Файл взят с сайта - http://www.natahaus.ru/

где есть ещё множество интересных и редких книг,
программ и прочих вещей.
Данный файл представлен исключительно в
ознакомительных целях.
Уважаемый читатель!
Если вы скопируете его,
Вы должны незамедлительно удалить его
сразу после ознакомления с содержанием.
Копируя и сохраняя его Вы принимаете на себя всю
ответственность, согласно действующему
международному законодательству .
Все авторские права на данный файл
сохраняются за правообладателем.
Любое коммерческое и иное использование
кроме предварительного ознакомления запрещено.
Публикация данного документа не преследует за
собой никакой коммерческой выгоды. Но такие документы
способствуют быстрейшему профессиональному и
духовному росту читателей и являются рекламой
бумажных изданий таких документов.

Все авторские права сохраняются за правообладателем.
Если Вы являетесь автором данного документа и хотите
дополнить его или изменить, уточнить реквизиты автора
или опубликовать другие документы, пожалуйста,
свяжитесь с нами по e-mail - мы будем рады услышать ваши
пожелания.

ПРЕДИСЛОВИЕ К КОМПЬЮТЕРНОМУ ИЗДАНИЮ

Уважаемый читатель !

Вашему вниманию предлагается электронный вариант книги «Казарин О.В. Теория и практика защиты программ». Данная работа является дополненным и существенно переработанным вариантом издания «Казарин О.В. Безопасность программного обеспечения компьютерных систем. - М.: МГУЛ, 2003», опубликованным, в том числе, и на данном сайте.

Автор по-прежнему будет очень признателен всем заинтересованным читателям, которые выскажут свои замечания, предложения и пожелания по сути и деталям представленной Вам работы.

С благодарностью,

автор

О.В. КАЗАРИН

Теория и практика защиты программ

Москва
2004

УДК 681.322

Казарин О.В.
Теория и практика защиты программ. – 2004. – 450 с.

В книге рассмотрены теоретические и прикладные аспекты проблемы защиты компьютерных программ от различного рода злоумышленных действий. Особое внимание уделено моделям и методам создания высокозащищенных и алгоритмически безопасных программ для применения в системах критических приложений.

Книга предназначена для ученых и практиков в области защиты программного обеспечения для современных компьютерных систем, предназначенных для применения в различных областях человеческой деятельности. Кроме того, книга может служить пособием по дисциплинам «Защита информации», «Программное обеспечение и средства его защиты», «Обеспечение безопасности программного обеспечения автоматизированных систем» для университетов, колледжей и курсов повышения квалификации.

Рецензенты: *д-р техн. наук, проф. Л.М. Ухлинов;*
канд. техн. наук, ст. научн. сотр. И.В. Марков;
В.С. Максимов

© О.В. Казарин, 2004

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ ... 8
ГЛАВА 1. ВВЕДЕНИЕ В ТЕОРИЮ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ... 12
1.1. Зачем и от кого нужно защищать программное обеспечение компьютерных систем ... 12
1.2. Угрозы безопасности программного обеспечения ... 16
1.3. Принятая аксиоматика и терминология ... 21
1.4. Жизненный цикл программного обеспечения компьютерных систем. Технологическая и эксплуатационная безопасность программ ... 25
1.5. Модели угроз безопасности программного обеспечения ... 27
1.6. Основные принципы обеспечения безопасности ПО ... 50
ГЛАВА 2. ФОРМАЛЬНЫЕ МЕТОДЫ ДОКАЗАТЕЛЬСТВА ПРАВИЛЬНОСТИ ПРОГРАММ И ИХ СПЕЦИФИКАЦИЙ ... 52
2.1. Общие положения ... 52
2.2. Предусловия и постусловия в доказательствах правильности ... 55
2.3. Правила вывода (доказательства) ... 56
2.4. Применение правил вывода ... 66
2.5. Пример доказательства правильности программы для алгоритма дискретного экспоненцирования ... 68
ГЛАВА 3. КОНФИДЕНЦИАЛЬНЫЕ ВЫЧИСЛЕНИЯ ... 74
3.1. Водные замечания по проблематике конфиденциальных вычислений ... 74
3.2. Описание используемых примитивов, схем и протоколов . 77
3.3. Обобщенные модели для сети синхронно и асинхронно взаимодействующих процессоров ... 81
3.4. Конфиденциальное вычисление функции ... 91
3.5. Проверяемые схемы разделения секрета как конфиденциальное вычисление функции ... 92
3.6. Синхронные конфиденциальные вычисления ... 101
3.7. Асинхронные конфиденциальные вычисления ... 113
3.8. RL-прототип модели синхронных конфиденциальных вычислений ... 142

ГЛАВА 4. САМОТЕСТИРУЮЩИЕСЯ И САМОКОРРЕКТИРУЮЩИЕСЯ ПРОГРАММЫ 147
4.1. Вводные замечания 147
4.2. Общие принципы создания двухмодульных вычислительных процедур и методология самотестирования 147
4.3. Устойчивость, линейная и единичная состоятельность ... 150
4.4. Метод создания самокорректирующейся процедуры вычисления теоретико-числовой функции дискретного экспоненцирования 152
4.5. Метод создания самотестирующейся расчетной программы с эффективным тестирующим модулем 156
4.6. Исследования процесса верификации расчетных программ 159
4.7. Области применения самотестирующихся и самокорректирующихся программ и их сочетаний 161
ГЛАВА 5. ЗАЩИТА ПРОГРАММ И ЗАБЫВАЮЩЕЕ МОДЕЛИРОВАНИЕ НА RAM-МАШИНАХ 178
5.1. Основные положения 178
5.2. Моделирование на забывающих RAM-машинах 180
5.3. Модели и определения 182
5.4. Преобразования, защищающие программное обеспечение 186
5.5. Определение забывающей RAM-машины и забывающего моделирования 188
5.6. Сведение защиты программ к забывающему моделированию на RAM-машине 191
5.7. Нетривиальное решение задачи забывающего моделирования 194
5.8. Заключительные замечания 201
ГЛАВА 6. КРИПТОПРОГРАММИРОВАНИЕ 203
6.1. Криптопрограммирование посредством использования инкрементальных алгоритмов 203
6.2. Основные элементы инкрементальной криптографии 205
6.3. Методы защиты данных посредством инкрементальных алгоритмов маркирования 207
6.4. Вопросы стойкости инкрементальных схем 212

6.5. Применение инкрементальных алгоритмов для защиты от вирусов ... 213
ГЛАВА 7. МЕТОДЫ И СРЕДСТВА АНАЛИЗА БЕЗОПАСНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ... 215
7.1. Общие замечания ... 215
7.2. Контрольно-испытательные методы анализа безопасности программного обеспечения ... 215
7.3. Логико-аналитические методы контроля безопасности программ ... 219
7.4. Сравнение логико-аналитических и контрольно-испытательных методов анализа безопасности программ ... 223
7.5. Способы тестирования программного обеспечения при испытаниях его на технологическую безопасность ... 226
7.6. Метод расчета вероятности наличия РПС на этапе испытаний программного обеспечения вычислительных задач ... 253
7.7. Подходы к исследованию сложных программных комплексов ... 262
ГЛАВА 8. МЕТОДЫ ОБЕСПЕЧЕНИЯ НАДЕЖНОСТИ ПРОГРАММ, ИСПОЛЬЗУЕМЫЕ ДЛЯ КОНТРОЛЯ ИХ ТЕХНОЛОГИЧЕСКОЙ БЕЗОПАСНОСТИ ... 274
8.1. Исходные данные, определения и условия ... 274
8.2. Краткий анализ существующих моделей надежности программного обеспечения ... 276
8.3. Описание модели Нельсона ... 280
8.4. Оценка технологической безопасности программ на базе метода Нельсона ... 285
ГЛАВА 9. ПОДХОДЫ К ЗАЩИТЕ РАЗРАБАТЫВАЕМЫХ ПРОГРАММ ОТ АВТОМАТИЧЕСКОЙ ГЕНЕРАЦИИ ИНСТРУМЕНТАЛЬНЫМИ СРЕДСТВАМИ ПРОГРАММНЫХ ЗАКЛАДОК ... 287
9.1. Способы внедрения РПС посредством инструментальных средств ... 287
9.2. Возможные методы защиты программ от потенциально опасных инструментальных средств ... 290

ГЛАВА 10. МЕТОДЫ ИДЕНТИФИКАЦИИ ПРОГРАММ И ИХ ХАРАКТЕРИСТИК ... 299
10.1. Идентификация программ по внутренним характеристикам ... 299
10.2. Способы оценки подобия целевой и исследуемой программ с точки зрения наличия программных дефектов ... 300
ГЛАВА 11. МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ПРОГРАММ ОТ КОМПЬЮТЕРНЫХ ВИРУСОВ ... 306
11.1. Общая характеристика и классификация компьютерных вирусов ... 306
11.2. Общая характеристика средств нейтрализации компьютерных вирусов ... 311
11.3. Классификация методов защиты от компьютерных вирусов ... 313
ГЛАВА 12. МЕТОДЫ ЗАЩИТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ОТ ИССЛЕДОВАНИЯ ... 318
12.1. Классификация средств исследования программ ... 318
12.2. Способы защиты программ от исследования ... 323
12.3. Способы встраивания защитных механизмов в программное обеспечения ... 327
12.4. Обфускация программ ... 329
ГЛАВА 13. МЕТОДЫ И СРЕДСТВА ОБЕСПЕЧЕНИЯ ЦЕЛОСТНОСТИ И ДОСТОВЕРНОСТИ ИСПОЛЬЗУЕМОГО ПРОГРАММНОГО КОДА ... 333
13.1. Методы защиты программ от несанкционированных изменений ... 333
13.2. Схема подписи с верификацией по запросу ... 335
13.3. Примеры применения схемы подписи с верификацией по запросу ... 341
ГЛАВА 14. ОСНОВНЫЕ ПОДХОДЫ К ЗАЩИТЕ ПРОГРАММ ОТ НЕСАНКЦИОНИРОВАННОГО КОПИРОВАНИЯ ... 343
14.1. Основные функции средств защиты от копирования ... 343
14.2. Основные методы защиты от копирования ... 344
ГЛАВА 15. ПРАВОВАЯ И ОРГАНИЗАЦИОННАЯ ПОДДЕРЖКА ПРОЦЕССОВ РАЗРАБОТКИ И ПРИМЕНЕНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ЧЕЛОВЕЧЕСКИЙ ФАКТОР ... 349

15.1. Стандарты и другие нормативные документы, регламентирующие защищенность программного обеспечения и обрабатываемой информации 349
15.2. Сертификационные испытания программных средств 359
15.3. Безопасность программного обеспечения и человеческий фактор. Психология программирования.. 362
ЗАКЛЮЧЕНИЕ 372
ПРИЛОЖЕНИЯ.. 390
Приложение 1. Основные результаты из теории сложности вычислений и криптологии 390
Приложение 2. Перечень типовых дефектов разработки, влияющих на безопасность ПО, и программных закладок, замаскированных под дефекты разработки. Формы проявления программных дефектов..................................... 420
Приложение 3. Характеристики программ с точки зрения влияния на их защищенность и результаты работы........... 423
Приложение 4. Перечень международных нормативных документов, связанных с проблематикой обеспечения безопасности ПО 424
Приложение 5. Практическая реализация технологии контроля отсутствия недекларированных возможностей в программных продуктах при их сертификации по требованиям безопасности информации 429
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ.. 448

ПРЕДИСЛОВИЕ

«Целью настоящего курса является углубление и развитие трудностей, лежащих в основе современной теории...»

А.А. Власов

Из кн. «Физики шутят». – М.: Мир, 1993.

Центральным информационно-активным звеном любых компьютерных систем является их математическое, программное, информационное и лингвистическое обеспечение. Современные компьютеры и сети компьютеров, обладающие «потрясающими» вычислительными, информационными и телекоммуникационными возможностями, со своим сложным «внутренним технологическим миром», остаются широким полем деятельности для человека, который создает и совершенствует и сами компьютеры, и те задачи, которые они решают. При этом основным техническим инструментом для этого является программное обеспечение, которое наряду с интеллектом человека, его навыками и знаниями, позволяет создавать сложные и порою удивительные компьютерные объекты, существенно расширяющие горизонты деятельности человека, облегчающие нашу повседневную жизнь и делающие ее активнее и разнообразнее.

Программное обеспечение современных компьютерных систем является очень сложным изделием, при создании и функционировании которого активно используются автоматизированные средства его разработки и общесистемное программное обеспечение, объем и сложность которого могут превышать прикладное программное обеспечение на порядки. Поэтому в общем случае, обеспечение абсолютного качества программных продуктов представляет собой практически неразрешимую задачу, что является причиной того, что ни один программист, ни одна организация-разработчик не гарантирует полноценной надежности создаваемого программного продукта. При этом

особую сложность, наряду с поиском, локализацией и устранением программных ошибок, представляет собой обнаружение дефектов, преднамеренно вносимых, как на этапе создания программных комплексов, так и их эксплуатации.

Кроме того, существенный урон производителю программных продуктов наносят такие несанкционированные действия, как несанкционированное копирование программ, их незаконное распространение и использование. Это наносит значительный нравственный и материальный ущерб фирмам-изготовителям программного обеспечения, а часто и легитимным потребителям программного продукта. Поэтому многие разработчики задаются вопросом, можно ли наряду с правовым и организационным обеспечением процесса разработки и эксплуатации программ, осуществить и научно-технические мероприятия, позволяющие защищаться от подобных злоумышленных действий.

Таким образом, необходимость внесения в программное обеспечение защитных функций на всем протяжении его жизненного цикла от этапа уяснения замысла на создание программ и их разработки до этапов испытаний, эксплуатации, модернизации и сопровождения программ, не вызывает сомнений.

В связи с этим, в гл. 1 рассмотрены методологические основы проблемы защиты программ различных объектов автоматизации. Описаны жизненный цикл современных программных комплексов, модели угроз и принципы обеспечения безопасности программного обеспечения.

В главах 2-10 рассмотрены современные методы обеспечения безопасности программ на этапе их разработки и испытаний. Важное место отводится методам создания алгоритмически безопасного программного обеспечения, позволяющим «игнорировать», а в ряде случаев и устранять, программные дефекты деструктивного характера. При этом в главах 2 и 3 рассматриваются методы высокоуровневой защиты программ от так называемых «программных закладок», а в главах 4 и 5 – теоретические основы защиты программ от компьютерных вирусов и от копирования. В главах 6-10 рассматриваются вопросы обеспечения технологической безопасности программ, реализуемые на этапах тестирования и испытания

программных комплексов и методы защиты программ от генерации программных закладок инструментальными средствами.

Современные методы обеспечения эксплуатационной безопасности программного обеспечения рассматриваются в главах 11-14. Основное внимание уделено методам обеспечения целостности и достоверности программного кода, защите программ от несанкционированного копирования и распространения. Для этих же целей служат также и нормативно-правовые и организационно-технические методы обеспечения качества разработки и эксплуатации программ, а также современные методы сертификации программных комплексов. Данные вопросы рассматриваются в заключительной 15-й главе книги. Кроме того, исследуются особенности поведения программиста – разработчика, который может осуществлять, в том числе, широкий набор злоумышленных действий.

В заключении публикуется обширный список литературы, посвященной современной проблематике защиты программного обеспечения, а также ряд приложений, в которых приводятся необходимые базовые сведения из теории сложности вычислений и криптологии, вспомогательные сведения о характеристиках и типах программ, так или иначе связанных с проявлением программных дефектов, список международных рекомендаций и стандартов, посвященных проблематике обеспечения безопасности программ. Кроме того, в приложении рассматриваются вопросы применения современных контрольно-испытательных методов для проведения сертификационных испытаний на отсутствие в заявленных для сертификации программах недекларированных возможностей.

Главы книги расположены так, что предлагаемый для прочтения материал, от начальных глав к последующим, соответствует жизненному циклу программного обеспечения современных компьютерных систем. При этом сущность процесса разработки и эксплуатация программ предполагает, что на начальных этапах их жизненного цикла необходимо больше уделять внимание теоретическим аспектам разработки безопасного программного обеспечения. Поэтому предлагаемый в начальных главах материал предполагает определенную теоретическую подготовку читателя в области теории алгоритмов, теории сложности вычислений, теории кодирования, теории информации, криптографии и криптоанализа и теории вероятностей. Тем не менее, в конце каждой главы автор

постарался привести прикладные аспекты применения предлагаемых теоретических конструкций.

Прикладные методы защиты программного обеспечения, к которым наряду с их теоретическими основаниями можно отнести такие методы как методы защиты программ от компьютерных вирусов, от несанкционированного копирования, методы обеспечения целостности и достоверности программного кода имеют достаточно широкую теоретическую и практическую базу и широко освещены в современной отечественной и зарубежной литературе. Тем не менее, для полноты изложения материала такие методы достаточно детально рассматриваются и предлагаются некоторые оригинальные решения вышеуказанных задач защиты.

Автор выражает надежду, что ему удалось затронуть большой спектр методов защиты современных программ от известных злоумышленных действий в их отношении. Таким образом, весь совокупный набор методов обеспечения безопасности программного обеспечения на всех этапах его жизненного цикла, позволяет говорить о возникновении и развитии современной методологии защиты программ и ее основных элементах. В то же время автор не претендует на полноту изложения материала в этой сложной и постоянно развивающейся области, а также на теоретическую глубину материала по отдельным направлениям этой методологии. И поэтому будет очень признателен всем заинтересованным читателям, которые выскажут свои замечания, предложения и пожелания по сути и деталям представленной работы.

С благодарностью,

автор

ГЛАВА 1. ВВЕДЕНИЕ В ТЕОРИЮ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. ЗАЧЕМ И ОТ КОГО НУЖНО ЗАЩИЩАТЬ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ КОМПЬЮТЕРНЫХ СИСТЕМ

Безопасность программного обеспечения в широком смысле является свойством данного программного обеспечения функционировать без проявления различных негативных последствий для конкретной компьютерной системы. Под уровнем безопасности программного обеспечения (ПО) понимается вероятность того, что при заданных условиях в процессе его эксплуатации будет получен *функционально пригодный результат*. Причины, приводящие к функционально непригодному результату, могут быть разными: сбои компьютерных систем, ошибки программистов и операторов, дефекты в программах. При этом дефекты принято рассматривать двух типов: преднамеренные и непреднамеренные. Первые являются, как правило, результатом злоумышленных действий, вторые - ошибочных действий человека.

При исследовании проблем защиты ПО от преднамеренных дефектов неизбежна постановка следующих вопросов:

- *кто потенциально может осуществить практическое внедрение программных дефектов деструктивного воздействия в исполняемый программный код?*
- *каковы возможные мотивы действий субъекта, осуществляющего разработку таких дефектов?*
- *как можно идентифицировать наличие программного дефекта?*
- *как можно отличить преднамеренный программный дефект от программной ошибки?*
- *каковы наиболее вероятные последствия активизации деструктивных программных средств при эксплуатации компьютерных систем?*

При ответе на первый вопрос следует отметить, что это - непосредственные разработчики алгоритмов и программ для компьютерных систем. Они хорошо знакомы с технологией разработки программных средств, имеют опыт разработки алгоритмов и программ для конкретных прикладных систем, знают тонкости существующей технологии отработки и испытаний программных компонентов и

представляют особенности эксплуатации и целевого применения разрабатываемой компьютерной системы (КС). Кроме того, при эксплуатации программных комплексов возможен следующий примерный алгоритм внесения программного дефекта: дизассемблирование исполняемого программного кода, получение исходного текста, привнесение в него деструктивной программы, повторная компиляция, корректировка идентификационных признаков программы (в связи с необходимостью получения программы «схожей» с оригиналом). Таким образом, манипуляции подобного рода могут сделать посторонние высококлассные программисты, имеющие опыт разработки и отладки программ на ассемблерном уровне.

В качестве предположений при ответе на второй вопрос следует отметить, что алгоритмические и программные закладки могут быть реализованы в составе программного компонента вследствие следующих факторов:

- в результате инициативных злоумышленных действий непосредственных разработчиков алгоритмов и программ;
- в результате штатной деятельности специальных служб и организаций, а также отдельных злоумышленников;
- в результате применения инструментальных средств проектирования ПО, несущих вредоносное свойство автоматической генерации деструктивных программных средств.

Для описания мотивов злоумышленных действий при разработке программных компонентов необходим психологический «портрет» злоумышленника, что требует проведения специальных исследований психологов и криминологов в области психологии программирования (психологии криминального программирования, см. главу 15). Однако некоторые мотивы очевидны уже сейчас и могут диктоваться следующим:

- неустойчивым психологическим состоянием алгоритмистов и программистов, обусловленным сложностью взаимоотношений в коллективе, перспективой потерять работу, резким снижением уровня благосостояния, отсутствием уверенности в завтрашнем дне и т.п., в результате чего может возникнуть, а впоследствии быть реализована, мысль отмщения;
- неудовлетворенностью личных амбиций непосредственного разработчика алгоритма или программы, считающего себя непризнанным талантом, в результате чего может появиться

стремление доказать и показать кому-либо (в том числе и самому себе) таким способом свои высокие интеллектуальные возможности;

- перспективой выезда за границу на постоянное место жительства (перспективной перехода в другую организацию, например, конкурирующую) с надеждой получить вознаграждение за сведения о программной закладке и механизме ее активизации, а также возможностью таким способом заблокировать применение определенного класса программных средств по избранному месту жительства (месту работы);
- потенциальной возможностью получить вознаграждение за устранение возникшего при испытаниях или эксплуатации системы «программного отказа» и т.п.

Кроме того, необходимо иметь в виду, что в конструировании вредоносной программы, так или иначе, присутствует притягательное творческое начало, которое само по себе может стать целью. При этом сам «творец» может слабо представлять все возможные результаты и последствия применения своей «конструкции», либо вообще не задумываться о них.

Таким образом, правомерно утверждать, что вредоносные программы, в отличие от широко применяемых электронных закладок, являются более изощренными объектами, обладающими большей скрытностью и эффективностью применения.

Ответы на три последних вопроса можно найти в рамках быстро развивающейся методологии обеспечения безопасности программных средств и оценки уровня их защищенности (разделы 2-14).

До сих пор мы рассматривали защиту программного обеспечения от разрушающих программных средств. Однако применение злоумышленником только этих деструктивных средств, не исчерпывает всего круга проблем, связанных с проблематикой обеспечения безопасности программ. Существует широкий спектр угроз, отнесенных к несанкционированному копированию, незаконному получению, распространению и использованию программных продуктов.

«Программное пиратство» является, в связи с большими вероятными потерями, одной из основных проблем для фирм-разработчиков, так или иначе, связанных с созданием и реализацией программного обеспечения. Программные пираты покупают или «берут на прокат» необходимое

программное обеспечение и, если в нем нет соответствующей защиты, они могут скопировать программы и использовать их без соответствующей оплаты (регистрации, подтверждения лицензионных соглашений, авторских прав и т.п.) по своему усмотрению. Таким образом, вопрос защиты ПО от несанкционированного копирования, распространения и использования является одним из наиболее важных в компьютерной практике. Все известные методы защиты ПО от несанкционированного копирования и распространения относятся к организационно-правовым и инженерно-техническим методам. При решении указанных задач защиты сегодня практически полностью отсутствуют теоретические основания. В частности, нет четкого определения собственно проблемы и определения того, что собственно должно являться ее удовлетворительным решением.

Задача защиты программ от несанкционированного копирования, на наш взгляд, заключается в обеспечении невозможности нахождения никакого эффективного метода создания выполнимых копий программ, в то время как задача защиты программ от несанкционированного распространения состоит в обеспечении того факта, что только производитель ПО смог бы доказать в суде, что *именно он разработал это ПО*.

При ответе на вопрос о несанкционированном копировании программ необходимо ответить на следующие вопросы:

- *что может сделать злоумышленник («пират») в ходе попыток изучения программы?*
- *что является существенными знаниями о программе?*
- *что является спецификацией программы?*

Для того чтобы ответить на вышеупомянутые вопросы необходимо рассматривать наиболее злонамеренное поведение злоумышленника. То есть необходимо рассматривать сценарий *наихудшего случая*, когда предполагается, что злоумышленник может выполнять преобразованную программу на произвольных данных по своему усмотрению и может модифицировать данные, циркулирующие в вычислительной системе, произвольным образом.

Таким образом, исходя из общего понимания проблемы необходимости защиты программ современных компьютерных систем, от целого набора злоумышленных действий, можно перейти к совокупному определению источников угроз безопасности программного обеспечения и обобщенных сценариев действий потенциального злоумышленника, к

подробному описанию и определению общего характера защитных действий.

1.2. УГРОЗЫ БЕЗОПАСНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.2.1. Разрушающие программные средства

Угрозы безопасности информации и программного обеспечения КС возникают как в процессе их эксплуатации, так и при создании этих систем, что особенно характерно для процесса разработки ПО, баз данных и других информационных компонентов КС.

Наиболее уязвимы с точки зрения защищенности информационных ресурсов являются так называемые *критические компьютерные системы*. Под критическими компьютерными системами будем понимать сложные компьютеризированные организационно-технические и технические системы, блокировка или нарушение функционирования которых потенциально приводит к потере устойчивости организационных систем государственного управления и контроля, утрате обороноспособности государства, разрушению системы финансового обращения, дезорганизации систем энергетического и коммуникационно - транспортного обеспечения государства, глобальным экологическим и техногенным катастрофам.

При решении проблемы повышения уровня защищенности информационных ресурсов КС необходимо исходить из того, что наиболее вероятным информационным объектом воздействия будет выступать программное обеспечение, составляющее основу комплекса средств получения, семантической переработки, распределения и хранения данных, используемых при эксплуатации критических систем.

В настоящее время одним из наиболее опасных средств информационного воздействия на компьютерные системы являются программы - вирусы или компьютерные вирусы.

Под *компьютерным вирусом* следует понимать программы, способные размножаться, прикрепляться к другим программам, передаваться по телекоммуникационным каналам.

В качестве основных средств вредоносного (деструктивного) воздействия на КС необходимо, наряду с компьютерными вирусами, рассматривать алгоритмические и программные закладки.

Под *алгоритмической закладкой* будем понимать преднамеренное завуалированное искажение какой-либо части алгоритма решения задачи,

либо построение его таким образом, что в результате конечной программной реализации этого алгоритма в составе программного компонента или комплекса программ, последние будут иметь ограничения на выполнение требуемых функций, заданных спецификацией, или вовсе их не выполнять при определенных условиях протекания вычислительного процесса, задаваемого семантикой перерабатываемых программой данных. Кроме того, возможно появление у программного компонента функций, не предусмотренных прямо или косвенно спецификацией, и которые могут быть выполнены при строго определенных условиях протекания вычислительного процесса.

Под *программной закладкой* будем понимать совокупность операторов и (или) операндов, преднамеренно в завуалированной форме включаемую в состав выполняемого кода программного компонента на любом этапе его разработки. Программная закладка реализует определенный несанкционированный алгоритм с целью ограничения или блокирования выполнения программным компонентом требуемых функций при определенных условиях протекания вычислительного процесса, задаваемого семантикой перерабатываемых программным компонентом данных, либо с целью снабжения программного компонента не предусмотренными спецификацией функциями, которые могут быть выполнены при строго определенных условиях протекания вычислительного процесса.

Действия алгоритмических и программных закладок условно можно разделить на три класса: изменение функционирования вычислительной системы (сети), несанкционированное считывание информации и несанкционированная модификация информации, вплоть до ее уничтожения. В последнем случае под информацией понимаются как данные, так и коды программ. Следует отметить, что указанные классы воздействий могут пересекаться.

В первом классе воздействий выделим следующие:

- уменьшение скорости работы вычислительной системы (сети);
- частичное или полное блокирование работы системы (сети);
- имитация физических (аппаратурных) сбоев работы вычислительных средств и периферийных устройств;
- переадресация сообщений;
- обход программно-аппаратных средств криптографического преобразования информации;

- обеспечение доступа в систему с непредусмотренных периферийных устройств.

Несанкционированное считывание информации, осуществляемое в автоматизированных системах, направлено на:

- считывание паролей и их отождествление с конкретными пользователями;
- получение секретной информации;
- идентификацию информации, запрашиваемой пользователями;
- подмену паролей с целью доступа к информации;
- контроль активности абонентов сети для получения косвенной информации о взаимодействии пользователей и характере информации, которой обмениваются абоненты сети.

Несанкционированная модификация информации является наиболее опасной разновидностью воздействий программных закладок, поскольку приводит к наиболее опасным последствиям. В этом классе воздействий можно выделить следующие:

- разрушение данных и кодов исполняемых программ внесение тонких, трудно обнаруживаемых изменений в информационные массивы;
- внедрение программных закладок в другие программы и подпрограммы (вирусный механизм воздействий);
- искажение или уничтожение собственной информации сервера и тем самым нарушение работы сети;
- модификация пакетов сообщений.

Из изложенного следует вывод о том, что алгоритмические и программные закладки имеют широкий спектр воздействий на информацию, обрабатываемую вычислительными средствами в КС. Следовательно, при контроле технологической безопасности программного обеспечения необходимо учитывать его назначение и состав аппаратных средств и общесистемного программного обеспечения (программно-аппаратную среду) КС.

С точки зрения времени внесения программных закладок в программы их можно разделить на две категории: априорные и апостериорные, то есть закладки, внесенные при разработке ПО (или «врожденные») и закладки, внесенные при испытаниях, эксплуатации или модернизации ПО (или «приобретенные») соответственно. Хотя последняя разновидность закладок и относятся больше к проблеме обеспечения эксплуатационной, а

не технологической безопасности ПО, однако методы тестирования программных комплексов, вероятностные методы расчета наличия программных дефектов и методы оценивания уровня безопасности ПО могут в значительной мере пересекаться и дополнять друг друга. Тем более что действие программной закладки после того как она была внесена в ПО либо на этапе разработки, либо на последующих этапах жизненного цикла ПО, практически не будет ничем не отличаться.

Таким образом, рассмотренные программные средства деструктивного воздействия по своей природе носят, как правило, разрушительный, вредоносный характер, а последствия их активизации и применения могут привести к значительному или даже непоправимому ущербу в тех областях человеческой деятельности, где применение компьютерных систем является жизненно необходимым. В связи с этим такие вредоносные программы будем называть *разрушающими программными средствами* (*РПС*), а их обобщенная классификация может выглядеть следующим образом:

- компьютерные вирусы - программы, способные размножаться, прикрепляться к другим программам, передаваться по линиям связи и сетям передачи данных, проникать в электронные телефонные станции и системы управления и выводить их из строя;
- программные закладки - программные компоненты, заранее внедряемые в компьютерные системы, которые по сигналу или в установленное время приводятся в действие, уничтожая или искажая информацию, или дезорганизуя работу программно-технических средств;
- способы и средства, позволяющие внедрять компьютерные вирусы и программные закладки в компьютерные системы и управлять ими на расстоянии.

1.2.2. Несанкционированное копирование, распространение и использование программ

Так как программное обеспечение в большинстве случаев является товаром, сама сущность человеческой психологии будет предполагать процесс его «купли-продажи», в том числе и незаконное приобретение, распространение и использование программ, в том числе и для спекулятивных целей, для незаконного проката и т.п.

Подобные злоумышленные действия направлены на преодоление юридических (правовых), административно-организационных,

административно-экономических и технических барьеров. Так как предлагаемая читателю книга носит, в основном, научно-технический характер мы не будем останавливаться на первых трех видах злоупотреблений. Отметим только, что вследствие того, что правовые вопросы в области защиты программ от копирования и незаконного распространения и использования в нашей стране практически не отрегулированы, для потенциального злоумышленника существует широкий спектр несанкционированных действий, которые не подпадают под действие существующего законодательства. Деятельность в данной области регламентируются в основном подзаконными актами и ведомственными организационно-распорядительными документами, совокупность которых, в свою очередь, не обладает полнотой и не носит системного характера (некоторые из этих нормативно-технических документов рассматриваются в главе 15 и в приложении).

Административно-экономические меры [РД], проводимые фирмами изготовителями программного обеспечения, предусматривают всяческое стимулирование легального приобретения программ. Такое стимулирование проводится только для зарегистрированных пользователей, легально покупающих программные продукты, что значительно позволяет сэкономить средства при использовании программного продукта, а иногда даже делает невыгодным его нелегальное копирование, использование и распространение. Для предотвращения незаконных действий с программным обеспечением возможны следующие сценарии поведения для зарегистрированных пользователей:

- периодическое получение новых версий программного продукта, соответствующей документации, специальных журналов;
- возможность получения оперативной консультации;
- проведение семинаров и курсов по обучению использованию программного продукта;
- предоставление скидки при покупке следующей версии продукта.

Применение технических методов и средств защиты программ от копирования, распространения и использования в некотором смысле ограничено, так как именно организационная и правовая составляющая совокупности всех мер, скорее всего, являются определяющей.

Действия злоумышленника по преодолению технической составляющей защиты сводятся по существу к реализации следующей очевидной парадигмы.

Так как любое программное обеспечение является по существу некоторой битовой строкой, то как любая последовательность битов оно может быть скопировано общедоступными системными средствами, если не существуют соответствующих аппаратных средств для защиты от копирования. Применяя известные или оригинальные методы исследования программ, задаваемых их объектным или исполняемым кодом, можно осуществить выделение алгоритма программы (или какой-либо интересующей его части) и выполнить его семантический анализ с целью получения ответа на интересующие вопросы, например, о степени ее защищенности, о возможных мерах по преодолению защиты от копирования и т.п.

www.kiev-security.org.ua
BEST rus DOC FOR FULL SECURITY

После копирования программы и ее глобального и детального изучения, злоумышленник может попытаться, изменив идентификационные признаки программы, с целью ее дальнейшей выдачи за свою собственную, осуществить незаконное распространение и/или продажу программного обеспечения.

Таким образом, защита (техническая составляющая) от подобных злоумышленных действий может заключаться в сочетании методов защиты от исследования и защиты от копирования, методов обеспечения целостности и достоверности (а в ряде случаев и конфиденциальности) программ, в обеспечении их неизменяемыми идентификационными характеристиками.

1.3. Принятая аксиоматика и терминология

1.3.1. Основные предположения и ограничения

В качестве вычислительной среды рассматривается совокупность установленных для данной КС алгоритмов использования системных ресурсов, программного и информационного обеспечения, которая потенциально может быть представлена пользователю для решения

прикладных задач. Операционной средой является совокупность функционирующих в данный момент времени элементов вычислительной среды, участвующих в процессе решения конкретной задачи пользователя.

Принципиально возможность программного воздействия определяется открытостью вычислительной системы, под которой понимается предоставление пользователю возможности формировать элементную базу вычислительной среды под свои задачи, а также возможность использовать в полном объеме системные ресурсы, что является неотъемлемым признаком автоматизированных рабочих мест на базе персональных ЭВМ.

В качестве средства борьбы с «пассивными» методами воздействия допускается создание служб безопасности, ограничивающих доступ пользователей к элементам вычислительной среды, в первую очередь к программам обработки чувствительной информации. Предполагается, что возможности «активных» методов воздействия значительно шире.

Необходимым условием для отнесения программы к классу разрушающих программных средств является наличие в ней процедуры нападения, которую можно определить как процедуру нарушения целостности вычислительной среды, поскольку объектом нападения разрушающего программного средства (РПС) всегда выступает элемент этой среды.

При этом необходимо учитывать два фактора:

- любая прикладная программа, не относящаяся к числу РПС, потенциально может содержать в себе алгоритмические ошибки, появление которых при ее функционировании приведет к непреднамеренному разрушению элементов вычислительной среды;
- любая прикладная или сервисная программа, ориентированная на работу с конкретными входными данными может нанести непреднамеренный ущерб элементам операционной или вычислительной среды в случае, когда входные данные либо отсутствуют, либо не соответствуют заданным форматам их ввода в программу.

Для устранения указанной неопределенности по отношению к испытываемым программам следует исходить из предположения, что процедура нарушения целостности вычислительной среды введена в состав ПО умышленно. Кроме условия необходимости, целесообразно

ввести условия достаточности, которые обеспечат возможность описания РПС различных классов:

- достаточным условием для отнесения РПС к классу компьютерных вирусов является наличие в его составе процедуры саморепродукции;
- достаточным условием для отнесения РПС к классу средств несанкционированного доступа являются наличие в его составе процедуры преодоления защиты и отсутствия процедуры саморепродукции;
- достаточным условием для отнесения РПС к классу программных закладок является отсутствие в его составе процедур саморепродукции и преодоления защиты.

Предполагается наличие в РПС следующего набора возможных функциональных элементов:

- процедуры захвата (получения) управления;
- процедуры самомодификации («мутации»);
- процедуры порождения (синтеза);
- процедуры маскировки (шифрования).

Этих элементов достаточно для построения обобщенной концептуальной модели РПС, которая отражает возможную структуру (на семантическом уровне) основных классов РПС.

1.3.2. Используемая терминология

Разработка терминологии в области обеспечения безопасности ПО является базисом для формирования нормативно-правового обеспечения и концептуальных основ по рассматриваемой проблеме. Единая терминологическая база является ключом к единству взглядов в области, информационной безопасности, стимулирует скорейшее развитие методов и средств защиты ПО. Термины освещают основные понятия, используемые в рассматриваемой области на данный период времени. Определения освещают толкование конкретных форм, методов и средств обеспечения информационной безопасности.

Термины и определения

Непреднамеренный дефект - объективно и (или) субъективно образованный дефект, приводящий к получению неверных решений (результатов) или нарушению функционирования КС.

Преднамеренный дефект - криминальный дефект, внесенный субъектом для целенаправленного нарушения и (или) разрушения информационного ресурса.

Разрушающее программное средство (РПС) - совокупность программных и/или технических средств, предназначенных для нарушения (изменения) заданной технологии обработки информации и/или целенаправленного разрушения извне внутреннего состояния информационно-вычислительного процесса в КС.

Средства активного противодействия - средства защиты информационного ресурса КС, позволяющие блокировать канал утечки информации, разрушающие действия противника, минимизировать нанесенный ущерб и предотвращать дальнейшие деструктивные действия противника посредством ответного воздействия на его информационный ресурс.

Несанкционированный доступ - действия, приводящие к нарушению целостности, конфиденциальности и доступности информационных ресурсов.

Нарушитель (злоумышленник, противник) - субъект (субъекты), совершающие несанкционированный доступ к информационному ресурсу.

Модель угроз - вербальная, математическая, имитационная или натурная модель, формализующая параметры внутренних и внешних угроз безопасности ПО.

Оценка безопасности ПО - процесс получения количественных и/или качественных показателей информационной безопасности при учете преднамеренных и непреднамеренных дефектов в системе.

Система обеспечения информационной безопасности - объединенная совокупность мероприятий, методов и средств, создаваемых и поддерживаемых для обеспечения требуемого уровня безопасности информационного ресурса.

Информационная технология - упорядоченная совокупность организационных, технических и технологических процессов создания ПО и обработки, хранения и передачи информации.

Технологическая безопасность - свойство программного обеспечения и информации не быть преднамеренно искаженными и (или) начиненными избыточными модулями (структурами) диверсионного назначения на этапе создания КС.

Эксплуатационная безопасность - свойство программного обеспечения и информации не быть несанкционированно искаженными (измененными) на этапе их эксплуатации.

1.4. Жизненный цикл программного обеспечения компьютерных систем. Технологическая и эксплуатационная безопасность программ

Необходимость определения этапов жизненного цикла (ЖЦ) ПО обусловлена стремлением разработчиков к повышению качества ПО за счет оптимального управления разработкой и использования разнообразных механизмов контроля качества на каждом этапе, начиная от постановки задачи и заканчивая авторским сопровождением ПО. Наиболее общим представлением жизненного цикла ПО является модель в виде базовых этапов – процессов [Лип1], к которым относятся:

- системный анализ и обоснование требований к ПО;
- предварительное (эскизное) и детальное (техническое) проектирование ПО;
- разработка программных компонент, их комплексирование и отладка ПО в целом;
- испытания, опытная эксплуатация и тиражирование ПО;
- регулярная эксплуатация ПО, поддержка эксплуатации и анализ результатов;
- сопровождение ПО, его модификация и совершенствование, создание новых версий.

Данная модель является общепринятой и соответствует как отечественным нормативным документам в области разработки программного обеспечения, так и зарубежным. С точки зрения обеспечения технологической безопасности целесообразно рассмотреть более подробно особенности представления этапов ЖЦ.

Графическое представление моделей ЖЦ позволяет наглядно выделить их особенности и некоторые свойства процессов. Первоначально была создана каскадная модель ЖЦ [Лип1], в которой крупные этапы начинались друг за другом с использованием результатов предыдущих работ. Наиболее специфической является спиралевидная модель ЖЦ [Лип1]. В этой модели внимание концентрируется на итерационном процессе начальных этапов проектирования. На этих этапах последовательно создаются концепции, спецификации требований,

предварительный и детальный проект. На каждом витке уточняется содержание работ и концентрируется облик создаваемого ПО.

Для проектирования ПО сложной системы, особенно системы реального времени, целесообразно использовать общесистемную модель ЖЦ, основанную на объединении всех известных работ в рамках рассмотренных базовых процессов. Эта модель предназначена для использования при планировании, составлении рабочих графиков, управлении различными программными проектами.

Совокупность этапов данной модели ЖЦ целесообразно делить на две части, существенно различающихся особенностями процессов, технико-экономическими характеристиками и влияющими на них факторами.

В первой части ЖЦ производится системный анализ, проектирование, разработка, тестирование и испытания ПО. Номенклатура работ, их трудоемкость, длительность и другие характеристики на этих этапах существенно зависят от объекта и среды разработки. Изучение подобных зависимостей для различных классов ПО позволяет прогнозировать состав и основные характеристики графиков работ для новых версий ПО.

Этой совокупности этапов ЖЦ ПО соответствует процесс внесения в разрабатываемые программы определенных защитных функций. Этот процесс называется *обеспечением технологической безопасности* и характеризуется необходимостью предотвращения модификации ПО за счет внедрения РПС априорного типа (алгоритмических и программных закладок).

Вторая часть ЖЦ, отражающая поддержку эксплуатации и сопровождения ПО, относительно слабо связана с характеристиками объекта и среды разработки. Номенклатура работ на этих этапах более стабильна, а их трудоемкость и длительность могут существенно изменяться, и зависят от массовости применения ПО. Для любой модели ЖЦ обеспечение высокого качества программных комплексов возможно лишь при использовании регламентированного технологического процесса на каждом из этих этапов. Такой процесс поддерживается CASE-средствами (средствами автоматизации разработки ПО), которые целесообразно выбирать из имеющихся или создавать с учетом объекта разработки и адекватного ему перечня работ.

Этапам эксплуатации и сопровождения ПО соответствует процесс *обеспечения эксплуатационной безопасности ПО*. Этот процесс характеризуется необходимостью защиты программ от компьютерных вирусов и программных закладок апостериорного типа. Последние могут

внедряться за счет злонамеренного использования методов исследования программ и их спецификаций. Кроме того, на этапе обеспечения эксплуатационной безопасности ПО применяются методы защиты программ от несанкционированного копирования, распространения и использования.

1.5. Модели угроз безопасности программного обеспечения

1.5.1. Вводные замечания

Использование при создании программного обеспечения КС сложных операционных систем, инструментальных средств разработки ПО увеличивают потенциальную возможность внедрения в программы преднамеренных дефектов диверсионного типа. Помимо этого, при создании прикладного программного обеспечения всегда необходимо исходить из возможности наличия в коллективе разработчиков программистов - злоумышленников, которые в силу тех или иных причин могут внести в разрабатываемые программы РПС.

Характерным свойством РПС в данном случае является возможность внезапного и незаметного нарушения или полного вывода из строя КС. Функционирование РПС реализуется в рамках модели угроз безопасности ПО, основные элементы которой рассматриваются в следующем разделе.

1.5.2. Подходы к созданию модели угроз технологической безопасности ПО

Один из возможных подходов к созданию модели угроз технологической безопасности ПО КС может основываться на обобщенной концепции технологической безопасности компьютерной инфосферы [Е1,ЕП, ЮП1,ЮП2], которая определяет методологический базис, направленный на решение, в том числе, следующих основных задач:

- создания теоретических основ для практического решения проблемы технологической безопасности ПО;
- создания безопасных информационных технологий;
- развертывания системы контроля технологической безопасности компьютерной инфосферы.

Модель угроз технологической безопасности ПО должна представлять собой официально принятый нормативный документ, которым должен руководствоваться заказчики и разработчики программных комплексов.

Модель угроз должна включать:

- полный реестр типов возможных программных закладок;
- описание наиболее технологически уязвимых мест компьютерных систем (с точки зрения важности и наличия условий для скрытого внедрения программных закладок);
- описание мест и технологические карты разработки программных средств, а также критических этапов, при которых наиболее вероятно скрытое внедрение программных закладок;
- реконструкцию замысла структур, имеющих своей целью внедрение в ПО заданного типа (класса, вида) программных закладок диверсионного типа;
- психологический портрет потенциального диверсанта в компьютерных системах.

В указанной Концепции [ЕП] также оговариваются необходимость содержания в качестве приложения банка данных о выявленных программных закладках и описания связанных с их обнаружением обстоятельств, а также необходимость периодического уточнения и совершенствования модели на основе анализа статистических данных и результатов теоретических исследований.

На базе утвержденной модели угроз технологической безопасности компьютерной инфосферы, как обобщенного, типового документа должна разрабатываться прикладная модель угроз безопасности для каждого конкретного компонента защищаемого комплекса средств автоматизации КС. В основе этой разработки должна лежать схема угроз, типовой вид которой применительно к ПО КС представлен на рис.1.1.

Наполнение модели угроз технологической безопасности ПО должно включать в себя следующие элементы: матрицу чувствительности КС к «вариациям» ПО (то есть к появлению искажений), энтропийный портрет ПО (то есть описание «темных» запутанных участков ПО), реестр камуфлирующих условий для конкретного ПО, справочные данные о разработчиках и реальный (либо реконструированный) замысел злоумышленников по поражению этого ПО. На рис.1.2 приведен пример типового реестра угроз для сложных программных комплексов.

1.5.3. Элементы модели угроз эксплуатационной безопасности ПО

Анализ угроз эксплуатационной безопасности ПО КС позволяет, разделить их на два типа: случайные и преднамеренные, причем последние подразделяются на активные и пассивные. Активные угрозы направлены на изменение технологически обусловленных алгоритмов, программ

функциональных преобразований или информации, над которой эти преобразования осуществляются. Пассивные угрозы ориентированы на нарушение безопасности информационных технологий без реализации таких модификаций.

Вариант общей структуры набора потенциальных угроз безопасности информации и ПО на этапе эксплуатации КС приведен в табл.1.1.

Рассмотрим основное содержание данной таблицы. Угрозы, носящие случайный характер и связанные с отказами, сбоями аппаратуры, ошибками операторов и т.п. предполагают нарушение заданного собственником информации алгоритма, программы ее обработки или искажение содержания этой информации. Субъективный фактор появления таких угроз обусловлен ограниченной надежностью работы человека и проявляется в виде ошибок (дефектов) в выполнении операций формализации алгоритма функциональных преобразований или описания алгоритма на некотором языке, понятном вычислительной системе.

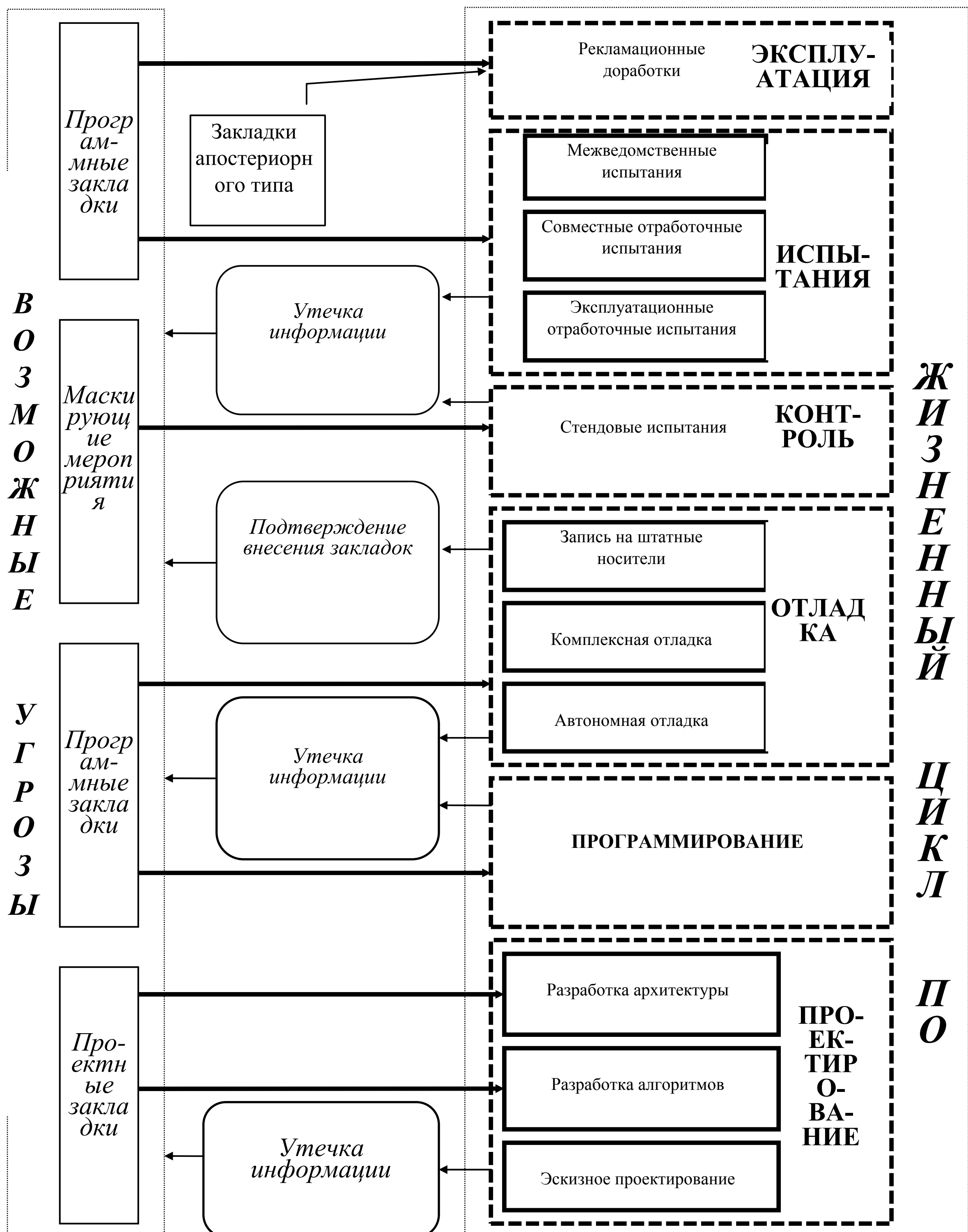


Рис.1.1. Схема угроз технологической безопасности ПО

ПРОЕКТИРОВАНИЕ

Проектные решения

Злоумышленный выбор нерациональных алгоритмов работы.

Облегчение внесения закладок и затруднение их обнаружения.

Внедрение злоумышленников в коллективы, разрабатывающие наиболее ответственные части ПО.

Используемые информационные технологии

Внедрение злоумышленников, в совершенстве знающих «слабые» места и особенности используемых технологий.

Внедрение информационных технологий или их элементов, содержащих программные закладки.

Внедрение неоптимальных информационных технологий.

Используемые аппаратно-технические средства

Поставка вычислительных средств, содержащих программные, аппаратные или программно-аппаратные закладки.

Поставка вычислительных средств с низкими эксплуатационными характеристиками.

Задачи коллективов разработчиков и их персональный состав

Внедрение злоумышленников в коллективы разработчиков программных и аппаратных средств.

Вербовка сотрудников путем подкупа, шантажа и т.п.

Рис.1.2. Пример типового реестра угроз технологической безопасности информации и ПО

КОДИРОВАНИЕ

Архитектура программной системы, взаимодействие ее с внешней средой и взаимодействие подпрограмм программной системы

Доступ к «чужим» подпрограммам и данным.

Нерациональная организация вычислительного процесса.

Неправильная организация динамически формируемых команд или параллельных вычислительных процессов.

Неправильная организация переадресации команд, запись злоумышленной информации в используемые программной системой или другими программами ячейки памяти.

Функции и назначение кодируемой части программной системы, взаимодействие этой части с другими подпрограммами

Формирование программной закладки, воздействующей на другие части программной системы.

Организация замаскированного спускового механизма программной закладки.

Формирование программной закладки, изменяющей структуру программной системы.

Технология записи программного обеспечения и исходных данных

Поставка программного обеспечения и технических средств со встроенными дефектами.

Продолжение рис.1.2.

ОТЛАДКА И ИСПЫТАНИЯ

Назначение, функционирование, архитектура программной системы

Встраивание программной закладки как в отдельные подпрограммы, так и в управляющую программу программной системы.

Формирование программной закладки с динамически формируемыми командами.

Неправильная организация переадресации отдельных команд программной системы.

Сведения о процессе испытаний (набор тестовых данных, используемые вычислительные средства, подразделения и лица, проводящие испытания, используемые модели)

Формирование набора тестовых данных, не позволяющих выявить программную закладку.

Поставка вычислительных средств, содержащих программные, аппаратные или программно-аппаратные закладки.

Формирование программной закладки, не обнаруживаемой с помощью используемой модели объекта в силу ее неадекватности описываемому объекту.

Вербовка сотрудников коллектива, проводящих испытания.

Продолжение рис.1.2.

КОНТРОЛЬ

Используемые процедуры и методы контроля

Формирование спускового механизма программной закладки, не включающего ее при контроле на безопасность.

Маскировка программной закладки путем внесения в программную систему ложных «непреднамеренных» дефектов.

Формирование программных закладок в ветвях программной системы, не проверяемых при контроле.

Формирование программных закладок, не позволяющих выявить их внедрение в программную систему путем контрольного суммирования.

Поставка программного обеспечения и вычислительной техники, содержащих программные, аппаратные и программно-аппаратные закладки.

Сведения о персональном составе контролирующего подразделения и испытываемых программных системах

Внедрение злоумышленников в контролирующее подразделение.

Вербовка сотрудников контролирующего подразделения.

Сбор информации об испытываемой программной системе.

Сведения об обнаруженных при контроле программных закладках

Разработка новых программных закладок при доработке программной системы.

Сведения об обнаруженных незлоумышленных дефектах и программных закладках

Сведения о доработках программной системы и подразделениях, их осуществляющих

Сведения о среде функционирования программной системы и ее изменениях.

Сведения о функционировании программной системы, доступе к ее загрузочному модулю и исходным данным, алгоритмах проверки сохранности программной системы и данных.

Продолжение рис.1.2.

Таблица 1.1

Угрозы нарушения безопасности ПО	Несанкционированные действия		
	Случайные	***Преднамеренные***	
		Пассивные	*Активные*
Прямые	Невыявленные ошибки программного обеспечения КС; отказы и сбои технических средств КС; ошибки операторов; неисправность средств защиты информации; скачки электропитания на технических средствах; старение носителей информации; разрушение информации под воздействием физических факторов (аварии, неправильная эксплуатация КС и т.п.).	Маскировка несанкционированных запросов под запросы ОС; обход программ разграничения доступа; чтение конфиденциальных данных из источников информации; подключение к каналам связи с целью получения информации («подслушивание» и/или «ретрансляция»); анализ трафика; использование терминалов и ЭВМ других операторов; намеренный вызов случайных факторов.	Включение в программы РПС, выполняющих функции нарушения целостности и конфиденциальности информации и ПО; ввод новых программ, выполняющих функции нарушения безопасности ПО; незаконное применение ключей разграничения доступа; обход программ разграничения доступа; вывод из строя подсистемы регистрации и учета; уничтожение ключей шифрования и паролей; подключение к каналам связи с целью модификации, уничтожения, задержки и переупорядочивания данных; несанкционированное копирование, распространение и использование программных средств КС; намеренный вызов случайных факторов.
Косвенные	нарушение пропускного режима и режима секретности; естественные потенциальные поля; помехи и т.п.	перехват ЭМИ от технических средств; хищение производственных отходов (распечаток, отработанных дискет и т.п.); использование визуального канала, подслушивающих устройств и т.п.; дистанционное фотографирование и т.п.	помехи; отключение электропитания; намеренный вызов случайных факторов.

Угрозы, носящие злоумышленный характер вызваны, как правило, преднамеренным желанием субъекта осуществить несанкционированные изменения с целью нарушения корректного выполнения преобразований, достоверности и целостности данных, которое проявляется в искажениях их содержания или структуры, а также с целью нарушения функционирования технических средств в процессе реализации функциональных преобразований и изменения конструктивных особенностей вычислительных систем и систем телекоммуникаций.

На основании анализа уязвимых мест и после составления полного перечня угроз для данного конкретного объекта информационной защиты, например, в виде указанной таблицы, необходимо осуществить переход к неформализованному или формализованному описанию модели угроз эксплуатационной безопасности ПО КС. Такая модель, в свою очередь, должна соотноситься (или даже являться составной частью) обобщенной модели обеспечения безопасности информации и ПО объекта защиты.

К неформализованному описанию модели угроз приходится прибегать в том случае, когда структура, состав и функциональная наполненность компьютерных системы управления носят многоуровневый, сложный, распределенный характер, а действия потенциального нарушителя информационных и функциональных ресурсов трудно поддаются формализации. Пример описания модели угроз при исследовании попыток несанкционированных действий в отношении защищаемой КС приведен на рис.1.3.

После окончательного синтеза модели угроз разрабатываются практические рекомендации и методики по ее использованию для конкретного объекта информационной защиты, а также механизмы оценки адекватности модели и реальной информационной ситуации и оценки эффективности ее применения при эксплуатации КС.

Таким образом, разработка моделей угроз безопасности программного обеспечения КС, являясь одним из важных этапов комплексного решения проблемы обеспечения безопасности информационных технологий, на этапе создания КС отличается от разработки таких моделей для этапа их эксплуатации.

Принципиальное различие подходов к синтезу моделей угроз технологической и эксплуатационной безопасности ПО заключается в различных мотивах поведения потенциального нарушителя информационных ресурсов, принципах, методах и средствах воздействия на ПО на различных этапах его жизненного цикла.

Рис. 1.3. Неформализованное описание модели угроз безопасности ПО на этапе исследований попыток несанкционированных действий в отношении информационных ресурсов КС

1.5.4. Модель разрушающего программного средства

Обобщенное описание и типизация РПС

Обобщенное описание и типизация РПС будет делаться в соответствии с работой [ПАС].

Разрушающим программным средством, в данном случае, будем считать некоторую программу, которая способна выполнить любое непустое подмножество перечисленных функций:

- сокрытия признаков своего присутствия в программной среде КС;
- реализации самодублирования, ассоциирования себя с другими программами и/или переноса своих фрагментов в иные (не занимаемые изначально указанной программой) области оперативной или внешней памяти;
- разрушения (искажения произвольным образом) кода программ в оперативной памяти КС;
- перемещения (сохранения) фрагментов информации из оперативной памяти в некоторые области оперативной или внешней памяти прямого доступа;
- искажения произвольным образом, блокировки и/или подмены выводимых во внешнюю память или в канал связи массивов информации, образовавшихся в результате работы прикладных программ или уже находящихся во внешней памяти, либо изменения их параметров.

Можно выделить следующие основные типы РПС.

РПС, отключающие защитные функции системы. Во многих случаях РПС, внедренные в защищенную систему, могут модифицировать машинный код или конфигурационные данные системы, тем самым полностью или частично отключая ее защитные функции. В защищенной системе создается «черный ход», позволяющий злоумышленнику работать в системе, обходя ее защитные функции.

Отключение программной закладкой защитных функций системы чаще всего используется для снятия защиты от несанкционированного копирования.

Перехватчики паролей. Перехватчики паролей перехватывают имена и пароли, вводимые пользователями защищенной системы в процессе идентификации и аутентификации. В простейшем случае перехваченные имена и пароли сохраняются в текстовом файле, более сложные

программные закладки пересылают эту информацию по сети на компьютер злоумышленника.

Существуют три основные архитектуры построения перехватчиков паролей.

1. Перехватчики паролей *первого рода* действуют по следующему сценарию. Злоумышленник запускает программу, содержащую программную закладку - перехватчик паролей. Она имитирует приглашение пользователю для входа в систему, и ждет ввода. Когда пользователь вводит имя и пароль, закладка сохраняет их в доступном злоумышленнику месте, после чего завершает работу и осуществляет выход из системы пользователя-злоумышленника. По окончании работы закладки на экране появляется настоящее приглашение для входа пользователя в систему. Пользователь, ставший жертвой закладки, видит, что он не вошел в систему и что ему снова предлагается ввести имя и пароль. Пользователь предполагает, что при вводе пароля произошла ошибка, и вводит имя и пароль повторно. После этого пользователь входит в систему, и дальнейшая его работа протекает нормально. Некоторые закладки, функционирующие по данной схеме, перед завершением работы выдают на экран правдоподобное сообщение об ошибке, например: «Пароль введен неправильно. Попробуйте еще раз».

2. Перехватчики паролей *второго рода* перехватывают все данные, вводимые пользователем с клавиатуры. Простейшие программные закладки данного типа просто сбрасывают все эти данные на жесткий диск компьютера или в любое другое место, доступное злоумышленнику. Более совершенные закладки анализируют перехваченные данные и отсеивают информацию, заведомо не имеющую отношения к паролям. Эти закладки представляют собой резидентные программы, перехватывающие одно или несколько прерываний, используемых при работе с клавиатурой. Информация о нажатой клавише и введенном символе, возвращаемая этими прерываниями, используется закладками для своих целей.

3. К перехватчикам паролей *третьего рода* относятся программные закладки, полностью или частично подменяющие собой подсистему аутентификации защищенной системы. Поскольку задача создания такой программной закладки гораздо сложнее, чем задача создания перехватчика паролей первого или второго рода, этот класс программных закладок появился совсем недавно и будем считать возможность злонамеренно

воздействовать на подсистемы идентификации и аутентификации пользователей при их входе в систему пока гипотетической.

Программные закладки, превышающие полномочия пользователя. Эти закладки применяются для преодоления тех систем защиты, в которых реализовано разграничение доступа пользователей к объектам системы (в основном эти закладки применяются против операционных систем). Закладки данного типа позволяют злоумышленнику осуществлять доступ к тем объектам, доступ к которым должен быть ему запрещен согласно текущей стратегии защиты. В большинстве систем, поддерживающих разграничение доступа, существуют администраторы безопасности, которые могут осуществлять доступ ко всем или почти всем объектам системы. Если программная закладка наделяет пользователя-злоумышленника полномочиями администратора и злоумышленник имеет практически неограниченный доступ к ресурсам системы, средства и методы, используемые такими закладками для превышения полномочий пользователя, в значительной мере определяются архитектурой системы. Чаще всего закладки данного класса используют ошибки в программном обеспечении системы.

Логические бомбы. Это программные закладки оказывают, как правило, разрушающие воздействия на атакованную систему и обычно нацелены на полное выведение системы из строя. В отличие от вирусов логические бомбы не размножаются или размножаются ограниченно. Логические бомбы всегда предназначены для атаки на конкретную компьютерную систему. После того как разрушающее воздействие завершено, то, как правило, логическая бомба уничтожается.

Иногда выделяют особый класс логических бомб - *временные бомбы,* для которых условием срабатывания является достижение определенного момента времени.

Характерным свойством логических бомб является то, что реализуемые ими негативные воздействия на атакованную систему носят исключительно разрушающий характер. Логические бомбы, как правило, не используются для организации НСД к ресурсам системы, их единственной задачей является полное или частичное разрушение системы.

Мониторы. Это программные закладки, перехватывающие те или иные потоки данных, протекающие в атакованной системе. В частности, к мониторам относятся перехватчики паролей второго рода.

Целевое назначение мониторов может быть самым разным:

- полностью или частично сохранять перехваченную информацию в доступном злоумышленнику месте;
- искажать потоки данных;
- помещать в потоки данных навязанную информацию;
- полностью или частично блокировать потоки данных;
- использовать мониторинг потоков данных для сбора информации об атакованной системе.

Мониторы позволяют перехватывать самые различные информационные потоки атакуемой системы. Наиболее часто перехватываются следующие потоки:

- потоки данных, связанные с чтение, записью и другими операциями над файлами;
- сетевой трафик;
- потоки данных, связанные с удалением информации с дисков или из оперативной памяти компьютера (так называемая «сборка мусора»).

Сборщики информации об атакуемой среде. Программные закладки этого класса предназначены для пассивного наблюдения за программной средой, в которую внедрена закладка. Основная цель применения подобных закладок заключается в первичном сборе информации об атакуемой системе. В дальнейшем эта информация используется при организации таких систем, возможно, с применением программных закладок других классов.

Кроме того, РПС можно классифицировать по методу и месту их внедрения и применения, т.е. по «способу доставки» в компьютерную систему (ниже приводится пример персонального компьютера) [ПАС]:

- РПС, ассоциированные с программно-аппаратной средой компьютерной системы (основная BIOS или расширенные BIOS);
- РПС, ассоциированные с программами первичной загрузки (находящиеся в Master Boot Record или BOOT-секторах активных разделов), - загрузочные РПС;
- РПС, ассоциированные с загрузкой драйверов DOS, драйверов внешних устройств других ОС, командного интерпретатора, сетевых драйверов, т.е. с загрузкой операционной среды;
- РПС, ассоциированные с системным программным обеспечением общего назначения (встроенные в клавиатурные и экранные

драйверы, программы тестирования компьютеров, утилиты и оболочки интерфейса с пользователем и т.п.);

- исполняемые модули, содержащие только код РПС (как правило, внедряемые в файлы пакетной обработки типа .BAT);
- модули-имитаторы, совпадающие по внешнему виду с некоторыми программами, требующими ввода конфиденциальной информации - наиболее характерны для Unix-систем;
- РПС, маскируемые под программные средства оптимизационного назначения (архиваторы, ускорители обмена с диском и т.д.);
- РПС, маскируемые в программные средства игрового и развлекательного назначения (как правило, используются для первичного внедрения закладок).

Для того чтобы РПС смогли выполнять какие-либо действия по отношению к прикладной программе или данным, они должны получить управление, т.е. процессор должен начать выполнять инструкции (команды), относящиеся к коду РПС. Это возможно только при одновременном выполнении двух условий:

1). РПС должно находиться в оперативной памяти до начала работы РПС, которое является целью воздействия закладки, а следовательно, РПС должно быть загружено раньше или одновременно с этой программой;

2). РПС должно активизироваться по некоторому общему как для закладки, так и для программы событию, т.е. при выполнении ряда условий в программно-аппаратной среде управление должно быть передано программе-закладке.

Обычно выполнение указанных условий достигается путем анализа и обработки закладкой общих относительно РПС и прикладной программы воздействий (как правило, прерываний) либо событий (в зависимости от типа и архитектуры операционной среды). Причем прерывания должны сопровождать работу прикладной программы или работу всего компьютера. Данные условия являются необходимыми (но недостаточными), т.е. если они не выполнены, то активизация кода закладки не произойдет и код не сможет оказать какого-либо воздействия на работу прикладной программы.

Кроме того, возможен случай [ПАС], когда при запуске программы (активизирующим событием является запуск программы) закладка разрушает некоторую часть кода программы, уже загруженной в оперативную память, и, возможно, систему контроля целостности кода или

контроля иных событий и на этом заканчивает свою работу. Данный случай не противоречит необходимым условиям.

С учетом замечания о том, что РПС должна быть загружена в ОП раньше, чем цель его воздействий, можно выделить РПС двух типов.

1). РПС резидентного типа - находятся в памяти постоянно с некоторого момента времени до окончания сеанса работы компьютера (выключения питания или перезагрузки). РПС может быть загружено в память при начальной загрузке компьютера, загрузке операционной среды или запуске некоторой программы (которая по традиции называется вирусоносителем или просто носителем), а также запущена отдельно.

2). РПС нерезидентного типа - начинают работу, как и РПС резидентного типа, но заканчивают ее самостоятельно через некоторый промежуток времени или по некоторому событию, при этом выгружая себя из памяти целиком.

Модели взаимодействия прикладной программы и программной закладки

Общая модель РПС можно представить в виде совокупности моделей, каждая из которых соответствует описанным выше типам РПС и характеризует действия злоумышленника, исходя из его образа мыслей и возможностей [ПАС].

1. *Модель «перехват».* Программная закладка встраивается (внедряется) в ПЗУ, ОС или прикладное программное обеспечение и сохраняет все или избранные фрагменты вводимой или выводимой информации в скрытой области локальной или удаленной внешней памяти прямого доступа. Объектом сохранения может быть клавиатурный ввод, документы, выводимые на принтер, или уничтожаемые файлы-документы. Для данной модели существенно наличие во внешней памяти места хранения информации, которое должно быть организовано таким образом, чтобы обеспечить ее сохранность на протяжении заданного промежутка времени и возможность последующего съема. Важно также, чтобы сохраняемая информация была каким-либо образом замаскирована от просмотра легальными пользователями.

2. *Модель «троянский конь».* Закладка встраивается в постоянно используемое программное обеспечение и по некоторому активизирующему событию моделирует сбойную ситуацию на средствах хранения информации или в оборудовании компьютера (сети). Тем самым могут быть достигнуты две различные цели: во-первых, парализована нормальная работа компьютерной системы и, во-вторых, злоумышленник

(например, под видом обслуживания или ремонта) может ознакомиться с имеющейся в системе или накопленной посредством использования модели «перехват» информацией. Событием, активизирующим закладку, может быть некоторый момент времени, либо сигнал из канала модемной связи (явный или замаскированный), либо состояние некоторых счетчиков (например, число запусков программ).

3. *Модель «наблюдатель».* Закладка встраивается в сетевое или телекоммуникационное программное обеспечение. Пользуясь тем, что данное ПО, как правило, всегда активно, программная закладка осуществляет контроль за процессами обработки информации на данном компьютере, установку и удаление закладок, а также съем накопленной информации. Закладка может инициировать события для ранее внедренных закладок, действующих по модели «троянский конь».

4. *Модель «компрометация».* Закладка либо передает заданную злоумышленником информацию (например, клавиатурный ввод) в канал связи, либо сохраняет ее, не полагаясь на гарантированную возможность последующего приема или снятия. Более экзотический случай – закладка инициирует постоянное обращение к информации, приводящее к росту отношения сигнал/шум при перехвате побочных излучений.

5. *Модель «искажение или инициатор ошибок».* Программная закладка искажает потоки данных, возникающие при работе прикладных программ (выходные потоки), либо искажает входные потоки информации, либо инициирует (или подавляет) возникающие при работе прикладных программ ошибки.

6. *Модель «сборка мусора».* В данном случае прямого воздействия РПС может и не быть; изучаются «остатки» информации. В случае применения программной закладки навязывается такой порядок работы, чтобы максимизировать количество остающихся фрагментов ценной информации. Злоумышленник получает либо данные фрагменты, используя закладки моделей 2 и 3, либо непосредственный доступ к компьютеру под видом ремонта или профилактики.

У рассмотренных различных по целям воздействия моделей закладок имеется важная общая черта - наличие операции записи, производимой закладкой (в оперативную или внешнюю память). При отсутствии данной операции никакое негативное влияние невозможно. Вполне понятно, что для направленного воздействия закладка должна также выполнять и операции чтения. Так, например, можно реализовать функцию

целенаправленной модификации данных в каком-либо секторе жесткого диска, которая возможна только после их прочтения.

Таким образом, исполнение кода закладки может быть сопровождено операциями несанкционированной записи (НСЗ), например, для сохранения некоторых фрагментов информации, и несанкционированного считывания (НСЧ), которое может происходить отдельно от операций чтения прикладной программы или совместно с ними. При этом операции считывания и записи могут быть не связаны с получением информации, например считывание параметров устройства или его инициализация - закладка может использовать для своей работы и такие операции, в частности, для инициирования сбойных ситуаций или переназначения ввода вывода.

Возможные комбинации несанкционированных действий (НСЧ и НСЗ), а также санкционированных действий прикладной программы или операционной среды по записи (СЗ) или считыванию (СЧ) приведены в табл. 1.2 [ПАС].

Ситуации 1 - 4 соответствуют нормальной работе прикладной программы, когда закладка не оказывает на нее никакого воздействия.

Ситуация 5 может быть связана либо с разрушением кода прикладной программы в оперативной памяти ЭВМ, поскольку прикладная программа не выполняет санкционированные действия по записи и считыванию, либо с сохранением уже накопленной в ОП информации.

Ситуация 6 связана с разрушением или с сохранением информации (искажение или сохранение выходного потока), записываемой прикладной программой.

Ситуация 7 связана с сохранением информации (сохранение входного потока) считываемой прикладной программой.

Ситуация 8 связана с сохранением информации закладкой при считывании или записи информации прикладной программой.

Ситуация 9 не связана с прямым негативным воздействием, поскольку прикладная программа не активна, а закладка производит только НСЧ (процесс «настройки»).

Ситуация 10 может быть связана с сохранением выводимой информации в оперативную память.

Ситуация 11 может быть связана с сохранением вводимой информации в оперативную память либо с изменением параметров процесса санкционированного чтения закладкой.

Ситуация 12 может быть связана с сохранением как вводимой, так и выводимой прикладной программой информации в оперативную память.

Ситуация 13 может быть связана с размножением закладки, сохранением накопленной в буферах ОП информации или с разрушением кода и данных в файлах, поскольку прикладная программа не активна.

Таблица 1.2

Номер ситуации	***НСЧ***	***НСЗ***	***Действия РПС***	***СЧ***	***СЗ***
1	0	0	Нет	0	0
2	0	0	Нет	0	1
3	0	0	Нет	1	0
4	0	0	Нет	1	1
5	0	1	Изменение (разрушение) кода прикладной программы в ОП	0	0
6	0	1	Разрушение или сохранение выводимых прикладной программой данных	0	1
7	0	1	Разрушение или сохранение вводимых прикладной программой данных	1	0
8	0	1	Разрушение или сохранение вводимых и выводимых данных	1	1
9	1	0	Нет	0	0
10	1	0	Перенос выводимых прикладной программой данных в ОП	0	1
11	1	0	Перенос вводимых прикладной программой данных в ОП	1	0
12	1	0	Перенос вводимых и выводимых данных в ОП	1	1
13	1	1	Процедуры типа «размножения вируса» (действия закладки независимо от операций прикладной программы)	0	0
14	1	1	Те же действия, что и в процедурах 6 – 8	0	1
15	1	1		1	0
16	1	1		1	1

Ситуации 14 - 16 могут быть связаны как с сохранением, так и с разрушением данных или кода и аналогичны ситуациям 6 - 8.

Несанкционированная запись закладкой может происходить:

- в массив данных, не совпадающий с пользовательской информацией, - сохранение информации;

- в массив данных, совпадающий с пользовательской информацией или ее подмножеством, - искажение, уничтожение или навязывание информации закладкой.

Следовательно, можно рассматривать три основные группы деструктивных функций, которые могут выполняться РПС [ПАС]:

- сохранение фрагментов информации, возникающей при работе пользователя, прикладных программ, вводе/выводе данных, во внешней памяти (локальной или удаленной) в сети или выделенном компьютере, в том числе сохранение различных паролей, ключей и кодов доступа, собственно конфиденциальных документов в электронном виде, либо безадресная компрометация фрагментов ценной информации (модели «перехват», «компрометация»);
- изменение алгоритмов функционирования прикладных программ (т.е. целенаправленное воздействие во внешней или оперативной памяти) - происходит изменение собственно исходных алгоритмов работы программ, например, программа разграничения доступа станет пропускать пользователей по любому паролю (модели «искажение», «троянский конь»);
- навязывание некоторого режима работы (например, при уничтожении информации - блокирование записи на диск, при этом информация, естественно, не уничтожается) либо замена записываемой информации данными, навязанными закладкой.

Методы внедрения РПС

Можно выделить следующие основные методы внедрения РПС [ПАС].

Маскировка закладки под «безобидное» программное обеспечение. Данный метод заключается в том, что программная закладка внедряется в систему под видом новой программы, на первый взгляд абсолютно безобидной. Программная закладка может быть внедрена в текстовый или графический редакторы, системную утилиту, компьютерную игру, хранитель экрана и т.д. После внедрения закладки ее присутствие в системе не нужно маскировать - даже если администратор заметит факт появления в системе новой программы, он не придаст этому значения, поскольку эта программа внешне совершенно безобидна.

Маскировка закладки под «безобидный» модуль расширения программной среды. Многие программные среды допускают свое расширение дополнительными программными модулями. Например, для

операционных систем семейства Microsoft Windows модулями расширения могут выступать динамически подгружаемые библиотеки (DLL) и драйверы устройств. В таких модулях расширения может содержаться РПС, которое может потенциально внедрено в систему. Данный метод фактически является частным случаем предыдущего метода и отличается от него только тем, что закладка представляет собой не прикладную программу, а модуль расширения программной среды.

Подмена закладкой одного или нескольких программных модулей атакуемой среды. Данный метод внедрения в систему программной закладки заключается в том, что в атакуемой программной среде выбирается один или несколько программных модулей, подмена которых фрагментами программной закладки позволяет оказывать на среду требуемые негативные воздействия. Программная закладка должна полностью реализовывать все функции подменяемых программных модулей.

Основная проблема, возникающая при практической реализации данного метода, заключается в том, что программист, разрабатывающий программную закладку, никогда не может быть уверен, что созданная им закладка точно реализует все функции подменяемого программного модуля. Если подменяемый модуль достаточно велик по объему или недостаточно подробно документирован, точно запрограммировать все его функции практически невозможно. Поэтому описываемый метод целесообразно применять только для тех программных модулей атакуемой среды, для которых доступна полная или почти полная документация. Оптимальной является ситуация, когда доступен исходный текст подменяемого модуля.

Прямое ассоциирование. Данный метод внедрения в систему программной закладки заключается в ассоциировании закладки с исполняемыми файлами одной или нескольких легальных программ системы. Сложность задачи прямого ассоциирования программной закладки с программой атакуемой среды существенно зависит от того, является атакуемая среда однозадачной или многозадачной, однопользовательской или многопользовательской. Для однозадачных однопользовательских систем эта задача решается достаточно просто. В то же время при внедрении закладки в многозадачную или многопользовательскую программную среду прямое ассоциирование

закладки с легальным программным обеспечением является весьма нетривиальной задачей.

Косвенное ассоциирование. Косвенное ассоциирование закладки с программным модулем атакуемой среды заключается в ассоциировании закладки с кодом программного модуля, загруженным в оперативную память. При косвенном ассоциировании исполняемый файл программного модуля остается неизменным, что затрудняет выявление программной закладки.

Для того чтобы косвенное ассоциирование стало возможным, необходимо, чтобы инсталлирующая часть закладки уже присутствовала в системе. Другими словами, программная закладка, внедряемая в систему с помощью косвенного ассоциирования, должна быть составной.

1.6. Основные принципы обеспечения безопасности ПО

В качестве объекта обеспечения технологической и эксплуатационной безопасности ПО рассматривается вся совокупность его компонентов в рамках конкретной КС. В качестве доминирующей должна использоваться стратегия сквозного тотального контроля технологического и эксплуатационного этапов жизненного цикла компонентов ПО. Совокупность мероприятий по обеспечению технологической и эксплуатационной безопасности компонентов ПО должна носить, по возможности, конфиденциальный характер. Необходимо обеспечить постоянный, комплексный и действенный контроль за деятельностью разработчиков и пользователей компонентов ПО. Кроме общих принципов, обычно необходимо конкретизировать принципы обеспечения безопасности ПО на каждом этапе его жизненного цикла. Далее приводятся один из вариантов разработки таких принципов.

Принципы обеспечения технологической безопасности при обосновании, планировании работ и проектном анализе ПО

Принципы обеспечения безопасности ПО на данном этапе включают принципы:

Комплексности обеспечения безопасности ПО, предполагающей рассмотрение проблемы безопасности информационно - вычислительных процессов с учетом всех структур КС, возможных каналов утечки информации и несанкционированного доступа к ней, времени и условий их возникновения, комплексного применения организационных и технических мероприятий.

Планируемости применения средств безопасности программ, предполагающей перенос акцента на совместное системное проектирование ПО и средств его безопасности, планирование их использования в предполагаемых условиях эксплуатации.

Обоснованности средств обеспечения безопасности ПО, заключающейся в глубоком научно-обоснованном подходе к принятию проектных решений по оценке степени безопасности, прогнозированию угроз безопасности, всесторонней априорной оценке показателей средств защиты.

Достаточности защищенности программ, отражающей необходимость поиска наиболее эффективных и надежных мер безопасности при одновременной минимизации их стоимости.

Гибкости управления защитой программ, требующей от системы контроля и управления обеспечением безопасности ПО способности к диагностированию, опережающей нейтрализации, оперативному и эффективному устранению возникающих угроз,

Заблаговременности разработки средств обеспечения безопасности и контроля производства ПО, заключающейся в предупредительном характере мер обеспечения технологической безопасности работ в интересах недопущения снижения эффективности системы безопасности процесса создания ПО.

Документируемости технологии создания программ, подразумевающей разработку пакета нормативно-технических документов по контролю программных средств на наличие преднамеренных дефектов.

Принципы достижения технологической безопасности ПО в процессе его разработки

Принципы обеспечения безопасности ПО на данном этапе включают принципы:

Регламентации технологических этапов разработки ПО, включающей упорядоченные фазы промежуточного контроля, спецификацию программных модулей и стандартизацию функций и формата представления данных.

Автоматизации средств контроля управляющих и вычислительных программ на наличие преднамеренных дефектов.

Создания типовой общей информационной базы алгоритмов, исходных текстов и программных средств, позволяющих выявлять преднамеренные программные дефекты.

Последовательной многоуровневой фильтрации программных модулей в процессе их создания с применением функционального дублирования разработок и поэтапного контроля.

Типизации алгоритмов, программ и средств информационной безопасности, обеспечивающей информационную, технологическую и программную совместимость, на основе максимальной их унификации по всем компонентам и интерфейсам.

Централизованного управления базами данных проектов ПО и администрирование технологии их разработки с жестким разграничением функций, ограничением доступа в соответствии со средствами диагностики, контроля и защиты.

Блокирования несанкционированного доступа соисполнителей и абонентов государственных и негосударственных сетей связи, подключенных к стендам для разработки программ.

Статистического учета и ведения системных журналов о всех процессах разработки ПО с целью контроля технологической безопасности.

Использования только сертифицированных и выбранных в качестве единых инструментальных средств разработки программ для новых технологий обработки информации и перспективных архитектур вычислительных систем.

Принципы обеспечения технологической безопасности на этапах стендовых и приемо-сдаточных испытаний

Принципы обеспечения безопасности ПО на данном этапе включают принципы:

Тестирования ПО на основе разработки комплексов тестов, параметризуемых на конкретные классы программ с возможностью функционального и статистического контроля в широком диапазоне изменения входных и выходных данных.

Проведения натурных испытаний программ при экстремальных нагрузках с имитацией воздействия активных дефектов.

Осуществления «фильтрации» программных комплексов с целью выявления возможных преднамеренных дефектов определенного назначения на базе создания моделей угроз и соответствующих сканирующих программных средств.

Разработки и экспериментальной отработки средств верификации программных изделий.

Проведения стендовых испытаний ПО для определения непреднамеренных программных ошибок проектирования и ошибок разработчика, приводящих к невыполнению целевых функций программ, а также выявление потенциально «узких» мест в программных средствах для разрушительного воздействия.

Отработки средств защиты от несанкционированного воздействия нарушителей на ПО.

Сертификации программных изделий КС по требованиям безопасности с выпуском сертификата соответствия этого изделия требованиям технического задания.

Принципы обеспечения безопасности при эксплуатации программного обеспечения

Принципы обеспечения безопасности ПО на данном этапе включают принципы:

Сохранения эталонов и ограничения доступа к ним программных средств, недопущение внесения изменений в эталоны.

Профилактического выборочного тестирования и полного сканирования программных средств на наличие преднамеренных дефектов.

Идентификации ПО на момент ввода его в эксплуатацию в соответствии с предполагаемыми угрозами безопасности ПО и его контроль.

Обеспечения модификации программных изделий во время их эксплуатации путем замены отдельных модулей без изменения общей структуры и связей с другими модулями.

Строгого учета и каталогизации всех сопровождаемых программных средств, а также собираемой, обрабатываемой и хранимой информации.

Статистического анализа информации обо всех процессах, рабочих операциях, отступлениях от режимов штатного функционирования ПО.

Гибкого применения дополнительных средств защиты ПО в случае выявления новых, непрогнозируемых угроз информационной безопасности.

http://www.natahaus.ru/

ГЛАВА 2. ФОРМАЛЬНЫЕ МЕТОДЫ ДОКАЗАТЕЛЬСТВА ПРАВИЛЬНОСТИ ПРОГРАММ И ИХ СПЕЦИФИКАИЦЙ

2.1. Общие положения

Традиционные методы анализа ПО включают, в том числе, методы доказательства правильности программ. Начало этому направлению было положено работами П. Наура и Р. Флойда, в которых сформулирована идея приписывания точке программы так называемого индуктивного, или промежуточного утверждения и указана возможность доказательства частичной правильности программы (то есть соответствия друг другу ее предусловия и постусловия), построенного на установлении согласованности индуктивных утверждений.

Фундаментальный вклад в теорию верификации внес Ч. Хоор, высказавший идеи проведения доказательства частичной правильности программы в виде вывода в некоторой логической системе, а Э. Дейкстра ввел понятие слабейшего предусловия, позволяющее одновременно как соответствие друг другу предусловия и постусловия, так и завершимость. Методы доказательства правильности программ принесли определенную пользу программированию. Было отмечено, что эти методы указывают способ рассуждения о ходе выполнения программ, дают удобную систему комментирования программ и устанавливают взаимосвязи между конструкциями языков программирования и их семантикой. Если принять более широкое толкование термина «анализ программ», подразумевая доказательство разнообразных свойств программ или доказательство теорем о программах, то ценность методов анализа станет более ясной. В частности можно исследовать характер изменения выходных значений программы, количество операций при выполнении программы, наличие зацикливаний, незадействованных участков программы. Таким образом, в некоторых частных случаях методы верификации могут применяться и для доказуемого обнаружения программных дефектов.

Формальное доказательство в виде вывода в некоторой логической системе вполне надежно, но сами доказательства оказываются очень длинными и часто необозримыми. Рассмотрим следующий фрагмент программы [ДДХ].

```
integer r, dd;
  r:=a; dd:=d;
  while dd≤r do dd:=2*dd;
  while dd≠d do
       begin dd:=dd/2;
           if dd≤r do r:=r-dd;
       end.
```

Должно соблюдаться условие, что целые константы a и d удовлетворяют отношениям $a \geq d$ и $d>0$.

Рассмотрим последовательность значений, заданную выражениями для:

$i=0 \quad dd_i=d$

$i>0 \quad dd_i=2*dd_i-1.$

Далее с помощью обычных математических приемов можно вывести, что:

$$dd_n=d*2^n \qquad (2.1)$$

Кроме того, поскольку $d>0$, можно сделать вывод, что для любого конечного значения r отношение $dd_k>r$ будет выполняться при некотором конечном значении k; первый цикл завершается при $dd=d*2^k$. Решая уравнение $d_i=2*d_{i-1}$ относительно d_{i-1}, получаем $d_{i-1}=d_i/2$, что позволяет утверждать, что второй цикл тоже завершится. По окончании первого цикла $dd=dd_k$ и поэтому выполняется соотношение

$$0 \leq r < dd \qquad (2.2)$$

Это соотношение сохраняется при выполнении повторяемого оператора второго заголовка. После завершения повторений (в соответствии с заголовком while $dd \neq d$ do) мы получаем $dd=d$. Отсюда и из соотношения 2.2 следует, что

$$0 \leq r < d \qquad (2.3)$$

Далее мы доказываем, что после начала работы программы всегда выполняется отношение:

$$dd \equiv 0 (\mathrm{mod}\ d) \qquad (2.4)$$

Это следует, например, из того, что возможные значения dd имеют вид (см. 2.1) $d*2^i$ при $0 \leq i \leq k$.

Следующая задача состоит в том, чтобы показать, что после присваивания r начального значения всегда выполняется отношение

$$a \equiv r (\mathrm{mod}\ d) \qquad (2.5)$$

Оно выполняется после начальных присваиваний.

Повторяемый оператор первого заголовка (dd:=2*dd) сохраняет отношение 2.5, и поэтому весь первый цикл сохраняет отношение 2.5.

Повторяемый оператор из второго цикла состоит из двух операторов. Первый dd:=2/dd сохраняет инвариантность 2.5; второй тоже сохраняет отношение 2.5, так как он либо не изменяет значение r, либо уменьшает r на текущее dd, а эта операция в силу 2.4 также сохраняет справедливость отношения 2.5. Таким образом, весь повторяемый оператор второго цикла сохраняет отношение 2.5.

Объединяя отношения 2.3 и 2.5, получаем, что окончательное значение r удовлетворяет условиям $0 \le r < d$ и $a \equiv r (\mathrm{mod}\, d)$, то есть r - наименьший неотрицательный остаток от деления a на d.

Как мы можем видеть, что программа, состоящая всего из нескольких строчек кода, может требовать формального доказательства ее правильности (т.е. доказательства соответствия предусловия и постусловия, а также доказательства завершимости), состоящего из нескольких страниц текста.

Таким образом, чтобы доказать правильность программы, сегмента программы или оператора, формулируется математическая теорема о результатах выполнения рассматриваемого сегмента программы. Затем эта теорема доказывается.

Почти всегда подобные теоремы формулируются следующим образом [Бей]. *«Если конкретное условие истинно непосредственно перед выполнением сегмента программы, тогда после выполнения этого сегмента тоже истинным будет некоторое условие (в общем случае другое)»*.

Как было сказано выше, такие условия принято называть *предусловиями* и *постусловиями*. Предусловия и постусловия основаны на переменных из программы. На самом деле в большинстве условий используются лишь *значения* переменных в программе. В данном случае мы будем рассматривать лишь предусловия и постусловия именно такого вида.

Для упрощения доказательства теорема о корректности в приведенной выше форме разбивается на две части. Одна часть служит для доказательства того, что рассматриваемый сегмент программы вообще работает, то есть его выполнение заканчивается получением некоторого определенного результата (без ошибок на фазе компиляции или

выполнения). В другой части доказывается корректность упомянутой теоремы в предположении, что программа завершает свою работу.

2.2. Предусловия и постусловия в доказательствах правильности

Условие — это алгебраическое выражение, значение которого либо «ложно», либо «истинно». Обычно в выражениях присутствуют имена переменных программы; каждому имени соответствует значение соответствующей переменной. Условия также называют *логическими* выражениями, *булевыми* выражениями или *суждениями.*

Если из истинности условия V непосредственно перед выполнением оператора S следует истинность условия P после выполнения этого оператора, то говорят, что V есть *предусловие для постусловия* P по отношению к оператору S. Подобная взаимозависимость V, P и S обычно записывается как $\{V\}\ S\ \{P\}$. Оператор S может быть простым или составным, содержащим любое число отдельных операторов, то есть может быть сегментом программы или же целой программой.

Результат выполнения оператора, сегмента программы или программы будет *корректным,* если он удовлетворяется заданному постусловию. Оператор программы (либо сегмент программы, либо программа) *завершается,* если его выполнение доходит до конца за конечное время и при этом отсутствуют ошибки фазы выполнения (и ошибки фазы компиляции) - то есть его выполнение приводит к получению вполне определенного результата.

Если выполнение оператора, сегмента программы или программы приводит к получению корректного результата, когда бы он ни завершился, то говорят, что оператор (сегмент программы или программа) *частично корректный*. Если, кроме того, доказано, что его выполнение действительно заканчивается, то говорят, что оператор (сегмент программы или программа) *полностью корректный*. За счет выделения таких двух аспектов в корректности упрощаются доказательства.

Предусловия и постусловия представляют ключевые моменты в доказательствах корректности. Они выражают понятие «корректности» конкретной программы или сегмента программы. При записи различным образом, предусловие и постусловие вместе образуют общую спецификацию рассматриваемой программы. Основные части стандартного доказательства правильности предусловия и постусловия и, в частности, их взаимозависимости. Иногда можно алгебраическим путем вывести предусловие для заданного постусловия и заданного оператора

(простого или составного). Такой подход особенно плодотворен в случае операторов присваивания. Часто задача сводится к проверке (доказательству) утверждения о корректности предусловий и постусловий сегмента программы. В таком случае доказываемое утверждение о корректности разбивается на утверждения о корректности частей рассматриваемого сегмента программы, и они доказываются раздельно. Ряд полезных правил, которые будут введены в последующих подразделах, окажут помощь при выполнении таких шагов.

2.3. Правила вывода (доказательства)

2.3.1. Правило усиления предусловия и ослабления постусловия

Предметом каждого правила вывода является взаимосвязь предусловия и постусловия. Начнем с правила вывода, которое в ряде случаев поможет упростить алгебраические преобразования выражений, возникающих в ходе доказательства. Это правило следует из вышеприведенного определения предусловий и постусловий и упрощает восприятие ряда других правил.

*Правило вывода **P1** – «Усиление предусловия и ослабление постусловия»*

If
$V \Rightarrow V1$ and
$\{V1\}\ OP\ \{P1\}$ and
$P1 \Rightarrow P$
then
$\{V\}\ OP\ \{P\}$ ■

Если условие V истинно перед выполнением оператора OP и если из V следует $V1$, то $V1$ истинно перед выполнением оператора OP. Если $V1$ является предусловием для $P1$ по отношению к OP, то $P1$ будет истинным после выполнения оператора OP. Если, наконец, из $P1$ следует P, то и P будет истинным после выполнения оператора OP. Иными словами, из истинности V перед выполнением OP следует истинность P после его выполнения. Следовательно, V является предусловием для P по отношению к OP.

Продвигаясь по программе в *обратном* направлении, можно *усилить* условия. Усиление условие может быть достигнуто путем добавления

произвольного элемента с помощью операции логического умножения and или отбрасыванием элемента, участвующего в операции логического сложения or.

Двигаясь по программе в *прямом* направлении, можно ослабить условия. Условие может быть ослаблено добавлением произвольного элемента с помощью операции логического сложения or или отбрасыванием элемента, участвующего в операции логического умножения and.

Разумное усиление предусловий или ослабление постусловий (что встречается реже) может привести к получению более простых выражений и сокращению (иногда значительному) объема алгебраических преобразований в процессе доказательства. Вместе с тем, здесь следует соблюдать известную осторожность, чтобы не усилить предусловие или не ослабить постусловие столь сильно, что нельзя будет завершить доказательство. В приводимых ниже примерах показывается, каким образом можно легко обойти такую потенциально возможную проблему.

2.3.2. Правило получения предусловия оператора присваивания

Чтобы получить предусловие для заданного постусловия P по отношению к заданному оператору присваивания $x{:=}E$, подставим выражение E вместо переменной с именем x всюду, где это имя встречается в постусловии P. Либо, используя введенные ранее обозначения, запишем то же в следующем виде.

*Правило вывода **A1** – «Получение предусловия оператора присваивания»*

$(P^{x}_{E}\}\ x{:=}E\{P\}$ ■

Значение x после выполнения оператора присваивания $x{:=}E$ будет таким же, как и значение E перед его выполнением. Значение y будет неизменным (на основании допущения, что не должно появляться «побочных эффектов»). Здесь переменная y обозначает все те переменные программы, которые отличны от x. Таким образом, значение $P(x,y)$ после выполнения оператора присваивания будет равно значению $P(E,y)$ до его выполнения. Следовательно, из истинности $P(E,y)$ до выполнения следует истинность $P(x,y)$ после выполнения, то есть $P(E,y)$ является предусловием для $P(x,y)$ по отношению к оператору присваивания $x{:=}E$.

2.3.3. Правило проверки предусловия оператора присваивания

*Правило вывода **A2** – «Проверка предусловия оператора присваивания»*

If
$V \Rightarrow P^x_E$
then
$\{V\}\ x{:=}E\{P\}$ ■

Правило вывода **A2** представляет собой комбинацию правил вывода **A1** и **P1**. Из правила вывода **P1** следует, что V является предусловием для P по отношению к оператору присваивания, если ($V{=}P^x_E$) and $\{P^x_E\}\ x{:=}E\{P\}$.

Из правила вывода **A1** следует, что $\{P^x_E\ \}\ x{:=}E\{P\}$. Поэтому достаточно показать, что $V \Rightarrow P^x_E$, если желательно проверить, что $\{V\}\ x{:=}E\{P\}$.

При применении правила вывода **A2** неявно используются два правила вывода (**A1** и **P1**). И действительно, правило вывода **A1** используется для получения предусловия. Затем проверяется, что из заданного предусловия вытекает полученное предусловие, то есть, что гипотеза правила вывода **P1** удовлетворяется.

2.3.4. Правило проверки предусловия условного оператора if

*Правило вывода **IF1** – «Проверка предусловия условного оператора if»*

If
$\{V$ and $B\}$ $OP1$ $\{P\}$ and
$\{V$ and not $B\}$ $OP2$ $\{P\}$
then
$\{V\}$ if B then $OP1$ else $OP2$ endif $\{P\}$ ■

Если условие V истинно перед выполнением условного оператора if, то и V, и B истинны непосредственно перед выполнением $OP1$ (если этот оператор выполняется). Поскольку (V and B) является предусловием для P по отношению к $OP1$, то P будет истинным после выполнения $OP1$.

Поступая аналогичным образом, получаем, что P будет истинным после выполнения $OP2$. Таким образом, из истинности V перед выполнением условного оператора в обоих случаях следует истинность P после его выполнения. Поэтому V является предусловием для P по отношению к условному оператору целиком.

2.3.5. *Правило получения предусловия условного оператора if*

Правило вывода ***IF2*** *– «Получение предусловия условного оператора if»*

If
{$V1$} $OP1$ {P} and
{$V2$} $OP2$ {P}
then
(($V1$ and B) or ($V2$ and not B)}
if B then $OP1$ else $OP2$ endif {P} ■

В действительности правило вывода **IF2** представляет правило вывода **IF1** с V=[($V1$ and B) or ($V2$ and not B)]. Правило вывода **IF2** следует из правил вывода **IF1** и **P1**. Применяя правило вывода **IF2**, можно получить предусловие заданного постусловия P по отношению к заданному условному оператору. Сначала получаем предусловия для P по отношению к частям then и else (в выражениях $OP1$ и $OP2$), воспользовавшись подходящими правилами вывода для этих операторов. Затем объединяем два предусловия приведенным выше образом.

2.3.6. *Правило условного оператора if №1*

Правило вывода ***IF3*** *– «Условный оператор if»*

If
{$V1$} $OP1$ {P} and
{$V2$} $OP2$ {P}
then
{$V1$ and $V2$} if B then $OP1$ else $OP2$ endif {P} ■

Правило вывода **IF3** представляет собой слабую теорему, которая следует из правил вывода **IF1** и **P1**. Благодаря простой форме образующих

ее выражений, на практике иногда она оказывается удобной для применения.

2.3.7. Правило условного оператора if №2

*Правило вывода **IF4** –«Условный оператор if»*

If
{V1 and B} OP1 {P} and
{V2 and not B} OP2 {P}
then
(V1 and V2} if B then OP1 else OP2 endif {P} ■

Правило вывода **IF4** также следует из правил вывода **IF1** и **P1**. Подобно правилу вывода **IF3** оно обладает определенной практической полезностью вследствие простоты формы предусловия (V1 and V2).

2.3.8. Правило последовательности операторов

*Правило вывода **S1** - «Последовательность операторов»*

If
{V} OP1 {P1} and
{P1} OP2 (P}
then
{V} (OP1;OP2) {P} ■

Правило вывода **S1** очевидным образом обобщается на последовательность операторов произвольной длины. Таким образом, для того чтобы отыскать предусловие для заданного постусловия по отношению к последовательности операторов, сначала отыщем предусловие для P по отношению к последнему оператору последовательности. Затем воспользуемся им в качестве постусловия по отношению к предыдущему, предпоследнему оператору и так далее, то есть будем продвигаться от оператора к оператору по последовательности в обратном направлении. Полученное таким образом предусловие по отношению к первому оператору последовательности является также предусловием для P по отношению ко всей последовательности.

2.3.9. Правило цикла с условием продолжения без инициализации

Правило вывода ***W1*** *- «Цикл с условием продолжения без инициализации»*

If
{*I* and *B*} *OP* {*I*}
then
{*I*} while *B* do *S* endwhile (*I* and not *B*} ■

Если условие *I* истинно непосредственно перед началом выполнения цикла с условием продолжения, то и *I*, и *B* будут истинными до первого выполнения тела цикла *OP*. Поскольку (*I* and *B*) является предусловием для *I* по отношению к OP, то *I* будет истинным после первого выполнения *OP*. Поэтому и *I*, и *B* будут истинными до второго выполнения тела цикла *OP* и так далее. Условие *I* будет истинным после каждого выполнения *OP*. Если когда-то наступит такой момент, что выполнение цикла закончится, то условие *I* будет истинным, а условие *B* ложным. То есть условие (*I* and not *B*) будет истинным при завершении цикла (если завершится его выполнение).

Значение условия *I* истинное, то есть константа, до и после каждого выполнения тела цикла *OP*. Поэтому условие *I* называют *инвариантом цикла.* Инвариант цикла является ключевым понятием в разработке и понимании существа цикла.

При применении правила вывода **W1** требуется, чтобы инвариант цикла *I* был истинным до выполнения цикла. Обычно инвариант *I* изначально истинен тривиальным образом. Другими словами, начальное состояние представляет особый случай инварианта цикла. При завершении выполнения цикла условие (*I* and not *B*) истинно. То есть конечное состояние также представляет особый случай инварианта *I*. Рассматривая инвариант цикла *I* с общих позиций, его можно считать результатом обобщения начального и конечного состояний. Из последнего вытекает следующее правило.

Эмпирическое правило определения инварианта цикла. Следует обобщить (ослабить) начальное и конечное состояния (предусловие и постусловие) для того, чтобы найти подходящий инвариант цикла.

Почти любому циклу предшествует его «инициализация» выполнение сегмента программы, *единственным* назначением которого является установка исходной истинности инварианта цикла.

Весьма часто желательно доказать корректность цикла совместно с его инициализацией. Для этого мы располагаем правилом вывода **W2**.

2.3.10. Правило цикла с условием продолжения с инициализацией

Правило вывода ***W2*** *– «Цикл с условием продолжения с инициализацией*

Пусть задано условие I (инвариант цикла).
If
$\{V\}$ инициализация $\{I\}$ and
$\{I$ and $B\}$ OP $\{I\}$ and
(I and not $B\}$ $\Rightarrow P$
then
$\{V\}$ (инициализация; while B do OP endwhile) $\{P\}$ ■

Правило вывода **W2** объединяет в себе (и следует из) правила вывода **S1**, **P1** и **W1**.

Чтобы доказать частичную корректность цикла с условием продолжения с помощью правила вывода **W2**, необходимо:

1. найти инвариант цикла I (если он не был уже указан разработчиком или программистом);
2. доказать, что $\{V\}$ инициализация $\{I\}$ (то есть I изначально истинно);
3. доказать, что $\{I$ and $B\}$ OP $\{I\}$ (то есть тело цикла обеспечивает сохранение истинности I) и
4. доказать, что (I and not $B\}\Rightarrow$ постусловие P (то есть P истинно при завершении цикла). Для доказательства, что цикл полностью корректный, необходимо дополнительно
5. показать, что цикл завершается, то есть существует верхняя граница числа выполнений оператора OP.

Доказательство завершения обычно заключается в показе того, что (1) значение некоторого выражения увеличивается или уменьшается по крайней мере на фиксированное значение при каждом выполнении OP и что (2) существует соответственно верхняя или нижняя граница значений такого выражения. Часто граница вытекает из условия B цикла while;

иногда она является частью инварианта цикла. Такое выражение называют *вариантом* цикла. Строго говоря, для шага *OP* требуется также показать, что весь цикл вообще выполняется, то есть не может возникнуть «ошибка фазы прогона» (например, переполнение, ссылка на неописанную переменную и т.п.).

2.3.11. Правило «разделяй и властвуй» №1

Правило вывода ***DC1*** *– «Разделяй и властвуй»*

If
{*V*1} *OP* {*P*1} and
{*V*2} *OP* {*P*2}
then
{*V*1 and *V*2} *OP* {*P*1 and *P*2} ■

Правило вывода **DC1** обобщается очевидным образом на произвольное число элементов (*P*1, *P*2, *P*3, *P*4 и т.д.).

Иногда при доказательстве корректности, например, в постусловии сегмента программы, возникает длинное выражение. Применяя правило вывода **DC1**, длинное постусловие, состоящее из двух или нескольких частей, объединенных операциями логического умножения and**,** можно разбить на более короткие части, получить предусловия отдельно для каждой части и затем объединить такие предусловия. При этом конечно же, не уменьшится объем усилий, затрачиваемых на доказательство, но оно обычно становится лучше организованным более ясным и более простым для восприятия. Отдельные шаги алгебраических преобразований часто оказываются более короткими и более простыми. Даже очень длинные и сложные выражения подчиняются стратегии «разделяй и властвуй».

Правило вывода **DC1** относится к элементам, связанным операциями логического умножения and. Аналогичное правило вывода существует и для элементов, связанных операциями логического сложения or.

2.3.12. Правило «разделяй и властвуй» №2-4

Правило вывода ***DC2*** *– «Разделяй и властвуй»*

If
{*V*1} *OP* {*P*1} and
{*V*2} *OP* {*P*2}

then
{$V1$ or $V2$} OP {$P1$ or $P2$} ■

*Правило вывода **DC3** – «Разделяй и властвуй»*

If
{V} OP {$P1$} and
{V} OP {$P2$}
then
{V} OP {$P1$ and $P2$} ■

Правило вывода **DC3** представляет собой правило вывода **DC1** при $V=V1=V2$.

*Правило вывода **DC4** – «Разделяй и властвуй»*

If
{V} OP {$P1$} and
{V} OP {$P2$}
then
{V} OP {$P1$ or $P2$} ■

Правило вывода **DC4** представляет собой правило вывода **DC2** при $V=V1=V2$.

2.3.13. Правило подпрограммы или сегмента программы №1

*Правило вывода **SP1** – «Подпрограмма или сегмент программы»*

Если значения всех переменных, присутствующих в условии B, остаются неизменными при выполнении сегмента программы S (например, подпрограммы, то

{B} S {B} ■

Если в условии B используются лишь такие переменные, значения которых одинаковы до и после выполнения S, очевидно, что значение B до исполнения S равно значению B после исполнения. Следовательно, если B истинно до исполнения, то B будет истинным и после, то есть B является предусловием для самого себя по отношению к S.

2.3.14. Правило подпрограммы или сегмента программы №2

Правило вывода ***SP2*** *– «Подпрограмма или сегмент программы»*

Если значения всех переменных, присутствующих в условии B, остаются неизменными при выполнении подпрограммы или сегмента программы S и

If
$\{V\}\ S\ \{P\}$
then
$\{V$ and $B\}\ S\ (P$ and $B\}$ and
$\{V$ or $B\}\ S\ \{P$ or $B\}$ ■

Правило вывода **SP2** непосредственно следует из правил вывода **SP1**, **DC1** и **DC2**.

Чтобы применить правило вывода **SP2**, необходимо разделить постусловие на две части. В одной части должны быть ссылки лишь на те переменные, чьи значения не изменяются в результате выполнения S. Эта часть постусловия является (согласно правилу вывода **SP1**) его собственным предусловием. Во второй части постусловия содержатся ссылки на все те переменные, значения которых изменяются (или могли бы измениться) в результате выполнения S. Тем самым будет получено предусловие для той части постусловия (применяя, например, соответствующие правила вывода или используя формальную спецификацию S). И, наконец, эти два частных предусловия нужно объединить, образуя требуемое предусловие.

2.3.15. Правило подпрограммы или сегмента программы №3

Правило вывода ***SP3*** *– «Подпрограмма или сегмент программы»*

Если значения всех переменных, присутствующих в условии B, остаются неизменными при выполнении подпрограммы или сегмента программы S и

If
$V \Rightarrow V1$ and

$\{V1\}\ S\ \{P1\}$ and

$P1 \Rightarrow P$

то цикл с условием продолжения инициализации

$\{V \text{ and } B\}\ S\ \{P \text{ and } B\}$

and

цикл с условием продолжения без инициализации

$\{V \text{ or } B\}\ S\ \{P \text{ or } B\}$ ■

Правило вывода **SP3** является комбинацией правил вывода **SP2** и **P1**.

Та часть постусловия, которая зависит от результата выполнения *S* (то есть *P* в вышеприведенном утверждении), может быть более слабой по сравнению с тем постусловием, которое в действительности устанавливается при выполнении *S* (то есть **P1).** Аналогично соответствующая часть предусловия, которая в действительности удовлетворяется до выполнения *S* (то есть *V*), может оказаться сильнее того предусловия, которое требуется для удовлетворительной работы *S* (то есть *V*1).

2.4. Применение правил вывода

Для доказательства правильности программы или сегмента программы, необходимо четко *определить смысл понятия «корректность»* по отношению к рассматриваемой конкретной программе. *Кроме того, необходимо знать, по крайней мере, постусловие. Часто также задается предусловие* и в таком случае необходимо доказать заданное утверждение о правильности, то есть проверить, действительно ли данное предусловие является предусловием для данного постусловия по отношению к данной программе. В остальных случаях необходимо получить предусловие для данного постусловия и данной программы.

Выбор для применения того или иного правила вывода зависит: во-первых, от вида рассматриваемого оператора программы и, во-вторых, от того, задано ли предусловие или же его следует получить.

Правила вывода **DC1 - DC4** можно применять для операторов всех типов и для обеих задач доказательства (проверки и получения предусловия) для разложения длинных условий на малые части. Из правил вывода и их практической применимости при доказательстве правильности

программ вытекает ряд требований, которым должна удовлетворять документация на программу.

Первое и наиболее важное из них состоит в том, что предусловия и постусловия необходимо включать в документацию. Их необходимо сформулировать в виде выражений алгебры логики. Эти формулы должны сопровождаться краткими пояснениями на естественном языке и схемами, если последние необходимы в качестве средства облегчения их восприятия читателем. При этом следует учитывать затраты времени и простоту восприятия.

В документации по подпрограмме необходимо четко указывать, на какие переменные не влияет ее выполнение. Обычно данное требование выполняется в результате перечисления *всех переменных*, которые изменяются (или, что более точно, могут изменяться) рассматриваемой подпрограммой. Совершенно обязательно в документации должен быть указан *инвариант цикла* для *каждого цикла*.

Кроме того, в документации необходимо указать те условия (суждения), которые должны быть истинными в определенных местах подпрограммы. В этом плане особенно удобными являются места между циклами и условными операторами, а также до и после последовательности операторов присваивания. Не требуется включать в документацию те условия, которые можно легко получить непосредственно из других условий. Но в документацию обязательно должны быть включены те условия, которые отражают принятые в работе решения или которые можно получить лишь в результате выполнения длинных алгебраических преобразований, занимающих много времени. Как отмечалось выше, содержащиеся в документации условия будут особенно полезны при последующем доказательстве правильности программы и ее сопровождении (изменении).

Любой программист, который составляет свою программу и включает в нее оператор вызова документированной подпрограммы, должен знать предусловия и постусловия для этой подпрограммы, а также переменные, которые изменяются в результате ее выполнения. Предусловие указывает ему, какое состояние должна подготовить его программа перед вызовом подпрограммы, то есть, что ему следует учитывать перед вызовом. Постусловие сообщает ему о том, что он может считать истинным после вызова подпрограммы.

И хотя методы доказательства правильности программ изо своей «громоздкости» существенно ограничены при доказательстве

безопасности большинства программных комплексов, тем не менее, есть области, где данные методы могут найти прикладное значение. Следующий пример характеризует это.

2.5. Пример доказательства правильности программы для алгоритма дискретного экспоненцирования

2.5.1. Вводные замечания

Большинство известных алгоритмов электронной цифровой подписи, криптосистем с открытым ключом и схем вероятностного шифрования (см. приложение) в качестве основной алгоритмической операции используют дискретное возведение в степень. Стойкость соответствующих криптографических схем основывается (как правило, гипотетически) или на сложности извлечения корней в поле $GF(n)$, n - произведение двух больших простых чисел, или на трудности вычисления дискретных логарифмов в поле $GF(p)$, p - большое простое число. Чтобы противостоять известным на данный момент методам решения этих задач операнды должны иметь длину порядка 512 или 1024 битов. Понятно, что выполнение вычислений над операндами повышенной разрядности (еще будет употребляться термин «операнды многократной точности» по аналогии с операндами однократной и двукратной точности [Кн]) требует высокого быстродействия рабочих алгоритмов криптографических схем.

2.5.2. Представление чисел

Пусть A, N, e - три целых положительных числа многократной точности, причем $A<N$. Тогда для любого e при вычислении $A^e(\mathrm{mod}\ N)\equiv C$, результат редукции $C\in\{1,N\text{-}1\}$. Если e представить n-разрядным двоичным вектором, то всю операцию возведения в степень можно свести к чередованию операций вида $A*B$ modulo N и $B*B$ modulo N, где $0<B<N\text{-}1$ [Кн, стр.482-510]. Таким образом, во всех дальнейших рассуждениях e будет представляться только как двоичная строка. Кроме того, числа A, B, N, а также P - частичное произведение и S - текущий результат будут представляться n-битовыми двоичными векторами, например, $N[1,n]$, где $N[1]$ и $N[n]$ - младший и старший биты N соответственно.

Алгоритм ***P*** (при разработке и доказательстве его правильности использованы результаты работы [Bk]), реализует вычислительную систему с фиксированной длиной слова, то есть A, B, N, P и S будут также рассматриваться как векторы $A[1,m]$, $B[1,m]$, $N[1,m]$, $P[1,m']$ и $S[1,m']$, где

каждый элемент вектора (элемент одномерного массива) есть цифра r-ичной системы счисления, $m'=m+h$, величина h будет изменяться в зависимости от вида алгоритма. Основание r такой системы будет ограничено длиной машинного слова λ и цифры такой системы имеют вид $0,1,...,r$-1 (r выбирается как целое положительное основание с неотрицательной базой). При этом n и m связаны соотношением $n=s*m$, где $s=\log_2 r$ (в дальнейших рассуждениях log - логарифм по основанию 2). Наиболее целесообразно выбрать основание $r=2^\lambda$ как наиболее экономное представление чисел в машине, так как при $r<2^\lambda$ на представление чисел уходит больше памяти. Например, широко принятое на практике представление десятичных чисел в двоично-десятичном коде требует на 20 % большего объема памяти, чем двоичное представление тех же чисел.

Тем не менее, иногда полезно представлять ситуацию, когда $r=10$ [Кн, стр.283] или $r=10^k$, например, при отладке программ.

Следует также обратить внимание на тот факт, что при выполнении арифметических операций над числами многократной точности, например, по классическим алгоритмам Кнута [Кн, стр.282-302], основание r следует уменьшать, чтобы не возникло переполнение разрядной сетки. Так, для операции сложения уменьшение выполняется до $r=2^{\lambda-1}$, для умножения - до $r=2^{\lambda/2}$. Однако если архитектурой вычислительной системы предусмотрен флаг переноса или хранение промежуточного результата с двойной точностью, то можно возвращаться к основанию $r=2^\lambda$.

*2.5.3. Алгоритм A*B modulo N - алгоритм выполнения операции модулярного умножения*

Операнды многократной точности для данного алгоритма представляются в виде одномерного массива целых чисел. Для знака можно зарезервировать элемент с нулевым индексом. Особенности представления чисел при организации взаимодействия алгоритма с другими рабочими программами, при организации ввода-вывода и т.д. рассматриваются, например, в работе [Hu]. В алгоритме использован известный вычислительный метод «разделяй и властвуй» и реализован способ вычисления «цифра-за-цифрой». Прямое умножение не следует делать по нескольким причинам: во-первых, произведение $A*B$ требует в два раза больше памяти, чем при методе «цифра-за-цифрой»; во-вторых, умножение двух многоразрядных чисел труднее организовать, чем умножение числа многократной точности на число однократной точности.

Так, в работе [BW] приводятся расчеты на супер-ЭВМ «CRAY-2» для 100-разрядных десятичных чисел: модулярное умножение методом «цифра-за-цифрой» выполняется примерно на 10% быстрее, чем прямое умножение и следующая за ним модульная редукция. Алгоритм модулярного умножения (алгоритм ***P***) приведен на рис.2.2, а на рис.2.1 представлен псевдокод процедуры **ADDK**.

Алгоритм ***ADDK***

```
carry:=0;
for i:=1 to m do
  begin
     t:=P[i]+k*N[i]+carry;
     P[i]:=t mod r;
     carry:=t div r;
  end;
P[m+1]:=carry;
write(P); {P - результат}
```

Рис.2.1. Псевдокод алгоритма вычисления $P+k*N$
(процедура ADDK)

Так как $B[i]\in[0,...,2^{\lambda/2}-1]$, то проверку «if $B[i]$<>0» в алгоритме ***P*** можно не вводить потому, что вероятность того, что $B[i]$ будет равняться 0 пренебрежимо мала, если значение λ не достаточно мало. Ошибка затем все равно будет алгоритмом обнаружена. Проверка

«if *p*_short-*k***n*_short>*n*_short DIV 2»

позволяет представлять k числом однократной точности и работать с наименьшими абсолютными остатками в каждой итерации. Значение операнда P_i на каждом шаге итерации должно быть ограничено величиной N.

Теорема 2.1. Пусть P_i - частичное произведение P в конце i-той итерации (т.е. в конце i-того цикла FOR алгоритма ***P***). Тогда для любого i (i=[1,...,n]) $\text{abs}(P_i)<N$, $r^{m-1}\le N\le r^m$.

Доказательство. Доказательство теоремы проведем методом индукции.

Если k=abs(p_short) DIV n_short, где DIV - целочисленное деление, то

$$p_\text{short}=(k+\delta)*n_\text{short}, \qquad (2.1.6)$$

где k - целое, $0\le k<r-1$ и $0\le\delta<1$.

Проверка «if p_short-k*n_short>n_short DIV 2» есть ни что иное, как проверка

$$\delta > 0.5 \quad (2.1.7)$$

На i-том шаге алгоритм вычисляет:

$$P' = P_{i-1} * r + A * B[i] \quad (2.1.8)$$

Алгоритм ***P***

```
m_shifts:=0;
while n[m_shifts]=0 do
  begin
     shift_left(N and A);
     m_shifts:=m_shifts+1;
  end;
m:=m_shifts;
reset(P);
n_short:=N[m];
for i:=n downto 1 do
  begin
     shift_left(P); {сдвиг на 1 элемент влево или умножение P*r}
     if b<>0 then
        addk(A*B[i],{to}P);
     let p_short represent the 2 high assimilated digits of P;
     k:=abs(p_short) div n_short;
     if p_short-k*n_short>n_short div 2 then k:=k+1;
     if k>0 then
        begin
          if p_short<0 then
            addk(k*N,{to} P)
          else
            addk(-k*N,{to} P);
        end;
     end;{for}
right shift P, N by m_shifts;
if P<0 then
  P:=P+N;
write(P); {P - результат}
```

Рис. 2.2. Псевдокод алгоритма модулярного умножения $A*B$ modulo N

Возможны два варианта:

Вариант 1. Если k=0, т.е. n_short>abs(p_short) (см. алгоритм), то суммирование при помощи процедуры **ADDK** не производится и утверждение теоремы выполняется, т.е. abs(P_i)<N.

Вариант 2. Если k>0, т.е.

n_short<abs(p_short) (2.1.9)

Здесь также возможны два варианта:

Вариант А:

p_short<0 (2.1.10)

Из (2.1.9) и (2.1.10) следует $P'<-N$ и так как $P_i=-P'+k*N$ (см. алгоритм), то согласно (2.1.7)

$P_i=\delta*N$, если $\delta\le 0.5$ (2.1.11)

и так как $P_i=-P'+(k+1)*N$, то

$P_i=-(1-\delta)*N$, если $\delta>0.5$ (2.1.12)

Вариант В:

p_short>0 (2.1.13)

Из (2.1.9) и (2.1.13) следует $P'>N$ и так как $P_i=P'-k*N$, то согласно (2.1.7)

$P_i=-\delta*N$, если $\delta\le 0.5$ (2.1.14)

и так как $P_i=P'-(k+1)*N$, то

$P_i=(1-\delta)*N$, если $\delta>0.5$ (2.1.15)

Во всех случаях (2.1.11), (2.1.12), (2.1.14) и (2.1.15), так как $0\le\delta<1$, то abs(P_i)<N.

Теорема доказана ■.

Примечание. Для чего нужна проверка (2.1.7)

«if p_short-k*n_short>n_short DIV 2» ?

Пусть в конце каждой итерации P принимает максимально возможное значение P_i-1=N-1, а N, A и $B[i]$ заведомо тоже велики: $N=r^{n+1}-1$, $A=r^{n+1}-2$, $B[i]=r-1$. Тогда на i-том шаге согласно (2.1.8):

$$P_i'=(r^{n+1}-2)*r+(r^{n+1}-2)*(r-1)=2*r^{n+2}-r^{n+1}-4*r+2 \quad (2.1.16)$$

$$\frac{2r^{n+2}-r^{n+1}-4r+2}{r^{n+1}-1}=2r-1-\frac{2r+1}{r^{n+1}-1} \quad (2.1.17)$$

При достаточно большом m, если не введена проверка (2.1.6), то $k<2*r-1$, по условию же $0<k<r-1$. И из (2.1.16) и (2.1.17) видно, что P придется представлять $m+2$ разрядами (что определяется слагаемым $2*r^{n+2}$), по условию же $m+1$. Если же ввести проверку (2.1.7), то даже при $\delta=0,5$ т.е. $P_{i-1}=(N-1)/2$ и с учетом того, что если неравенство (2.1.7) выполняется, то P_i меняет знак на противоположный (сравн. (2.1.11), (2.1.12), (2.1.14) и (2.1.15)), из (2.1.6) и (2.1.7) получим, что $0 \le k<(1/2)*r-1$, что удовлетворяет наложенному на k условию, и для представление P достаточно $m+1$ разряда.

В алгоритме ***P*** используется также известный метод, когда для получения частного от деления двух многоразрядных чисел, используются только старшие цифры этих чисел (см., например, алгоритм ***D*** в работе [Кн, стр.291-295]). Чем больше основание системы счисления r, тем меньше вероятность того, что пробное частное k от деления первых цифр больших чисел не будет соответствовать действительному частному.

Методы доказательства правильности программ могут быть применены при разработке ПО при существенных ограничениях на размеры и сложность создаваемых программ. И в частных случаях они могут оказаться более эффективными, чем другие известные методы анализа программ, которые исследуются в следующих разделах данной работы.

http://www.natahaus.ru/
BEST rus DOCY

ГЛАВА 3. КОНФИДЕНЦИАЛЬНЫЕ ВЫЧИСЛЕНИЯ

3.1. ВОДНЫЕ ЗАМЕЧАНИЯ ПО ПРОБЛЕМАТИКЕ КОНФИДЕНЦИАЛЬНЫХ ВЫЧИСЛЕНИЙ

В последнее время появилась насущная необходимость в создании новых информационных технологий разработки ПО, исходно ориентированных на создание безопасных программных продуктов относительно заданного класса решаемых задач. В этом случае проблема исследований сводится к разработке таких математических моделей, которые представляются адекватной формальной основой для создания методов защиты программного обеспечения на этапе его проектирования и разработки. При этом изначально предполагается, что:

- один или несколько участников проекта являются (или, по крайней мере, могут быть) злоумышленниками;
- в процессе эксплуатации злоумышленник может вносить в программы изменения (например, внедрение компьютерных вирусов, внесение программных закладок);
- средства вычислительной техники, на которых выполняются программы, не свободны от аппаратных закладок.

Тогда, исходя из этих допущений, формулируется следующая неформальная постановка задачи: «Требуется разработать программное обеспечение таким образом, что, несмотря на указанные выше «помехи», оно функционировало бы правильно». Одно из основных достоинств здесь состоит в том, что одни и те же методы позволяют защищаться от злоумышленника, действующего как на этапе разработки, так и на этапе эксплуатации программного обеспечения. Однако, это достигается за счет некоторого замедления вычислений, а также повышения затрат на разработку программного обеспечения.

В рамках указанного выше подхода на данный момент известны несколько направлений, к числу которых, в том числе, относится *теория конфиденциальных вычислений* или ее основная составляющая – *теория конфиденциального вычисления функции*, введение в которую дается ниже.

Задачу конфиденциального вычисления функции, которая решается посредством многостороннего интерактивного протокола можно описать в следующей постановке. Имеется n участников протокола или n

процессоров вычислительной системы, соединенных сетью связи. Изначально каждому процессору известна своя «часть» некоторого входного значения x. Требуется вычислить $f(x)$, f - некоторая известная всем участникам вычислимая функция, таким образом, чтобы выполнились следующие требования:

- *корректности,* когда значение $f(x)$ должно быть вычислено правильно, даже если некоторая ограниченная часть участников произвольным образом отклоняется от предписанных протоколом действий;
- *конфиденциальности*, когда в результате выполнения протокола не один из участников не получает никакой дополнительной информации о начальных значениях других участников (кроме той, которая содержится в вычисленном значении функции).

Можно представить следующий сценарий использования этой модели для разработки безопасного программного обеспечения. Имеется некоторый процесс, для управления которым необходимо вычислить функцию f. При этом последствия вычисления неправильного значения таковы, что представляется целесообразным пойти на дополнительные затраты, связанные с созданием сети из n процессоров и распределенного алгоритма для вычисления f. В системе имеется еще один абсолютно надежный участник, который имеет доступ к секретному значению x и имеет возможность выделить каждому участнику свою «долю» x. Название протоколы «конфиденциальное вычисление функции» отражает тот факт, что требование конфиденциальности является основным, то есть значение x не должно попасть в руки злоумышленника.

Задача конфиденциального вычисления была впервые сформулирована А. Яо для случая с двумя участниками в 1982 г. [Y1]. В 1987 г. О. Голдрайх, С. Микали и А. Вигдерсон показали [GMW], как безопасно вычислить любую функцию, аргументы которой распределены среди участников в вычислительной установке (то есть в конструкции, где потенциальный противник ограничен в действиях вероятностным полиномиальным временем). В их работе рассматривалась синхронная сеть связи из n участников, где каналы связи небезопасны и стороны, также как и противник, ограниченны в своих действиях вероятностным полиномиальным временем. В своей модели вычислений они показали, что в предположении существования односторонних функций с секретом можно построить многосторонний протокол (с n участниками)

конфиденциальных вычислений любой функции в присутствие пассивных противников (т.е., противников, которым позволено только «прослушивать» коммуникации). Для некоторых типов противников (для византийских сбоев) авторы привели протокол для конфиденциального вычисления любой функции, где ($\lceil n/2 \rceil$-1) участников протокола являются нечестными (или ($\lceil n/2 \rceil$-1)-протокол конфиденциальных вычислений).

В дальнейшем изучались многосторонние протоколы конфиденциальных вычислений в модели с защищенными каналами. Было показано, что если противник пассивный, то существует ($\lceil n/2 \rceil$-1)-протокол для конфиденциального вычисления любой функции. Если противник активный (т.е., противник, которому позволено вмешиваться в процесс вычислений), тогда любая функция может быть вычислена посредством ($\lceil n/3 \rceil$-1)-протокола конфиденциальных вычислений. Эти протоколы являются безопасными в присутствие неадаптивных противников (т.е., противников, действующих в схеме вычислений, в которой множество участников независимо, но фиксировано).

В последнее время исследуются вычисления для случая активных противников, ограниченных в работе вероятностным полиномиальным временем, где часть участников вычислений может быть «подкуплена» противником и многосторонние конфиденциальные вычисления при наличии незащищенных каналов и с вычислительно неограниченным противником, а также исследовались многосторонние конфиденциальные вычисления с нечестным большинством участников при наличии защищенных каналов. Кроме того, изучаются многосторонние протоколы конфиденциальных вычислений при наличии защищенных каналов и динамического противника (т.е., противника, который может «подкупать» различных участников в разные моменты времени). В фундаментальной работе [MR] предложены определения для многосторонних конфиденциальных вычислений при наличии защищенных каналов и в присутствии адаптивных противников.

В работе [BCG], авторы начали комплексные исследования асинхронных конфиденциальных вычислений. Они рассмотрели полностью асинхронную сеть (т.е., сеть, где не существует глобальных системных часов), в которой стороны соединены защищенными каналами связи. Авторы привели первый асинхронный протокол византийских соглашений с оптимальной устойчивостью, где противник может «подкупать» $\lceil n/3 \rceil$-1 из n участников вычислений.

3.2. Описание используемых примитивов, схем и протоколов

3.2.1. Общие определения

В качестве одного из основных математических объектов в данной работе используются односторонние функции, то есть вычислимые функции, для которых не существует эффективных алгоритмов инвертирования. Необходимо отметить, что односторонняя функция - гипотетический объект, поэтому называть конкретные функции односторонними математически некорректно. Впервые такую гипотетическую конструкцию для конкретного криптографического приложения, - открытого распределения ключей, предложили У. Диффи и М. Хеллман в 1976 г. [DH]. Они показали, что вычисление степеней в мультипликативной группе вычетов над конечным полем является простой задачей с точки зрения состава необходимых вычислений, в то время как извлечение дискретных логарифмов над этим полем – предположительно сложная вычислительная задача.

Неформально говоря, для двух независимых множеств X и Y функция $f{:}X\to Y$ называется *односторонней*, если для каждого $x\in X$ можно легко вычислить $f(x)$, в то время как почти для всех $y\in Y$ вычислительно трудно получить такой $x\in X$, что $f(x)=y$, при условии , что такой x существует.

Все формальные определения односторонних функций в терминах теории сложности вычислений даны в приложении.

3.2.2. (n,t)-Пороговые схемы

Используемая в данном подразделе (n,t)-пороговая схема, известная как схема разделения секрета Шамира, – это протокол между $n+1$ участниками, в котором один из участников, именуемый дилер - **Д**, распределяет частичную информацию (доли) о секрете между n участниками так, что:

- любая группа из менее чем t участников не может получить любую информацию о секрете;
- любая группа из не менее чем t участников может вычислить секрет за полиномиальное время.

Пусть секрет s - элемент поля $\boldsymbol{F}$. Чтобы распределить s среди участников $P_1,\ldots,P_n$, (где $n<|\boldsymbol{F}|$) дилер выбирает полином $f\in\boldsymbol{F}[x]$ степени не более t-1, удовлетворяющий $f(0)=s$. Участник P_i получает $s_i=f(x_i)$ как свою

долю секрета s, где $x_i \in \boldsymbol{F} \backslash \{0\}$ – общедоступная информация для P_i ($x_i \neq x_j$ для $i \neq j$).

Вследствие того, что существует один и только один полином степени не менее k-1, удовлетворяющий $f(x_i)=s_i$ для k значений i, схема Шамира удовлетворяет определению (n,t)-пороговых схем. Любые t участников могут найти значение f по формуле:

$$f(x) = \sum_{l=1}^{k} (\prod_{h \neq l} \frac{x - x_{i_l}}{x_{i_l} - x_{i_h}}) f(x_{i_l}) = \sum_{l=1}^{k} (\prod_{h \neq l} \frac{x - x_{i_l}}{x_{i_l} - x_{i_h}}) s_{i_l} .$$

Следовательно

$$s = \sum_{j=1}^{k} a_j s_{i_j} ,$$

где a_1,...,a_k получаются из

$$a_j = \prod_{h \neq j} \frac{x_{i_h}}{x_{i_h} - x_{i_j}}$$

Таким образом, каждый a_i является ненулевым и может быть легко вычислен из общедоступной информации.

3.2.3. Проверяемая схема разделения секрета

Пусть имеется n участников вычислений и t^* (значение t^* не более порогового значения t) из них могут отклоняться от предписанных протоколом действий. Один из участников назначается дилером **Д**, которому (и только ему) становится известен секрет (секретная информация) s. На первом этапе дилер вне зависимости от действий нечестных участников осуществляет привязку к уникальному параметру u. Идентификатор дилера известен всем абонентам системы. На втором этапе осуществляется открытие (восстановление) секрета s всеми честными участниками системы. И если дилер **Д** – честный, то $s=u$.

Проверяемая схема разделения секрета ***ПРС*** представляет собой пару многосторонних протоколов (**РзПр**,**ВсПр**), - а именно протокола разделения секрета и проверки правильности разделения **РзПр** и протокола восстановления и проверки правильности восстановления секрета **ВсПр**, при реализации которых выполняются следующие условия безопасности. (Все обозначения даны в соответствии с приложением).

Условие полноты. Для любого s, любой константы $c>0$ и для достаточно больших n вероятность

Prob((n,t,t^*)**РзПр**=(Да,s)↲$t^*<t$ & **Д** - честный)>1-n^{-c}.

Условие верифицируемости. Для всех возможных эффективных алгоритмов **Прот**, любой константы $c>0$ и для достаточно больших n вероятность

$\text{Prob}((t^*,(\text{Да},u))$**ВсПр**$=(s=u)\hookleftarrow(n,t,t^*)$**РзПр**$=(\text{Да},u)\&t^*<t$ & **Д** - честный$)<n^{-c}$.

Условие неразличимости. Для секрета $s^*\in_R S$ вероятность

$\text{Prob}(s^*=s\hookleftarrow(n,t,t^*)$**РзПр**$=(\text{Да},s^*)$ & **Д** - честный$)<1/|S|$.

Свойство полноты означает, что если дилер **Д** честный и количество нечестных абонентов не больше t, тогда при любом выполнении протокола **РзПр** завершится корректно с вероятностью близкой к 1. Свойство верифицируемости означает, что все честные абоненты выдают в конце протокола **ВсПр** значение u, а если **Д** – честный, тогда все честные абоненты восстановят секрет $s=u$. Свойство неразличимости говорит о том, что при произвольном выполнении протокола **РзПр** со случайно выбранным секретом s^*, любой алгоритм **Прот** не может найти $s^*=s$ лучше, чем простым угадыванием.

3.2.4. Широковещательный примитив (Br-протокол)

Введем следующее определение. Протокол называется *t-устойчивым широковещательным протоколом*, если он инициализирован специальным участником (дилером **Д**), имеющим вход m (сообщение, предназначенное для распространения) и для каждого входа и коалиции нечестных участников (числом не более t) выполняются условия.

Условие завершения. Если дилер **Д** - честный, то все честные участники обязательно завершат протокол. Кроме того, если любой честный участник завершит протокол, то все честные участники обязательно завершат протокол.

Условие корректности. Если честные участники завершат протокол, то они сделают это с общим выходом m^*. Кроме того, если дилер честный, тогда $m^*=m$.

Необходимо подчеркнуть, что свойство завершения является более слабым, чем свойство завершения в византийских соглашений (см. далее). Для *Br*-протокола не требуется, чтобы честные участники завершали протокол, в том случае, если дилер нечестен.

Br-протокол

Код для дилера (по входу m):

1. Послать сообщение (*сбщ*,m) всем участникам и завершить протокол с выходом m.

Код для участников:

2. После получения первых сообщений (*сбщ*,*m*) или (*эхо*,*m*), послать (*эхо*,*m*) ко всем участникам и завершить протокол с выходом *m*.

Предложение 3.1. *Br*-протокол является *n*-усточивым широковещательным протоколом для противников, которые могут приостанавливать отправку сообщений.

Доказательство. Если дилер честен, то все честные участники получают сообщение (*сбщ*,*m*) и, таким образом, завершают протокол с выходом *m*. Если честный участник завершил протокол с выходом *m*, то он посылает сообщение (*эхо*,*m*) ко всем другим участникам и, таким образом, каждый честный участник завершит протокол с выходом *m*.

Ниже описывается $\lfloor (n\text{-}1)/3 \rfloor$-устойчивый широковещательный протокол, который именуется **BB**, где $t \le \lfloor (n\text{-}1/3) \rfloor$ - количество участников, которые могут быть нечестными. В протоколе принимается, что идентификатор дилера содержится в параметре *m*.

Протокол BB

Код для дилера (по входу m):

1. Послать сообщение (*сбщ*,*m*) ко всем участникам.

Код для участника P_i:

2. После получения сообщения (*сбщ*,*m*), послать сообщение (*эхо*,*m*) ко всем участникам.
3. После получения *n*-1 сообщений (*сбщ*,*m'*), которые согласованы со значением *m'*, послать сообщение (*гот*,*m'*) ко всем участникам.
4. После получения *t*+1 сообщений (*гот*,*m'*), которые согласованы со значением *m'*, послать (*гот*,*m'*) ко всем участникам.
5. После получения *n*-1 сообщений (*сбщ*,*m'*), которые согласованы со значением *m'*, послать сообщение (*OK*,*m'*) ко всем участникам и принять *m'* как распространяемое сообщение.

3.2.5. Протокол византийского соглашения (BA-протокол)

При *византийских соглашениях* или при реализации *протокола византийских соглашений* для любого начального входа x_i, $i \in [1,...,n]$

участника i и некоторого параметра d (соглашения) должны быть выполнены следующие условия.

Условие завершения. Все честные участники вычислений в конце протокола принимают значение d.

Условие корректности. Если существует значение x такое, что для честных участников $x_i=x$, тогда $d=x$.

3.3. Обобщенные модели для сети синхронно и асинхронно взаимодействующих процессоров

3.3.1. Вводные замечания

Протоколы конфиденциального вычисления функции относятся к протоколам, которые предназначены, прежде всего, для защиты процесса вычислений от действия «разумного» противника (злоумышленника), то есть от противника, который всегда выбирает наихудшую для нас стратегию.

В моделях конфиденциальных вычислений вводится дополнительный параметр t, $t<n$, - максимальное число участников, которые могут отклоняться от предписанных протоколом действий, то есть максимальное число злоумышленников, n – общее число участников протокола. Поскольку злоумышленники могут действовать заодно, обычно предполагается, что против протокола действует один злоумышленник, который может захватить и контролировать любые t из n участников по своему выбору.

3.3.2. Обобщенные модели сбоев и противника

Рассматривается сеть взаимодействующих процессоров. Некоторые процессоры могут «сбоить». При *сбоях, приводящих к останову* (*FS-сбоях*), *сбоящий процессор* может приостановить в некоторый момент времени отправку своих сообщений. В то же время предполагается, что сбоящие процессоры продолжают получение сообщений и могут выдавать «некую информацию» в свои выходные каналы. При *Византийских сбоях* (*By-сбоях*) процессоры могут произвольным образом сотрудничать друг с другом с целью получения необходимой для них информации или с целью нарушения процесса вычислений. При *By*-сбоях сбоящие процессоры могут объединять свои входы и изменять их. В то же время это должно происходить при условии невозможности изучения любой информации о входах *несбоящих процессоров*.

Сбои могут быть *статическими* и *динамическими*. При статических сбоях множество сбоящих процессоров фиксировано в начале и процессе вычислений, при динамических – множество сбоящих процессоров может меняться, как может меняться и характер самих сбоев. Сбоящие процессоры будут также называться *нечестными* участниками вычислений, в противоположность *честным* участникам, которые выполняют предписанные вычисления.

Противника можно представлять как некий универсальный процессор, действия которого заключаются в стремлении «сделать» процессоры сети сбоящими («подкупить» их).

По аналогии со сбоями противник может быть динамическим и статическим. *Статическим противником* является противник, который «сотрудничает» с фиксированным количеством сбоящих процессоров («подкупленных» противником). При действиях *динамического противника* количество сбоящих процессоров может меняться (в том числе непредсказуемым образом).

Кроме того, противник может быть *активным* и *пассивным*, а также с *априорными* и *апостериорными* протокольными и раундовыми действиями. Пассивный противник не может изменять сообщения, циркулируемые в сети. Активный противник может знать все о внутренней конфигурации сети, может читать и изменять все сообщения сбоящих процессоров и может выбирать сообщения, которые будут посылать сбоящие процессоры при вычислениях. Активный динамический противник может в начале каждого раунда «подкупить» несколько новых процессоров. Таким образом, он может изучить информацию, получаемую ими в текущем раунде, и принять решение «подкупить» ли ему новый процессор, или нет. Такой противник может собирать и изменять все сообщения от сбоящих процессоров к несбоящим. Некоторые сценарии могут предполагать, что все сообщения (в том числе, между честными процессорами) будут доставлены адресатам в конце текущего раунда. Действия аналогичного характера активный противник может выполнить не только в течение раунда (в его начале и конце), но и до начала выполнения протокола и после его завершения.

Противник называется *t-противником*, если он сотрудничает с t процессорами.

3.3.3. Получестные модели

Обычно основная цель при конфиденциальных вычислениях (как, впрочем, и в большинстве криптографических протоколов) состоит в том, чтобы разработать протоколы, которые могут противостоять любому возможному поведению противника. При конфиденциальных вычислениях это делается, как правило, за два этапа. Сначала рассматривается так называемый «удобный» противник, называемый в данном случае *получестным* и строятся протоколы, которые являются безопасными в отношении такого противника. Затем, показывается как процессоры «ведут себя» в присутствии такого противника. А затем показывается, как преобразовать любой протокол, безопасный в получестной модели, в протокол, который является безопасным против любого возможного поведения противника.

В получестной модели имеется участник, именуемый *получестным процессором*, который следует предписанным протоколом действиям, за исключением того, что он может хранить информацию обо всех своих промежуточных вычислениях. Фактически достаточно хранить только внутренние случайно сгенерированные параметры (результаты подбрасывания монеты) и все сообщения, полученные от других процессоров. Таким образом, получестный процессор «подбрасывает честную монету» (генерирует случайный бит с вероятностью ½) и посылает сообщения в соответствии с инструкциями специальной программы (как функции от входа, результатов подбрасываний монеты и входных сообщений). Отметим, что получестный процессор соответствует честному проверяющему в доказательствах с нулевым разглашением (см. приложение).

Такие получестные модели могут иметь самостоятельный интерес. В частности фактическое отклонение от инструкций специальной программы (которая может представлять собой достаточно сложное внутреннее программное обеспечение) может быть более сложным, чем простое изменение содержания некоторых коммуникационных регистров. Например, содержание этих регистров может быть доступно для некоторых стандартных операций операционной системы. Таким образом, в то время как к полностью честному поведению (в отличие от получестного) предъявляется повышенные требования, получестное поведение может рассматриваться как приемлемое во многих криптографических конструкциях.

Получестная модель является неявной в следующем определении безопасности. Неформально говоря, протокол конфиденциально вычисляет функцию f, если любой получестный процессор, принимавший участие в работе протокола, мог бы быть по существу процессором, имеющим только вход и выход, доступные такому процессору. Это делается с использованием соответствующей парадигмы моделирования. Для этого достаточно (эффективно) «промоделировать взгляд» каждого (получестного) процессора, так что все, что может быть получено после участия получестного процессора в протоколе, может быть получено из этого «промоделированного взгляда» (см. далее).

3.3.4. Злонамеренные модели

Действия процессоров в злонамеренной модели

Теперь будем рассматривать независимо возможное отклонение процессоров от предопределенных протоколом действий. Для этого необходимо сделать несколько замечаний. Во-первых, для противника не существует никакого способа принудить процессоры участвовать в выполнении каких-либо действий при выполнении протокола. То есть возможное злонамеренное поведение противника не может состоять в инициализации протокола, в его приостановке или прерывания работы в любой желательной точке в какое-либо время.

Во-вторых, необходимо отметить, что не существует никакого способа для противника зафиксировать корректный вход протокола. То есть, процессор всегда может модифицировать свой локальный вход и не существует никакого способа для противника, чтобы предотвратить это. Отметим, что оба подобных сценария не могут, по очевидной причине, произойти в получестной модели, так как предполагалось, что такие процессоры в такой модели не отклоняются указанным ниже образом от протокола:

- процессоры могут отказаться участвовать в протоколе (когда инициируется протокол);
- процессоры подменяют свои локальные входы (и могут участвовать в протоколе с входами других процессоров);
- процессоры преждевременно прерывают протокол (например, перед посылкой своего последнего сообщения).

Вычисления в идеальной модели

Теперь перенесем все вышеупомянутые рассуждения в определения идеальной модели. Будем предполагать, что процессоры имеют в своем распоряжении доверенное третье лицо. Определим его как *доверенный процессор* или *TP-процессор*. Однако даже такая сторона не может предотвращать специфические злонамеренные действия. В идеальной модели нечестному процессору позволяется отказаться участвовать в протоколе или подменять свои локальные входы. Таким образом, вычисления в идеальной модели для случая с двумя процессорами могут выполняться следующим образом. (Более подробно идеальный и реальный сценарии для различных моделей вычислений рассматриваются далее).

Вычисления в идеальной модели

Вход. Каждый из двух процессоров получает вход z.

Посылка входов доверенному процессору. Несбоящий процессор всегда посылает z доверенному процессору. Сбоящий процессор может, в зависимости от z, или прервать, или послать некоторый $z' \in \{0,1\}^{|z|}$ доверенному процессору.

Доверенный процессор «отвечает» первому процессору. В случае если была получена входная пара (x,y) доверенный процессор, вычисляя f, сначала отвечает первому процессору $f_1(x,y)$. В противном случае (то есть, когда получен только один вход) доверенный процессор отвечает обеим сторонам с специальным символом ∇.

Доверенный процессор «отвечает» второму процессору. В случае если первый процессор нечестен, то доверенный процессор посылает ∇ второму процессору и останавливается. В противном случае (то есть, если не останавливается) доверенный процессор посылает $f_2(x,y)$ второй стороне.

Выход. Несбоящий процессор всегда выводит сообщение, которое было получено от доверенного процессора. Сбоящий процессор может выдавать значение независимой (полиномиально вычислимой) функции от начального входа и сообщения, полученное от доверенного процессора.

Вычисления в реальной модели

Рассматривается реальная модель, в которой выполняется реальный протокол (и в ней не существует никаких доверенных третьих лиц). В этом случае, сбоящий процессор может следовать любой возможной стратегии, то есть любой стратегии, реализуемой схемой полиномиальной длины. В

частности сбоящий процессор может прервать в какое-либо время свою работу в любой точке протокола.

Мы также будем предполагать, что противник в реальной модели - детерминирован. Интуитивно, (неоднородный) детерминированный противник рассматривается как мощный (с вычислительной точки зрения) и рандомизированный противник. *При определении безопасности требуется эффективное преобразование противника для реальной модели в противника для идеальной модели.* Если такое преобразование применяется к детерминированному противнику, оно обязательно применяется к рандомизированному противнику.

3.3.5. Модель взаимодействия

Взаимодействие процессоров может осуществляться посредством *конфиденциальных и/или открытых каналов связи*. Каждые два процессора могут быть иметь симплексное, полудуплексное или дуплексное соединение. При этом каждое из соединений, в зависимости от решаемой задачи, в некоторый промежуток времени, может быть либо конфиденциальным, либо открытым.

Кроме того, может существовать *широковещательный канал связи*, то есть канал, посредством которого один из процессоров может одновременно распространить свое сообщение всем другим процессорам вычислительной системы. Такой канал еще называется *каналом с общей шиной*.

Взаимодействие в сети характеризуется *трафиком T*, который может представлять собой совокупность сообщений, полученных процессорами в r-том раунде в установленной модели взаимодействия.

3.3.6. Синхронная модель вычислений

Общее описание модели

Ниже рассматривается модель вычислений, которая будет использоваться в дальнейших рассуждениях при исследовании синхронных конфиденциальных вычислений. Таким образом, рассматривается сеть процессоров, функционирование которой синхронизируется общими часами (синхронизатором). Каждое локальное (внутреннее) вычисление соответствует одному моменту времени (одному такту часов). В данной модели отрезок времени между i и i+1 тактами называется *раундом при синхронных вычислениях.* За один раунд протокола каждый процессор может получать сообщения от любого другого

процессора, выполнять локальные (внутренние) вычисления и посылать сообщения всем другим процессорам сети. Имеется входная лента «только-для-чтения», которая первоначально содержит строку $\Lambda(k)$ (например вида 1^k), при этом k является *параметром безопасности* системы. Каждый процессор имеет ленту случайных значений, конфиденциальную рабочую ленту «только-для-чтения» (первоначально содержащую строку Λ), конфиденциальную входную ленту «только-для-чтения» (первоначально содержащую строку x_i), конфиденциальную выходную ленту «только-для-записи» (первоначально содержащую строку Λ) и несколько коммуникационных лент. Между каждой парой процессоров i и j существует конфиденциальный канал связи, посредством которого i посылает безопасным способом сообщение процессору j. Данный канал (коммуникационная лента) исключает запись для i и исключает чтение для j. Каждый процессор i имеет также широковещательный канал, представляющий собой ленту, исключающую запись для i и являющийся каналом «только-для-чтения» для всех остальных процессоров сети.

Предполагается, что в раунде r для каждого процессора i существует однозначно определенное сообщение (возможно пустое), которое распространяет процессор i в этом раунде и для каждой пары процессоров i и j существует единственное сообщение, которое i может безопасным образом послать j в данном раунде. Все каналы являются помеченными так, что каждый получатель сообщения может идентифицировать его отправителя.

Процессор i запускает программу π_i, совокупность которых, реализует распределенный алгоритм Π. *Протоколом* работы сети называется n-элементный кортеж $\boldsymbol{P}=(\pi_1,\pi_2,..,\pi_n)$. Протокол называется *$t$-устойчивым*, если t процессоров являются сбоящими во время выполнения протокола. *Историей* процессора i являются: содержание его конфиденциального и общего входов, все распространяемые им сообщения, все сообщения, полученные i по конфиденциальным каналам связи, и все случайные параметры, сгенерированные процессором i во время работы сети.

Идеальный и реальный сценарии

Для доказуемо конфиденциального вычисления вводятся понятия *идеального* и *реального сценариев* [MR]. Как было показано выше в идеальном сценарии дополнительно вводится доверенный процессор. Процессоры конфиденциально посылают свои входы доверенному

процессору, который вычисляет необходимый результат (выход) и также конфиденциально посылает его обратно процессорам сети. Противник может манипулировать с этим результатом (вычислить или изменить его) следующим образом. До начала вычислений он может подкупить один из процессоров и изучить его секретный вход. Основываясь на этой информации, противник может подкупить второй процессор и изучить его секретный вход. Это продолжается до тех пор пока противник не получит всю необходимую для него информацию. Далее у противника есть два основных пути. Он может изменить входы сбоящих процессоров. После чего те, вместе с корректными входами несбоящих процессоров, направляют свои новые измененные входы *TP*-процессору. По получению от последнего выходов (значения вычисленной функции) противник может приступить к изучению выхода каждого нечестного процессора. Второй путь заключается в последовательном изучении входов и выходов процессоров, подключая их всякий раз к числу нечестных. В данном случае рассматривается противник, который не только может изучать входы нечестных процессоров, но и менять их, пробовать изучать по полученному значению функции конфиденциальные входы честных процессоров.

В реальном сценарии не существует доверенного процессора, и все процессоры моделируют его поведение посредством выполнения многостороннего интерактивного протокола.

Грубо говоря, считается, что *вычисления в действительности (в реальном сценарии) безопасны, если эти вычисления «эквивалентны» вычислениям в идеальном сценарии*. Точное определение (формальное определение) *понятия эквивалентности* в этом контексте является одной из основных проблем в теории конфиденциальных вычислений.

3.3.7. Асинхронная модель вычислений

Общее описание модели

В данном подразделе рассматривается полностью асинхронная сеть из n процессоров, которые соединены конфиденциальными каналами связи. Именно такая модель взаимодействия будет исследоваться далее. При этом не существует единых глобальных часов. Любое сообщение в сети *может быть задержано независимым образом*. В то же время считается, что каждое посланное сообщение *обязательно будет получено адресатом*. Вопрос о переупорядочивании сообщений не исследуется.

Вычисления в асинхронной модели рассматривается как последовательность *шагов*. На каждом шаге активизируется один из процессоров. При этом *активизация процессора происходит по получении им сообщения*. После чего он выполняет внутренние вычисления и возможно выдает сообщения в свои выходные каналы. Порядок шагов определяется *планировщиком* (D), неограниченным в вычислительной мощности. В данной модели вычислений каждый шаг рассматривается как *раунд при асинхронных вычислениях*.

Асинхронные идеальный и реальный сценарии

Идеальный сценарий в асинхронной модели с доверенным процессором заключается в добавлении этого процессора в существующую асинхронную сеть при наличии t потенциальных сбоев (сбоящих процессоров). При этом несбоящие процессоры, также как и доверенный процессор, не могут ожидать наличия более, чем n-t входов для вычислений с целью получения их выходов, так как t процессоров (сбоящие процессоры) могут никогда не присоединиться к вычислениям.

В начале вычислений процессоры посылают свои входы доверенному процессору. В то же время, существует планировщик D, который доставляет сообщения от процессоров некоторому базовому подмножеству процессоров, мощностью не меньше n-t, обозначаемому как C и являющемуся независимым от входов честных процессоров. Доверенный процессор по получению входов - аргументов функции (возможно некорректных) из множества C, предопределенно оценивает значение вычисляемой функции, основываясь на C и входах процессоров из C. (Здесь для корректности может использоваться следующее предопределенное оценивание: установить входы из C в 0 и вычислить данную функцию). Затем доверенный процессор посылает значение оценочной функции обратно процессорам совместно с базовым множеством C. Наконец несбоящие процессоры выдают то, что они получили от доверенного процессора. Сбоящие процессоры выдают значение некоторой независимой функции, информацию о которой они «собирали» в процессе вычислений. Эта информация может состоять из их собственных входов, случайных значений, используемых при вычислениях и значения оценочной функции.

Так же как и в синхронной модели, *вычисления в реальной асинхронной модели безопасны, если эти вычисления «эквивалентны» вычислениям в сценарии с доверенным процессором.*

Далее в соответствии с работой [BCG] сделаем попытку формализовать понятия полноты и безопасности протокола асинхронных конфиденциальных вычислений.

Безопасность асинхронных вычислений

Сначала напомним, что как и сбоящие процессоры, так и планировщик имеют неограниченную вычислительную мощность.

Для вектора $\vec{x} = x_1 ... x_2$ и множества $C \subseteq [n] \cong \{1,...,n\}$, пусть $\vec{x}_C$ определяет вектор $\vec{x}$, спроектированный на индексы из C. Для подмножества $B \subseteq [n]$ и вектора $\vec{z} = z_1 ... z_{|B|}$, пусть $\vec{x}/_{(B,\vec{z})}$ определяет вектор, полученный из $\vec{x}$ подстановкой входов из B соответствующими входами из $\vec{z}$. Используя эти определения, можно определить оценочную функцию f с базовым множеством $C \subseteq [n]$ как $f_C(\vec{x}) \cong f(\vec{x}/_{(\bar{C},\vec{0})})$.

Пусть A – область возможных входов процессоров и пусть R – область случайных входов. *TP-противник* это кортеж $\boldsymbol{A}=(B,h,c,O)$, где $B \subseteq [n]$ – множество сбоящих процессоров, $h{:}A^{|B|} \times R \rightarrow A^{|B|}$ - функция подстановки входов, $c{:}A^{|B|} \times R \rightarrow \{C \subseteq [n] \mid |C| \geq n\text{-}t\}$ – функция выбора базового множества и $O{:}A^{|B|} \times R \times A \rightarrow \{0,1\}^*$ - функция выхода для сбоящих процессоров.

Функции h и O представляют собой программы сбоящих процессоров, а функция c – комбинацию планировщика и программ сбоящих процессоров.

Пусть $f{:}A^n \rightarrow A$ для некоторой области A. Выход функции вычисления f в TP-сценарии по входу $\vec{x}$ и с TP-противником $\boldsymbol{A}=(B,h,c,O)$ – это n-размерный вектор $\tau(\vec{x},\boldsymbol{A}) = \tau_1(\vec{x},\boldsymbol{A}) ... \tau_n(\vec{x},\boldsymbol{A})$ случайных переменных, удовлетворяющих для каждого $1 \leq i \leq n$:

$$\tau_i(\vec{x},\boldsymbol{A}) = \begin{cases} (C, f_C(\vec{y})) & i \notin B \\ O(x_{\vec{B}}, r, f_C(\vec{y})) & i \in B \end{cases},$$

где r - объединенный случайный вход сбоящих процессоров, $C=(\vec{x}_B, r)$ и $\vec{y} = \vec{x}/_{(B,h(x_B,r))}$

Акцентируем внимание на то, что выход сбоящих процессоров и выход несбоящих процессоров вычисляется на одном и том же значении случайного входа r.

Далее формализуем понятие вычисления «в реальной жизни».

1. Пусть $\boldsymbol{B}=(B,\beta)$ – коалиция нечестных процессоров, где $B\subset[n]$ – множество нечестных процессоров и β - их совместная стратегия.

2. Пусть $\pi_i(\vec{x},B,D)$ определяет выход процессора P_i после выполнения протокола π_i по входу $\vec{x}$, с планировщиком D и коалиции $\boldsymbol{B}$. Пусть также $\boldsymbol{P}(\vec{x},B,D)=\pi_1(\vec{x},B,D)...\pi_n(\vec{x},B,D)$.

Кроме того, пусть $f:A^n\to A$ для некоторой области A и пусть $\boldsymbol{P}$ - протокол для n процессоров. Будем говорить, что $\boldsymbol{P}$ безопасно t-вычисляет функцию f в асинхронной модели для каждой коалиции $\boldsymbol{B}$ с не более, чем из t сбоящих процессоров, если выполняются следующие условия.

Условие завершения (условие полноты). По всем входам все честные процессоры завершают протокол с вероятностью 1.

Условие безопасности. Существует TP-противник $\boldsymbol{A}$ такой, что для каждого входа $\vec{x}$ векторы $\tau(\vec{x},\boldsymbol{A})$ и $\boldsymbol{P}(\vec{x},B,D)$ идентично распределены (эквивалентны).

3.4. Конфиденциальное вычисление функции

Общие положения. Для некоторых задач, решаемых в рамках методологии конфиденциальных вычислений, достаточно введения определения *конфиденциального вычисления функции* [MR].

Пусть в сети N, состоящей из n процессоров $P_1,P_2,...,P_n$ со своими секретными входами $x_1,x_2,...,x_n$, необходимо корректно (даже при наличии t сбоящих процессоров) вычислить значение функции $(y_1,y_2,...,y_n)=f(x_1,x_2,...,x_n)$ без разглашения информации о секретных аргументах функции, кроме той, информации, которая может содержаться в вычисленном значении функции.

По аналогии с идеальным и реальным сценариями, приведенными выше, можно ввести понятия «реальное и идеальное вычисление функции f» [MR].

Пусть множество входов и выходов обозначается как X и Y соответственно, размерности этих множеств $|X|=\chi$ и $|Y|=\mu$. Множество случайных параметров, используемых всеми процессорами сети, обозначается через R, размерность - $|R|=\nu$. Кроме того, через W

обозначим рабочее пространство параметров сети. Через $T^{(r)}$ обозначается трафик в r-раунде, через $t_i^{(r)}$ – трафик для процессора i в r-том раунде, r_0 и r_k – инициализирующий и последний раунды протокола ***P*** соответственно и r^* - заданный неким произвольным образом раунд выполнения протокола ***P***.

Пусть функцию f можно представить как композицию d функций (суперпозицию функций) $g_1 \circ \ldots \circ g_\eta \circ g_{\eta+1} \circ \ldots \circ g_d$:

$$f(x_1,\ldots,x_n)=$$
$$=g_1((w_1,\ldots,w_n),(t_1^{(r_0)},\ldots,t_n^{(r_0)})) \circ \ldots \circ g_\eta((w_1,\ldots,w_n),(t_1^{(r^*)},\ldots,t_n^{(r^*)})) \circ \ldots \circ$$
$$\circ g_d((w_1,\ldots,w_n),(t_1^{(r_k)},\ldots,t_n^{(r_k)})).$$

Аргументы функции g_η являются рабочими параметрами $w_1,\ldots,w_n$ участников протокола с трафиком $(t_1,\ldots,t_n)$ в r раунде. Значения данной функции g_η являются аргументами (рабочими параметрами протокола с трафиком $(t_1,\ldots,t_n)$ в r+1 раунде) для функции $g_{\eta+1}$.

Из определения следует, что функция $f:(X^n \otimes R^n \otimes W) \rightarrow Y$, где $\otimes$ - декартово произведение множеств, реализует:

$$f(x_1,\ldots,x_n)=g_d((w_1,\ldots,w_n),(t_1^{(r^*)},\ldots,t_n^{(r^*)}))=((y_1,\ldots,y_n),(t_1^{(r_k)},\ldots,t_n^{(r_k)})).$$

Введем понятие моделирующего устройства ***M***. Здесь можно проследить некоторые аналогии с моделирующей машиной в интерактивных системах доказательств с нулевым разглашением (см., например, [Ка14,КУ4]).

Пусть $_{\mathbf{p}}\Theta^{\mathbf{Прот}}$ - распределение вероятностей над множеством историй (трафика T и случайных параметров) сбоящих процессоров во время выполнения протокола ***P***.

Моделирующее устройство, взаимодействующее со сбоящими процессорами, осуществляет свое функционирование в рамках идеального сценария. Моделирующее устройство ***M*** создает распределение вероятностей параметров взаимодействия $_{\mathbf{и}}\Theta^{\mathbf{Прот}}$ между ***M*** и сбоящим процессорами.

Протокол ***P*** *конфиденциально вычисляет функцию* $f(x)$, если выполняются следующие условия:

Условие корректности. Для всех несбоящих процессоров P_i функция

$$f(x_1,\ldots,x_n)=$$
$$=g_1((w_1,\ldots,w_n),(t_1^{(r_0)},\ldots,t_n^{(r_0)})) \circ \ldots \circ g_\eta((w_1,\ldots,w_n),(t_1^{(r^*)},\ldots,t_n^{(r^*)})) \circ$$
$$\circ g_{\eta+1}((w_1,\ldots,w_n),(t_1^{(r^*)},\ldots,t_n^{(r^*)})) \circ \ldots \circ g_d((w_1,\ldots,w_n),(t_1^{(r_k)},\ldots,t_n^{(r_k)}))=$$
$$=((y_1,\ldots,y_n),(t_1^{(r_k)},\ldots,t_n^{(r_k)}))$$

вычисляется с вероятностью ошибки близкой к 0.

Условие конфиденциальности. Для заданной тройки $(x,r,w)\in(X^n\otimes R^n\otimes W)$ распределения $_{\mathbf{р}}\Theta^{\mathbf{Прот}}$ и $_{\mathbf{и}}\Theta^{\mathbf{Прот}}$ являются *статистически не различимыми*.

Функция f, удовлетворяющая условиям предыдущего определения называется *конфиденциально вычислимой*.

3.5. Проверяемые схемы разделения секрета как конфиденциальное вычисление функции

3.5.1. Описание проверяемой схемы разделения секрета

Для вышеприведенных определений проверяемой схемы разделения секрета **ПРС** и обобщенных моделей противника, сбоев, взаимодействия и вычислений синтезируем схему разделения секрета, которая являлась бы работоспособной в протоколах конфиденциальных вычислений.

Рассматривается полностью синхронная сеть взаимодействующих процессоров в условиях проявления *Ву-сбоев.* Противник представляется как активный динамический t-противник. Взаимодействие процессоров осуществляется посредством конфиденциальных каналов связи. Кроме того, существует широковещательный канал связи.

Схема проверяемого разделения секрета, рассматриваемая как схема конфиденциального вычисления функции, значение которой является распределенный среди процессоров проверенный на корректность и затем восстановленный и проверенный секрет, обозначается как **ПРСК**.

Пусть сеть N состоит из n процессоров $P_1,P_2,\ldots,P_{n\text{-}1},P_n$, где P_n – дилер **Д** сети N. Множество секретов обозначается через S, размерность этого множества $|S|=l$. Множество случайных параметров, используемых всеми процессорами сети, обозначается через R, размерность $|R|=v$. Через W обозначается рабочее пространство параметров сети.

Требуется вычислить посредством выполнения протокола $\boldsymbol{P}$=(**РзПр**,**ВсПр**) функцию $f(x)$, где f – представляется в виде композиции двух функций $g\circ h$. Пусть функция g:$(S^n\otimes R^n\otimes W)\rightarrow W$, а функция h:$W\rightarrow S$.

Проверяемая схема разделения секрета ПРСК называется *t-устойчивой*, если протокол разделения секрета и проверки правильности разделения **РзПр** и протокол восстановления секрета **ВсПр** являются *t-устойчивыми*. Функция f является конфиденциально вычислимой, если конфиденциально вычислимы функции g и h.

t-Устойчивая верифицируемая схема разделения секрета **ПРСК** – есть пара протоколов **РзПр** и **ВсПр**, при реализации которых выполняются следующие условия.

Условие полноты. Пусть событие A_1 заключается в выполнении тождества

$f(w_1,...,w_{n-1},s)=$

$=g((w_1,...,w_{n-1},s\circ r),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s_1,...,s_{n-1},w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$.

$h((s_1,...,s_{n-1}),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s,s,...,s,w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$.

Тогда для любой константы $c>0$ и для достаточно больших n вероятность $\mathrm{Prob}(A_1)>1-n^{-c}$.

Условие корректности. Пусть событие A_2 заключается в выполнении тождества

$f(w_1,...,w_{n-1},s)=$

$=g((w_1,...,w_{n-1},s\circ r),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s_1,...,s_{n-1},w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$.

$h((s_1,...,s_{n-1}),(t_1^{(r_0)},...,t_n^{(r_0)}))=((u_1,...,u_j,...,u_{n-1},w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$,

где $u_j=s$ для $j\in G$ и G – разрешенная коалиция процессоров.

Тогда для любой константы $c>0$, достаточно больших n, для границы $t^*<t$ и любого алгоритма **Прот** вероятность $\mathrm{Prob}(A_2)<n^{-c}$.

Условие конфиденциальности.

Функция $g((w_1,...,w_{n-1},s\circ r),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s_1,...,s_{n-1},w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$ – конфиденциально вычислима.

Функция $h((s_1,...,s_{n-1}),(t_1^{(r_0)},...,t_n^{(r_0)}))=((u_1,...,u_j,...,u_{n-1},s),(t_1^{(r_k)},...,t_n^{(r_k)}))$ – конфиденциально вычислима.

Свойство полноты означает, что если все процессоры **РзПр** и **ВсПр** следуют предопределенным вычислениям, тогда любая коалиция несбоящих процессоров может восстановить секрет s. Свойство корректности означает, что при действиях t-противника **Прот**, любая разрешенная коалиция из n процессоров сети может корректно разделить, восстановить и проверить секрет. Свойство конфиденциальности вытекает из свойства конфиденциальности функции g и h.

Схема ПРСК

Предлагаемая схема **ПРСК** [GM], является $(n/3-1)$-устойчивой и использует схему разделения секрета Шамира.

Пусть $n=3t+4$ и все вычисления выполняются по модулю большого простого числа $p>n$. Из теории кодов с исправлением ошибок известно, что если мы вычисляем полином f степени $t+1$ в

n-1различных точках i, где i=1,...,n-1, тогда по данной последовательности $s_i=f(i)$, можно восстановить коэффициенты полинома за время, ограниченное некоторым полиномом, даже если t элементов в последовательности произвольно изменены. Это вариант кода Рида-Соломона, известный как код Берлекампа-Велча (см., например, [КС]). Пусть далее K – параметр безопасности и K/n означает $\lceil K/n \rceil$.

Протокол РзПр

1. Дилер **Д** выбирает случайный полином $f_0(x)$ степени t+1 с единственным условием: $f_0(0)=s$ - его секрет. Затем он посылает процессору P_i его долю $s_i=f_0(i)$. Кроме того, он выбирает 2K случайных полиномов $f_1,...,f_{2K}$ степени t+1 и посылает процессору P_i значения $f_j(i)$ для каждого j=1,...,2K.
2. Каждый процессор P_i распространяет (посредством широковещательного канала) K/n случайных битов $\alpha(i\text{-}1)_{K/n+j}$, для j=1,...,K/n.
3. Дилер **Д** распространяет полиномы $g_j=f_j+\alpha_j f_0$ для всех j=1,...,K.
4. Процессор P_i проверяет, удовлетворяют ли его значения полиномам, распространяемым дилером. Если он обнаруживает ошибку, он ее декларирует для всех. Если более чем t процессоров сообщают об этом, тогда дилер считается нечестным и все процессоры принимают по умолчанию значение нуля как секрет дилера **Д**.
5. Если менее чем t процессоров сообщили об ошибке, дилер распространяет значения, который он посылал в первом раунде тем процессорам, которые сообщали об ошибке дилера.
6. Каждый процессор P_i распространяет K/n случайных битов $\beta(i\text{-}1)_{K/n+j}$ для j=1,...,K/n.
7. Дилер **Д** распространяет полиномы $h_j=f_{K+j}+\beta_j f_0$ для всех j=1,...,K.
8. Процессор P_i проверяет, удовлетворяют ли значения, которые он имеет, полиномам, распространяемым дилером **Д** в 5-м раунде. Если он находит ошибку, он декларирует об этом всем процессорам сети. Если более чем t процессоров сообщают об ошибке, тогда дилер нечестный и все процессоры принимают по умолчанию значение нуля как секрет дилера **Д**.

Протокол ВсПр

1. Каждый процессор P_i выбирает случайный многочлен h_i степени t+1 такой, что $h_i(0)=s_i$ - его собственная входная доля секрета. Он посылает процессору P_j значение $h_i(j)$.

2. Каждый процессор P_i выбирает случайные полиномы $p_i(x)$, $q_{i,1}(x),...,q_{i,2K}(x)$ степени t+1 со свободным членом 0 и посылает процессору P_j значения $p_i(j)$, $q_{i,1}(j),...,q_{i,2K}(j)$.
3. Каждый процессор P_i распространяет K случайных битов $\gamma_{l,(i-1)K/n+m}$ для l=1,...,n и m=K/n.
4. Каждый процессор распространяет следующие полиномы r_j=$q_{i,j}$+$\gamma_{i,j}p_i$ для каждого j=1,...,K.
5. Каждый процессор P_i проверяет, что информация процессора P_l, посланная ему в 1-м раунде, соответствует тому, что P_l распространяет в 3-м раунде. Если имеется ошибка или P_l распространяет полином с ненулевым свободным членом, процессор P_i распространяет сообщение bad_l. Если более чем t процессоров распространяют bad_l, процессор P_l исключается из сети и все другие процессоры принимают как 0 долю процессора P_l. В противном случае, P_l распространяет информацию, которую он посылал в раунде 1 процессорам, распространявшим сообщение bad_l.
6. Каждый процессор P_i распространяет K случайных битов $\delta_{l,(i-1)K/n+m}$ для l=1,...,n и m=1,...,K/n.
7. Каждый процессор P_i распространяет следующие полиномы r_j=$q_{i,K+j}$+$\delta_{i,j}p_i$ для каждый j=1,...,K.
8. Каждый процессор P_i проверяет, что информация, посланная процессором P_l, в 1-м раунде и распространенная в 5-м раунде соответствует полиномам процессора P_l, распространенным в 7-м раунде. Если имеется ошибка или P_l распространил полином с ненулевым свободным членом процессор P_i распространяет bad_l^r. Если более чем t процессоров распространили bad_l, тогда P_l – нечестен и все процессоры принимают его долю, равную 0.
9. Каждый процессор P_l распределяет всем другим процессорам следующее значение s_i+$p_1(i)$+$p_2(i)$+...+$p(i)$, затем интерполирует полином $F(x)$=$f_0(x)$+$p_1(x)$+$p_2(x)$+...+$p_n(x)$ с использованием алгоритма с исправлением ошибок Берлекампа-Велча. Секрет будет равен s=$F(0)$=$f(0)$.

Заметим, что если дилер **Д** честен, в конце протокола **ВсПр** противник, даже зная секрет s и t долей «подкупленных» процессоров, ничего не узнает о долях секрета несбоящих процессоров, так как полином имеет степень t+1, а ему необходимо для интерполяции t +2 точки.

3.5.2. Доказательство безопасности схемы проверяемого разделения секрета

Теорема 3.1. Схема **ПРСК** является (n/3-1)-устойчивой.

Доказательство. Пусть

$g((w_1,...,w_{n-1},s\circ r),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s_1,...,s_{n-1},w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$

и

$h((s_1,...,s_{n-1},w_n),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s,s,...,s,w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$,

где $s_i=f_0(i)$, $f_0(x)=s+a_1x+...+a_{t+1}x^{t+1}$ и $r=a_1\circ....\circ a_d$ (то есть полином, созданный, с использованием случайного параметра r дилера **Д**).

При рассмотрении протокола **РзПр** напомним, что t_i – трафик процессора P_i. Ясно, что для всех процессоров P_i, $i\leq n$, функция входа всегда возвращает пустую строку $I_i(t_i)=\varepsilon$, так как процессоры не вносят никакие входы в процессе вычисления функции g. Для дилера **Д**, **Д**=P_{n+1}, функция входа немного сложнее. Пусть m_i - сообщение, которое дилер распространяет процессору P_i в 5-м раунде, если P_i сообщил об ошибке в 4-м раунде или сообщение, которое дилер послал процессору P_i в 1-м раунде, если P_i не заявлял об ошибке. Тогда $I_D(t_D)=f(0)$, где $f=BW(m_1,...,m_n)$ – полином степени t+1, следующий из интерполяции Берлекампа-Велча. Функция выхода более простая: $O_i(t_i)=m_i$, где $m_D=\varepsilon$.

При рассмотрении протокола **ВсПр**, определим вход и выход функции g. Функция входа I_i для процессора P_i определена как следующая: пусть $m_{i,j}$ – сообщение, посланное процессором P_i процессору P_j в 1-м раунде; $I_i(t_i)=h_i(0)$, где $h_i=BW(m_{i,1},...,m_{i,n})$ - многочлен степени t+1, следующий из интерполяции Берлекампа-Велча. Если такого полинома не существует, то $I_i(t_i)=0$. Функция выхода следующая: пусть M_i – сообщение, распространяемая процессором P_i в раунде 9; $O_i(t_i)=F(0)=s$, где $F=BW(M_1,...,M_n)$ – полином степени t+1, следующий из интерполяции Берлекампа-Велча.

Далее для доказательства теоремы необходимо доказать выполнения условий полноты, корректности и конфиденциальности.

Полнота. Если дилер **Д** - честный, исходя из свойств схемы **ПРСК**, любой несбоящий процессор может восстановить секрет s с вероятностью 1, так как посредством определенных выше функций g и h

$g((w_1,...,w_{n-1},s\circ r),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s_1,...,s_{n-1},w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$,

$h((s_1,...,s_{n-1},w_n),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s,s,...,s,w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$

реализуется

$f(w_1,...,w_{n-1},s)=((s,s,...,s,w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$.

Корректность. Для доказательства теоремы необходимо доказать следующие две леммы.

Лемма 3.1. Пусть функция g имеет вид

$g((w_1,...,w_{n-1},s\circ r),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s_1,...,s_{n-1},w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$.

Тогда g корректно вычисляет доли секрета $s_1,...,s_{n-1}$.

Доказательство. Сначала мы должны доказать, что для всех несбоящих процессоров P_i, значение $I_i(t_i)$ равно корректному входу. Если дилер **Д -** честный, то $m_i=f(i)$, где f - многочлен степени $t+1$ со свободным членом s (секретом). Таким образом, $I_D(t_D)=s$, если дилер честный. Второе условие корректности состоит в том, что с высокой вероятностью должно выполняться $O(t)=g(I(t))$. В нашем случае это означает, что с высокой вероятностью значения m_i, находящиеся у несбоящих процессоров, должны предназначаться для единственного полинома степени $t+1$. Это справедливо с вероятностью $\geq 2^{-\frac{2K}{3}}$, где не менее $\frac{2K}{3}$ битов выбраны действительно случайно несбоящими процессорами в раундах 2 и 6. Каждый бит представляет запрос, на который нечестный дилер, распределивший «плохие» доли, должен будет ответить правильно в следующем раунде только с вероятностью 1/2 (то есть, если он предскажет правильно значение бита). Следовательно, это и есть граница для вероятности ошибки. ■

Лемма 3.2. Пусть функция h имеет вид

$h((s_1,...,s_{n-1}),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s,s,...,s,w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$.

Тогда h корректно восстанавливает секрет s.

Доказательство. Понятно, что для всех несбоящих процессоров $I_i(t_i)=s_i$ – корректная доля входа. В этом случае необходимо проверить, что с высокой вероятностью $O(t)=h(I,t)$, а это означает, что необходимо доказать, что

$h((s_1,...,s_{n-1}),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s,s,...,s,\varepsilon),(t_1^{(r_k)},...,t_n^{(r_k)}))$.

Это равенство не выполняется, если:

- любой сбоящий процессор P_l «преуспевает» в получении случайного «мусора» вместо значений $p_l(j)$ в раунде 2 (в этом случае, сообщения M_i не будут интерполировать полином);

- процессор P_i распределяет $p_l(j)$ в раунде 2 и использует полином со свободным членом, отличным от нуля (в этом случае, M_i восстановит другой секрет).

Так как мы уже знаем, что P_l «преуспевает» в любом из двух описанных случаев с вероятностью $2^{-\frac{2K}{3}}$, то, следовательно, имеется не более, чем $n/3$ сбоящих процессоров и вероятность того, что протокол вычисляет неправильный выход не более, чем $n/3(2^{-\frac{2K}{3}})$, что для достаточного большого K, является экспоненциально малым.

Конфиденциальность. Для доказательства теоремы необходимо доказать следующие две леммы.

Лемма 3.3. Пусть функция g имеет вид $g((w_1,...,w_{n-1},s\circ r),(t_1^{(r_0)},...,t_n^{(r_0)}))=$ $=((s_1,...,s_{n-1},w_n),(t_1^{(r_k)},...,t_n^{(r_k)}))$.

Тогда g конфиденциально вычислима в отношение долей секрета $s_1,...,s_{n-1}$.

Доказательство. Доказательство условия конфиденциальности для функции g заключается в описании работы моделирующего устройства $\boldsymbol{M}$, которое взаимодействует со сбоящими процессорами (в том числе с нечестным дилером) и создает «почти» такое же распределение вероятностей, которое возникает у сбоящих процессоров во время реального выполнения протокола **РзПр**.

Необходимо рассмотреть два случая.

Случай А: Дилер нечестен до начала 1-ого раунда. Моделирующее устройство будет следовать только командам процессоров с единственным исключением, что оно будет «передавать» их в какое-либо время противнику в случае «сговора». Так как процессоры не сотрудничают по любому входу, то это сводит моделирование к работе схемы проверяемого разделения секрета с нечестным дилером. Так что моделирование будет неразличимо с точки зрения противника.

Случай В: Дилер честен до начала 1-ого раунда. Моделирующее устройство в 1-м раунде будет создавать случайный «ложный» секрет s' и распределять его процессорам в соответствии с командами протокола с полиномом f'. Если дилер честен в течение всего протокола, тогда он будет выполняться с точки зрения противника как обычный протокол проверяемого разделения секрета с честным дилером. Если дилер нечестен после

1-ого раунда противник и моделирующее устройство получит из оракула истинный вход s дилера. В этом случае моделирующее устройство передает управление от дилера к противнику и меняет полином, используемый для разделения на новый полином f'' такой, что $f''(0)=s$ и $f''(i)=f'(i)$ для всех процессоров P_i, которые были «подкуплены» противником. Изменения моделирующего устройства в соответствии со случайным полиномом f_i, используемым для доказательств с нулевым разглашением (см, например, [Ка14,КУ4] и приложение) делает их совместимыми с любым широковещательным каналом. Моделирующее устройство может всегда сделать это, так как противник имеет не более t точек полинома степени t+1. Далее моделирующее устройство использует полином f'' для работы несбоящих процессоров, все еще находящихся под его управлением. Можно утверждать, что для противника эти вычисления не отличимы от реальных вычислений. Единственный момент, отличающийся от реальных вычислений, - это тот факт, что доли секрета, которые противник получает до того, как дилер становится нечестным, созданы с использованием другого полинома. Но благодаря свойствам полиномов – это не является проблемой для моделирующего устройства, в том случае, если дилер нечестен. ■

Лемма 3.4. Пусть функция h имеет вид

$$h((s_1,...,s_{n-1}),(t_1^{(r_0)},...,t_n^{(r_0)}))=((s,s,...,s,w_n),(t_1^{(r_k)},...,t_n^{(r_k)})).$$

Тогда h конфиденциально вычислима в отношение секрета s.

Доказательство. Работа моделирующего устройства $\boldsymbol{M}$ сводится к следующему.

Описание работы моделирующего устройства *M*

1. В раунде 1, $\boldsymbol{M}$ моделирует процессор P_i, выбирая случайный полином h'_i степени t+1 и посылает $h'_i(j)$ к P_j. В этом месте моделирующему устройству позволено получить из оракула выход функции, так что $\boldsymbol{M}$ будет изучать истинный секрет s. Если такой процессор является «подкупленным» противником **Прот** в конце этого раунда (или в следующих раундах), то и $\boldsymbol{M}$, и **Прот** узнают истинную долю s_l и $\boldsymbol{M}$ должен изменить полином h'_l в соответствии с тем, что $h'_l(0)=s_l$, но без изменения значения в точках, уже известных противнику. Моделирующее устройство $\boldsymbol{M}$ может всегда сделать это, потому что противник имеет не более t точек полинома степени t+1.

2-8. В течение раундов 2-8 моделирующее устройство полностью следует явно командам процессоров. Так как все что делают процессоры в этих раундах полностью случайно и нет влияния на их входы, ***M*** будет всегда способно создать неразличимые распределения.

10. Моделирующее устройство ***M*** выбирает полином g степени t+1 такой, что $g(0)=s$ и затем для каждого процессора P_i, устройство ***M*** распространяет $g(i)+p_i(i)+...+p_n(i)$, где p_j - полином, распределенный процессором P_j в течение раундов 2-8 моделирования. Интерполяция Рида-Соломона этих значений даст как результат секрет s. Если процессор P_l является сбоящим в конце этого раунда, тогда и ***M***, и **Прот** узнают из оракула истинную долю входа s_l. Если $s_l \neq g(l)$, тогда ***M*** только изменит значение p_l в точке l так, чтобы сделать полную сумму соответствующей такой широковещательной передаче.

Моделирование неотличимо от реального выполнения с точки зрения противника. Фактически, в раундах 2-8 все сообщения случайны и не связаны с входом, так что моделирующее устройство может легко играть роль несбоящих процессоров. В раунде 1 противник просматривает не более t долей реального входа несбоящих процессоров. В соответствии со свойствами схемы разделения секрета Шамира, эти доли полностью случайны и, таким образом, могут моделироваться даже без знания реального входа (как и в случае с моделирующим устройством). В раунде 9 реальная доля распространена «скрытым образом» как случайный «мусор», а это позволит моделирующему устройству распространять сообщение несбоящих процессоров с необходимым распределением даже без знания реального входа. ■

С доказательством лемм 3.1 – 3.4 доказательство теоремы 3.1 следует считать законченным. ■

Здесь уместно сделать следующее замечание. Схемы (протоколы) конфиденциальных вычислений являются чрезвычайно сложным объектом исследований. Как видно из сказанного выше, даже описание и доказательство безопасности одного из базовых примитивов в протоколах конфиденциального вычисления функции, - схемы проверяемого разделения секрета являются чрезвычайно громоздкими и сложными.

3.6. Синхронные конфиденциальные вычисления

3.6.1. Примитив «Забывающий обмен»

Пусть k - фиксированное целое и пусть $b_1,b_2,...,b_k \in \{0,1\}$ и $i \in \{1,...,k\}$.

Тогда забывающий обмен $\mathbf{OT}^k{}_1$ определяется как функциональная зависимость:

$\mathbf{OT}^k{}_1((b_1,b_2,...,b_k),i)=(\lambda,b_i)$.

Эта вычислимая функция является в явном виде асимметричной и реализуется посредством протокола взаимодействия двух участников (процессоров). Обычно первый процессор, который содержит вход $(b_1,b_2,...,b_k)$, называется *отправителем* (или **S**), а второй процессор, который содержит вход $i \in \{1,...,k\}$, называется *получателем* (или **R**).

Интуитивно, цель забывающего обмена состоит в том, чтобы для отправителя обменяться i-тым битом с получателем так, чтобы не позволить последнему получить какую-либо информацию относительно значения любого другого своего бита и так, чтобы не позволить отправителю получить какую-либо информацию о бите, который был затребован получателем.

С использованием любой *перестановки с секретом* $\{f_i\}_{i \in I}$ (см. Приложение), ниже описывается протокол для конфиденциально вычисления $\mathbf{OT}^k{}_1$. Так как мы имеем дело с конечными вычислимыми функциями, безопасность будет рассматриваться в терминах некоторого дополнительного параметра n, именуемого далее *параметром безопасности*.

Протокол $\mathbf{OT}_{\text{ПЧМ}}$

Вход: Отправитель **R** имеет вход $b_1,b_2,...,b_k \in \{0,1\}^k$, а получатель **S** имеет вход $i \in \{1,...,k\}$ и обе стороны имеют дополнительный параметр безопасности 1^n.

1. Отправитель **S** равномерно выбирает пару *односторонних функций с секретом* (α,t) (см. Приложение), выполняя алгоритм их генерации G по входу 1^n. Параметр α отсылается **R**.

2. Получатель **R** равномерно и независимо выбирает $e_1,...,e_k \in_{R*} D_\alpha$, устанавливает $y_i=f(e_i)$ и $y_j=e_j$ для каждого $j \neq i$ и отсылает $(y_1,y_2,...,y_k)$ к отправителю **S**. Для этого:

2.1. Равномерно и независимо выбирает $e_j \in_{R^*} D_\alpha$ каждый раз, вызывая алгоритм генерации $e_j = D(\alpha, r_j)$, $r_j \in_{R^*} Z$, $j=1,...,k$.

2.2. Вычисляет $y_i = f(e_i)$.

2.3. Для $j \neq i$ присваивает $y_j = e_j$.

2.4. Получатель **R** отсылает $(y_1, y_2, ..., y_k)$ к отправителю **S**. Таким образом, получатель знает $f_\alpha^{-1}(y_i) = e_i$, однако не может предсказать $b(f_\alpha^{-1}(y_i))$ для $j \neq i$.

3. После получения $(y_1, y_2, ..., y_k)$, используя знание секрета для инвертирования односторонней функции с секретом, отправитель **S** вычисляет $x_j = f_\alpha^{-1}(y_j)$ для $j \in \{1,...,k\}$. В свою очередь, **S** посылает $(b_1 \oplus b(x_1), b_2 \oplus b(x_2), ..., b_k \oplus b(x_k))$ получателю **R**.

4. По получении $(c_1, c_2, ..., c_k)$ получатель локально выдает $c_i \oplus b(e_i)$.

Отметим сначала, что вышеупомянутый протокол корректно вычисляет $\mathbf{OT}^k{}_1$ так как

$c_i \oplus b(e_i) = b_i \oplus b(x_i)) \oplus b(e_i) = b_i \oplus b(f_\alpha^{-1}(f_\alpha(e_i))) \oplus b(e_i) = b_i$.

Аргументы для доказательства того, что протокол действительно конфиденциально вычисляет $\mathbf{OT}^k{}_1$ следующие. Интуитивно, отправитель **S** не получает никакой информации при выполнении протокола, так как для любого возможного значения i все потенциальные отправители «видят» одно и то же распределение равномерно и независимо выбранных элементов из D_α.

Интуитивно, получатель **R** не получает никакой (с вычислительной точки зрения) информации при выполнении протокола, так как для $j \neq i$, единственные данные, которые могут «говорить» о b_j – это кортеж $(\alpha, e_j, b_j \oplus (f_\alpha^{-1}(e_j)))$, из которого невозможно предсказать b_j лучше, чем простым угадыванием.

Формальные аргументы даны в работе [Go1].

3.6.2. Двухсторонние вычисления

Безопасные протоколы для получестной модели

В этом подразделе описывается метод построения протоколов конфиденциального вычисления любой заданной вычислимой функции. Конструкция представляет собой булеву схему для реализации заданной функции, вычисления в которой выполняются в соответствии с протоколом. Конструкция носит модульный характер.

Сначала определяется соответствующее понятие *редукции* и показывается, как получить протокол для конфиденциального вычисления функции g по данной редукции (конфиденциально вычислимой) функции g к (конфиденциально вычислимой) функции f вместе с протоколом для конфиденциального вычисления f. В частности мы сводим конфиденциальное вычисление обобщенных вычислимых функций к конфиденциальному вычислению детерминированных вычислимых функций.

Затем, без потери общности, рассматриваются булевы схемы с логическими вентилями **and** и **xor** с коэффициентом объединения по входу равным 2. В общем случае можно представить следующий сценарий. Первый процессор «содержит» параметры a_1,b_1, а второй процессор - a_2,b_2, где a_1+a_2 - значение одной входной линии и b_1+b_2 - значение другой входной линии. Необходимо обеспечить каждый из процессоров «случайной» долей значения выходной линии, то есть долей значения $(a_1+a_2)\cdot(b_1+b_2)$. Другими словами нас интересует конфиденциальное вычисление следующей рандомизированной вычислимой функции:

$$((a_1,b_1),(a_2,b_2))\to(c_1,c_2), \qquad (3.1)$$

где $c_1+c_2=(a_1+a_2)\cdot(b_1+b_2)$. (3.2)

То есть значения (c_1,c_2) равномерно выбраны среди всех решений $c_1+c_2=(a_1+a_2)\cdot(b_1+b_2)$. Вышеупомянутая вычислимая функция имеет конечную область значений и может быть вычислена (в общем случае) сведением к одному из вариантов забывающего обмена (**OT**). Ниже показывается, как это может быть сделано при условии существования односторонней перестановки с секретом (см. Приложение).

Конфиденциальное вычисление c_1+c_2

Теперь мы непосредственно обращаемся к вычислимой функции, определенной в равенствах (3.1)-(3.2). Все арифметические операции выполняются в поле $GF(2)$. Ниже приводится алгоритм редукции (конфиденциальным образом) этой вычислимой функции к вычислению $\mathbf{OT}^4{}_1$.

Редукция к $\mathbf{OT}^4{}_1$

Вход. Процессор i содержит $(a_i,b_i)\in\{0,1\}\times\{0,1\}$, $i=1,2$.

1. Первый процессор равномерно выбирает $c_1\in_R\{0,1\}$.

2. Оба процессора вызывают субпротокол $\mathbf{OT}^4{}_1$, где первый процессор играет роль отправителя **S**, а второй процессор играет роль получателя **R**.

Тогда вход для отправителя – это 4-элементный кортеж $(c_1+a_1 \cdot b_1, c_1+a_1 \cdot (b_1+1), c_1+(a_1+1) \cdot b_1, c_1+(a_1+1) \cdot (b_1+1))$, а вход получателя - $1+2a_2+b_2 \in \{1,2,3,4\}$.

Выход. Первый процессор выдает c_1, в то время как второй процессор выдает результат, полученный при вызове $\mathbf{OT}^4{}_1$.

В столбцах таблицы 3.1 приведены значения выходов процессора 2 как функции от значений его собственных входов и входы и выходы процессора 1 (то есть, a_1,b_1,c_1). Значения, с которыми процессор 2 инициализирует субпротокол **OT** (то есть, $1+2a_2+b_2$) показаны во второй строке и значения выходов (и **OT**, и всего протокола) показаны в третьей строке. Отметим, что в каждом случае, выход процессора 2 равняется $c_1+(a_1+a_2) \cdot (b_1+b_2)$.

Таблица 3.1

Значения (a_2,b_2)	(0,0)	(0,1)	(1,0)	(1,1)
***Выход* OT**	1	2	3	4
Значения выхода	$c_1+a_1 \cdot b_1$	$c_1+a_1 \cdot (b_1+1)$	$c_1+(a_1+1) \cdot b_1$	$c_1+(a_1+1) \cdot (b_1+1)$

Вначале отметим, что вышеупомянутая редукция корректна, так как выход процессора 2 равняется $c_1+(a_1+a_2) \cdot (b_1+b_2)$. Это следует из анализа вышеприведенной таблицы истинности, которая описывает значения выхода процессора 2, как функции от ее собственных входов a_1,b_1,c_1. Необходимо подчеркнуть, что выходная пара (c_1,c_2) равномерно распределена среди всех пар, для которых $c_1+c_2=(a_1+a_2) \cdot (b_1+b_2)$.

Таким образом, каждый из локальных выходов (т.е., либо процессора 1, либо процессора 2) равномерно распределен, хотя оба локальных выхода зависимы друг от друга (как в уравнении (3.2). Таким образом, легко увидеть, что редукция конфиденциально вычислима. Формальные аргументы приведены в [Go2].

Протокол вычислений на арифметической схеме над GF(2)

Теперь мы покажем, что процесс вычисления любой детерминированной вычислимой функции, которая вычисляется

посредством арифметической схемы над $GF(2)$, конфиденциально сводим к вычислению функции в соответствии с уравнениями (3.1) - (3.2). Напомним, что последняя вычислимая функция соответствует частному вычислению на мультипликативных вентилях для входов, общедоступных обоим процессорам. Таким образом, можно констатировать, что эта вычислимая функция может рассматриваться как эмулирование мультипликативного вентиля, как это описано выше. В частности разделение битового значения v между обоими процессорами означает равномерно выбранную пару битов (v_1,v_2) так, что $v=v_1+v_2$, где первый процессор содержит v_1, второй - v_2. Наша цель заключается в разделении, через частные вычисления, долей входных линий схемы на доли всех линий схемы так, чтобы, в конце концов, мы получили бы доли выходных линий схемы.

В начале мы рассмотрим нумерацию всех линий схемы. Входные линии схемы (n) для каждого процессора будут пронумерованы $1,2,...,2n$ так, что для $j=1,...,n$ j-тый вход процессора i соответствует $((i\text{-}1)n+j)$-той линии. Линии будут пронумерованы так, чтобы выходные линии каждого вентиля имеют большую нумерацию, чем его входные линии. Для простоты предположим, что каждый процессор получает n выходных битов, и что выходные биты второго процессора соответствуют последним n линиям схемы.

Ниже приводится алгоритм сведения любой детерминированной вычислимой функции к вычислениям на мультипликативном вентиле.

Редукция к МВ

Вход. Процессор i содержит строку битов $x_i^1...x_i^n \in \{0,1\}^n$, $i=1,2$.

1. *Разделение входов*. Каждый процессор делит каждый из своих входных битов с другим процессором. То есть для каждого $i=1,2$ и $j=1,...,n$ процессор i равномерно выбирает бит r_i^j и посылает его другому процессору, как долю входной линии (с нумерацией $((i\text{-}1)n+j)$ последнего. Процессор i устанавливает свою собственную долю на входной линии с нумерацией $(i\text{-}1)n+j$ в значение $x_i^j+r_i^j$.

2. *Эмуляция схемы*. Следуя нумерации линий, процессоры используют свои доли двух входных линий схемы, чтобы конфиденциально вычислить доли для выходной линии вентиля.

Предположим, что процессоры имеют общие две входные линии вентиля, то есть процессор 1 содержит доли a_1,b_1, а

процессор 2 содержит доли a_2,b_2, где a_1,a_2 - доли на первой линии и b_1,b_2 - доли на второй линии. Рассмотрим два случая.

2.1. *Эмуляция аддитивного вентиля.* Процессор 1 только устанавливают долю выходной линии вентиля в значение a_1+b_1, а процессор 2 устанавливают долю выходной линии вентиля в значение a_2+b_2.

2.2. *Эмуляция мультипликативного вентиля.* Доли выходной линии вентиля получаются посредством вызова оракула для вычисления функции (см.(3.1)-(3.2)), где процессор 1 обеспечивает вход (часть запроса) (a_1,b_1), а процессор 2 обеспечивает вход (a_2,b_2). После ответов оракула, каждый из процессоров устанавливают свою долю выходной линии вентиля в значение равное своей части ответа оракула.

3. *Восстановление выходных битов.* Как только доли выходных линий схемы вычислены, каждый процессор посылает долю с каждой из таких линий процессору, с которым линия связана. То есть доли на последних n линиях посылаются процессором 1 процессору 2, в то время как доли n предшествующих линий посылаются процессором 2 процессору 1. Каждый из процессоров восстанавливает соответствующие выходные биты, складывая две доли; то есть доли, которую он получил на шаге 2 и доли, полученной на текущем шаге.

Выход. Каждый процессор локально выдает биты, восстановленные на шаге 3.

Сначала необходимо проверить, что выходы действительно корректны. Это можно сделать методом индукции, которая состоит в том, что значение доли каждой линии прибавляется к корректному значению на линии. Базис индукции – номер входной линии схемы. А, именно, $((i\text{-}1)n+j)$-тая линия имеет значение x_i^j и ее доли - r_i^j и $r_i^j+x_i^j$. На каждом шаге индукции рассматривается эмуляция аддитивного или мультипликативного вентиля. Предположим, что значения входных линий – a и b и что их доли a_1,a_2 и b_1,b_2, которые удовлетворяют $a_1+a_2=a$ и $b_1+b_2=b$. В случае аддитивного вентиля доли на выходной линии устанавливаются в значения a_1+b_1 и a_2+b_2, что удовлетворяет $(a_1+b_1)+(a_2+b_2)=(a_1+a_2)+(b_1+b_2)=a+b$. В случае мультипликативного вентиля доли выходной линии были устанавливаются в значения c_1 и c_2 так, что $c_1+c_2=(a_1+a_2)\cdot(b_1+b_2)$. Таким образом, $c_1+c_2=a\cdot b$, что и требовалось показать.

В работе [Go2] приводятся формальные аргументы для конфиденциального сведения вычисления на арифметической схеме над $GF(2)$ к функции, вычислимой в соответствии с уравнениями (3.1)-(3.2). А следующая теорема, устанавливает главный результат для конфиденциального вычисления любой вычислимой функции для случая двухстороннего протокола взаимодействия.

Теорема 3.2. Предположим, что односторонние перестановки с секретом существуют. Тогда любая вычислимая функция для получестной модели конфиденциально вычислима.

Основной результат для злонамеренной модели

Следующая теорема устанавливает основной результат для случая двухстороннего протокола для злонамеренной модели.

Теорема 3.3. Предположим, что односторонние перестановки с секретом существуют. Тогда любая вычислимая функция для двухстороннего процесса взаимодействия и злонамеренной модели противника конфиденциально вычислима.

Эта теорема устанавливает возможность компиляции любого протокола для получестной модели в «эквивалентный» протокол для злонамеренной модели. Существование таких компиляторов показано в ряде работ (см., например, [Ca1,Ca2,Go1]. Сам процесс построения компиляторов для целей, необходимых для построения наших протоколов, является достаточно сложным, и использует в качестве инструментальных средств («крупных примитивов») такие схемы как: схемы привязки, системы доказательств с нулевым разглашением для **NP**-утверждений, для **NP**-свидетельств и др. [Go1].

3.6.3. Многосторонние протоколы

Общая идея

Также как и для случая двухсторонних протоколов, основная наша цель состоит в том, чтобы разработать протоколы, которые могут противостоять любому возможному поведению противника в случае многостороннего взаимодействия. Это также делается в два этапа. Как и в предыдущем случае, сначала рассматривается получестный противник (в роли которого выступают получестные процессоры) и разрабатываются протоколы, которые являются безопасными относительно такого противника. Определение этого типа противника такое же, как в случае с двумя сторонами. Однако в случае многосторонних протоколов при изучении поведения противника рассматриваются две модели. Первая

модель злонамеренного поведения подобна злонамеренной модели для противников в случае с двумя процессорами. Это модель позволяет противнику контролировать большую часть процессоров, однако она не рассматривает неизбежно возникающие нарушения безопасности (так как «внешне» процессоры «ведут себя» нормально). Во второй модели злонамеренного поведения предполагается, что противник может контролировать только ограниченную часть сторон. В этой модели, которая была бы пустой в случае с двумя сторонами, нарушения безопасности могут быть эффективно предотвращены. В данном подразделе будет показано, как преобразовать протоколы, безопасные в получестной модели в протоколы, безопасные в каждой из двух моделей со злонамеренным поведением противника.

www.kiev-security.org.ua
BEST rus DOC FOR FULL SECURITY

Конструкции в целом получаются посредством внесения соответствующих изменений в двухсторонние протоколы. Фактически, построение многосторонних протоколов для получестной модели как, впрочем, и построение компилятора для злонамеренной модели первого типа – это незначительное изменение аналогичных конструкций для двухпроцессорного взаимодействия. В компиляции протоколов для получестной модели в протоколы для второй злонамеренной модели используется схема проверяемого разделения секрета, которая необходима для «эффективного предотвращения» прерывания работы протокола меньшей частью процессоров. Для этого фактически, необходимо только преобразовать протоколы, безопасные в первой злонамеренной модели, в протоколы, безопасные во второй злонамеренной модели.

Получестная модель

Построение многосторонних протоколов (безопасных против получестного поведения) конфиденциального вычисления для любой данной вычислимой функции от нескольких аргументов основывается на соответствующем протоколе для случая с двумя процессорами. Для простоты, зафиксируем число процессоров - m. (В общем случае для предлагаемых конструкций значение m может быть как функция полинома).

Рассмотрим схему с вычислениями над $GF(2)$ для оценки значения m-арной вычислимой функции f. Сначала каждый из процессоров

объединяет свои входные биты со всеми другими процессорами так, чтобы сумма всех долей равнялась некоторому входному биту.

Следуя от входных линий к выходным линиям, мы выполняем конфиденциальное вычисление доли на каждой линии схемы так, чтобы сумма долей равнялась бы корректному значению. Перед нами стоит только одна проблема. При вычислениях на мультипликативном вентиле, мы имеем процессор i, имеющий биты a_i и b_i и нам необходимо выполнить конфиденциальное вычисление так, чтобы этот процессор закончил работу со случайным битом c_i и выполнялось бы: $(\sum_{i=1}^{m} a_i) \cdot (\sum_{i=1}^{m} b_i) = (\sum_{i=1}^{m} c_i)$.

Более точно, нас интересует конфиденциальное вычисление следующих рандомизированной вычислимой m-арной функции

$$((a_1,b_1),...,(a_m,b_m)) \rightarrow (c_1,...,c_m) \in_{R^*} \{0,1\}^m, \tag{3.3}$$

и

$$(\sum_{i=1}^{m} a_i) \cdot (\sum_{i=1}^{m} b_i) = (\sum_{i=1}^{m} c_i). \tag{3.4}$$

Таким образом, необходимо конфиденциально вычислить посредством m-стороннего протокола вышеупомянутую вычислимую функцию. Это делается посредством конфиденциального сведения (для независимого m) вычисления уравнений (3.3)-(3.4) к вычислению тех же самых функциональных зависимостей для случая m=2, которые, в свою очередь, соответствуют уравнениям (3.1)-(3.2).

Конфиденциальное вычисление $(\sum_{i=1}^{m} c_i) = (\sum_{i=1}^{m} a_i) \cdot (\sum_{i=1}^{m} b_i)$

Необходимо отметить, что арифметика над $GF(2)$ реализует также, что –1=+1.

Имеют место следующие соотношения:

$$(\sum_{i=1}^{m} a_i) \cdot (\sum_{i=1}^{m} b_i) = \sum_{i=1}^{m} a_i b_i + \sum_{1 \le i < j \le m} (a_i b_j + a_j b_i) =$$

$$=(m\text{-}2) \cdot \sum_{i=1}^{m} a_i b_i + \sum_{1 \le i < j \le m} (a_i + a_j)(b_i + b_j)$$

Из этого соотношения можно увидеть, что каждый процессор может сам вычислить элемент $(m\text{-}2)a_i b_i$, в то время как каждые два процессора из двухэлементного подмножества $\{i,j\}$ могут конфиденциально вычислить

доли по элементам $(a_i+a_j)\cdot(b_i+b_j)$ (в соответствии с теоремой 3.2). Отсюда следует алгоритм сведения вычисления на основании (3.3) и (3.4) m-арной функции к вычислению на основании (3.1) и (3.2) функции для двух аргументов.

Редукция к КВ$^{m=2}$

Вход. Процессор i содержит $(a_i,b_i)\in\{0,1\}\times\{0,1\}$, $i=1,...,m$.

1. *Редукция*. Каждая пара процессоров (i,j), где $i<j$, вызывает 2-арную функцию (см. (3.1)-(3.2)). Процессор i обеспечивает входную пару (a_i,b_i), в то время как процессор j обеспечивает - (a_j,b_j). Кроме того, определим ответ оракула процессору i как $c_j^{\{i,j\}}$.

2. Процессор i устанавливает $c_i=(m-2)a_ib_i+\sum_{j\neq i} c_i^{\{i,j\}}$.

3. Процессор i выдает c_i.

Вышеприведенное сведение является корректным, так как выходы всех процессоров складываются в соответствии с алгоритмом. Конфиденциальность сведения определяется следующим предложением.

Предложение 3.1. Алгоритм «Редукция к КВ$^{m=2}$» конфиденциально сводит вычисления на основании (3.3) и (3.4) m-арной функции к вычислению на основании (3.1) и (3.2) функции для двух аргументов.

В работе [Go2] приводятся формальные аргументы для доказательства корректности и конфиденциальности такой редукции. А следующая теорема, устанавливает главный результат для конфиденциального вычисления любой вычислимой функции для случая многостороннего протокола взаимодействия.

Теорема 3.4. Предположим, что односторонние перестановки с секретом существуют. Тогда любая вычислимая m-арная функция для получестной модели конфиденциально вычислима.

Многосторонний протокол схемного вычисления

Ниже описывается m-арный аналог вышеописанного субпротокола «Редукция КВ$^{m=2}$». Показывается, что вычисления любой детерминированной функции, вычислимой посредством арифметической схемы над $GF(2)$, конфиденциально сводимы к вычислениям функции из уравнений (3.3) - (3.4).

В частности разделение битового значения v между m процессорами означает равномерно выбранную m-арную последовательность битов

$(v_1,...,v_m)$ так, что $v=\sum_{i=1}^{m} v_i$, где i-тый процессор содержит v_i. Наша цель заключается в разделении, через частные вычисления, долей входных линий схемы на доли всех линий схемы так, чтобы, в конце концов, мы получили бы доли выходных линий схемы.

В начале мы рассмотрим нумерацию всех линий схемы. Входные линии схемы (n) для каждого процессора будут пронумерованы $1,2,...,mn$ так, что для $j=1,...,n$ j-тый вход процессора i соответствует $((i-1)n+j)$-той линии. Линии будут пронумерованы так, чтобы выходные линии каждого вентиля имеют большую нумерацию, чем его входные линии. Для простоты предположим, что каждый процессор получает n выходных битов, и что j-тый выходной бит j-того процессора соответствуют линии $N-(m+1-i)+j$, где N – размер схемы.

Ниже приводится алгоритм сведения любой детерминированной m-арной вычислимой функции ($m\geq 2$) к функции, вычислимой в соответствии с уравнениями (3.3) и (3.4).

Редукция к КВ^m

Вход. Процессор i содержит строку битов $x_i^1...x_i^m \in \{0,1\}^m$, $i=1,...,m$.

1. *Разделение входов*. Каждый процессор делит каждый из своих входных битов с другим процессором. То есть для каждого $i=1,...,m$ и $j=1,...,n$ и каждого $k\neq i$ процессор i равномерно выбирает бит $r_k^{(i-1)n+j}$ и посылает его процессору k, как долю входной линии (с нумерацией $(i-1)n+j$) последнего. Процессор i устанавливает свою собственную долю на входной линии с нумерацией $(i-1)n+j$ в значение $x_i^j+\sum_{k\neq i} r^{(i-1)n+j}$.

2. *Эмуляция схемы*. Следуя нумерации линий, процессоры используют свои доли двух входных линий схемы, чтобы конфиденциально вычислить доли для выходной линии вентиля.

Предположим, что процессоры имеют общие две входные линии вентиля, то есть для $i=1,...,m$ процессор i содержит доли a_i,b_i, где $a_1,...,a_m$ - доли на первой линии и $b_1,...,b_m$ - доли на второй линии. Рассмотрим два случая.

2.1. *Эмуляция аддитивного вентиля.* Каждый процессор только устанавливают свою долю выходной линии вентиля в значение a_i+b_i.

2.2. *Эмуляция мультипликативного вентиля*. Доли выходной линии вентиля получаются посредством вызова оракула для вычисления функции (см.(3.4)-(3.5), где процессор i обеспечивает вход (часть запроса) (a_i,b_i). После ответов оракула, каждый из процессоров устанавливают свою долю выходной линии вентиля в значение равное своей части ответа оракула.

3. *Восстановление выходных битов*. Как только доли выходных линий схемы вычислены, каждый процессор посылает долю с каждой из таких линий процессору, с которым линия связана. То есть для i=1,...,m и j=1,...,n каждый процессор посылает свою долю с линии N-(m+1-i)n+j процессору i. Каждый из процессоров восстанавливает соответствующие выходные биты, складывая соответствующие m долей; то есть доли, которую он получил на шаге 2 и m-1 долей, полученных на текущем шаге.

Выход. Каждый процессор локально выдает биты, восстановленные на шаге 3.

Сначала необходимо проверить, что выходы действительно корректны. Это можно сделать методом индукции, которая состоит в том, что значение доли каждой линии прибавляются к корректному значению на линии. Базис индукции – номер входной линии схемы. А именно, ((i-1)n+j)-тая линия имеет значение x_i^j и ее доли прибавляются x_i^j. На каждом шаге индукции рассматривается эмуляция аддитивного или мультипликативного вентиля. Предположим, что значения входных линий – a и b и что их доли $a_1,...,a_m$ и $b_1,...,b_m$ действительно удовлетворяют $\sum_{i=1}^{m} a_i = a$ и $\sum_{i=1}^{m} b_i = b$. В случае аддитивного вентиля доли на выходной линии установить в значения $a_1+b_1,...,a_m+b_m$, что удовлетворяет $\sum_{i=1}^{m}(a_i+b_i) = (\sum_{i=1}^{m} a_i) + (\sum_{i=1}^{m} b_i) = a+b$. В случае мультипликативного вентиля доли выходной линии были установлены в значение $c_1,...,c_m$ так, что $\sum_{i=1}^{m} c_i = (\sum_{i=1}^{m} a_i) + (\sum_{i=1}^{m} b_i)$. Таким образом, $\sum_{i=1}^{m} c_i = a \cdot b$, что и требовалось показать.

В работе [Go2] приводятся формальные аргументы для конфиденциального сведения вычисления на арифметической схеме над $GF(2)$ к

m-арной функции, вычислимой в соответствии с уравнениями (3.3)-(3.4). А следующая теорема, устанавливает главный результат для конфиденциального вычисления любой вычислимой функции для многостороннего протокола взаимодействия.

Теорема 3.5. Предположим, что односторонние перестановки с секретом существуют. Тогда любая вычислимая m-арная функция для получестной модели и m-стороннего протокола взаимодействия конфиденциально вычислима.

Основной результат для злонамеренной модели

Следующая теорема устанавливает основной результат для случая двухстороннего протокола и для злонамеренной модели.

Теорема 3.6. Предположим, что односторонние перестановки с секретом существуют. Тогда любая вычислимая функция для m-стороннего процесса взаимодействия и злонамеренной модели противника конфиденциально вычислима.

Эта теорема устанавливает возможность компиляции любого протокола для получестной модели в «эквивалентный» протокол для двух злонамеренных моделей, обсуждаемых выше. Существование таких компиляторов показано в работе [Go2]. Построение компилятора для первой модели аналогично построению компилятора для злонамеренной модели при двухстороннем взаимодействии. После получения такого компилятора, строится компилятор [Go2] для преобразования любого протокола для первой злонамеренной модели в безопасный протокол для второй злонамеренной модели.

3.7. Асинхронные конфиденциальные вычисления

3.7.1. Вводные замечания

В данном подразделе сначала приводятся необходимые примитивы и схема асинхронного проверяемого разделения секрета **АПРС**, а затем приводятся схема вычислений на мультипликативном вентиле для разных типов сбоев как схема конфиденциальных вычислений.

3.7.2. Примитив «Соглашение об аккумулируемом множестве» (СОАМ-субпротокол)

Предположим, что каждый процессор инициализирует Br-субпротокол с некоторым входным значением. Процессоры желают договориться об общем аккумулируемом множестве с не менее n-t процессорами, которые успешно завершили Br-субпротокол. Понятно, что

каждый процессор может пожелать дождаться n-t локальных завершений Br-субпротокола и сохранить «этих отправителей сообщений». Однако если более чем n-t завершений Br-субпротокола глобально осуществлены, тогда мы не можем быть уверены в том, что все несбоящие процессоры сохранят то же самое множество. Для этого необходимо всем процессорам выполнить **COAM**-субпротокол, в котором выход несбоящих процессоров этого субпротокола есть общее множество из не менее n-t процессоров, которые завершили Br-субпротокол.

Неформально говоря, выходы процессоров после выполнения **COAM**-субпротокола это множества переменных, которые аккумулируют процессоры во время выполнения субпротокола, то есть элементы множества постоянно добавляются в него по мере выполнения субпротокола. Такой тип входа будет называться *динамическим входом* и, такое множество будет называться *аккумулируемым множеством переменных*.

Определение 3.1. Пусть $m, M \in N$ (в данном контексте $m=n-t$ и $M=n$) и пусть $\boldsymbol{U}_1 ... \boldsymbol{U}_n \subseteq [M]$ – коллекция аккумулируемых множеств такая, что процессор P_i имеет $\boldsymbol{U}_i$. Будем говорить, что коллекция является (m,t)-*однородной*, если для каждого планировщика и каждой коалиции из не более чем t сбоящих процессоров выполняются следующие условия:

- каждый несбоящий процессор P_i будет обязательно иметь $|\boldsymbol{U}_i| \geq m$;
- каждых два несбоящие процессора P_i и P_j будут обязательно иметь $\boldsymbol{U}_i = \boldsymbol{U}_j$.

Определение 3.2. Пусть $m, M \in N$ и пусть $\boldsymbol{P}$ - протокол, где вход каждого процессора P_i есть аккумулируемое множество $\boldsymbol{U}_i$. Протокол $\boldsymbol{P}$ называется *t-устойчивым **COAM**-субпротоколом* для n процессоров (с параметрами n, M), если для каждого планировщика и каждой коалиции из не более чем t сбоящих процессоров выполняются следующие условия:

Условие завершения. Если коллекция $\boldsymbol{U}_1 ... \boldsymbol{U}_n$ (m,t)-однородна, тогда с вероятностью 1 все несбоящие процессоры обязательно завершат протокол.

Условие корректности. Все несбоящие процессоры завершат протокол с общим выходом $C \subseteq [M]$ так, что $|C| \geq m$. Кроме того, каждый несбоящий процессор имеет $C \subseteq \boldsymbol{U}_i^*$, где $\boldsymbol{U}_i^*$ значение $\boldsymbol{U}_i$ при завершении протокола.

Схема «Соглашение об аккумулируемом множестве»

Пусть $n \geq 3t+1$. Субпротокол **COAM** состоит из 2-х этапов (с параметрами m и M). На первом этапе каждый процессор сначала ждет пока его динамический вход станет размером m. После чего, он выполняет $\log_2 n$ итераций. В каждой итерации процессор посылает текущее содержание его динамического входа всем другим процессорам, после чего собирает множества, посланные другими процессорами в текущей итерации. Как только накопятся n-t таких множеств в динамическом входе процессора, он приступает к следующей итерации. Это продолжается до тех пор, пока пересечение динамических входов всех несбоящих процессоров, которые получены во всех $\log_2 n$ итерациях станет размером не менее m.

На втором этапе процессоры последовательно запускают M раз BA-субпротокол. На j-том шаге процессоры решают, принадлежит ли элемент $j \in [M]$ согласованному множеству C. То есть вход каждого процессора на j-том шаге BA-субпротокола будет 1 тогда и только тогда, когда j принадлежит динамическому входу процессора. Элемент j принадлежит множеству C тогда и только тогда, когда общий выход j-того шага BA-субпротокола является 1. Свойства BA-субпротокола гарантируют нам, что множество C содержит пересечение динамических входов всех процессоров, которые получены при завершении первого этапа, и что каждый элемент $j \in C$ будет в динамическом входе каждого несбоящего процессора.

Протокол COAM

Процессор P_i выполняет следующие шаги по входу m, M и аккумулируемому множеству $\boldsymbol{U}_i$.

1. Ожидает пока $|\boldsymbol{U}_i| \geq m$.
2. Выполнить в цикле $r = \lfloor \log n \rfloor$ раз
   2.1. Послать $(r, \boldsymbol{U}_i)$ всем другим процессорам;
   2.2. Пусть $F_i^r = \{P_j$, если (r, S_j) получено от $P_j\}$. Ждет пока $S_j \subseteq \boldsymbol{U}_i$ для не менее чем n-t процессоров $P_j \in F_i^r$.
3. Выполняет M раз BA-субпротокол $BA_1 \ldots BA_M$, где вход 1 в j-том выполнении субпротокола, тогда и только тогда, когда $j \in U_i$.

4. Устанавливает C_i={j, если выход BA_j равен 1}. Ждет пока $C_i \subseteq U_i$.

5. Выдает C_i.

Предложение 3.2. Протокол **COAM** – является $\min(r,(\lceil n/3 \rceil\text{-}1))$-устойчивым протоколом для сети из n процессоров, где r – граница устойчивости для используемого протокола соглашений.

Доказательство. Сначала докажем условие завершения. Предположим, что аккумулируемые множества $U_1,...,U_n$ определяют (m,t)-однородную коллекцию. Покажем, что каждый несбоящий процессор P_i будет событийно завершать протокол.

Замечание. Напомним, что в асинхронных моделях выполнение каждой конкретной операции начинается сразу после приема сигнала об окончании предыдущей операции, т.е. после наступления некоторого события, не обязательно привязанного к работе синхронизатора. Такая операция (действие) здесь и далее будет называться *событийной* (*событийным*).

Шаг 1 будет завершен в любом случае. Индукция по числу итераций показывает, что шаг 2 будет также завершен. В каждой итерации r процессор P_i будет событийно получать сообщение (r,S_j) от каждого несбоящего процессора. Кроме того, для каждого процессора P_j процессор будет событийно иметь $S_j \subseteq U_i$ (так как система $U_1,...,U_n$ является однородной). Шаг 3 будет завершен с вероятностью 1, так как все протоколы византийских соглашений завершаются с вероятностью 1. Чтобы доказать, что шаг 4 будет также завершен, необходимо отметить, что если BA-субпротокол выдает 1, тогда существует несбоящий процессор P_k, который запускает BA_j-протокол, где $j \in U_k$ и, таким образом, процессор P_i будет событийно иметь $j \in U_i$.

Доказательство свойства корректности начинается со следующего. Согласование выходов процессоров непосредственно следует из определения византийских соглашений. Шаг 4 протокола гарантирует, что $C \subseteq U_i^*$ для каждого несбоящего процессора P_i.

В заключение необходимо показать, что $|C| \geq m$. Пусть U_i^r обозначает значение аккумулируемого множества U_i по окончании r-ой итерации на шаге 2 при функционировании процессора P_i. Покажем индукцией по r, что каждое множество D из не более чем 2^r несбоящих процессоров, которые завершили итерацию r, удовлетворяет $|\cap_{j \in D} U_j^r| \geq m$. Основание

индукции (r=0) следует непосредственно из шага 1 протокола. Для шага индукции рассмотрим множество D из не более чем 2^r несбоящих процессоров. Отметим, что для каждых двух процессоров $P_j,P_k \in D$ имеем $|F_j^r \cap F_k^r| \geq t+1$, где $n \geq 3t+1$ и $|F_j^r| \geq n-t$ для каждого j. Поэтому существует не менее одного несбоящего процессора $P_l \in F_j^r \cap F_k^r$. Процессор P_l будем называть *арбитром* P_j и P_k. Процессор P_j (в отн. P_k) получает сообщение (r,S_l). Кроме того, $U_l^{r-1} \subseteq S_l$. Таким образом, $U_l^{r-1} \subseteq U_j^r$ (в отн. $U_l^{r-1} \subseteq U_k^r$).

Рассмотрим независимое деление множества D на две части, выберем арбитра для каждой пары и пусть D' является множеством этих арбитров. Таким образом, получим $|D'| \leq 2^{r-1}$. По индукции имеем $m \leq |\cap_{j \in D'} U_j^{r-1}|$. Каждый процессор из D имеет арбитр из D' и, таким образом, $\cap_{j \in D'} U_j^{r-1} \subseteq \cap_{j \in D} U_j^r$. Отсюда, $m \leq |\cap_{j \in D} U_j^r|$.

Пусть D – множество несбоящих процессоров, которые начинают работу на шаге 3 протокола и пусть $C' = \cap_{j \in D} U_j^{\log n}$. По индукции получим $|C'| \geq m$. Для каждого $j \in C$ выходы всех несбоящих процессоров в BA_j-субпротоколах являются 1. По этому по определению византийских соглашений выход каждого BA_j-субпротокола является 1. Таким образом, $C' \subseteq C$ и $C \geq m$. ■

3.7.3. Схема (n,t)-звезды

Пусть OK_i-граф – это неориентированный граф с вершинами $[n]$, где ребро (j,k) существует тогда и только тогда, когда процессор P_i завершил распространение двух сообщений (OK,j,k), инициированное P_j и (OK,k,j), инициированное P_k.

Определение 3.3. Пусть G - граф с вершинами $[n]$. Тогда пара множеств (C,D) такая, что $C \subseteq D \subseteq [n]$ является (n,t)-звездой в G, если выполняются следующие условия:

- $|C| \geq n-2t$ и $|D| \geq n-t$;
- для каждого $j \in C$ и для каждого $k \in D$ в G существует ребро (j,k).

Процессор P_i получает доступ к разделенному секрету, когда находит звезду по OK_i-графу. Лемма 3.5 (см. далее) реализует, что если $n \geq 4t+1$, доли несбоящих процессоров в звезде OK_i- графа, определяют единственным образом полином степени t от двух переменных. Лемма 3.6. говорит о том, что полиномы, определенные звездами для каждой из двух сторон равны. Таким образом, только несбоящие процессоры могут найти звезду, определяющую уникальный секрет.

В принципе, можно было бы допустить процессор, ждущий клику размера n-t взамен звезды. Если дилер честен, то OK_i - граф будет иметь такую клику. Однако задача нахождения клики максимального размера является **NP**-полной задачей. Поэтому стороны будут пытаться найти (n,t)-звезду.

Кроме того, необходимо удостовериться, что если несбоящий процессор находит звезду по OK-графу, то все несбоящие процессоры найдут звезду, даже если дилер нечестен и OK-граф не содержит клики. После получения сообщения $(OK,\cdot,\cdot)$ процессор, который не нашел звезду проверяет какая из предложенных звезд является звездой для OK-графа. Отметим, что ребро в OK-графе для несбоящего процессора будет, в конечном счете, ребром в OK-графе для каждого несбоящего процессора (так как все сообщения $(OK,\cdot,\cdot)$ распространяются посредством Br-субпротокола. Таким образом, звезда в OK-графе некоторого несбоящего процессора будет в конечном счете звездой в OK-графе любого другого несбоящего процессора.

Далее описывается процедура нахождения (n,t)-звезды в графе с n вершинами (см. определение 3.3), где граф содержит клику размера n-t. А именно, нижеописываемая процедура **ЗВЗ**, в качестве выхода выдает либо звезду в графе, либо выдает сообщение «звезда не найдена». Как только граф содержит клику размера n-t, процедура выдает звезду в графе. Аналогичная процедура описана в [ГДж].

Алгоритм работает следующим образом. Сначала находится максимальное паросочетание в комплиментарном графе и выдается множество несравнимых вершин. (Паросочетание в графе – это произвольное подмножество попарно несмежных ребер. Паросочетание является *максимальным*, если оно не содержится в паросочетании с большим числом ребер [Ле]. Комплиментарный граф – это граф, в котором ребро существует тогда и только тогда, когда оно не существует в оригинальном графе).

Выход алгоритма является независимым множеством в комплиментарном графе и, таким образом, формируется клика в оригинальном графе. Кроме того, если граф содержит клику размера n-k, тогда максимальное паросочетание в комплиментарном графе включает не более k ребер и $2k$ вершин. Далее описание алгоритма ведется в терминах комплиментарного графа. Если входной граф имеет независимое множество мощностью n-t, находится множества $C \subseteq D$ вершин, такие, что

$|C| \geq n\text{-}2t$ $|D| \geq n\text{-}t$ и не существует ребер между вершинами из C и вершинами из C∪D. Эта пара множеств будет называться $\overline{(n,t)}$-*звездой*. Далее находится максимальное паросочетание в графе (см., например, [Ал, стр.18]).

На основе этого паросочетания вычисляются множества C и D вершин, и проверяется, формирует ли пара (C,D) $\overline{(n,t)}$-звезду в графе. Если входной граф содержит независимое множество мощностью $n\text{-}t$, тогда полученные множества C и D формируют $\overline{(n,t)}$-звезду в входном графе.

Далее показывается, как вычисляются множества C и D. Вершина называется *трехглавой*, если она не входит в паросочетание и две ее соседние вершины являются парой паросочетаний (а именно, ребро между этими двумя соседними вершинами являются паросочетанием). Пусть C - множество вершин, которые не являются трехглавыми и B - множество вершин, которые имеют соседние вершины из C и пусть D=[n]-B.

Алгоритм ЗВЗ

Вход: неориентированный граф G с вершинами [n] и параметром t.

Выход: $\overline{(t)}$-звезда в графе G, или сообщение «звезда не найдена».

1. Найти максимальное паросочетание M в G. Пусть N - множество согласованных вершин (а именно, концевые точки ребер в M) и пусть $\overline{N}$ =[n]-N.
2. Проверить, что паросочетание M имеет свойство Λ:
   - 2.1. Пусть T – множество трехглавых вершин, а именно $T=\{i \in \overline{N} \mid \exists\, j,k$ такие, что $(i,j),(i,k) \in G\}$. Пусть $C=\overline{N}-T$.
   - 2.2. Пусть B – множество вершин, имеющих соседние вершины в C, а именно $B:=\{j \in N \mid \exists\, i \in C$ такие, что $(i,j) \in \mathrm{G}\}$. Пусть D:=[n]-B.
   - 2.3. Если $|C| \geq n\text{-}2t$ и $|D| \geq n\text{-}t$ выдать (C,D), в противном случае выдать «звезда не найдена».

Предложение 3.3. Предположим, что процедура **ЗВЗ** выдает (C,D) на входном графе G. Тогда, (C,D) формирует $\overline{(t)}$-звезду в G.

Доказательство. Ясно, если алгоритм **ЗВЗ** выдает (C,D), тогда затем $|C|\geq n\text{-}2t$ $|D|\geq n\text{-}t$ и $C\subseteq D$. Мы показываем, что для каждого $i\in C$ и для каждого $j\in D$ вершины i и j не являются соседними в G.

Предположим, что $i\in C$ и $j\in D$ и что (i,j) – ребро в G. Так как $j\in D$, мы имеем $j\notin B$. По определению множества B, мы имеем $j\notin N$ (если $i\in C$ и $j\in N$, тогда $j\in B$). Кроме того, $i\in C\subseteq \overline{N}$. Таким образом, и i, и j не входят в паросочетание. Следовательно, ребро (i,j) может быть добавлено к паросочетанию для создания наибольшего паросочетания, и, следовательно, паросочетание не является максимальным. ■

Предложение 3.4. Пусть G - граф с n вершинами, содержащий независимое множество мощностью $n\text{-}t$. Тогда процедура **ЗВЗ** выдает $\overline{(t)}$-звезду в G.

Доказательство. Сначала покажем, что если входной граф G содержит независимое множество мощности $n\text{-}t$, тогда множества C и D, определенные на шагах 2.a и 2.b. алгоритма **ЗВЗ** являются достаточно большими. Следовательно, алгоритм выдает (C,D). В предложении 3.3 утверждается, что (C,D) формирует звезду в G. Далее покажем, что $|C|\geq n\text{-}2t$. Пусть $I\subseteq[n]$ – независимое множество мощности $n\text{-}t$ в G и пусть $\overline{I}=[n]\text{-}I$. Далее необходимо конкретизировать значения N, T и C в алгоритме **ЗВЗ**.

Пусть $F=I\text{-}C$. Далее покажем, что $|F|\leq|\overline{I}|$. В то же время $|\overline{I}|\leq t$. Следовательно, $|C|\geq|I|\text{-}|F|\geq n\text{-}2t$.

Так как $|F|\leq|\overline{I}|$, покажем взаимно однозначное соответствие $\phi{:}F\to\overline{I}$. Пусть $i\in F$ и так как $i\notin C$, мы имеет либо $i\in N$, либо $i\in T$.

Случай 1: $i\in N$. Тогда, пусть ϕ(i) - вершина, сочетаемая с i в множестве M. Ясно, что $\phi(i)\in\overline{I}$, в противном случае мы имели бы ребро $(i,\phi(i))$, где и i и $\phi(i)$ принадлежит независимому множеству.

Случай 2: $i\in T$. По определению множества T, вершина i имеет две соседние вершины j и k такие, что $(j,k)\in M$. Произвольно множество $\phi(i)=j$. Понятно, что и j, и k принадлежат $\overline{I}$.

Покажем, что ϕ - взаимно однозначная функция. Рассматривая две различные вершины $l,m\in F$, мы различаем три случая.

Случай 1: $l,m\in N$. В этом случае, $\phi(l)\neq\phi(m)$, так как M - паросочетание.

Случай 2: $l \in N$ и $m \in T$. Так как $m \in T$, то существует ребро между m и вершиной, сочетаемой с $\phi(m)$. Так как $l \in N$ вершина, сочетаемая с $\phi(l)$ – это вершина l. Далее, предположим, что $\phi(l)=\phi(m)$. Таким образом, (l,m) – ребро в G. Однако и l, и m принадлежат множеству I. Что противоречит условию.

Случай 3: $l,m \in T$. Предположим, что $\phi(l)=\phi(m)$. Пусть a - вершина, сочетаемая с $\phi(m)$ из M. Тогда и l, и m – соседние вершины для $\phi(m)$ и a. Однако, в этом случае паросочетание M - не является максимальным, например, $M-\{(\phi(m),a)\} \cup \{(\phi(m),l),(a,m)\}$ – является наибольшим паросочетанием.

В заключение покажем, что $D \geq n-t$. Напомним, что $D=[n]-B$. Далее покажем, что $|B| \leq |M|$. Так как G содержит независимое множество мощности $n-t$, мы имеем $|M| \leq t$. Таким образом, $|D|=n-|B| \geq n-|M| \geq n-t$.

Для того, чтобы понять, что $|B| \leq |M|$, мы покажем, что не более одной концевой точки каждого ребра $(a,b) \in M$ находится в B. От обратного предположим, что и a, и b имеют соседние вершины в C и пусть $c,d \in C$ – соседние вершины a и b, соответственно. Конечно, $c \neq d$ (в противном случае, вершина c была бы трехглавой, и мы имели бы $c \notin C$). Однако в этом случае паросочетание M не является максимальным. Например, $M-\{(a,b)\} \cup \{(a,c),(b,d)\}$ является наибольшим паросочетанием. ■

3.7.4. Схема, корректирующая ошибки

Схема, корректирующая ошибки, работающая в префиксном режиме, обозначаемая в дальнейшем **СКОП**, позволяет каждому процессору вычислить полином $p(\cdot)$ степени d, удовлетворяющий $p(j)=a_j$ для каждого полученного им сообщения a_j. А именно, процессор будет исследовать этот полином после получения сообщений от других процессоров и определять единственным образом интерполируемый полином степени d [BCG].

Определение 3.4. Пусть δ и ρ - целые числа и пусть множество $S \subseteq [n] \times F$ такое, что для каждых двух элементов (i,a) и (i',a') из S, получаем $i=i'$. Будем считать, что S является (δ,ρ)-*интерполируемым*, если существует полином $p(\cdot)$ степени δ такой, что не более ρ элементов $(i,a) \in S$ не удовлетворяет $p(a)=e$. Будем говорить, что $p(\cdot)$ является (δ,ρ)-*интерполируемым полиномом* множества S.

Определение 3.1. Пусть I – аккумулируемое множество (см. выше). Тогда I называется *событийно* (δ,ρ)-*интерполируемым*, если для каждого

планировщика, каждой коалиции из не более чем τ сбоящих процессоров и каждого запуска алгоритма существует целое число $0 \le \rho \le \tau$ такое, что I будет событийно содержать (δ,ρ)-интерполируемое множество мощностью не менее $\delta+\tau+\rho+1$.

При использовании этих обозначений динамический вход процедуры, описываемой ниже, является (d,t)-интерполируемым аккумулируемым множеством I. Требуемый выход этой процедуры является (d,t)-интерполируемым полином множества I. Определение 3.5 гарантирует, что, по крайней мере, $d+t+1$значений из I будут сопряжены с полиномом степени t. По крайней мере $t+1$ из них соответствуют несбоящим процессорам. Следовательно, (d,t)-интерполируемый полином множества I определен значениями несбоящими процессорами.

Процедура **СКОП** состоит из $t+1$ итераций. На итерации r процессор ожидает, пока мощность аккумулируемого множества I станет не менее $d+t+r+1$. Затем, каждый процессор запускает ниже описываемую процедуру, которая определяет, является ли множество I - (d,r)-интерполируемым и вычисляет соответствующий интерполируемый полином. Если (d,r)-интерполируемый полином найден, тогда оно подается на выход и работа завершается. В противном случае, мы переходим к итерации $r+1$ (так как I – является событийно интерполируемым, он ограничен, по крайней мере, еще одним элементом и, таким образом, итерация $r+1$ будет завершена).

Далее описывается, как определить, является ли данное множество (d,r)-интерполируемым и как вычислять интерполируемый полином, используя теорию кодирования. Рассмотрим следующий код. Слово $W=(i_1,a_1)...(i_l,a_l)$ является ключевым, тогда и только тогда, когда существует полином $p(\cdot)$ степени d такой, что $p(i_j)=a_j$ для каждый $1 \le j \le l$. Такой код является обобщенным кодом Рида-Соломона, имеющим эффективную процедуру, корректирующую ошибки. Данная процедура обнаруживает и исправляет не менее r ошибок во входном слове W, где $|W| \ge d+2r+1$ (см., например, [КС]).

Пусть **ПКО** обозначает следующую процедуру. А именно, пусть S - (d,r)-интерполируемое множество мощностью не менее $d+2r+1$. Тогда процедура **ПКО** по входу (d,r,S), выдает (d,r)-интерполируемый полином из множества S.

Процедура СКОП

Выполняется в t циклах по r.

1. Пусть I_r определяет содержимое аккумулируемого множества I, где I содержит $d+t+r+1$ элементов.
2. Ожидать пока $|I| \geq d+t+r+1$. Тогда, запустить процедуру **ПКО** со входом (d,r,I_r), и пусть $p(\cdot)$ – выходной полином.
3. Если $p(\cdot)$ - (d,r)- интерполируемый полином I_r (а именно, если для $d+t+1$ элементов $(i,a) \in I_r$, мы имеем $p(i)=a$), выдать $p(\cdot)$. В противном случае, перейти к следующей итерации.

Предложение 3.5. Пусть I – событийно (d,t)-интерполируемое аккумулируемое множество. Тогда процедура **СКОП** останавливается и выдает (d,t)-интерполируемый полином множества I.

Доказательство. Пусть r', являющийся наименьшим из r, такой, что $I_{r'}$ является $(d,I_{r'})$-интерполируемым. Так как I – является событийно (d,t)-интерполируемым, то $r' \leq t$. Все итерации вплоть до r' будут неудачно завершены. Тогда (d,t)-интерполируемый полином I будет найден на итерации r'. ■

3.7.5. Асинхронная схема проверяемого разделения секрета

Общие определения

Следующее определение формализует требования к схеме проверяемого разделения секрета, но в асинхронной установке [BCG].

Определение 3.2. Пусть (**АРзПр,АВсПр**) - пара многосторонних протоколов таких, что каждый несбоящий процессор, который завершает протокол **АРзПр** впоследствии вызывает протокол **АВсПр** с локальным выходом протокола **АРзПр** как локальным входом. Схема (**АРзПр**,**АВсПр**) называется *t-устойчивой схемой асинхронного разделения секрета* – **АПРС** для n процессоров, если для каждой коалиции из не более, чем t процессоров и каждого планировщика выполняются следующие условия.

Условие завершения. 1. Если дилер честен, тогда каждый несбоящий процессор, в конечном счете, завершит протокол **АРзПр**. 2. Если некоторый несбоящий процессор завершил протокол **АРзПр**, то все несбоящие процессоры, в конечном счете, завершат протокол **АРзПр**. 3. Если несбоящий процессор завершил протокол **АРзПр**, то он завершит и протокол **АВсПр**.

Условие корректности. Как только несбоящий процессор завершил протокол **АРзПр**, тогда существует уникальное значение r такое, что: 1. все несбоящие процессоры выдают r (а, именно, r - восстанавливаемый секрет); 2. если дилер честен и делит секрет s, тогда $r=s$.

Условие конфиденциальности. Если дилер честен и до тех пор пока никакой из несбоящих процессоров не вызвал протокол **АВсПр**, тогда сбоящий процессор не получает никакой информации о разделенном секрете.

Необходимо подчеркнуть, что для несбоящего процессора не требуется, чтобы он завершал протокол **АРзПр** в случае, если дилер нечестен. Мы не различаем случай, где несбоящий процессор не завершает протокол **АРзПр** и случай, где несбоящий процессор завершает протокол **АРзПр** неудачно.

Схема АПРС

Первый из двух протоколов асинхронного разделения секрета **АРзПр** состоит из трех стадий. Сначала, каждый процессор ждет получения своей доли секрета от дилера. Затем, стороны совместно пытаются проверить, что их доли определяют единственным образом секрет. Как только процессор убеждается, что секрет уникален, он локально вычисляет и выдает свою «скорректированную долю» секрета (с использованием информации, накапливаемой на стадии верификации).

В протоколе **АВсПр** каждый процессор посылает свою долю процессорам, принадлежащим некоторому предопределенному множеству E. Затем каждый процессор ждет получения достаточного количества долей для определения единственным образом секрета и, в конечном счете, для реконструкции секрета.

Схема АПРС

Схема **АПРС** имеет следующее свойство. Существует полином $p(\cdot)$ степени t такой, что каждый выход несбоящего процессора P_i, протокола **АРзПр** – $p(i)$ и $p(0)$ является разделенным секретом. Это свойство лежит в основе построения схемы **АРзПр** в условиях *Ву*-сбоев.

Протокол АРзПр

Вычисления дилера (по входу s):

1. Выбрать случайный полином $h(\cdot,\cdot)$ степени t от двух переменных такой, что $h(0,0)=s$ (Т.е. $h(x,y)=\sum_{i=0}^{t}\sum_{j=0}^{t} h_{i,j}x^i y^j$, где $h_{0,0}=s$ и все другие коэффициенты $h_{0,1},...,h_{t,t}$ выбраны однородно и независимо над F). Для каждого $1\le i\le n$ посылает полиномы $f_i(\cdot)=h(\cdot,i)$ и $g_i(\cdot)=(h_i,\cdot)$ процессору P_i.

Вычисления процессора P_i:

2. После получения $f_i(\cdot)$ и $g_i(\cdot)$ от дилера и для каждого $1\leq j\leq n$ посылает $f_i(j)$ процессору P_j.

3. После получения $v_{i,j}$ от процессора P_j проверяет, если $v_{i,j}=g_i(j)$, тогда этот процессор распространяет (OK,i,j).

4. После получения (OK,j,k), контролирует существование звезды в OK, используя процедуру **ЗВЗ**, описанную выше. Если звезда (C_i,D_i) найдена, переходит к шагу 6 и посылает (C_i,D_i) всем процессорам.

5. После получения сообщения (C_j,D_j) добавляет (C_j,D_j) к множеству «предлагаемых звезд». Пока звезда не найдена, всякий раз, когда получено распространяемое сообщение (OK,k,l), проверяет, формируют ли (Cj,D_j) звезду в графе OK_i.

6. После нахождения звезды (C_i,D_i) и если $i\notin D_i$ скорректировать полином $g_i(\cdot)$, основываясь на сообщении о верификации, полученном от процессоров из D_i и с использованием кодов с исправлением ошибок (процедуры **СКОП**, описанной ниже). А именно, пусть $V_i=\{(j,v_i,j)\,|\,j\in D_i\}$, тогда установить (t,V_i)**СКОП=(**$g_i(\cdot)$).

7. Как только $g_i(\cdot)$ локально скорректирован, выдает g (0).

Протокол АВсПр

Вычисления для процессора P_i (по входу i и с параметрами $R\subseteq\{P_1,\ldots,P_n\}$).

7. Посылает a_i процессорам из R.

9. Пусть $S_i=\{(j,a_j)\,|\,a_j$ получен из P_j. Если $P_i\in R$, заканчивает без выхода. В противном случае установить (t,S_i)**СКОП=**$z_i(\cdot)$ и выдает $z_i(0)$.

Доказательство безопасности схемы АПРС

Теорема 3.7. Пара протоколов (**АРзПр**, **АВсПр**) является t-устойчивой схемой **АПРС** в сети из n процессоров, если $n\geq 4t+1$.

Доказательство. Из определения 3.6 видно, что нам необходимо доказать условия завершения, корректности и конфиденциальности. Начнем с доказательства корректности схемы **АПРС**.

Корректность. С каждым несбоящим процессором P_i, который завершает протокол **АРзПр** мы ассоциируем уникальный полином $h_i(\cdot,\cdot)$ степени t от двух переменных и показываем, что каждые два несбоящих процессора P_i и P_j имеют $h_i(\cdot,\cdot)=h_j(\cdot,\cdot)$. Затем, мы показываем, что условия 1 и 2 корректности выполняются, относительно $r=h_i(0,0)$ (для некоторого

несбоящего процессора P_i). Кроме того, мы показываем, что выход процессора P_i протокола **АРзПр** есть $h_i(i,0)$.

Для демонстрации этого мы используем две технических леммы, которые приведем без доказательства [BCG]. В лемме 3.5 показывается, что если $4t+1$ долей несбоящих процессоров, находящихся в звезде в графе OK определяют единственный полином степени t от двух переменных. Лемма 3.6 демонстрирует тот факт, что полиномы индуцированные звездами каждой из двух процессоров одинаковы. Таким образом, только несбоящий процессор находит звезду, и, следовательно, секрет определяется единственным образом.

Лемма 3.5. Пусть $m \geq d+1$, и пусть $f_1(\cdot),...,f_m(\cdot)$ и $g_1(\cdot),...,g_m(\cdot)$ - полиномы степени d над полем F с $|F| \geq m$ такие, что для каждого $1 \leq i \leq d+1$ и каждого $1 \leq j \leq m$ мы имеем $f_i(j)=g_j(i)$ и $g_i(j)=f_j(i)$. Тогда существует единственный полином $h(\cdot,\cdot)$ степени d от двух переменных такой, что для каждого $1 \leq i \leq m$ мы имеем $h(\cdot,i)=f_i(\cdot)$ и $h(i,\cdot)=g_i(\cdot)$.

Лемма 3.6. Пусть $h(\cdot,\cdot)$, $h'(\cdot,\cdot)$ – два полинома степени d от двух переменных над полем F с $|F| \geq d$ и пусть $v_1,...,v_{d+1}$ - различные элементы в F. Предположим, что для каждого $1 \leq i,j \leq d+1$ мы имеем $h(v_i,v_j)=h'(v_i,v_j)$. Тогда $h(\cdot,\cdot)=h'(\cdot,\cdot)$.

Далее, пусть P_i – несбоящий процессор, который завершил протокол **АРзПр** и пусть (C_i,D_i) - звезда, найденная процессором P_i. Пусть D_i' - множество несбоящих процессоров в D_i пусть C_i' - множество несбоящих процессоров в C_i и, таким образом, $|D_i'| \geq |D_i|-t \geq n-2t$ и $|C_i'| \geq |C_i|-t \geq n-3t \geq t+1$ (так как $4t+1$). В соответствии с леммой 3.5 полиномы $f_j(\cdot),g_j(\cdot)$ процессоров $j \in D_i'$ определяют единственный полином степени t от двух переменных. Пусть $h_i(\cdot,\cdot)$ обозначает этот полином. А именно, $h_i(\cdot,\cdot)$ - полином, ассоциированный с P_i. Отметим также, что $h_i(\cdot,\cdot)$ фиксирован, как только P_i завершает протокол **АРзПр**. Для каждого другого несбоящего процессора P_j пусть $I_{i,j}$ - множество несбоящих процессоров из $D_i \cap D_j$. Так как $n \geq 4t+1$, мы имеем $|D_i \cap D_j| n-2t \geq 2t+1$ и, таким образом, $|I_{i,j}| \geq t+1$. Для каждых двух процессоров $k,l \in I_{i,j}$, мы имеем $h_i(k,l)=v_{k,l}=h_j(k,l)$, где $v_{k,l}$ – верификационная часть, посланная P_k к P_l на шаге 2 протокола **АРзПр**. Применяя результаты леммы 3.6, мы имеем $h_i(\cdot,\cdot)=h_j(\cdot,\cdot)$. Значение r, требуемое в условии корректности, является равным $h_i(0,0)$.

Мы утверждаем, что условие 1 (а именно, что если дилер честен и выдает значение s, то $r=s$). Если дилер честен и он выбрал полином $h(\cdot,\cdot)$ на

шаге 1, тогда для каждых двух процессоров $P_k,P_l \in D_i'$ мы имеем $h_i(k,l)=h(k,l)$. В соответствии с леммой 3.6 мы снова можем получить $h_i(\cdot,\cdot)=h(\cdot,\cdot)$. Далее нижний индекс в полиноме $h(\cdot,\cdot)$ опускается.

Далее мы показываем, что выход каждого несбоящего процессора P_i протокола **АРзПр** есть $h(i,0)$. Полином $h(i,\cdot)$ - интерполируемый полином аккумулируемого множества U_i процессора P_i на шаге 6. Следовательно, выход процедуры **СКОП** на шаге 6 будет $h(i,\cdot)$ и выход протокола **АРзПр** будет $h(i,0)$.

В заключение мы показываем, как выполняется условие 2 (а именно, что выход P_i протокола **АВсПр** есть r). Каждый несбоящий процессор P_j распространяет $h(j,0)$ на шаге 7 и, таким образом, $h(\cdot,0)$ - интерполируемый полином аккумулируемого множества U_i процессора P_i на шаге 8. Следовательно, выход процедуры **СКОП** на шаге 8 будет $h(\cdot,0)$ и выход протокола **АВсПр** будет $h(0,0)=r$.

Завершение. Условие 1. Если дилер честен, то для каждых двух несбоящих процессоров P_j и P_k, оба сообщения (OK,j,k) и (OK,k,j) будут распространены так как $f_j(k)=h(k,j)=g_k(j)$ и $g_j(k)=h(j,k)=f_k(j)$. Таким образом, каждый несбоящий процессор P_i будет в конечном счете иметь клику размера n-t в графе OK_i. Поэтому процедура **ЗВЗ** будет находить звезду в OK_i и шаг 4 будет завершен. Шаг 6 будет завершен, так как вход процедуры **СКОП** (а именно, аккумулируемое множество U_i, которое базируется на звезде, найденной на шагах 4 или 5) - событийно (t,t)-интерполируем.

Условие 2. Пусть P_i – несбоящий процессор, который завершил протокол **АРзПр** и пусть (C_i,D_i) - звезда, найденная процессором P_i. Тогда (C_i,D_i) будет в конечном счете звездой в графе OK_j для каждого несбоящего процессора P_j, даже если P_j не завершил протокол **АРзПр**. Кроме того, процессор P_j получит (C_i,D_i) сообщение (посланное P_i на шаге 4), и будет проверять на шаге 5, что множества (C_i,D_i) формируют звезду в OK_j. После нахождения звезды процессор P_j выполнит шаг 6 и завершит протокол **АРзПр**.

Условие 3. Если все несбоящие процессоры начали выполнять протокол **АВсПр**, тогда аккумулируемое множество S_i на шаге 8 для каждого несбоящего процессора P_i, событийно (t,t)-интерполируемо. Таким образом, все несбоящие процессоры завершат процедуру **СКОП** и протокол **АВсПр**.

Конфиденциальность. Мы используем следующую систему обозначения. Для значения v пусть H_v обозначает множество полиномов степени t от двух переменных со свободным коэффициентом v. Будем говорить, что последовательность $f_1(\cdot),...,f_t(\cdot)$, $g_1(\cdot),...,g_t(\cdot)$ полиномов *чередуемы*, если для каждых $1 \le i,j \le t$ мы имеем $f_i(j)=g_j(i)$. Пусть I обозначает множество чередуемых последовательностей $2t$ полиномов степени t.

Лемма 3.7. Пусть F – поле с $|F| \ge d$ и пусть $s \in F$. Тогда для каждой чередуемой последовательности $f_1(\cdot),...,f_d(\cdot)$, $g_1(\cdot),...,g_d(\cdot)$ в I существует единственный полином $h(\cdot,\cdot) \in H_s$ такой, что для каждого $1 \le i \le d$ мы имеем $h(\cdot,i)=f_i(\cdot)$ и $h(i,\cdot)=g_i(\cdot)$.

Доказательство. См. работу [BCG].

Далее предположим, что дилер честен и пусть s – разделяемое значение. Тогда дилер имеет на шаге 1 протокола **АРзПр** полином $h(\cdot,\cdot)$ с равномерным распределением вероятностей над H_s. Кроме того, вся необходимая информация о множестве из t процессоров, полученная во время выполнения протокола **АРзПр**, является чередуемой последовательностью $f_1(\cdot),...,f_t(\cdot)$, $g_1(\cdot),...,g_t(\cdot)$ в I такой, что для каждого $1 \le i \le t$ мы имеем $h(\cdot,i)=f_i(\cdot)$ и $h(i,\cdot)=g_i(\cdot)$.

Из леммы 3.7 следует, что для каждого разделяемого значения $s \in F$ это соответствие между полиномами в H_s и чередуемыми последовательностями в I – является взаимнооднозначным. Следовательно, равномерное распределение над полиномами из H_s индуцирует равномерное распределение вероятностей над чередуемыми последовательностями в I. ■

Асинхронная схема глобального проверяемого разделения секрета

Протокол асинхронного глобального проверяемого разделения секрета, обозначаемый как **АГРз**, состоит из двух этапов. Сначала, каждый процессор делит секрет среди других процессоров, а затем процессоры используют протокол **СОАМ**, чтобы договориться о множестве C размером не менее чем из n-t процессоров, которые успешно разделили свои секреты. Выход процессора P_i в протоколе **АГРз** – это множество процессоров, которые успешно разделили свои входы, вместе с i-той долей входа каждого процессора в этом множестве. Протокол **АГРз** имеет параметр безопасности d. Исходя из контекста, этот параметр будет устанавливаться либо в t, либо в $2t$.

Протокол АГРз

Код для процессора P_i по входу x_i, d.

1. Осуществить разделение x_i:

1.1. Выбрать случайным образом полином $h_i(\cdot)$ степени d такой, что $h_i(0)=x_i$.

1.2. Для каждого $1 \leq j \leq n$ послать $h_i(j)$ процессору P_j.

1.3. Разослать сообщение «Сторона P_i завершила разделение».

2. После получения доли $s_{i,j}$ секрета процессора P_j и сообщения «Сторона P_j завершила разделение» добавить j к множеству C_i процессоров, которые успешно разделили свои секреты. Установить (n,t,C_i)**СОАМ**$=C$.

3. Подать на выход C и $\{s_{i,j} \mid j \in C\}$.

Необходимо отметить, что аккумулируемые множества $C_1,...,C_n$ на шаге 2 определяют $(n\text{-}t,t)$-однородную коллекцию. Таким образом, все несбоящие процессоры завершают протокол **СОАМ** (и, следовательно, завершают протокол **АПРз**) с желаемым выходом. Кроме того, сбоящие процессоры не получают информацию о значениях, разделенных несбоящими процессорами.

В протоколе **АГВс**, описываемом ниже процессоры восстанавливают секрет из долей. Параметры этого протокола – параметр безопасности d и множество R процессоров, для которого секрет может быть вскрыт. Вход процессора P_i является i-той долей секрета, обозначаемой как s_i.

<u>Протокол АГВс</u>

Код для процессора P_i по входу s_i и параметрам $d \in \{t,2t\}$ и $R \subseteq \{P_1,...,P_n\}$.

1. Послать s_i всем процессорам из R.

2. Если процессор $P_i \notin R$, подать на выход 0. В противном случае после получения d+1 значений (значение v_j от процессора P_j) интерполировать полином $p_i(\cdot)$ степени d такой, что $p(j)=v_j$ для каждого полученного значения v_j. Подать на выход $p_i(0)$.

Асинхронная схема глобального проверяемого разделения секрета (**АГПРС**), описанная ниже, является обобщением предыдущей схемы для случая *Ву*-сбоев. Схема состоит из двух этапов: сначала, каждый процессор разделяет вход, используя протокол **АРзПр** схемы **АПРС**, а затем процессоры используют субпротокол **СОАМ** для того, чтобы договориться о множестве C не менее чем $n\text{-}t$ процессорами, которые объединили свои входы. Выход процессора P_i в субпротоколе **АГПРС** и

есть это множество C и i-тая доля каждого секрета включена процессором P_i в C. В данном протоколе параметр безопасности фиксирован: $d=t$.

Субпротокол АГПРС

Код для процессора P_i по входу x_i.

1. Инициировать **АРзПр**(x_i) с P_i в качестве дилера. Для $1 \le j \le n$ участвовать в протоколе в **АРзПр**$_j$. Пусть v_j – есть выход **АРзПр**$_j$.

2. Пусть $U_i=\{j \mid$ **АРзПр**$_j$ был завершен$\}$. Множество C вычислить как: (n,t,U_i)**COAM**$=C$.

3. Как только множество C вычислено, подать на выход $(C, v_j \mid j \in C)$.

3.7.6. Вычисления на мультипликативном вентиле

Вычисления при FS-сбоях

В данном разделе показывается, как надежно при наличии *FS*-сбоях t-вычислить любую функцию, вход которой разделен между n процессорами, когда $n \ge 3t+1$.

Пусть F – конечное поле, известное всем процессорам и $|F|>n$. Пусть также $f{:}F^n \rightarrow F$ - вычислимая функция. Предположим, что процессоры имеют арифметическую схему, вычисляющую функцию f. Схема состоит из вентилей сложения и умножения степени 2. Добавление константы рассматривается как частный случай сложения. Все вычисления в выполняются в поле F.

Общее описание протокола выглядит следующим образом [BCG]. Пусть x_i - вход процессора P_i. На первом шаге каждый процессор разделяет вход среди других процессоров, используя, например, схему разделения секрета Шамира (см. выше). А именно, для каждого процессора P_i, который успешно разделил свой вход, случайным образом генерируется полином $p_i(\cdot)$ степени t, сгенерирован такой, что каждый процессор P_j имеет $p_i(j)$ и $p_i(0)$ - значение входа процессор P_i. Будем говорить, что $p_i(j)$ - доля $p_i(0)$ процессора P_i. Затем, стороны договариваются, используя протокол **COAM,** о множестве C процессоров, которые успешно разделили свои входы. Как только C вычислено, процессоры начинают вычислять $f_C(\vec{x})$, следующим образом. Сначала, входные значения процессоров не принадлежащие C – устанавливаются в некоторое значение по умолчанию, скажем в 0. Затем процессоры вычисляют данную схему, вентиль за

вентилем способом, описанном ниже. Заметим, что выход протокола будет зафиксирован, как только множество C зафиксировано.

Для каждого вентиля процессоры используют свои доли входных линий для совместного и безопасного «генерирования» случайного полинома $p(\cdot)$ степени t, такого, что каждый процессор P_i вычисляет $p(i)$ и $p(0)$ - является выходным значением этого вентиля. А именно, $p(i)$ - доля процессора P_i на выходной линии этого вентиля. Как только процессоры вычислили свои доли на выходных линиях всей схемы, процессоры восстанавливают свои доли на выходной линии и интерполируют выходное значение.

Вычисления на линейном вентиле

Ниже описывается вычисления на линейном вентиле вместо простого аддитивного вентиля. Это более обобщенное описание будет удобно для протокола вычисления на мультипликативном вентиле.

Вычисления на линейном вентиле достаточно просты и не требуют взаимодействия между процессорами. Пусть $c = \sum_{j=1}^{k} \alpha_j a_j$ - линейный вентиль, где $a_1,...,a_k$ – входные линии вентиля, $\alpha_1,...,\alpha_k$ - фиксированные коэффициенты и c – выходная линия. Пусть $A_j(\cdot)$ – полином, ассоциированный с j-той входной линией. А именно доля процессор P_i на этой линии – есть $a_{i,j}=A_j(i)$. Каждый процессор P_i локально устанавливает свою долю выходной линии в $c_i = \sum_{j=1}^{k} \alpha_j a_{i,j}$. Можно легко увидеть, что доли $c_1,...,c_n$ определяют случайный полином $C(\cdot)$ степени t с корректным свободный коэффициентом и с $C(i)=C_i$ для всех i.

Вычисления на мультипликативном вентиле

Пусть $c=a\cdot b$ – мультипликативный вентиль и пусть $A(\cdot)$, $B(\cdot)$ – полиномы, ассоциированные с входными линиями, а именно доли каждого процессора P_i этих линий - $A(i)$ и $B(i)$ соответственно. Процессоры совместно вычисляют свои доли случайного полинома $C(\cdot)$ степени t, удовлетворяющий $C(0)=A(0)\cdot B(0)$, а именно доля каждого несбоящего процессора P_i на выходной линии будет $C(i)$.

Сначала каждый процессор локально вычисляет свою долю полинома $E(\cdot)=A(\cdot)\cdot B(\cdot)$, устанавливая $E(i)=A(i)\cdot B(i)$. Ясно, что $E(\cdot)$ имеет требуемый свободный коэффициент. Однако, полином $E(\cdot)$ имеет степень $2t$ и, кроме того, полином не является равномерно распределенным. Процессоры используют свои доли полинома $E(\cdot)$, чтобы вычислить свои доли

требуемого полинома $C(\cdot)$ безопасным способом. Вычисления осуществляются в два этапа: на первом этапе процессоры совместно генерируют случайный полином $D(\cdot)$ степени $2t$ такой, чтобы $D(0)=E(0)$. А именно, каждый процессор P_i будет иметь $D(i)$. Затем, стороны используют их доли $D(\cdot)$, чтобы совместно вычислить свои доли полинома $C(\cdot)$. Эти шаги описаны ниже.

Рандомизация. Сначала покажем, как генерируется полином $D(\cdot)$. Сначала процессоры генерируют случайный полином $H(\cdot)$ степени $2t$ и с $H(0)=0$; а именно, каждая сторона P_i будет иметь $H(i)$. Мы сначала описываем, как полином $H(\cdot)$ делится в синхронной модели вычислений [BGW].

Каждый процессор P_i выбирает случайный полином $H_i(\cdot)$ степени $2t$ с $H_i(0)=0$ и делит его среди процессоров; а именно, каждый процессор P_j получает $H_i(j)$. Полином $H(\cdot)$ – устанавливается в $H(\cdot)=\sum_{j=1}^{n} H_j(\cdot)$; а именно каждый процессор P_i вычисляет $H(i)=\sum_{j=1}^{n} H_j(i)$.

В асинхронной модели вычислений процессор не может ждать, пока получит свою долю от всех других процессоров. Вместо этого, стороны соглашаются, используя субпротокол **СОАМ**, о множестве C процессоров, которые успешно разделили полиномы H_i. Затем полином $H(\cdot)$ будет установлен в $H(\cdot)=\sum_{j\in C} H_j(\cdot)$. Другими словами, процессоры запускают протокол **АГПРС,** получают на выходе $(C, f\{H_j(i) \mid j\in C\})$ и каждый процессор P_i вычисляет $H(i)=\sum_{j\in C} H_j(i)$.

Полином $D(\cdot)$теперь определен как $D(\cdot)=E(\cdot)+H(\cdot)$; а именно каждый процессор P_i вычисляет $D(i)=E(i)+H(i)$. Ясно, что $D(0)=A(0)\cdot B(0)$ и все другие коэффициенты $D(\cdot)$ равномерно и независимо распределены над F.

Редукция степени. На этом этапе процессоры используют свои доли полинома $D(\cdot)$, чтобы совместно и безопасно вычислить свои доли случайного полинома $C(\cdot)$ степени t с $C(0)=D(0)$. Полином $C(\cdot)$ будет устанавливать «усечение» полинома $D(\cdot)$ к степени t; а именно $t+1$ коэффициентов $C(\cdot)$ являются коэффициентами $t+1$ более низкой степени полинома $D(\cdot)$. Важный момент здесь состоит в том, что информация, которую накапливают сбоящие процессоры (а именно t долей полинома $D(\cdot)$ вместе с t долями усеченного полинома $C(\cdot)$), независима от $C(0)$.

Пусть $\vec{d}=D(1),...,D(n)$ и пусть $\vec{c}=C(1),...,C(n)$. В работе [BGW] отмечено, что существует фиксированная матрица M размерностью $n\times n$ такая, что $\vec{c}=\vec{d}M$. Отсюда следует, что требуемый выход каждого процессора P_i - линейная комбинация входов процессоров: $C(i)=[\vec{d}M]_i=\sum_{j=1}^{n} D(i)\cdot M_{i,j}$. Будем называть такую операцию «умножение входов на фиксированную матрицу». В этом случае, входы процессоров являются их долями полинома $D(\cdot)$.

В синхронной модели вычислений умножение входов на фиксированную матрицу может быть выполнено посредством безопасного вычисления соответствующих n фиксированных линейных комбинаций входов таких, что значение i-той линейной комбинации вскрываемо только процессором P_i. Линейные комбинации безопасно вычисляются следующим образом. Сначала каждый процессор разделяет свой вход; затем каждый процессор вычисляет линейную комбинацию своих долей и открывает эту линейную комбинацию специальному процессору; в заключение специальный процессор вычисляет значение выхода, интерполируя полином степени t из полученных комбинаций [Ca2].

Линейные комбинации всех входов не могут быть вычислены в асинхронной модели вычислений и, следовательно, синхронный метод, описанный выше, не может использоваться. Ниже приводится решение для асинхронной модели вычислений. Сначала мы описываем метод для умножения входов на фиксированную матрицу в асинхронной модели для случая, когда матрица и множество входов связаны специальным образом. Кроме того, мы отметим также, что матрица M и множество возможных входов (на этапе редукции степени) тоже связаны специальным образом.

Определение 3.7. Пусть A – матрица размерностью $n\times n$ и пусть $S\subseteq F^n$ - множество входных векторов. Будем говорить, что S – *t-умножаемо* на A, если для каждого множества $G\subseteq[n]$ с $|G|\geq n-t$ существует (легко вычислимая) матрица A_G размером $|G|\times n$ такая, что для каждого входа $\vec{x}\in S$ мы имеем $\vec{x}_G\cdot A_G=\vec{x}A$.

Пусть S – t-умножаемо на A. Тогда протокол, описанный ниже, «умножает входы из множества S на матрицу». Пусть $\vec{x}\in S$. Процессоры сначала выполняют протокол **АГПРС** (с входным вектором $\vec{x}$). Как только общее множество G вычислено, каждый процессор локально

вычисляет A_G. Затем, процессоры запускают n протоколов восстановления секрета **АВс**, где в i-том протоколе **АВс** процессоры позволяют процессору P_i вычислить $\vec{x}_G \cdot A_G$, посылая ему соответствующую линейную комбинацию своих долей. Ниже описывается протокол для умножения входа на фиксированную матрицу.

Протокол МАТ (x_i,A)

Процессор P_i на входе x_i и матрице A работает следующим образом.

1. Устанавливает (t,x_i)**АГПРС**$=(G,\{s_{i,j} \mid G\})$.
2. Нумерует $G=g_1,\ldots,g_{|G|}$ и пусть в этом случае $\vec{s} = s_{i,g_1},\ldots,s_{i,g_{|G|}}$.
3. Вычисляет A_G.
4. Для $1 \le k \le n$ устанавливает $([\vec{s}_i \cdot A_G]_k,t,\{k\})$**Авс**$=y_k$.

Подает на выход y_i.

Далее будем говорить, что входной вектор $\vec{x}$ есть – *d-сгенерирован*, если существует полином $P(\cdot)$ степени d, удовлетворяющий $x_i=P(i)$ для каждого i; множество возможных входов на шаге редукции степени – это множество $2t$-сгенерированных векторов.

Для дальнейших рассуждений нам необходима матрица особого вида. Эта матрица, назовем ее M, описана в работе [BGW] и строится как $M=V^{-1}TV$, где V – матрица Вандермонда размерностью $n \times n$ (см, например, [Кн, стр.474]), определенная как $V_{i,j}=i^j$, а T построена установкой всех элементов, кроме элементов первых $t+1$ столбцов, единичной матрицы в нули. (Пусть $\vec{c}\,,\vec{d}$ - вектора, определенные выше). Для того чтобы увидеть, что $\vec{d} \cdot M=\vec{c}$ необходимо заметить, что $\vec{d} \cdot V^{-1}$ – это вектор коэффициентов полинома $D(\cdot)$ и, таким образом, $\vec{d}\, V^{-1}T$ – вектор коэффициентов полинома $C(\cdot)$ и $\vec{d}\, V^{-1}TV=\vec{c}$).

Пусть $G \subseteq [n]$ с $|G| \ge n-t$ и пусть $G=g_1,\ldots,g_{|G|}$. Матрица M_G строится следующим образом. Пусть матрица V^G размерностью $|G| \times |G|$ - это матрица V, спроектированная на индексы из матрицы G; а именно $V^G_{i,j}=(g_i)^j$.Далее строиться матрица $\widetilde{V}$ размерностью $|G| \times n$, присоединением n-$|G|$ нулевых столбцов к $(V^G)^{-1}$. В итоге установить $M_G=\widetilde{V}\,TV$, где T – определена выше.

Так как $n \geq 3t+1$ мы имеем $|G| \geq 2t+1$. Можно проверить, что в этом случае $\vec{x}_G \cdot \tilde{V}$ - тоже вектор коэффициентов полинома $D(\cdot)$, а именно $\vec{x}_G \cdot \tilde{V} = \vec{x} \cdot V^1$. Таким образом, $\vec{x}_G\ \tilde{V}\ TV = \vec{x}\ V^1 TV = \vec{x}\ M$. ■

Объединяя шаги рандомизации и редукции степени, мы получаем протокол для вычислений на мультипликативном вентиле. Этот протокол, обозначенный **MUL**, представлен ниже.

Субпротокол MUL(a_i,b_i)

Код для процессора P_i по входу a_i,b_i.

1. Установить (t)**АГПРС=**$(C', \{h_{i,j} \mid j \in C'\})$.
2. Пусть $d_i = a_i \cdot b_i + \sum_{j \in C'} h_{i,j}$.
3. Пусть $M_{n \times n}$ – матрица, определенная выше как M.
4. Установить (d_i, M)**MAT=**c_i.

Подать на выход c_i.

Основной протокол

Пусть $f{:}F^n \rightarrow F$ – арифметическая схема A. Нижеприведенный протокол безопасно t-вычисляет функцию f.

Протокол АВФ

Код для процессора P_i по локальному входу x_i и данной схеме A.

1. Установить (t,x_i)**АГПРС=**$(C, \{s_{i,j} \mid j \in C\})$. Для линии l в схеме пусть $l^{(i)}$ определяет долю процессора P_i, находящуюся на этой линии. Если l – j-тая входная линия схемы, тогда установить $l^{(i)} = s_{i,j}$, если $j \in G$ и $l^{(i)} = 0$ в противном случае.
2. Для каждого вентиля g в схеме, процессор P_i ожидает, пока i-тые доли всех входных линий вентиля g будут вычислены. Тогда он делает следующее.
   2.1. Если g – аддитивный вентиль с выходной линией l и входными линиями l_1 и l_2, тогда вычисляет $l^{(i)} = l_1^{(i)} + l_2^{(i)}$.
   2.2. Если g – аддитивный вентиль $l = l_1 \cdot l_2$, тогда вычисляет $l^{(i)} =$ **MUL**$(l_1^{(i)}, l_2^{(i)})$.
3. Пусть l_{out} – выходная линия схемы. Как только значение $l^{(i)}_{out}$ – вычислено, послать сообщение «Готово» всем другим процессорам.

4. Дождаться получения n-t сообщений «Готово» от других процессоров. Установить $(t,n,l^{(i)}_{out})$**Авс**=y.

Подать на выход (C,y).

Теорема 3.8. Пусть f:$F^n \rightarrow F$ для некоторого поля F с $|F|>n$ и пусть A – схема, вычисляющая функцию f. Тогда протокол **АВФ** асинхронно ($\lceil n/3 \rceil$-1)-безопасно вычисляет f в модели с ограничено защищенными каналами в условиях проявления FS-сбоев.

Доказательство. Приведено в работе [Са2].

Вычисления при Ву-сбоях

Как и в случае FS-сбоев нам необходимо вычислить функцию f:$F^n \rightarrow F$. Предположим, что процессоры имеют общую арифметическую схему для вычисления f. Ниже описывается n-сторонний протокол для безопасного t-вычисления f в асинхронной сети с произвольными (то есть, при Ву-сбоях) противниками, при условии $n \geq 4t+1$.

Основная идея заключается в адаптации вышеописанного протокола для FS-сбоев к аналогичному протоколу, но для *Ву*-сбоев. Для этого используем вышеописанную схему **АПРС**, а затем, этап умножения адаптируется к аналогичной конструкции, но для *Ву*-сбоев.

Пусть $c=a \cdot b$ – мультипликативный вентиль и пусть $A(\cdot)$, $B(\cdot)$ – полиномы, ассоциированные с входными линиями, а именно доли каждого процессора P_i этих линий - $A(i)$ и $B(i)$ соответственно и $A(0)=a$ и $B(0)=b$. Как и в случае FS-сбоев процессоры совместно вычисляют свои доли случайного полинома $C(\cdot)$ степени t, где $C(0)=A(0) \cdot B(0)$, так, что доля каждого несбоящего процессора P_i на выходной линии будет $C(i)$.

Процедура умножения при *Ву*-сбоях следует из соответствующей процедуры, но для FS-сбоев. А именно, процессоры сначала генерируют случайный полином $D(\cdot)$ степени $2t$ со свободными коэффициентами $D(0)=A(0) \cdot B(0)$. Тогда процессоры вычисляют свои доли усеченного полинома $D(\cdot)$ степени t и этот усеченный полином есть выходной полином $C(\cdot)$.

Собственно сама процедура умножения начинается с описания двух модифицированных шагов: умножения и редукции степени.

Рандомизация. Отличие от случая для FS-сбоев состоит в том, как каждый процессор P_i делит полином $H_i(\cdot)$ степени $2t$ с $H_i(0)=0$. Для этого используется следующий метод [BGW]. Каждый процессор P_i делит t равномерно выбранных значений, используя t вызовов субпротокола

АРзПр. Пусть $z_{i,j,k}$, - выход процессора P_k при j-том вызове **АРзПр**, где P_i - дилер. После завершения всех t вызовов **АРзПр**, каждый процессор P_k локально вычисляет $H_i(k)=\sum_{j=1}^{t} k^j \cdot z_{i,j,k}$.

Для этого пусть $S_{i,j}(\cdot)$ – полином степени t определенный j-тым вызовом **АРзПр**, инициированным P_i (а именно, $S_{i,j}(k)=z_{i,j,k}$ для каждого несбоящего процессора P_k). Полином $H_i(\cdot)$ теперь определен как $H_i(x)=\sum_{j=1}^{t} x^j \cdot S_{i,j}(x)$. Каждый процессор P_k локально вычисляет $H_i(k)=\sum_{j=1}^{t} k^j \cdot S_{i,j}(k) = \sum_{j=1}^{t} k^j \cdot z_{i,j,k}$. Пусть $s_{i,j,l}$ - коэффициент x^l в $S_{i,j}(x)$. Это можно показать как

$$
\begin{aligned}
H_i(x)= & \\
= & s_{i,1,0}x + s_{i,1,1}x^2 + \ldots + s_{i,1,t-1}x^t + s_{i,1,t}x^{t+1} + \\
& s_{i,2,0}x^2 + \ldots + s_{i,2,t-2}x^t + s_{i,2,t-1}x^{t+1} + s_{i,2,t}x^{t+2} + \\
& \ldots \\
& s_{i,t,0}x^t + s_{i,t,1}x^{t+1} + \ldots + s_{i,t,t}x^{2t}
\end{aligned}
$$

Свободный коэффициент $H_i(\cdot)$ равен 0 и, таким образом, $H_i(0)=0$. Каждый полином $S_{i,j}(\cdot)$ имеет степень t и, таким образом, $H_i(x)$ имеет степень $2t$. Кроме того, можно заметить, что коэффициенты членов $x,\ldots,x^t$ в $H_i(x)$ равномерно распределены над F. Следовательно, коэффициенты всех ненулевых степеней усеченного полинома $C(\cdot)$ равномерно распределены над F.

Ясно, что сбоящие процессоры накапливают некоторую дополнительную информацию относительно t долей $H_i(\cdot)$. Однако эта информация независима от $A(0)\cdot B(0)$, так как процессор P_i выбирает t^2+t случайных коэффициентов, а сбоящие процессоры получает только t^2 значений.

Редукция степени. Процессоры используют свои доли полинома $D(\cdot)$ для совместного и конфиденциального вычисления своих долей «усечения» полинома $D(\cdot)$ степени t. А именно $t+1$ коэффициентов выходного полинома $C(\cdot)$ являются коэффициентами $D(\cdot)$ с более низкой степенью.

В протоколе для FS-сбоев процессоры вычисляли свои доли $C(\cdot)$, вызывая протокол для «умножения вектора входов на фиксированную матрицу».

В протоколе для By-сбоев процессоры используют субпротоколы **АРзПр** и **АВсПр** вместо простой схемы разделения секрета для FS-сбоев. Однако проблема заключается в том, что согласованное множество G может содержать сбоящие процессоры P_i, совместно использующие некоторое значение, отличное от ожидаемого значения $D(i)$. В этом случае, процессоры не будут иметь требуемых выходов.

Далее мы описываем, как процессоры P_i могут удостовериться в том, что значение, связанное с каждым процессором P_i в согласованном множестве (а именно, свободный коэффициент полинома степени t, определенного долями несбоящих процессоров P_i) – действительно $D(i)$.

Для процессоров P_i пусть s_i - значение, связанное с P_i; для множества A процессоров, пусть $S_A=f\{(i,s_i) \mid P_i \in A\}$. Сначала отметим, что достаточно договориться о множестве G из, по крайней мере, $3t+1$ процессоров, такое, что S_G является $(2t,0)$-интерполируемым (а именно, все значения, общедоступные для процессоров из G «находятся на полиноме степени $2t$»). Это так, потому, что множество G размером $3t + 1$, содержит, по крайней мере, $2t+1$ несбоящих процессоров. Таким образом, интерполируемый полином S_G связан (ограничен) с полиномом $D(\cdot)$.

Далее описывается протокол для соглашения по множеству A процессоров такому, что S_A является $(2t,0)$-интерполируемым. Этот протокол, обозначаемый **СОИМ** - *Соглашения об интерполируемом множестве*, является «распределенным вариантом» схемы **СКОП**, описанной выше.

Протокол **СОИМ** состоит из t итераций. На итерации r $(0 \le r \le t)$, процессоры сначала используют протокол **СОАМ** (см. выше) чтобы договориться о множестве G_r, состоящее из, по крайней мере, $3t+1+r$ процессоров, которые успешно объединяют свои входы. Затем, стороны выполняют вычисления, описываемые ниже, для проверки, является ли множество S_{G_r} $(2t,r)$-интерполируемым. Если S_{G_r} является $(2t,r)$-интерполируемым, тогда существует множество $G_r' \subseteq G_r$ размером из, по крайней мере, $3t+1$ процессоров такое, что S_{G_r} является $(2t,0)$-интерполируемым. Тогда процессоры вычисляют и выдают G_r'. В противном случае (т.е., когда S_{G_r} - не является $(2t,r)$-интерполируемым),

процессоры переходят к следующей итерации. Подчеркиваем, что стороны не будут знать интерполируемый полином для каждого S_{G_r}. Они будут только знать, что такой полином существует.

Остается только описать, как проверить по данному множеству G размера $3t+1+r$ является ли S_G $(2t,r)$-интерполируемым, и как вычислить соответствующее множество G' (то есть, $G'\subseteq G$, $|G'|\geq 3t+1$ и $S_{G'}$ является $(2t,0)$-интерполируем). Как и в процедуре **СКОП**, мы используем обобщенные коды с исправлением ошибок Рида-Соломона. Однако, в процедуре **СКОП** «слово» S_G, было (динамическим) входом для одного процессора и, таким образом, каждый процессор мог бы локально запускать процедуру, реализующую схему с исправлением ошибок. В данной установке, каждый процессор имеет только одну долю каждого элемента S_G; стороны будут вызывать совместные вычисления, выполняющее специфическую процедуру, реализующую схему с исправлением ошибок и использовать это для проверки, является ли G - $(2t,r)$-интерполируемым и вычисления G'.

Сначала опишем частичную процедуру с исправлением ошибок. Входы для этой процедуре - (d,r,W). Если $|W|\geq d+2r+1$ и W является (d,r)-интерполируем, тогда выход - интерполируемый полином W. (В противном случае, выводится соответствующее сообщение). Процедура состоит из трех шагов.

<u>Процедура СОИМ</u>

Вычисление синдрома. Для входного слова $W=(i_1,s_1)...(i_l,s_l)$, пусть $V_{l\times l}$ – матрица Вандермонда (см., например, [Кн, стр.474]), определенная как $V_{j,k}=(i_k)^j$. Пусть $\vec{a}=s_1...s_l$. Синдром W есть l-$(d+1)$ наибольших правых элементов l-размерного вектора произведения $\vec{a}\cdot V^{-1}$. Таким образом, на этом шаге вычисляется синдром W.

Вычисление вектора погрешности. Вектор погрешности – это l-размерный вектор $\vec{e}=e_1...e_l$, где e_j – «смещение s_j от правильного значения». А именно, предположим, что W - (d,r)-интерполируем и пусть $P(\cdot)$ - (d,r)-интерполируемый полином W. Тогда $e_j=P(i_j)$-s_j алгоритм для каждого элемента $(i_j,s_j)\in W$. Вектор погрешности уникален, так как интерполируемый полином $P(\cdot)$ – уникален. (Отметим, что вектор погрешности может быть вычислен только на основании синдрома. Если входное слово W -

не является (d,r)-интерполируемым, тогда вычисленный вектор погрешности может быть ошибочным).

Вычисление выходного полинома. Выбрать $2t+1$ корректных элементов из W (а именно, элементы (i_j,a_j) такие, что $e_j=0$) для использования их, чтобы интерполировать $P(\cdot)$. (Этот шаг не будет выполнен).

Важно отметить, что синдром может быть представлен как функция только от вектора погрешности и, таким образом, он не содержит никакой информации относительно (d,r)-интерполируемого полинома $P(\cdot)$. А именно, пусть $\vec{P}$ (в отн. $\vec{Q}$) - вектор коэффициентов полинома $P(\cdot)$ (в отн. $Q(\cdot)$) длины l. (Полином $Q(\cdot)$ - полином, удовлетворяющий $Q(i)=a$ для каждой пары $(i,a)\in W$). Тогда $\vec{Q}=\vec{a}\cdot V^{-1}=(\vec{P}\cdot V+\vec{e})\cdot V^{-1}=\vec{P}+\vec{e}\cdot V^{-1}$.

Последние l-$(d+1)$ элементов из $\vec{P}$ является нулевыми. Так как $l-(d+1)<i\leq l$, мы имеем $Q_i=[\vec{e}\cdot V]_i$. Следовательно, последние l-$(d+1)$ элементов из $\vec{Q}$ (т.е. синдром) являются линейной комбинацией $\vec{e}$ и, таким образом, процессоры могут совместно вычислять синдром. То есть для данного согласованного множества G, каждый элемент синдрома S_G вычисляется следующим образом. Каждый процессор вычисляет соответствующую линейную комбинацию долей и вызывает субпротокол **АВсПр** с результатом этой линейной комбинации как входом. Как только все эти субпротоколы завершаться, каждый процессор будет иметь полный синдром S_G.

Как только синдром вычислен, каждый процессор использует алгоритм Берлекампа-Месси, чтобы локально вычислить вектор погрешности. Если на итерации r S_{G_r} является $(2t,r)$-интерполируем, тогда вычисленный вектор погрешности является «истинным» вектором погрешности множества S_{G_r}, однако, если S_{G_r} не является $(2t,r)$-интерполируемым, тогда вычисленный вектор погрешности может быть некорректным. Следовательно, процессоры должны выполнить следующие действия. (Каждый выполняет эти операции локально, а все несбоящие процессоры выполняют одни и те же действия). Если вычисленный вектор погрешности, обозначаемый $\vec{e}\,'$ содержит более чем r ненулевых элементов, то несомненно S_{G_r} не является $(2t,r)$-интерполируемым и процессоры переходят к следующей итерации. Если $\vec{e}\,'$ содержит не более

r ненулевых элементов, то процессоры все еще должны проверять, что $\vec{e}\,'$ - корректный вектор погрешности. Для этого пусть G_r' - множество процессоров P_i такое, что $e_i'=0$, тогда процессоры повторно вычисляют синдром, основанный только на G_r'. Если повторно вычисленный синдром представляет собой одни нули, тогда $S_{G_r'}$ является $(2t,0)$-интерполируем и процессоры выдают G_r' и заканчивают. Если повторно вычисленный синдром ненулевой, тогда процессоры заключают, что S_{G_r} - не является $(2t,r)$-интерполируемым и переходят к следующей итерации.

Далее опишем непосредственно протокол **СОИМ**. Пусть $z_{i,j}$ - доля P_i' значения общего с P_j. Динамический вход каждого процессора P_i является следующим аккумулируемым множеством, обозначаемым как Z_i: как только P_i успешно завершил субпротокол **АРзПр** для P_j, пара $(j,z_{i,j})$ добавляется к Z_i. Общий выход сторон – это множество G из, по крайней мере, $3t+1$ процессоров такое, что каждый несбоящий процессор P_i завершает субпротокол **АРзПр** для каждой стороны из G (а именно, $G\subseteq\{P_j \mid (j,z_{i,j})\in Z_i\}$) и S_G – является $(2t,0)$-интерполируем.

Утверждение 3.1. Предположим, что протокол **СОИМ** инициализируется с динамическими входами $Z_1,...,Z_n$ как описано выше. Тогда все несбоящие процессоры завершают протокол **СОИМ** с общим множеством G из, по крайней мере, $3t+1$ процессоров, такое, что S_G является $(2t,0)$-интерполируемым.

Доказательство. Приведено в работе [Ca1].

Протокол СОИМ(Z_i)

Код для процессора P_i на динамическом входе Z_i.

Выполнить для $0\le r\le t$.

1. Пусть $U_i=\{P_j \mid (j,z_{i,j})\in Z_i\}$. Установить (t,r,n,U_i)**СОБМ=**G.
2. Как только G вычислено, вычислять синдром S_G.

 2.1. Пусть V - матрица индексов Вандермонда из G. А именно пусть $G=k_1...k_{|G|}$, тогда $V_{i,j}=(k_j)^i$.

 2.2. Пусть $\vec{z}_i = z_{i,k_1},...,z_{i,|G|}$.

 2.3. Для $2t+1<j\le |G|$ установить $([\vec{z}_i\cdot V^1]_j,[n])$**АВсПр=**$\sigma_j$.

 2.4. Пусть $\vec{\sigma} = \sigma_1...\sigma_{t+r}$, где σ - синдром S_G.

3. Инициализировать алгоритм Берлекампа-Месси для $\vec{\sigma}$ и пусть $\vec{e}\,'$ - выход.

3.1. Если $\vec{e}\,'$ имеет более чем r ненулевых элементов, продолжить следующую итерацию (в этом случае S_G, не является $(2t,r)$-интерполируемым).

3.2. Если $\vec{e}\,'$ имеет не более чем r ненулевых элементов, проверить, что $\vec{e}\,'$ корректен.

3.2.1. Пусть G' - множество процессоров из G, чьи соответствующие записи в $\vec{e}\,'$ являются нулевыми. Повторить шаг 2 в отношении G'.

3.2.2. Если синдром $S_{G'}$ является нулевым вектором, выдать G' и остановиться. В противном случае перейти к следующей итерации (S_G не является $(2t,r)$-интерполируемым).

При объединении этапов рандомизации и редукции степени, мы получаем протокол для мультипликативного вентиля.

<u>Протокол *By*MUL</u>

Код для процессора P_i, на входах a_i и b_i.

1. Рандомизация.
   1.1. Для $1 \le k \le t$ выполнить **АРзПр**$_{i,k}$ по входу r_k, где $r_k \in_R F$ и P_i - дилер.
   1.2. Для $1 \le j \le n$ и для $1 \le k \le t$ участвуют в субпротоколах **АРзПр**$_{j,k}$.
   1.3. Пусть $h_{i,j,k}$ – выход процессора P_i для субпротокола **АРзПр**$_{j,k}$.
   1.4. Пусть $U_i = \{P_j \mid$ **АРзПр**$_{j,k}$ завершен для всех $1 \le k \le n\}$.
   1.5. Установить (n,t,U_i)**СОАМ=**G.
   1.6. Установить $h_i \sum_{j \in G} \sum_{k=1}^{t} i^k \cdot h_{i,j,k}$ и $d_i = a_i \cdot b_i + h_i$.
2. Редукция степени.
   2.1. Как только d_i вычислено, выполнить **АРзПр**$_i(d_i)$, где P_i - дилер. Для $1 \le j \le n$ участвовать в субпротоколе **АРзПр**$_j$.
   2.2. Пусть $z_{i,j}$ – доля (процессора P_i) общего с P_j секрета и пусть $Z_i' = \{(j, z_{i,j}) \mid$ **АРзПр**$_j$ был завершен$\}$. Установить (Z_i)**СОИМ=**G'.
   2.3. Пусть $V^{G'}$ - матрица, используемая на шаге умножения для FS-сбоев и пусть $\vec{z}_i = z_{i,j_1}, \ldots, z_{i,j_{|G'|}}$,

где $j_1 ... j_{|G'|}$ - индексы процессоров из G'. Для $1 \leq j \leq n$ установить $([\vec{z}^{\,i}_{\mathcal{C}} \cdot V^{G''}]_j, \{j\})$**АРзПр**=$c_j$.

2.4. Как только c_i вычислен, выдать c_i.

Полная структура протокола для *By*-сбоев точно такая же, как для *FS*-сбоев. А именно, в таком протоколе, обозначаемом ***By*Выч**, процессоры выполняют аналогичные инструкции протокола для *FS*-сбоев, за исключением того, что протоколы **АГРз**, **MUL**, и **АГВс** заменены на протоколы **АГПРС**, **ByMUL** и **АВсПр** соответственно.

Теорема 3.9. Пусть $f: F^n \to F$ для некоторой области F с $|F| > n$ и пусть A - схема, вычисляющая f. Тогда протокол ***By*Выч** по входу A асинхронно $(\lceil n/4 \rceil - 1)$-безопасно вычисляет f в установке с ограниченно защищенными каналами и в присутствии адаптивных противников.

Доказательство. Приведено в работе [Ca2].

3.8. RL-ПРОТОТИП МОДЕЛИ СИНХРОННЫХ КОНФИДЕНЦИАЛЬНЫХ ВЫЧИСЛЕНИЙ

Зададимся вопросом «Существует ли реальный вычислительный аналог представленной модели синхронных конфиденциальных вычислений ?». Такой вопрос важен и с прикладной, и с содержательной точки зрения.

Рассмотрим многопроцессорную суперЭВМ семейства **S-1** на базе сверхбыстродействующих процессоров MARK IIA (MARK V). Такую вычислительную систему назовем *RL*-прототипом (real-life) модели синхронных конфиденциальных вычислений в реальном сценарии.

Проект семейства систем высокой производительности для военных и научных применений (семейства **S-1**) является в США частью общей программы развития передовых направлений в области числовой обработки информации. Исходя из целей программы, можно сделать вывод о возможности применения указанной вычислительной системы в автоматизированных системах критических приложений. Поэтому требования высокой надежности и безопасности программного обеспечения систем являются обязательными.

В указанной многопроцессорной системе используются специально разработанные однопроцессорные машины, образующие семейство, то есть имеющие однотипную архитектуру. В систему может входить до 16

однопроцессорных ЭВМ, сравнимых по производительности с наиболее мощными из существующих суперЭВМ. В дополнение к обычному универсальному оборудованию процессоры семейства **S-1** оснащены специализированными устройствами, позволяющими выполнять высокоскоростные вычисления в тех прикладных областях, где планируется использование данной многопроцессорной системы. К таким операциям относятся вычисления функций синуса, косинуса, возведения в степень, логарифмирования, операции быстрого преобразования Фурье и фильтрации, операции умножения над матрицами и транспонирования.

Системы семейства **S-1** предоставляют в распоряжение пользователя большую сегментированную память с виртуальной адресацией в несколько миллиардов 9-разрядных байтов. Процессоры соединены между собой с помощью матричного переключателя, который обеспечивает прямую связь каждого процессора с каждым блоком памяти (см. рис.3.1). При обращениях к той или иной ячейке памяти процессоры всегда получают последнее записанное в ней значение. Команды выполняются в последовательности: «чтение - операция – запись». С целью повышения быстродействия памяти в составе каждого процессора имеются две кэш-памяти: первая – объемом 64 Кбайт для хранения данных, вторая – объемом 16 Кбайт (с перспективой наращивания). В обоих типах кэш-памяти длина набора составляет 4, а длина строки 64 байта.

В операционной системе **Amber**, предназначенной для вычислительных систем семейства **S-1**, предусмотрена программа планировщик, который на нижнем уровне архитектуры системы обеспечивает эффективный механизм оперативного планирования заданий на одном процессоре. Основным правилом планирования здесь является назначения в порядке очереди. На уровне такого одноприоритетного планирования каждое задание выполняется до тех пор, пока не возникает необходимость ожидания какого-либо внешнего события или не истечет выделенный квант процессорного времени. Планировщик высокого уровня осуществляет более сложное планирование, в основу которого положена некоторая общая стратегия. Результатом его работы являются соответствующим образом измененные параметры планировщика нижнего уровня: приоритеты заданий или размеры квантов времени.

Одной из важнейших особенностей многопроцессорной системы семейства S-1 является наличие общей памяти большой емкости,

позволяющей осуществлять широкомасштабный обмен данными между заданиями. Если два задания работают с одним и тем же сегментом памяти, они пользуются его единственной физической копией, в то время как другие области пространства адресов остаются в их собственном владении. Таким образом, задания оказываются защищенными от неосторожных попыток изменить их внутренне состояние. Между двумя заданиями можно установить жесткий режим чтения-записи, когда одному заданию разрешаются операции чтения-записи, а другому только операции чтения из данного сегмента.

Операционная система **Amber** имеет большие возможности для реконфигурации системы в случае возникновения сбоев (внештатных ситуаций). Если требуется вывести из конфигурации процессор, выполнение на нем приостанавливается и производится перераспределение процессоров на другие процессоры. Когда процессор или память вводятся в рабочую конфигурацию, они становятся обычными ресурсами системы и загружаются по мере необходимости.

Таким образом, можно проследить широкий круг аналогий между моделью конфиденциальных вычислений и ее вычислительным прототипом. В этом случае центральный коммутатор совместно с устройствами памяти можно рассматривать и как широковещательный канал, и как набор конфиденциальных каналов связи между процессорами. Конфиденциальность каналов может рассматриваться в том случае, если существует возможность предотвратить несанкционированный доступ к блокам памяти или хранить и передавать данные в зашифрованном виде. Привязка во времени многопроцессорной системы **S-1** осуществляется посредством устройства

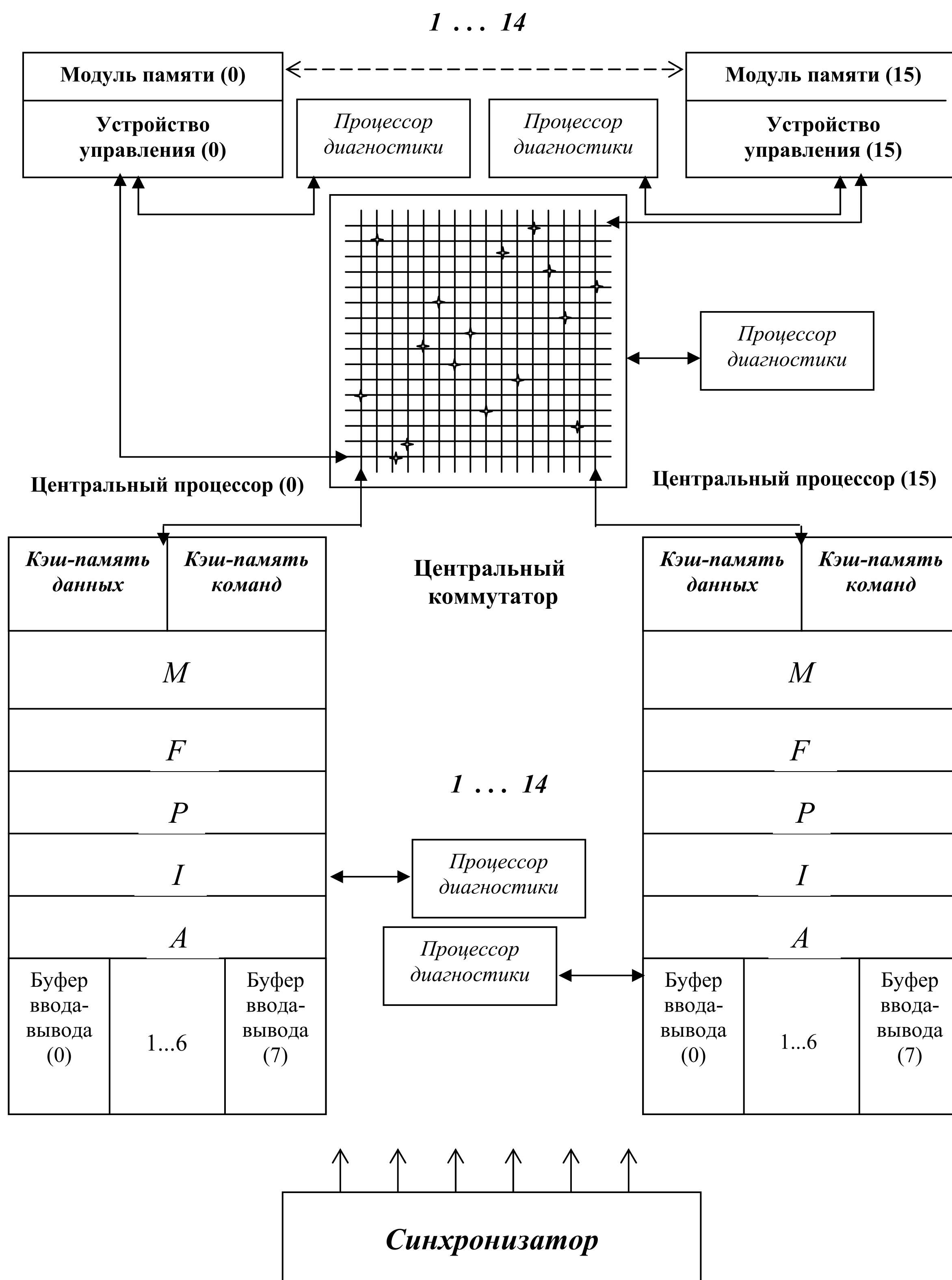


Рис. 3.1. *RL*-прототип модели синхронных конфиденциальных вычислений

синхронизации. Параметром безопасности системы может являться длина строки (64-256 Кбайт).

Вычислительная система типа **S-1** позволяет осуществлять разработку прототипов распределенных вычислительных систем и исследование характеристик многосторонних протоколов различного типа, как криптографического характера, так и нет. К такой разновидности распределенных вычислений можно отнести следующие протоколы, имеющие, в том числе, и прикладное значение (определения даны в этой главе и приложении).

1. *Протоколы византийского соглашения*.
2. *Протоколы разделения секрета*.
3. *Протоколы электронного голосования*.
4. *Протоколы выработки общей случайной строки* и многие другие.

Таким образом, методы конфиденциальных вычислений могут позволить для многопроцессорных систем проектировать высокозащищенную программно-аппаратную среду для использования в автоматизированных системах различных объектов информатизации.

ГЛАВА 4. САМОТЕСТИРУЮЩИЕСЯ И САМОКОРРЕКТИРУЮЩИЕСЯ ПРОГРАММЫ

4.1. Вводные замечания

Основополагающей работой в области *проверки программ*, использующих некоторые присущие им внутренние свойства для контроля результатов своей работы, следует считать, написанную в 1989 г., работу [BK], в которой были формализованы основные идеи построения *программных чекеров*. Соответствующие определения *самотестирующихся и самокорректирующихся программ* в 1993 г. были введены в работах [BLR,L1]. В свою очередь, методология самотестирования явилась результатом объединения трех основных идей из криптографии, вероятностных алгоритмов и тестирования программ [BK]. К ним относятся интерактивные системы доказательств и интерактивные системы доказательств с нулевым разглашением [GMR]. Кроме этого, большое значение имели работы М.О. Рабина по вероятностным вычислениям (см., например, [Ра]) и работа латышского математика Р. Фрейвалдса [F], написанная им еще в 1979 году. Он предложил простой и элегантный чекер для задачи умножения матриц, который можно также эффективно применить и для целочисленного умножения и для умножения полиномов.

4.2. Общие принципы создания двухмодульных вычислительных процедур и методология самотестирования

Пусть необходимо написать программу P для вычисления функции f так, чтобы $P(x)=f(x)$ для всех значений x. Традиционные методы верификационного анализа и тестирования программ не позволяют убедиться с вероятностью близкой к единице в корректности результата выполнения программы, хотя бы потому, что тестовый набор входных данных, как правило, не перекрывают весь их возможный спектр. Один из методов решения данной проблемы заключается в создании так называемых самокорректирующихся и самотестирующихся программ, которые позволяют оценить вероятность некорректности результата выполнения программы, то есть, что $P(x)\neq f(x)$ и корректно вычислить $f(x)$

для любых x, в том случае, если сама программа P на большинстве наборах своих входных данных (но не всех) работает корректно.

Чтобы добиться корректного результата выполнения программы P, вычисляющей функцию f, нам необходимо написать такую программу T_f, которая позволяла бы оценить вероятность того, что $P(x) \neq f(x)$ для любых x. Такая вероятность будет называться *вероятностью ошибки* выполнения программы P. При этом T_f может обращаться к P как к своей подпрограмме.

Обязательным условием для программы T_f является ее принципиальное *временное отличие* от любой корректной программы вычисления функции f, в том смысле, что время выполнения программы T_f, не учитывающее время вызовов программы P, должно быть значительно меньше, чем время выполнения любой корректной программы для вычисления f. В этом случае, вычисления согласно T_f некоторым количественным образом должны отличаться от вычислений функции f и *самотестирующаяся программа* может рассматриваться как независимый шаг при верификации программы P, которая предположительно вычисляет функцию f. Кроме того, желательное свойство для T_f должно заключаться в том, чтобы ее код был насколько это возможно более простым, то есть T_f должна быть *эффективной* в том смысле, что время выполнения T_f даже с учетом времени, затраченного на вызовы P должно составлять константный мультипликативный фактор от времени выполнения P. Таким образом, самотестирование должно лишь незначительно замедлять время выполнения программы P.

Пусть π - означает некоторую вычислительную задачу и/или некоторую задачу поиска решения. Для x, рассматриваемого в качестве входа задачи, пусть $\pi(x)$ обозначает результат решения задачи π. Пусть P – программа (предположительно предназначенная) для решения задачи π, которая останавливается (например, не имеет зацикливаний) на всех входах задачи π. Будем говорить, что P *имеет дефект*, если для некоторого входа x задачи π имеет место $P(x) \neq \pi(x)$.

Определим (*эффективный*) *программный чекер* C_π для задачи π следующим образом. Чекер $C_\pi^P(I,k)$ – является произвольной вероятностной машиной Тьюринга, удовлетворяющей следующим условиям. Для любой программы P (предположительно решающей задачу π), выполняемой на всех входах задачи π, для любого элемента I

задачи π и для любого положительного k (параметра безопасности) имеет место:

- если программа P не имеет дефектов, т.е. $P(x)=\pi(x)$ для всех входов x задачи π, тогда $C_\pi^P(I,k)$ выдаст на выходе ответ «Норма» с вероятностью не менее $1-1/2^k$;
- если программа P имеет дефекты, т.е. $P(x)\neq\pi(x)$ для всех входов x задачи π, тогда $C_\pi^P(I,k)$ выдаст на выходе ответ «Сбой» с вероятностью не менее $1-1/2^k$.

Самокорректирующаяся программа – это вероятностная программа C_f, которая помогает программе P скорректировать саму себя, если только P выдает корректный результат с низкой вероятностью ошибки, то есть для любого x, C_f вызывает программу P для корректного вычисления $f(x)$, в то время как собственно сама P обладает низкой вероятностью ошибки.

Самотестирующейся/самокорректирующейся программной парой называется пара программ вида (T_f,C_f). Предположим, что пользователь может взять любую программу P, которая целенаправленно вычисляет f и тестирует саму себя при помощи программы T_f. Если P проходит такие тесты, тогда по любому x, пользователь может вызвать программу C_f, которая, в свою очередь, вызывает P для корректного вычисления $f(x)$. Даже если программа P, которая вычисляет значение функции f некорректно для некоторой небольшой доли входных значений, ее в данном случае все равно можно уверенно использовать для корректного вычисления $f(x)$ для любого x. Кроме того, если удастся в будущем написать программу P' для вычисления f, тогда некоторая пара (T_f,C_f) может использоваться для самотестирования и самокоррекции P' без какой-либо ее модификации. Таким образом, имеет смысл тратить определенное количество времени для разработки самотестирующейся/ самокорректирующейся программной пары для прикладных вычислительных функций.

Перед тем как перейти к более формальному описанию определений самотестирующихся и самокорректирующихся программ необходимо дать определение вероятностной оракульной программе (по аналогии с вероятностной оракульной машиной Тьюринга).

Вероятностная программа M является *вероятностной оракульной программой*, если она может вызывать другую программу, которая является исполнимой во время выполнения M. Обозначение M^A означает, что M может делать вызовы программы A.

Пусть P - программа, которая предположительно вычисляет функцию f. Пусть I является объединением подмножеств I_n, где n принадлежит множеству натуральных чисел N и пусть $D^p=\{D_n|n\in N\}$ есть множество распределений вероятностей D_n над I_n. Далее, пусть $err(P,f,D_n)$ – это вероятность того, что $P(x)\neq f(x)$, где x выбрано случайно в соответствии с распределением D_n из подмножества I_n. Пусть β - есть некоторый параметр безопасности. Тогда $(\varepsilon_1,\varepsilon_2)$-*самотестирующейся программой* для функции f в отношении D^p с параметрами $0\leq\varepsilon_1<\varepsilon_2\leq 1$ - называется вероятностная оракульная программа T_f, которая для параметра безопасности β и любой программы P на входе n имеет следующие свойства:

- если $err(P,f,D_n)\leq\varepsilon_1$, тогда программа T_f^P выдаст на выходе ответ «норма» с вероятностью не менее $1-\beta$.
- если $err(P,f,D_n)\geq\varepsilon_2$, тогда программа T_f^P выдаст на выходе «сбой» с вероятностью не менее $1-\beta$.

Оракульная программа C_f с параметром $0\leq\varepsilon<1$ называется ε-*самокорректирующейся программой* для функции f в отношении множества распределений D^p, которая имеет следующее свойство по входу n, $x\in I_n$ и β. Если $err(P,f,D_n)\leq\varepsilon$, тогда $C_f^P=f(x)$ с вероятностью не менее $1-\beta$.

$(\varepsilon_1,\varepsilon_2,\varepsilon)$-*самотестирующейся/ самокорректирующейся программной парой* для функции f называется пара вероятностных программ (T_f,C_f) такая, что существуют константы $0\leq\varepsilon_1<\varepsilon_2\leq\varepsilon<1$ и множество распределений D^p при которых T_f -есть $(\varepsilon_1,\varepsilon_2)$-самотестирующаяся программа для функции f в отношении D^p и C_f - есть ε-самокорректирующаяся программа для функции f в отношении распределения D^p.

Свойство случайной самосводимости. Пусть $x\in I_n$ и пусть $c>1$ - есть целое число. Свойство случайной самосводимости заключается в том, что существует алгоритм A_1, работающий за время пропорциональное $n^{O(1)}$, посредством которого функция $f(x)$ может быть выражена через вычислимую функцию F от x, $a_1,...,a_c$ и $f(a_1),...,f(a_c)$ и алгоритм A_2, работающий за время пропорциональное $n^{O(1)}$, посредством которого по данному x можно вычислить $a_1,...,a_c$, где каждое a_i является случайно распределенным над I_n в соответствии с D^p.

4.3. Устойчивость, линейная и единичная состоятельность

Пусть свойство I выражается уравнением $I(x_1,...,x_k)=0$, где кортеж $(\langle x_1,...,x_k)$ выбирается с распределением E из пространства D^k. Пара (I,E) характеризует семейство функций F, где $f\in F$ тогда и только тогда, когда для всех $(x_1,...,x_k)$ с ненулевой выборкой элементов кортежа из E, $I^f(x_1,...,x_k)=0$. Базовой техникой самотестирования является идентификация свойства *устойчивости* для семейства функций F. Неформально (D,D')-*устойчивость* пары (I,E) для семейства функций G реализует, что если для программы $P\in G$, свойство $I^P(x_1,...,x_k)=0$ удовлетворяется с высокой вероятностью, когда $\langle x_1,...,x_k\rangle$ выбрано с распределением E из D^k, тогда существует функция $g\in F\cap G$, которая согласуется с P на большей части входов из D'.

Рассмотрим некоторое свойство линейности (I,E), где свойство $I^f(x_1,x_2,x_3)$ тождественно $f(x_1)+f(x_2)=f(x_3)$ и E означает $(x_1\in_{\mathrm{R}}Z_p,x_2\in_{\mathrm{R}}Z_p,x_1+x_2)$. Пара (I,E) характеризует $F=\{f(x)=cx \mid c\in Z_p\}$ – множество всех линейных функций над Z_p. В этом примере G – тривиальное множество всех функций и пара (I,E) устойчива для G.

Как только мы убедились посредством рандомизированных попыток, что программа P удовлетворяет свойству устойчивости, можно переходить к тестированию программы на линейную и единичную состоятельность.

Существует два базовых теста для самотестирующихся программ – *тест линейной состоятельности* и *тест единичной состоятельности* [BLR]. Продемонстрируем построение таких тестов на примере следующей тривиальной модулярной функции. Пусть x,R – положительные целые, тогда $f_R(x)\equiv x \pmod R$, где R - фиксировано.

Пусть x_1 и x_2 случайно, равновероятно и независимо от других событий выбраны из Z_{R2^n} и x принимает значение: $x{:}\equiv x_1+x_2 \pmod{R2^n}$. Необходимо отметить, что $f_R(x)\equiv[f_R(x_1)+f_R(x_2)] \pmod R$ – линейная функция по всем входам (аргументам). Тогда *тест линейной состоятельности* заключается в выполнении или не выполнении равенства: $P_R(x)\equiv[P_R(x_1)+P_R(x_2)] \pmod R$, а *ошибка линейной состоятельности* – есть вероятность того, что данный тест не выполнился.

Пусть z случайно выбрано из Z_{R2^n} в соответствии с распределением $U_{Z_{R2^n}}$ и z принимает значение: $z'{:}\equiv z+1 \pmod{R2^n}$. Отметим также, что $f_R(z')\equiv[f_R(z)+1] \pmod R$. Тогда *тест единичной состоятельности* заключается в выполнении или не выполнении равенства:

$P_R(z')\equiv[P_R(z)+1](\mathrm{mod}\,R)$, а *ошибка единичной состоятельности* – есть вероятность того, что данный тест не выполнился.

4.4. Метод создания самокорректирующейся процедуры вычисления теоретико-числовой функции дискретного экспоненцирования

4.4.1. Обозначения и определения для функции дискретного возведения в степень вида g^xmodulo M.

Пусть $I_n=Z_q$ представляет собой множество $\{1,...,q\}$, где $q=\varphi(M)$ - значение функции Эйлера для модуля M, а Z^*_M - множество вычетов по модулю M, где $n=\lceil \log_2 M \rceil$. Пусть распределение D^p есть равномерное распределение вероятностей. Оракульной программой, в данном случае, является программа вычисления функции g^xmodulo$M$, по отношению к исследуемым самотестирующейся и самокорректирующейся программам.

Алгоритм A^xmodulo N можно вычислить многими способами [Кн и др.], один из наиболее общеизвестных и применяемых на практике, - это алгоритм, основанный на считывании индекса (значения степени) слева направо. Этот метод достаточно прост и нагляден, история его восходит еще к 200 г. до н.э. Он еще также известен как русский крестьянский метод.

Пусть $x[1,...,n]$ - двоичное представление положительного числа x и A, B и N - положительные целые числа в r-ичной системе счисления, тогда псевдокод алгоритма A^xmodulo N, реализованного программой Exp(•) имеет следующий вид.

```
Program Exp(x,N,A,R); {вход x,N,A, выход R}
 begin
     B:=1;
     for i:=1 to n do
        begin
          B:=(B*B)modN;
          if [i]=1 then B:=(A*B)modN;
        end;
     R:=B;
 end.
```

Рис. 4.1. Псевдокод алгоритма A^xmodulo$N$

Из описания алгоритма видно, что число обращений к операции вида (A*B)modulo$N$ будет $\log x+\beta(x)$, где $\beta(x)$ число единиц в двоичном представлении операнда x или вес Хэмминга x.

4.4.2. Построение самотестирующейся/самокорректирующейся программной пары для функции дискретного экспоненцирования

Сначала рассмотрим следующие 4 алгоритма (см. рис.4.2 - 4.5). Для доказательства полноты и безопасности указанной самотестирующейся/ самокорректирующейся программной пары доказывается следующая теорема.

Теорема 4.1. Пара программ S_K_exp(x,M,q,g,β) и S_T_exp(x,M,q,g,β) является (1/288,1/8,1/8)-самокорректирующейся/ самотестирующейся программной парой для функции g^xmodulo$M$ с входными значениями, выбранными случайно и равновероятно из множества I_n.

Для доказательства теоремы необходимо доказать две леммы.

Лемма 4.1. Программа S_K_exp(M,q,g,β) является (1/8)-самокорректирующейся программой для вычисления функции g^xmodulo$M$ в отношении равномерного распределения D_n.

Доказательство. Полнота программы S_K(·) означает, что если оракульная программа $P(x)$, обозначаемая как Exp(·) выполняется корректно, то и самокорректирующаяся программа S_K(·) будет выполняться корректно. В данном случае полнота программы очевидна. Если $P(x)$ корректно вычислима, то из $[P_{M,g}(x_1)\cdot P_{M,g}(x_2)](\mathrm{mod}M)$ следует, что $f_{M,g}(x)=f_{M,g}(x_1)\circ f_{M,g}(x_2)= g^{[x_1+x_2](\mathrm{mod}\varphi(M))}(\mathrm{mod}M) \equiv g^x(\mathrm{mod}\ M)\equiv R_k$.

```
Program S_K_exp(x,M,q,g, Rk); {вход n,x,M,q,g, выход Rk}
  begin
    for l=1 to 12ln(1/β)
      begin
        x1:=random(q); {random- функция случайного
                                равновероятного выбора из
                                целочисленного отрезка
                                [1,...,q-1]}
        x2:=(x-x1)modq;
        Exp(g,x1,M,R1); {Exp-   процедура вычисления
                                g^x moduloM=R}
        Exp(g,x2,M,R2);
        R0:=(R1·R2)modM;
      end;
        Rk=choice(R0(l)); {choice- функция выбора из массива,
                                состоящего из 12ln(1/β)
                                элементов, ответов, который
```

```
                                      повторяется наибольшее
                                      количество раз}
  end.
```

Рис. 4.2. Псевдокод алгоритма S_K_exp

```
Program S_T_exp(x,M,q,g,β); {вход x,M,q,g, выход значение
                                   предиката output}
  begin
    t1:=0;t2:=0;
    for l=1 to ⌈576ln(4/β)⌉
      begin
        L_T(g,M,q,Rl); {L_T -  процедура, реализующая
                               тест линейной
                               состоятельности, выход -
                               Rl}
        t1:=t1+Rl;
      end;
    if t1/⌈576ln(4/β)⌉>1/72 then output:=«false»;
    for l=1 to ⌈32(4/β)⌉
      begin
        N_T(g,M,q,Re); {N-T -  процедура, реализующая
                               тест единичной
                               состоятельности, выход -
                               Re}
        t2:=t2+R;
      end;
    if t2/⌈32ln(4/β)⌉>1/4 then output:= «false»
    else output:=«true»
  end.
```

Рис. 4.3. Псевдокод алгоритма S_T_exp

```
Program L_T(n,R); {вход g,M,q, выход Rl}
  begin
    x1:=random(q);
    x2:=random(q);
    x:=(x1+x2)modq;
    Exp(g,x1,M,R1);
    Exp(g,x2,M,R2);
    Exp(g,x,M,R);
    if R1·R2=R then Rl:=1
    else Rl:=0;
  end.
```

Рис. 4.4. Псевдокод алгоритма теста линейной состоятельности L_T

```
Program N_T(n,R); {вход g,M,q, выход Re}
  begin
    x1:=random(q);
    x2:=(x1+1)modq;
    Exp(g,x1,M,R1);
    Exp(g,x2,M,R2);
    if R1·g=R2 then Re:=1
    else Re:=0;
  end.
```

Рис. 4.5. Псевдокод алгоритма теста единичной состоятельности N_T

Для доказательства безопасности сначала необходимо отметить, что так как $x_1 \in_R Z_q$, то и x_2 имеет равномерное распределение вероятностей над Z_q. Так как вероятность ошибки $\varepsilon \leq 1/8$, то в одном цикле вероятность $\mathrm{Prob}[R_k = f_{M,g}(x)] \geq 3/4$. Чтобы вероятность корректного ответа была не менее чем $1-\beta$, исходя из оценки Чернова [Be1,BLR], необходимо выполнить не менее $12\ln(1/\beta)$ циклов ■.

<u>Лемма 4.2.</u> Программа S_T_exp(n,M,q,g,β) является (1/288,1/8)-самотестирующейся программой, которая контролирует результат вычисления значения функции g^xmodulo$M$ с любым модулем M.

<u>Доказательство.</u> Полнота программы S_T(·) доказывается аналогично доказательству полноты в лемме 4.1, где $x_1, x_2 \in_R Z_q$. Полнота теста единичной состоятельности вытекает исходя из следующего очевидного факта. Корректное выполнение теста $[P_{M,g}(x_1) \cdot P_{M,g}(1)](\mathrm{mod}\, M)$ соответствует вычислению функции:

$$f_{M,g}(x) = f_{M,g}(x_1) \circ f_{M,g}(1) = g^{[x_1+1](\mathrm{mod}\,\varphi(M))} \equiv g^{x_1} \cdot g(\mathrm{mod}\, M) \equiv g^x(\mathrm{mod} M) \equiv R_e.$$

Для доказательства условия самотестируемости необходимо отметить, что так как и в лемме 4.1 для того, чтобы вероятность корректных ответов R_l и R_e в обоих тестах была не более $1-\beta$ достаточно выполнить тест

линейной состоятельности $\lceil 576\ln(4/\beta) \rceil$ раз и тест единичной состоятельности $\lceil 32\ln(4/\beta) \rceil$ раз.

Можно показать, основываясь на теоретико-групповых рассуждениях, что возможно обобщение программы S_T(·) и для других групп (вышеописанные алгоритмы основываются на вычислениях в мультипликативной группе вычетов над конечным полем). То есть для всех $y \in G$, $P(y) \in G^*$, где G^* -представляет собой любую группу, кроме групп G^{**}. К группам последнего вида относятся бесконечные группы, не имеющие конечных подгрупп за исключением $\{O'\}$, где O' - тождество группы. Таким образом, можно показать (если параметры выбираются независимо, равновероятно и случайным образом), что программа вида S_T(·) является $(\varepsilon/36,\varepsilon)$-самотестирующейся программой. Из всего сказанного, следует доказательство утверждения леммы ■.

Исходя из определения самотестирующейся/ самокорректирующейся программной пары и основываясь на результатах доказательств лемм 1 и 2, очевидным образом следует доказательство теоремы 1 ■.

Замечания. Как видно из псевдокода алгоритма $A^x \text{modulo}\, N$ в нем используется операция $AB \text{modulo}\, N$. Используя ту же технику доказательств, как и для функции дискретного возведения в степень, можно построить (1/576,1/36,1/36)-самокорректирующуюся/ самотестирующуюся программную пару для вычисления функции модулярного умножения. Это справедливо, исходя из следующих соображений. Вычисление функции $f_M(ab)=f_M((a_1+a_2)(b_1+b_2))$ следует из корректного выполнения программы с 4-х кратным вызовом оракульной программы $P(a,b)$, то есть:

$$[P_M(a_1,b_1)+P_M(a_1,b_2)+P_M(a_2,b_1)+P_M(a_2,b_2)](\text{mod}\, M).$$

Алгоритм вычисления $A^x \text{modulo}\, N$ выполняется для c=2. Однако, декомпозиция x, как следует из свойства самосводимости функции $A^x \text{modulo}\, N$, может осуществляться на большее число слагаемых. Хотя это приведет к гораздо большему количеству вызовов оракульной программы, но в то же время позволит значительно снизить вероятность ошибки.

4.5. Метод создания самотестирующейся расчетной программы с эффективным тестирующим модулем

В качестве расчетной программы рассматривается любая программа, решающая задачу получения значения некоторой вычислимой функции.

При этом под верификацией расчетной программы понимается процесс доказательства того, что программа будет получать на некотором входе истинные значения исследуемой функции. Иными словами, верификация расчетной программы направлена на доказательство отсутствия преднамеренных и/или непреднамеренных программных дефектов в верифицируемой программе.

В данном случае предлагается метод создания самотестирующихся программ для верификации расчетных программных модулей [КС2]. Данный метод не требует вычисления эталонных значений и является независимым от используемого при написании расчетной программы языка программирования, что существенно повышает оперативность исследования программы и точность оценки вероятности отсутствия в ней программных дефектов.

Пусть для функции $Y = f(X)$ существует пара функций $(g_c, h_c)^Y$ таких, что:

$Y = g_c(f(a_1), ..., f(a_c))$,

$X = h_c(a_1, ..., a_c)$.

Легко увидеть, что если значения a_i выбраны из I_n в соответствии с распределением D^p, тогда пара функций $(g_c, h_c)^Y$ обеспечивает выполнение для функции $Y = f(X)$ свойства случайной самосводимости. Пару функций $(g_c, h_c)^Y$ будем называть *ST-парой функций* для функции $Y = f(X)$.

Предположим, что на *ST*-пару функций можно наложить некоторую совокупность ограничений на сложность программной реализации и время выполнения. В этом случае, пусть длина кода программ, реализующих функции g_c и h_c , и время их выполнения составляет константный мультипликативный фактор от длины кода и времени выполнения программы P.

Предлагаемый метод верификации расчетной программы P на основе *ST*-пары функций для некоторого входного значения вектора X^* заключается в выполнении следующего алгоритма. (Всюду далее, если осуществляется случайный выбор значений, этот выбор выполняется в соответствии с распределением вероятностей D^p).

Алгоритм ST

1. Определить множество $A^* = \left\{a_1^*, \ldots, a_c^*\right\}$ такое, что $X^* = h_c\left(a_1^*, \ldots, a_c^*\right)$, где $a_1^*, \ldots, a_c^*$ выбраны случайно из входного подмножества I_n.

2. Вызвать программу P для вычисления значения $Y_0^* = f\left(X^*\right)$.

3. Вызвать c раз программу P для вычисления множества значений $\left\{f\left(a_1^*\right), \ldots, f\left(a_c^*\right)\right\}$.

4. Определить значения $Y_1^* = g_c\left(f\left(a_1^*\right), \ldots, f\left(a_c^*\right)\right)$.

5. Если $Y_0^* = Y_1^*$, то принимается решение, что программа P корректна на множестве значений входных параметров $\left\{X^*, a_1^*, \ldots, a_c^*\right\}$, в противном случае данная программа является некорректной.

Таким образом, данный метод не требует вычисления эталонных значений и за одну итерацию позволяет верифицировать корректность программы P на (n+1) значении входных параметров. При этом время верификации можно оценить как $T = \sum_{i=1}^{c} t_i + t_x + t_g + t_{h^{-1}} < T_P\left(X\right) \cdot \left(1 + c + K_{gh}\left(X, c\right)\right)$,

где t_i и t_x - время выполнения программы P при входных значениях a_i и X^* соответственно;

t_g и $t_{h^{-1}}$ - время определения значения функции g_c и множества A^* соответственно:

$T_P(X)$ - временная (не асимптотическая) сложность выполнения программы P;

$K_{gh}(X, c)$ - коэффициент временной сложности программной реализации функции g_c и определения A^* по отношению к временной сложности программы P (по предположению он составляет константный мультипликативный фактор от $T_P(X)$, а его значение меньше 1). Для традиционного вышеуказанного метода тестирования время выполнения и сравнения полученного результата с эталонным значением составляет:

$$T_0 = \sum_{i=1}^{c} t_i + t_x + \sum_{i=1}^{c} t_i^e + t_x^e > 2 \cdot T_P(X) \cdot (1 + c),$$

где t_i^e и t_x^e - время определения эталонных значений функции $Y = f(X)$ при значениях a_i и X^* соответственно (в общем случае, не может быть меньше времени выполнения программы).

Следовательно, относительный выигрыш по оперативности предложенного метода верификации (по отношению к методу тестирования программ на основе ее эталонных значений):

$$\Delta T = \frac{T}{T_0} = \frac{\sum_{i=1}^{c} t_i + t_x + t_g + t_{h^{-1}}}{\sum_{i=1}^{c} t_i + t_x + \sum_{i=1}^{c} t_i^e + t_x^e} < \frac{1 + c + K_{gh}}{2 \cdot (1 + c)} = {}^1\!/_2 + \frac{K_{gh}}{2 \cdot (1 + c)}$$

Так как, коэффициент $K_{gh} < 1$, а $c \geq 2$, то получаем относительный выигрыш по оперативности испытания расчетных программ указанного типа (обладающих свойством случайной самосводимости) более чем в 1.5 раза.

4.6. Исследования процесса верификации расчетных программ

В качестве примера работоспособности предложенного метода рассмотрим верификацию программы вычисления функции дискретного возведения в степень:

$$y = f_{AM}(x) = A^x \text{ modulo } M.$$

Выбор данной функции обусловлен тем, что она является одной из основных функций в различных теоретико-числовых конструкциях, например, в схемах электронной цифровой подписи, системах открытого распределения ключей и т.п. Это, в свою очередь, демонстрирует возможность применения предложенного метода при исследовании расчетных программ, решающих конкретные прикладные задачи. Кроме того, очевидно, что данная функция обладает свойством случайной самосводимости, а, исходя из вышеприведенных рассуждений и работы [BLR], рассуждений можно показать, что для данной функции существует (1/288, 1/8)-самотестирующаяся программа.

Для экспериментальных исследований была выбрана программа EXP из библиотеки базовых криптографических функций CRYPTOOLS [КС2], которая реализует функцию дискретного экспоненцирования в степень (размерность переменных и констант не превышает 64 или 128 байтов).

Была разработана интегрированная оболочка для проведения верификации, включающая интерфейс с пользователем и программные процедуры, реализующие некоторую совокупность проверочных тестов. Экспериментальные исследования состояли из определения временных характеристик процесса верификации на основе использования *ST*-пары функций и определения возможности обнаружения предложенным методом преднамеренно внесенных программных ошибок.

Для этого были определены следующие *ST*-пары функций:

$g_2(a_1,a_2) = [f_{AM}(a_1) \cdot f_{AM}(a_2)](\text{mod } M)$ и $h_2(a_1,a_2) = a_1 + a_2$;

$g_3^1(a_1,a_2,a_3) = [f_{AM}(a_1) \cdot f_{AM}(a_2) \cdot f_{AM}(a_3)](\text{mod } M)$ и

$h_3^1(a_1,a_2,a_3) = \sum_{i=1}^{3} a_i$;

$g_3^2(a_1,a_2,a_3) = \left[f_{f_{AM}(a_1)}(a_2) \cdot f_{AM}(a_3)\right](\text{mod } M)$ и

$h_3^2(a_1,a_2,a_3) = a_1 \cdot a_2 + a_3$;

В процессе исследований менялась используемая *ST*-пара функций и варьировалась размерность параметров *A*, *M* и аргумента *X*. Результаты экспериментов полностью подтвердили приведенные выше временные зависимости (технические результаты исследований автор в данной работе опускает).

Исследование возможности обнаружения предложенным методом преднамеренно внесенных закладок заключалось в написании программы EXPZ. Спецификация для программ EXP и EXPZ одна и та же, отличие же заключается в том, что программа EXPZ содержит программную закладку деструктивного характера. Преднамеренно внесенная закладка при исследованиях гарантировала сбой работы программы вычисления значения функции $y = f_{AM}(x) = A^x \text{ modulo } M$ (то есть обеспечивала получение неправильного значения функции) примерно на каждой восьмой части входных значений экспоненты *x*.

Было проведено свыше 2000 экспериментов [КС2]. Все входные значения, на которых произошел сбой программы, были обнаружены, что в дальнейшем подтвердилось проверочными тестами, основанными на использовании малой теоремы Ферма и теореме Эйлера. Этот факт, в свою очередь, экспериментально показал, что программа, реализующая алгоритм **ST**, является (1/8,1/288)-самотестирующейся программой.

Таким образом, предложенный метод позволяет в значительной степени сократить время испытания расчетных программ на предмет выявления непреднамеренных и преднамеренных программных дефектов. При этом по результатам испытаний можно получить количественные оценки вероятности наличия программных дефектов в верифицируемой расчетной программе. Однако, разработка формальных методов получения *ST*-пары функций для заданной расчетной программы, а также разработка методик ее испытания с помощью рассмотренного алгоритма требуют дальнейших теоретических и прикладных исследований.

4.7. Области применения самотестирующихся и самокорректирующихся программ и их сочетаний

4.7.1. Общие замечания

Применение чекеров, самотестирующихся, самокорректирующихся программ и их сочетаний возможно в самых различных областях вычислительной математики, а, следовательно, в самых разнообразных областях человеческой деятельности, где широко применяются вычислительные методы. К ним относятся такие направления как цифровая обработка сигналов (а, следовательно, решение проблем в системах распознавания изображений, голоса, в радио- и гидроакустике), а также методы математического моделирования процессов изменения народонаселения, экономических процессов, процессов изменения погоды и т.п. Идеи самотестирования могут найти самое широкое применение в системах защиты информации, например, в системах открытого распределения ключей, в криптосистемах с открытым ключом, в схемах идентификации пользователей вычислительных систем и аутентификации данных, где базовые вычислительные алгоритмы обладают некоторыми алгоритмическими свойствами, например, свойством случайной самосводимости, описанным выше.

4.7.2. Вычислительная математика

Активными исследованиями в области создания самотестирующихся и самокорректирующихся программ ученые и практики начали заниматься с начала 90-х годов. В этот период были разработаны программные чекеры для ряда теоретико-числовых и теоретико-групповых задач, для решения

задач с матрицами, полиномами, линейными уравнениями и рекуррентными соотношениями. В дальнейшем, исходя из контекста работы, по необходимости, будут приводиться наиболее интересные и необходимые схемы, протоколы, теоремы и их доказательства. В некоторых случаях детали схем и доказательств будут опускаться, но необходимые ссылки на литературные источники приводятся в обязательном порядке.

Целочисленная арифметика и арифметика многократной точности

Пусть $M(n)$ – время выполнения программы P на входе размером n. В работе [BLR] были разработаны программные чекеры для функций, представленных в таблице 4.1. Там же приведены ресурсозатраты на выполнение самотестирующейся/ самокорректирующейся программной пары для указанных целочисленных (в том числе модулярных с операндами многократной точности) функций. Во второй колонке показано время выполнения самотестирующейся/ самокорректирующейся программной пары без учета времени вызова программы, реализующей функции, приведенные в первой колонке. В третьей колонке приведено общее время выполнения с учетом времени вызовов программы P. В это время не включается время выполнения программ реализации функций, зависящее от параметра безопасности k, который обычно составляет $O(\log(1/k))$.

Таблица 4.1.

Функция	**Без вызова программы**	**Общее время выполнения**
A mult B	n	$M(n)$
A div B	$n \log n$	$M(n) log\ n$
A modulo N	n	$M(n)$
AB modulo N	n	$M(n)$
A^b modulo N (с известной факторизацией модуля)	n	$M(n)$
A^b modulo N (с неизвестной факторизацией модуля)	$n \log^4 n$	$M(n)\log^4 n$

Теоретико-групповые и теоретико-числовые вычисления

В работе [ВК] приведены чекеры для решения некоторых теоретико-групповых и теоретико-числовых проблем, некоторые из которых приведены ниже.

Проблема эквивалентного поиска. Пусть S – множество и G – группа, групповые действия в которой, осуществляются над элементами множества S. Для $a,b \in S$ элемент $a \equiv_G b$, тогда и только тогда, когда $g(a)=b$ для некоторого g из G. Проблема эквивалентного поиска заключается в нахождении g такого, что $g(a)=b$, если $a \equiv_G b$ для a и b, принадлежащих множеству S. Если существует эффективный вероятностный алгоритм поиска $g \in G$, тогда существует [ВК] эффективный программный чекер для данной проблемы. Примеров решения задач эффективного эквивалентного поиска достаточно много. К ним относятся проблема поиска изоморфизма графов (см. дальше), решение задачи квадратных вычетов, обобщенная проблема дискретных логарифмов, задачи подобные «Кубику Рубика», ряд задач из теории кодирования и др. [ВК].

Проблема пересечения групп. Пусть G и H – группы перестановок, определенные некоторыми генераторами групп. Генераторы представляются как перестановки над $[1,...,n]$. Проблема пересечения групп заключается в нахождении генераторов для $G \cap H$. В работе показывается [ВК], что можно построить программный чекер для данной проблемы.

Проблема расширенного нахождения НОД. Проблема расширенного нахождения наибольшего общего делителя (которая отличается от нахождения НОД посредством алгоритма Евклида) заключается в нахождении для двух положительных целых a и b целого d=НОД(a,b) и целых u и v таких, что $au+bv=d$.

Чекер для решения расширенного нахождения НОД по входу двух положительных целых a и b, целого d и целых u и v выдать «Сбой», если d не делит a или d не делит b или $au+bv \neq d$. В противном случае выдать «Норма». Эффективность и корректность данного чекера легко доказывается.

Вычисления над полиномами

Существует достаточно простой способ построения самокорректирующейся программы, который основывается на существовании следующего интерполяционного тождества, соответствующего значения функций между точками: для всех

одномерных полиномов f степени не более d, для всех $x,t \in F$, $\sum_{i=0}^{d+1} \alpha_i f(x+a_i \cdot t)=0$, где a_i – различные элементы из F, α_i=-1 и α_i зависит от F,d и не зависит от x,t. Тогда самокорректирующаяся программа для вычисления $f(\vec{x})=f(x_1,...,x_n)$ заключается в выполнении следующего алгоритма. Случайно и равновероятно выбирается $\vec{t}=(t_1,...,t_n)$ и выдается $\sum_{i=0}^{d+1} \alpha_i P(\vec{x}+i \cdot \vec{t})=0$. С вероятностью не менее 2/3 все вызовы программы будут возвращать корректные результаты и, следовательно, выход программы будет корректным. Рис.4.6 демонстрирует самотестирующуюся программу для полинома f.

В ряде работ было показано, как строить самотестирующиеся и самокорректирующиеся программы для задач сложения и умножения полиномов [BLR,GLR] и аппроксимирующие чекеры для полиномов [EKR] (см. далее).

Вычисления над матрицами

Одной из первых работ в области вероятностных алгоритмов, которая, в конечном счете, явилась одной из основополагающей в области методологии самотестирования явилась работа Р. Фрейвалдса, написанная им еще в 1979 году. Он предложил следующий простой и элегантный чекер для задачи умножения матриц (рис.4.7).

```
Program P_S_T(P,ε,β,x,f(x)); {вход P,ε,β,(x1,f(x1)),...,xd+1,f(xd+1))),
                                 выход («Норма»,«Сбой»)}
 begin
    for i=1 to O((1/k)log(1/β)) do
      begin
        x,t:=random(Zp);        {random - функция случайного
                                 равновероятного выбора из
                                 множества вычетов по
                                 модулю p};
              if Σ_{i=0}^{d+1} α_i P(x+i·t) ≠ 0 (более, чем в ε итерациях) then
              output «Сбой»;
      end;
 output «Норма»;
    for j=0 to d do
```

```
begin
   for i=1 to O(log(d/β)) do
        begin
            t:=random(Z_p); {random - функция случайного
                             равновероятного выбора из
                             множества вычетов по
                             модулю p};
```

if $\sum_{i=0}^{d+1} \alpha_i P(x_j + i \cdot t) \neq f(x_j)$ (более, чем в ¼ итерациях) then output «Сбой»;

```
        end;
   end;
 output «Норма»;
end.
```

Рис.4.6. Псевдокод алгоритма самотестирующейся программы для полинома *f*

Пусть матрицы *A* и *B* матрицы размером $n \times n$ определены над конечным полем *F*.

Время выполнения программы, реализующей чекер Фрейвалдса, составляет $O(n^2 \lceil \log(1/k) \rceil)$.

Используя чекер Фрейвалдса, можно построить самотестирующуюся/ самокорректирующуюся программную пару для умножения матриц (рис.4.8 – 4.9).

```
Program C_F(A,B,C,k); {вход A,B,C,k, выход («Норма»,«Сбой»)}
 begin
    for i=1 to ⌈log(1/k)⌉ do
       begin
          R:=random(F);        {random - функция случайного
                                равновероятного выбора 0-1-
                                вектора размером (n×1) из F};
              if C·R≠A·(B·R) then output «Сбой»;
       end;
    output «Норма»;
 end.
```

Рис.4.7. Псевдокод алгоритма, реализующего чекер Фрейвалдса

```
Program S_K_multAB(A,B,k); {вход A,B,k, выход C}
 begin
    for i=1 to ∞ do
      begin
        A1:=random(F);        {random - функция случайного
                               равновероятного выбора матрицы
                               размером (n×n) из F};
        B1:=random(F);        {random - функция случайного
                               равновероятного выбора матрицы
                               размером (n×n) из F};
        A2:=A-A1;
        B2:=B-B1;
        C:=P(A1,B1)+P(A1,B2)+P(A2,B1)+P(A2,B2);
        if C_F(A,B,C,k)=«Норма» then output C and goto 1;
      end;
 1:end.
```

Рис.4.8. Псевдокод алгоритма самокорректирующейся программы умножения матриц

Время выполнения программы S_K_multAB составляет $O(M(n)+n^2\log(1/k))$, где $M(n)$ – время выполнения программы умножения матриц размером $n\times n$ [BLR].

Самотестирующаяся программа для умножения матриц строится достаточно просто.

```
Program S_T_multAB(A,B,k); {вход A,B,k, выход («Норма»,«Сбой»)}
 begin
    for i=1 to O(log(1/k))∞ do
      begin
        A:=random(F);         {random - функция случайного
                               равновероятного и независимого
                               выбора матрицы размером (n×n) из F};
        B:=random(F);         {random - функция случайного
                               равновероятного и независимого
                               выбора матрицы размером (n×n) из F};
        C:=P(A,B);
        if C_F(A,B,C,1/4)=«Норма» then output 0 and goto 1;
        if C_F(A,B,C,1/4)=«Сбой» then output 1 and goto 1;
      end;
```

1:end.

Рис. 4.9. Псевдокод алгоритма самотестирующейся
программы умножения матриц

Можно легко удостовериться, что, если $err(P,f,U_n)\geq 1/8$, то количество единиц будет не менее 1/16 с вероятностью не менее 1-k и если $err(P,f,U_n)\leq 1/32$, то количество единиц будет не менее 1/16 с вероятностью не менее 1-k. Таким образом, вышеприведенная программа будет (1/32,1/8)-самотестирующейся программой для умножения матриц.

Некоторым аналогичным образом строятся самотестирующиеся/ самокорректирующиеся программные пары для других операций над матрицами. Данные по ресурсозатратам сведены в таблицу 4.2, где обозначения в таблице точно такие же, как и в таблице 4.1.

Таблица 4.2.

Функция	***Без вызова программы***	***Общее время выполнения***
Умножение матриц	N	$M(n)$
Определение детерминанта	N	$M(n)$
Инверсия матрицы	N	$M(n)$
Определение ранга матрицы C	N	$M(n)$
Определение ранга матрицы T	$n\sqrt{n}$	$M(n)\sqrt{n}$

Линейные рекуррентные соотношения

В работе [KS] исследовались вопросы построения самотестирующихся и самокорректирующихся программ для линейных рекуррентных соотношений, т.е. соотношений вида $f(n)=\sum_{i=1}^{d} c_i f(n-i)$. Такие последовательности являются основными для многих комбинаторных и теоретико-числовых последовательностей, таких как последовательность Фибоначчи и последовательность Лукаша.

Линейные рекуррентные соотношения часто рассматриваются в неявном виде в качестве однородных линейных уравнений вида: $\sum_{i=0}^{d} c_i f(n+d-i)=0$. Линейные рекуррентные соотношения часто используются в таких прикладных областях как моделирование динамики изменения народонаселения, различных экономических процессах, при анализе различных трафиков (потоков всевозможных данных, процессов) и т.п. Кроме того, такие последовательности используются при описании различных процессов в робототехнике и цифровой обработке сигналов.

Аппроксимирующие функции

Пусть для данной функции f и границы ошибки δ, входа x, программа $P(x)$ вычисляющая функцию f, *приблизительно корректна*, если $|P(x)-f(x)|\le\delta$. Это обозначается следующим образом: $P(x)\approx_\delta f(x)$. Будем также считать, что $P(\delta,\varepsilon)$ *аппроксимирует функцию* f на области D, если $|P-f|\le\delta$ на $1-\varepsilon$ элементах области D.

Исходя из работ [GLRSW,RS1], можно показать, как тестировать программы, которые вычисляют полиномы и функции, определенные теоремами сложения, когда выход программы, реализующей вычисления над этими полиномами или функциями является приблизительным. В этих работах также показано, как выполнить аппроксимирующие проверку, самотестирование и самокоррекцию полиномов и функциональных уравнений.

В таблице 4.3 показаны некоторые теоремы сложения для функциональных уравнений, где верна следующая форма $f(x+y)=G[f(x),f(y)]$.

Таблица 4.3.

$G[f(x),f(y)]$	$f(x)$
$f(x)+f(y)$	Ax
$\dfrac{f(x)+f(y)}{1-f(x)f(y)}$	tg Ax
$\dfrac{f(x)f(y)-1}{f(x)+f(y)}$	ctg Ax
$\dfrac{f(x)+f(y)-1}{2f(x)+2(y)-2f(x)f(y)-1}$	$\dfrac{1}{1+tgAx}$

$\frac{f(x)+f(y)-2f(x)f(y)}{1-f(x)f(y)}$	$\frac{-Ax1}{1-Ax}$
$\frac{f(x)+f(y)}{1+(f(x)f(y))A^2}$	A th Bx
$\frac{f(x)f(y)}{f(x)+f(y)}$	$\frac{A}{x}$
$f(x)f(y)$-$\sqrt{1-f(x)^2}\sqrt{1-f(y)^2}$	cos Ax
$\frac{f(x)+f(y)-2f(x)f(y)\cos a}{1-f(x)f(y)}$	$\frac{\sin Ax}{\sin Ax+a}$
$\frac{f(x)+f(y)-2f(x)f(y)\text{ch } a}{1-f(x)f(y)}$	$\frac{\text{sh } Ax}{\text{sh } Ax+a}$
$\frac{f(x)+f(y)+2f(x)f(y)\text{ch } a}{1-f(x)f(y)}$	$\frac{-\text{sh } Ax}{\text{sh } Ax+a}$
$\frac{f(x)+f(y)+2f(x)f(y)}{1-f(x)f(y)}$	$\frac{Ax}{1-Ax}$
$f(x)f(y)$+$\sqrt{f(x)^2-1}\sqrt{f(y)^2-1}$	ch Ax

Кроме того, в ряде работ (см., например, упоминания в [EKR]) было показано, как строить аппроксимирующие чекеры для ряда модулярных и логарифмических операций, а также функций sin, cos, для умножения и инвертирования матриц, решения систем линейных уравнений и определения детерминантов. Также исследовались проблемы тестирования операций деления с плавающей точкой, одномерных полиномов степени до 9 включительно и многомерных полиномов, а также тестирования ряда других тригонометрических и гиперболических функций.

4.7.3. Криптография, интерактивные доказательства

Вводные замечания

Основная идея использования задач самотестирования в криптографии заключается в девизе «Защитить самих защитников». Так как криптографические методы используются для высокоуровневого обеспечения конфиденциальности и целостности информации, собственно программно-техническая реализация этих методов должна быть свободна от программных и/или аппаратных дефектов. Таким образом, самотестирование и самокоррекция программ может эффективно

применяться в современных системах защиты информации от несанкционированного доступа.

Распределение ключей, цифровая подпись, схемы аутентификации

Функция дискретного экспоненцирования, описанная выше, широко используется в современной криптографии, в частности, при открытом распределении ключей Диффи-Хеллмана, для генерации и верификации подписей в схемах электронной цифровой подписи, для построения различных схем аутентификации сообщений, идентификации пользователей вычислительных систем и т.п. Следовательно, существует принципиальная возможность построения программных чекеров, самотестирующихся, самокорректирующихся программ. Продемонстрируем это на примере схемы цифровой подписи *RSA* (рис. 4.10).

Пусть программа P предположительно вычисляет *RSA*-функцию и для $x,y,z \in Z_{>0}$ с $x,y<z$ и НОД$(x,z)=1$. Тогда чекер $C^{P}_{RSA}(x,y,z;k)$ работает следующим образом [KA].

Доказательства существования чекера для *RSA*-криптоалгоритма приведены в работе [KA]. Там же приведены *RSA*-чекеры для фиксированного модуля криптосистемы.

```
Program C_RSA_(x,y,z;k); {вход x,y,z,k выход («Норма»,«Сбой»)}
 begin
     t1:=⌈-klog_{99/100}2⌉;
     t2:=⌈-k/log_{4/5}2⌉;
     for l=1 to t1 do
        begin
           i:=random(Z);          {random - функция случайного
                                  равновероятного выбора из
                                  [1,...,z[};
           if P(x,i,z)≡0 (mod z) output «Сбой» and STOP;
           i,j:=random(Z);        {random - функция случайного
                                  равновероятного выбора из
                                  [1,...,z[};
```

```
        if  P(x,i,z)P(x,j,z)≠P(x,i+j,z)(mod z)  output  «Сбой»  and
                                  STOP;
        i:=random(Z);             {random - функция случайного
                                  равновероятного выбора из
                                  [1,...,z[};
        if  P(x,i,z)≡P(x,1,z)≠P(x,i+1,z)(mod z)  output  «Сбой»  and
                                  STOP;
      end;
    for l=1 to t2 do
      begin
        r:=random(Z);             {random - функция случайного
                                  равновероятного выбора из
                                  [1,...,z[};
        if  P(x,y,z)P(x,r,z)≠P(x,y+r,z)(mod z)  output  «Сбой»  and
                                  STOP;
      end;
    output «Норма»;
  end.
```

Рис.4.10. Псевдокод алгоритма *RSA*-чекера

Интерактивные системы доказательств

Пусть **A** и **B** - две интерактивные, т. е. взаимодействующие через общую коммуникационную ленту, вероятностные машины Тьюринга. Через $[B(x),A(x)]$ обозначается случайная величина - выходное слово машины **A**, когда **A** и **B** работают на входном слове x. Через $|x|$ обозначается длина слова x.

Интерактивным доказательством для языка L называется пара интерактивных машин Тьюринга (**P**,**V**) такая, что выполняются следующие два условия.

1. (*Полнота*). Для всех $x \in L$ вероятность $\mathrm{Prob}\{[P(x),V(x)]=1\}=1$.
2. (*Корректность*). Для любой машины Тьюринга $\mathbf{P}^*$, для любого полинома p и для всех $x \in L$ достаточно большой длины $\mathrm{Prob}\{[P^*(x),V(x)]=1\}<1/p(|x|)$.

Формальные определения интерактивных систем доказательств и интерактивных систем доказательств с нулевым разглашением даны в приложении.

Задача «Изоморфизм графа»

Приведем в качестве примера протокол доказательства с абсолютно нулевым разглашением для языка «*Изоморфизм графов*» из работы [GMW]. Входным словом является пара графов $G_1=(U,E_1)$ и $G_0=(U,E_0)$. Здесь U - множество вершин, которое можно отождествить с множеством натуральных чисел $\{1,...,n\}$, E_1 и E_0 множества ребер такие, что $|E_1|=|E_0|=m$. Графы G_1 и G_2 называются изоморфными, если существует перестановка на множестве U такая, что $(u,v)\in E_0$ тогда и только тогда, когда $(\varphi(u),\varphi(v))\in E_1$ (обозначается $G_1=\varphi G_0$). Задача распознавания изоморфизма графов хорошо известная математическая задача, для которой на данный момент неизвестно полиномиальных алгоритмов ее решения. С другой стороны, неизвестно, является ли эта задача **NP**-полной, хотя есть веские основания предполагать, что не является.

Протокол IG

Пусть φ изоморфизм между G_1 и G_0. Следующие четыре шага выполняются в цикле m раз, каждый раз с независимыми случайными величинами.

1. **P** выбирает случайную перестановку π на множестве U, вычисляет $H=-mG_1$ и посылает этот граф **V.**
2. **V** выбирает случайный бит α и посылает его **P**.
3. Если $\alpha=1$, то **P** посылает **V** перестановку π, в противном случае - перестановку $\pi\circ\varphi$.
4. Если перестановка, полученная **V**, не является изоморфизмом между G_α и H, то **V** останавливается и отвергает доказательство. В противном случае выполнение протокола продолжается.

Если проверки п.4 дали положительный результат во всех m циклах, то **V** принимает доказательство.

Необходимо отметить, что если в протоколе **IG** машина **P** получает изоморфизм в качестве дополнительного входного слова, то ей для выполнения протокола не требуются неограниченные вычислительные ресурсы. Более того, в этом случае **P** может быть полиномиальной вероятностной шиной Тьюринга.

Теорема 4.1. Протокол **IG** является доказательством с абсолютно нулевым разглашением для языка «Изоморфизм графов».

Доказательство. Подробно приведено в работе [Ba1].

Чекер для задачи «Изоморфизм графа»

Пусть G и H – два графа и k – некоторый параметр безопасности. Чекер $C_{GI}^{P}(G,H,k)$ проверяет программу P на входных графах G и H. На выходе чекера будет результат «Норма», если графы изоморфны и «Сбой», если неизоморфны.

Следующая теорема [BK] определяет эффективность и корректность чекера для решения задачи «ИЗОМОРФИЗМ ГРАФОВ» - **IG**.

Пусть P – программа, которая останавливается на всех входах и всегда выдает либо «Норма», либо «Сбой». Пусть также G и H - два любых графа и пусть $C_{GI}^{P}(G,H;k)$ – вышеуказанный чекер.

Теорема 4.2. Если P корректная программа для решения задачи **IG**, тогда чекер $C_{GI}^{P}(G,H;k)$ всегда выдаст на выходе «Норма». Если P - некорректна, т.е. $P(G,H)GI(G,H)$, тогда вероятность $\mathrm{Prob}\{C_{GI}^{P}(G,H;k)=«Норма»\}<1/2^{k}$.

Доказательство. Достаточно очевидно и приведено в работе [BK].

Интерактивные системы доказательств и интерактивные системы доказательств с нулевым разглашением [Ba1,Ba2] могут эффективно применяться в системах защиты информации от несанкционированного доступа, например, в схемах интерактивной идентификации пользователей системы [Ка14,КУ4,КУ6]. В целом интерактивный протокол доказательств для задачи **IG** может применяться для идентификации, однако для практических целей удобно использовать системы доказательств, основанные на трудности решения некоторых теоретико-числовых задач. Тем более, как было показано выше, существует принципиальная возможность построения самотестирующейся/ самокорректирующейся программной пары, например, для функции дискретного экспоненцирования.

Чекер $C_{GI}^{P}(G,H,k)$

1. Вычислить $P(G,H)$.
2. Если $P(G,H)$=«Изоморфизм», тогда использовать P для поиска изоморфизма их G в H. Проверить, является ли результирующее соответствие изоморфизмом. Если нет, тогда подать на выход «Сбой», если является, тогда подать на выход «Норма».
3. Если $P(G,H)$=«Не изоморфизм», тогда выполнить следующие шаги k раз.
   - 3.1. Выработать случайный бит b.
   - 3.2. Если b=1, тогда

3.2.1. Сгенерировать случайную перестановку G' графа G.

3.2.2. Вычислить $P(G,G')$.

3.2.3. Если $P(G,G')$=«Не изоморфизм», тогда подать на выход «Сбой».

3.3. Если b=0, тогда

3.3.1. Сгенерировать случайную перестановку H' графа H.

3.3.2 Вычислить $P(G,H')$.

3.4. Если $P(G,H')$=«Изоморфизм», тогда подать на выход «Сбой».

4. Подать на выход «Норма».

4.7.4. Другие направления

В работе [BM] был построен псевдослучайный генератор, в котором центральным звеном является конструкция, которую можно рассматривать как самокорректирующуюся программу для решения задачи, эквивалентной проблеме дискретных логарифмов [BLR]. В работе [BLR] указывается, что Р. Рубинфилд ввел понятия программных чекеров для параллельных программ и использовал идею самотестирования для построения схемы константной глубины для проверки мажоритарных функций. В работе [BK] показаны методы построения программных чекеров для решения некоторых задач сортировки. В частности для двух массивов целых X и Y, представляющих некоторые мультимножества, чекер выдает «Сбой», если множество Y не упорядочено или если $X \neq Y$. В противном случае, если данная программа выполняет корректную сортировку, чекер выдает «Норма».

4.7.5. Применение самотестирующихся и самокорректирующихся программ

Вводные замечания

Как было показано выше применение программных чекеров, самотестирующихся, самокорректирующихся программ и их сочетаний возможно в самых различных областях человеческой деятельности, где программное обеспечение является центральным информационно-активным звеном автоматизированных и автоматических систем управления объектами информатизации, либо автоматизированными системами обработки данных. На следующем примере непосредственно

показывается, как использовать идеи самотестирования на практике, а именно в системах гидролокации для подавления шума, подавления помех и спектрального анализа по методу максимума энтропии [УСБ].

Применение вычислительных методов к задачам гидролокации

При решении проблемы подавления шума полагают, что выходной сигнал гидрофона представляет собой линейную комбинацию напряженности окружающего акустического поля с составляющими локальных источников шума, такими, как, например, шум двигателя собственного корабля. Чтобы выделить только сигнал шума, вблизи двигателей можно установить вспомогательные датчики. Для дальнейшего улучшения этого метода на входы устройства подавления могут быть поданы задержанные копии сигналов каждого эталонного источника шума. Тогда линейные комбинации этих задержанных копий будут аппроксимировать отфильтрованный сигнал шума, принятый гидрофоном от источника после его распространения по корпусу корабля.

Задача подавления помех аналогична задаче подавления шума, за исключением того, что входными сигналами адаптивного устройства компенсации являются выходные сигналы лучеобразующих устройств. Для формирования одного луча без помех в заданное положение направляется обычный луч и формируется S «нулевых лучей». Чувствительность нулевых лучей равна нулю при приеме сигнала с заданного направления, и они используются для приема помех. В общем случае требуется столько линейно независимых нулевых лучей, сколько помех необходимо подавить. Если выходы каждой из этих лучеобразующих схем объединить через весовой сумматор с выходом лучеобразующей схемы, принимающей сигнал с заданного направления, то минимизация общей мощности уменьшит помехи в выходном сигнале. Если составляющие помехи не коррелированы с сигналом, принимаемым с заданного направления, то вклад полезного сигнала в выходную мощность остается постоянным. Однако при определенных условиях это допущение может нарушаться. При многолучевом распространении полезный сигнал может быть принят нулевым лучом так же, как и лучом, ориентированным в заданном направлении. В этом случае подстройка весовых коэффициентов, направленная на уменьшение общей мощности выходного сигнала, может привести к уменьшению, как помех, так и полезного сигнала.

Подавление шума и помех, как правило, осуществляется адаптивными трансверсальньнми фильтрами, реализующими метод градиентного спуска посредством алгоритма минимизации средней квадратической ошибки [УСБ]. Цель таких способов реализации - обеспечить адаптацию при небольшом числе операций умножения, т.е. при относительно простой аппаратной реализации. Скорость адаптации уменьшается при большом разбросе собственных значений ковариационной матрицы данных, что имеет место при сильных источниках помех. Это может привести к времени сходимости, значительно превышающему период, в течение которого процессы, воздействующие на систему подавления, можно считать стационарными. В этом случае более быстрая сходимость может быть получена непосредственным обращением ковариационной матрицы выборок и решением нормальных уравнений, что приведет к более эффективному (в статистическом смысле) использованию имеющихся данных.

Современные методы спектрального анализа обеспечивают повышенную разрешающую способность при использовании параметрической модели сигнала. В методе максимума энтропии сигнал моделируется как выходной сигнал фильтра, имеющего только полюсы, на вход которого подан белый шум. Обратным такому фильтру служит трансверсальный фильтр, преобразующий сигнал в белый шум, весовые коэффициенты которого рассчитываются путем решения задачи линейного прогноза сигнала на один шаг вперед. Известно, что для стационарного процесса ошибка прогноза по методу наименьших квадратов представляет собой белый шум. Тогда оцениваемая функция спектральной плотности пропорциональна величине, обратной квадрату передаточной функции прогнозирующего фильтра. Этот метод спектральной оценки, обеспечивающий повышенную разрешающую способность в том случае, когда модель применима, и отношение сигнал/шум достаточно велико, может быть также использован для формирования луча.

Задачи, решаемые с помощью метода наименьших квадратов применительно к подавлению шума, подавлению помех и спектральному анализу методом максимальной энтропии, сведены в табл. 4.4., где z и d – случайная величина и случайный вектор в методе наименьших квадратов.

Таблица 4.4.

Задача	*z*	*d*
Подавление шума	$z_1,\ldots,z_s$ (эталонные источники шума)	Сигнал + шум
Подавление помех (адаптивная обработка с фиксированными лучами)	$b_1,\ldots,b_s$ (эталонные источники шума)	b_0 (выходной сигнал луча, ориентированного в данном направлении)
Спектральный анализ методом максимума энтропии	$x_1,\ldots,x_s$ (эталонные источники шума)	x_{s+1}

Метод наименьших квадратов и задача самотестирования

Рассмотрим метод наименьших квадратов применительно к задачам двух типов: детерминированной и стохастической с известными вторыми моментами.

Детерминированная задача состоит в выборе вектора x таким образом, чтобы минимизировать значение ε_1^2 в уравнении:

$$\varepsilon_1^2 = \|\mathbf{Ax}-\mathbf{y}\|^2 = \mathbf{x}^T\mathbf{A}^T\mathbf{Ax} - 2\mathbf{y}^T\mathbf{Ax} + \|\mathbf{y}\|^2, \quad (4.1)$$

где **A** — известная матрица, а **y** — известный вектор, индекс T обозначает транспонирование матрицы или вектора.

Стохастическая задача наименьших квадратов состоит в выборе детерминированного весового вектора **w** для минимизации средней квадратической ошибки ε_2^2:

$$\varepsilon_2^2 = E(\mathbf{w}^T\mathbf{z}-d)^2 = \mathbf{w}^T{}_{\mathbf{z}}\mathbf{R_z w} - 2\mathbf{R}^T{}_{d\mathbf{z}}\mathbf{w} + E(d^2), \quad (4.2)$$

где **z** — случайный вектор; d — случайная величина. Скалярная случайная величина $\mathbf{w}^T{}_{\mathbf{z}}$ представляет собой линейную комбинацию компонентов случайного вектора **z** и используется как линейная оценка случайной величины d. Математическое ожидание обозначается символом E, через $\mathrm{R}_\mathbf{z}$ обозначена матрица вторых моментов случайного вектора **z**: $\mathbf{R_z}=E_{\mathbf{zz}}{}^T$. Аналогично вектор $\mathbf{R}_{d\mathbf{z}}$ определяется как $\mathbf{R}_{d\mathbf{z}}=E(d\mathbf{z})$.

В работе [УСБ] показана вычислительная эквивалентность стохастической задачи наименьших квадратов с оценкой вторых моментов и соответствующей детерминированной задачи. Поэтому достаточно для

наших целей рассматривать вычислительные методы только для детерминированной задачи наименьших квадратов.

Из формул 4.1 и 4.2 видно, что задача наименьших квадратов в итоге сводится к умножению вектора на вектор, умножению матрицы на вектор, сложению и умножению матриц, транспонированию матриц, нахождению математического ожидания и нахождению нормы вектора. Исходя из результатов, приведенных в этой главе, можно констатировать, что для большинства данных функций существуют программные чекеры и самотестирующиеся/ самокорректирующиеся программные пары. Следовательно, существует принципиальная возможность последовательной, либо параллельной организации работы программных чекеров, самотестирующихся и самокорректирующихся программ для решения задачи наименьших квадратов. В то же время проблемы, связанные с композицией указанных программ в данной работе не рассматриваются.

Таким образом, можно сделать вывод о том, что основные вычислительные процедуры при решении задач обработки сигналов в реальном масштабе времени могут быть сведены к набору операций над матрицами. Такой набор включает умножение матрицы на вектор, сложение и умножение матриц, обращение матриц, решение систем линейных уравнений, приближенное решение линейных систем по методу наименьших квадратов, вычисление собственных значений, нахождение сингулярного разложение матриц [УСБ]. Поэтому можно говорить о хороших перспективах использования идей самотестирования в системах цифровой обработки сигналов.

ГЛАВА 5. ЗАЩИТА ПРОГРАММ И ЗАБЫВАЮЩЕЕ МОДЕЛИРОВАНИЕ НА RAM-МАШИНАХ

5.1. Основные положения

5.1.1. Вводные замечания

В этом разделе рассматриваются *теоретические аспекты защиты программ от копирования. Эта задача защиты может сводиться к задаче эффективного моделирования RAM-машины* (машины с произвольным доступом к памяти (см. приложение и [АХУ]) *посредством забывающей RAM-машины*. Следует заметить, что основные результаты по данной тематике принадлежат О. Голдрайху и Р. Островски [GO,O] и эти исследования активно продолжаются в настоящее время.

Машина является *забывающей*, если последовательность операций доступа к ячейкам памяти эквивалентна для любых двух входов с одним и тем же временем выполнения. Например, *забывающая машина Тьюринга* – это машина, для которой перемещение головок по лентам является идентичным для каждого вычисления и, таким образом, перемещения не зависят от фактического входа.

Необходимо выделить следующую формулировку ключевой задачи изучения программы по особенностям ее работы. «Как можно эффективно моделировать независимую *RAM*-программу на вероятностной забывающей *RAM*-машине». В предположении, что односторонние функции существуют, далее показывается, как можно сделать некоторую схему защиты программ стойкой против полиномиально-временного противника, которому позволено изменять содержимое памяти в процессе выполнения программы в динамическом режиме.

5.1.2. Центральный процессор, имитирующий взаимодействие

Неформально, будем говорить, что *центральный процессор, имитирует взаимодействие* с соответствующими зашифрованными программами, если никакой вероятностный полиномиально-временной противник не может различить следующие два случая, когда по данной зашифрованной программе как входу:

- противник экспериментирует с оригинальным защищенным центральным процессором, который пытается выполнить зашифрованную программу, используя память;

- противник экспериментирует с «поддельным» (фальсифицированным) центральным процессором.

Взаимодействие поддельного центрального процессора с памятью почти идентично тому, которое оригинальный центральный процессор имел бы с памятью при выполнении фиксированной фиктивной программы. Выполнение фиктивной программы не зависит по времени от числа шагов реальной программы. Не зависимо от времени поддельный центральный процессор (некоторым «волшебным» образом) записывает в память тот же самый выход, который подлинный центральный процессор написал бы, выполняя «реальную» программу.

При создании центрального процессора, который имитирует эксперименты, имеются две проблемы. Первая заключается в том, что необходимо скрывать от противника значения, сохраненные и восстановленные в/из памяти, и предотвращать попытки противника изменять эти значения. Это делается с использованием традиционных криптографических методов (например, методов вероятностного шифрования и аутентификации сообщений). Вторая проблема заключается в необходимости скрывать от противника последовательность адресов, к которым осуществляется доступ в процессе выполнения программы (здесь и далее это определяется как *сокрытие модели доступа*).

5.1.3. Сокрытие модели доступа

Сокрытие оригинальной модели доступа к памяти – это абсолютно новая проблема и традиционные криптографические методы здесь не применимы.

Программу, которую бы будем защищать, будем называть *оригинальной программой*, а последовательность операций доступа к памяти во время выполнения такой программы будем называть *оригинальной последовательностью операций доступа*. Программу, реализующую модель сокрытия доступа будем называть *завуалированной программой*.

Цель при решении задачи сокрытия модели доступа состоит в том, чтобы сделать невозможным для противника получить о программе что-либо полезное из модели доступа. В этом случае центральный процессор не будет выполнять программу обычным способом, - он будет заменять каждый оригинальный цикл «выборки/запоминания» многими циклами «выборки/запоминания». Это должно «запутывать» противника и предупреждать его от изучения оригинальной последовательности

операций доступа к памяти, т.е. от фактической последовательности. Следовательно, противник не должен улучшить свои возможности по реконструкции оригинальной программы.

Ценой, которую необходимо заплатить за защиту программ, таким образом, является быстродействие вычислений. Неформально говоря, *затраты на защиту программ* определяются как отношение числа шагов выполнения защищенной программы к числу шагов исходного кода программы. Далее показывается, что эти затраты полиномиально связаны с параметром безопасности односторонней функции, что подтверждается следующим тезисом.

Предположим, что односторонние функции существуют и пусть k - параметр безопасности функции. Тогда существует эффективный способ преобразования программ в пары, состоящие из физически защищенного центрального процессора с k битами внутренней защищенной памяти и соответствующей завуалированной программы такой, что центральный процессор имитирует взаимодействие с завуалированной программой, реализуемое за время, ограниченное poly(k). Кроме того, взамен t команд оригинальной программы будет выполняться менее чем $tk^{O(1)}$ команд завуалированной программы, а увеличение размера внешней памяти ограничивается фактором k.

Вышеупомянутый результат доказывается посредством сведения задачи создания центрального процессора, который имитирует эксперименты, к задаче эффективного сокрытия модели доступа. По-существу, последняя задача формулируется как задача моделирования независимых *RAM*-машин на забывающей *RAM*-машине (см. ниже).

5.2. Моделирование на забывающих RAM-машинах

Для каждой приемлемой модели вычислений существует преобразование независимых машин в эквивалентные забывающие машины (т.е., в забывающие машины, вычисляющие ту же самую функцию) [GO,O]. Вопрос заключается в ресурсозатратах для этих преобразований, а именно в определении времени замедления работы забывающей машины. Например, машина Тьюринга с одной лентой может моделироваться посредством забывающей машины Тьюринга с двумя лентами с логарифмическим замедлением времени выполнения. Ниже исследуется подобный процесс, но для модели вычислений с произвольным доступом к памяти (*RAM*-машины). Основное достоинство *RAM*-машины – это способность мгновенно получать доступ к

произвольным ячейкам памяти. В контексте настоящей работы, приводится следующий основной неформальный результат для *RAM*-машины [GO,O].

Пусть $RAM(m)$ означает *RAM*-машину с m ячейками памяти и доступом к случайному оракулу [ГДж]. Тогда t шагов независимой $RAM(m)$-программы может моделироваться менее чем за $O(t(\log_2 t)^3)$ шагов на забывающей $RAM(m(\log_2 m)^2)$.

Таким образом, можно увидеть, как провести моделирование независимой *RAM*-программы на забывающей *RAM*-машине с полилогарифмическим увеличением размера памяти и полилогарифмическим замедлением времени выполнения. В то же время, простой комбинаторный аргумент показывает, что любое забывающее моделирование независимой *RAM*-машины должно иметь среднее число $\Omega(\log t)$ затрат. В связи с эти приводится следующий аргумент.

Пусть машина $RAM(m)$ определена как показано выше. Тогда каждое забывающее моделирование $RAM(m)$-машины должно содержать не менее $\max\{m,(t-1)\log m\}$ операций доступа к памяти при моделировании t шагов оригинальной программы.

Далее рассмотрим сценария наихудшего случая, при котором противник активно пытается получить информацию, вмешиваясь в процесс вычислений. Неформально говоря, моделирование *RAM*-машины на забывающей *RAM*-машине *является доказуемо защищенным от вмешательства,* если моделирование остается забывающим (т.е. не вскрывает какой-либо информации о входе за исключением его длины) даже в случае, когда независимый «мощный» противник исследует и изменяет содержимое ячеек памяти. В связи с этим приводится следующий аргумент.

В условиях определения $RAM(m)$-машины t шагов независимой $RAM(m)$-программы могут быть промоделированы (доказуемо защищенным от вмешательства способом) менее чем за $O(t(\log_2 t)^3)$ шагов на забывающей машине $RAM(m(\log_2 m)^2)$.

Необходимо отметить, что вышеприведенные результаты относятся к *RAM*-машинам с доступом к случайному оракулу. Чтобы получить результаты для более реалистичных моделей вероятностных *RAM*-машин, необходимо заменить случайный оракул, используемый выше, псевдослучайной функцией. Такие функции существуют в предположении существования односторонних функций с использованием короткого

действительно случайно выбранного начального значения (см., например, [Ba1]).

5.3. Модели и определения

5.3.1. RAM-машина как пара интерактивных машин

Далее рассматривается модель *RAM* как пара интерактивных машин с ограниченными ресурсами, и даются два базовых понятия: понятие защиты программ и понятие моделирования на забывающей *RAM*-машине.

В данном разделе *RAM*-машина представляется как две интерактивные машины: *центральный процессор* (**ЦП**) и *модуль памяти* (**МП**). Задача исследований сводится к изучению взаимодействия между этими машинами. Для лучшего понимания необходимо начать с конкретизации (для целей данного раздела) определения интерактивной машины Тьюринга.

Интерактивная машина Тьюринга – многоленточная машина Тьюринга (см. приложение), имеющая следующие ленты:

- *входная лента «только-для-чтения»;*
- *выходная лента «только-для-записи»;*
- *рабочая лента «для-записи-и-для-чтения»;*
- *коммуникационная лента «только-для-чтения»;*
- *коммуникационная лента «только-для-записи».*

Под $ITM(c,w)$ обозначается машина Тьюринга с рабочей лентой длины w и коммуникационными лентами, разделенными на блоки c-битной длины, которая функционирует следующим образом. Работа $ITM(c,w)$ на входе y начинается с копирования y в первые $|y|$ ячеек ее рабочей ленты. В случае если $|y|>w$, выполнение приостанавливается немедленно. В начале каждого раунда, машина читает следующий c-битный блок с коммуникационной ленты «только-для-чтения». После некоторого внутреннего вычисления, использующего рабочую ленту, раунд завершается записью c битов на коммуникационную ленту «только-для-записи». Работа машины может завершиться в некоторой точке с копированием префикса ее рабочей ленты на выходную ленту машины.

Теперь можно определить **ЦП** и **МП** как интерактивные машины Тьюринга, которые взаимодействуют друг с другом, а также можно ассоциировать коммуникационную ленту «только-для-чтения» **ЦП** с коммуникационной лентой «только-для-записи» **МП** и наоборот. Кроме того, и **ЦП**, и **МП** будут иметь одну и ту же длину сообщений, то есть,

параметр c, определенный выше. **МП** будет иметь рабочую ленту размером, экспоненциальным от длины сообщений, в то время как **ЦП** будет иметь рабочую ленту размером, линейным от длины сообщений. Каждое сообщение может содержать «адрес» на рабочей ленте **МП** и/или содержимое регистров **ЦП**.

Далее используем k как параметр, определяющий и длину сообщений, и размер рабочих лент **ЦП** и **МП**. Кроме того, длина сообщений будет равна $k+2+k'$, а размер рабочей ленты будет равен $2^k k'$, где $k'=O(k)$.

Для каждого $k \in \mathbf{N}$ определим MEM_k как машину $ITM(k+2+O(k), 2^k O(k))$, работающую точно так, как определено выше. Рабочая лента разбивается на 2^k слов, каждое размером $O(k)$. После копирования входа на рабочую ленту машина MEM_k становится *машиной, управляемой сообщениями*. После получения сообщения (i,a,v), где $i \in \{0,1\}^2$={«запомнить»,«выборка»,«стоп»}, $a \in \{0,1\}^k$ и $v \in \{0,1\}^{O(k)}$, машина MEM_k работает следующим образом.

Если i=«запоминание», тогда машина MEM_k копирует значение v из текущего сообщения в число a рабочей ленты.

Если i=«выборка», тогда машина MEM_k посылает сообщение, состоящее из текущего содержания слова a (на рабочей ленте).

Если i=«стоп», тогда машину MEM_k копирует префикс рабочей ленты (как специальный символ) на выходную ленту и останавливается.

Далее, пусть для каждого $k \in \mathbf{N}$ определим CPU_k как машину $ITM(k+2+O(k), O(k))$, работающую точно так, как определено выше. После копирования входа на свою рабочую ленту, машина CPU_k выполняет вычисления за время, ограниченное poly(k), используя рабочую ленту, и посылает сообщение, полученное в этих вычислениях. В следующих раундах CPU_k – является машиной, управляемой сообщениями. После получения нового сообщения машина CPU_k копирует сообщение на рабочую ленту и, основываясь на вычислениях на рабочей ленте, посылает свое сообщение (копирует его на свою выходную ленту). Число шагов каждого вычисления на рабочей ленте ограничено фиксированным полиномом от k.

Единственная роль входа **ЦП** заключается в инициализации регистров **ЦП**, и этот вход в дальнейшем может игнорироваться. «Внутренние» вычисления **ЦП** в каждом раунде соответствует элементарным операциям над регистрами. Следовательно, число шагов, принимаемых в каждом таком вычислении, является фиксированным полиномом от длины

регистра (которая равна $O(k)$). Теперь можно определить *RAM*-модель вычислений, как семейство RAM_k-машин для каждого k.

Определение 5.1. Для каждого $k \in \mathbf{N}$ определим машину RAM_k как пару (CPU_k, MEM_k), где ленты «только-для-чтения» машины CPU_k совпадают с лентами «только для записи» машины MEM_k, а ленты «только-для-записи» машины CPU_k совпадают с лентами «только-для-чтения» машины MEM_k. Вход RAM_k – это пара (s,y), где s - вход (инициализация) для CPU_k и y – вход для MEM_k. Выход машины RAM_k по входу (s,y), обозначаемый как $RAM_k(s,y)$, определен как выход $MEM_k(y)$ при взаимодействии с $CPU_k(s)$.

Для того, чтобы рассматривать *RAM*-машину как универсальную машину, необходимо разделить вход y машины MEM_k на «программу» и «данные». То есть, вход y памяти разделен (специальным символом) на две части, названные программой (обозначенной П) и данными (обозначаемыми x).

Пусть RAM_k и s фиксированы и $y=(\mathbf{\Pi},x)$. Определим выход программы **П** на входных данных x, обозначаемый через $\Pi(x)$ как $RAM_k(s,y)$. Определим время выполнения **П** на данных x, обозначаемое через $t_\Pi(x)$, как сумму величины $(|y|+|\Pi(x)|)$ и числа раундов вычисления $RAM_k(s,y)$. Определим также размер памяти программы **П** для данных x, обозначаемый через $s_\Pi(x)$ как сумму величины $|y|$ и числа различных адресов, появляющихся в сообщениях, посланных CPU_k к MEM_k в течение работы $RAM_k(s,y)$.

Легко увидеть, что вышеупомянутая формализация непосредственно соответствует модели вычислений с произвольным доступом к памяти. Следовательно, «выполнение **П** на x» соответствует раундам обмена сообщениями при вычислении $RAM_k(\cdot,(\mathbf{\Pi},x))$. Дополнительный член $|y|+|\Pi(x)|$ в $t_\Pi(x)$ поясняет время, потраченное при чтении входа и записи выхода, в то время как каждый раунд обмена сообщениями представляет собой единственный цикл в традиционной *RAM*-модели. Член $|y|$ в $s_\Pi(x)$ объясняет начальное пространство, взятое по входу.

5.3.2. Дополнения к базовой модели и вероятностные RAM-машины

Приводимые ниже результаты верны для *RAM*-машин, которые являются вероятностными в очень строгом смысле. А именно **ЦП** в этих машинах имеет доступ к случайным оракулам. Однако в предположении существования односторонних функций, случайные оракулы могут быть эффективно реализованы посредством псевдослучайных функций.

Определение 5.2. Для каждого $k \in \mathbf{N}$ определим *оракульный* CPU_k как CPU_k с двумя дополнительными лентами, названными *оракульными лентами*. Одна из этих лент является «только-для-чтения», а другая «только-для-записи». Всякий раз, когда машина входит в специальное состояние вызова оракула, содержимое оракульной ленты «только-для-чтения» изменяется мгновенно (т.е., за единственный шаг) и машина переходит к другому специальному состоянию. Строка, записанная на оракульной ленте «только-для-записи» между двумя вызовами оракула называется запросом, соответствующим последнему вызову. Будем считать, что CPU_k имеет доступ к функции f, если делается запрос q и оракул отвечает и изменяет содержимое оракульной ленты «только-для-чтения» на $f(q)$. *Вероятностная* машина CPU_k – это оракульная машина CPU_k с доступом к однородно выбранной функции $f:\{0,1\}^{O(k)} \to \{0,1\}$.

Определение 5.3. Для каждого $k \in \mathbf{N}$ определим *оракульную* RAM_k-*машину* как RAM_k-*машину*, в которой машина CPU_k заменена на оракульную CPU_k. Скажем, что эта RAM_k-машина имеет доступ к функции f, если CPU_k имеет доступ к функции f и будем обозначать как RAM_k^f. Вероятностная RAM_k-машина – это RAM_k-машина, в которой машина CPU_k заменена вероятностной CPU_k. (Другими словами, вероятностная RAM_k-машина – это оракульная RAM_k-машина с доступом к однородно выбранной функции).

5.3.3. Повторные выполнения RAM-машины

Для решения проблемы защиты программ необходимо использовать повторные выполнения «одной и той же» *RAM*-программы на нескольких входах. Задача состоит в том, что *RAM*-машина начинает каждый следующий цикл своей работы с рабочими лентами **ЦП** и **МП**, имеющих содержимое, идентичное их содержимому по окончании предыдущего цикла.

Для каждого $k \in \mathbf{N}$, *повторные выполнения* RAM_k-*машины* на входной последовательности $y_1, y_2, \ldots$ рассматриваются как последовательность вычислений RAM_k-машины, при котором первое вычисление началось с входа y_1, когда рабочие ленты и CPU_k, и MEM_k пусты и i-тое вычисление начинается с входа y_i, когда рабочая лента каждой машины (т.е., и CPU_k, и MEM_k) содержит ту же самую строку, которую она содержала по окончании i-1 вычисления.

5.3.4. Эксперименты с RAM-машиной

Рассматриваются два типа противников. Оба могут неоднократно инициировать работу *RAM*-машины на входах по своему выбору. Различия между двумя типами противников состоит в их способности модифицировать коммуникационные ленты **ЦП** и **МП** в процессе вычислений. *Вмешивающемуся противнику* позволено как читать, так и записывать на эти ленты свою информацию (т.е., просматривать и изменять содержание взаимодействия), в то время как *невмешивающемуся противнику* позволено только читать эти ленты (то есть, только просматривать сообщения). В любом случае противнику не разрешено иметь те же самые права доступа к рабочей ленте **МП**, так как содержимое этой ленты полностью определено начальным входом и сообщениями, посланными **ЦП**. Кроме того, в обоих случаях противник не имеет никакого доступа к рабочим и оракульным лентам **ЦП**.

Для простоты, основное внимание будет уделяться противникам с экспоненциально ограниченным временем выполнения. Кроме того, время выполнения противника ограничено сверху 2^n, где n - размер рабочей ленты **ЦП**. На практике противник будет ограничен по времени некоторым полиномом от длины рабочей ленты **ЦП**.

Определение 5.4. *Невмешивающийся противник*, обозначаемый как *ADV*, является вероятностной машиной, которая на входе k и завуалированной программе α, которая имеет доступ к оракульной RAM_k-машине. Машина *ADV* инициирует повторные выполнения RAM_k-машины на входах по своему выбору до тех пор, пока общее время выполнения не стане равным 2^k. В процессе каждого из этих выполнений, машина *ADV* имеет доступ «только-для-чтения» к коммуникационным лентам между CPU_k и MEM_k.

Определение 5.5. *Вмешивающийся противник* определяется аналогично невмешивающемуся противнику за исключением того, что в процессе повторных выполнений противник имеет доступ для чтения и записи к коммуникационным лентам между CPU_k и MEM_k.

5.4. Преобразования, защищающие программное обеспечение

Определим компиляторы, которые по данной программе **П**, производят пару (f,$\mathbf{П}_f$), где f - случайно выбранная функция и $П_f$ – завуалированная программа, которая соответствует **П** и f. Здесь имеется в виду оракульная *RAM*-машина, которая на входе ($\mathbf{П}_f$,x) и при доступе к

оракулу f, моделирует выполнение **П** на данных x так, чтобы это моделирование «защищало бы» оригинальную программу **П**.

Далее даются определения *компиляторов* как набора преобразований программ в программно-оракульные пары, которые при выполнении оракульных *RAM*-программ являются функционально эквивалентными выполнениям оригинальных программ.

Определение 5.5. *Компилятор*, обозначаемый через C, является вероятностным отображением, которое по входу целочисленного параметра k и программы **П** для RAM_k возвращает пару $(f,\mathbf{\Pi}_f)$ так, чтобы:

- $f:\{0,1\}^{O(k)}\to\{0,1\}$ – случайно выбранная функция;
- $|\Pi_f|=O(|\Pi|)$;
- для $k'=k+O(\log k)$ существует оракульная $RAM_{k'}$-машина такая, что для каждой П, каждой f и каждого $x\in\{0,1\}$ инициируется $RAM_{k'}$ на входе (Π_f,x) и при доступе к оракулу f обеспечивает выход $\Pi(x)$.

Оракульная $RAM_{k'}$-машина отличается от RAM_k-машины в том, что $RAM_{k'}$ имеет доступ к оракулу, в то время как RAM_k нет. Понятно, что $RAM_{k'}$ имеет большую память, а именно $RAM_{k'}$-машина состоит из $2^{k'}=\text{poly}(k)2^k$ слов, в то время как память RAM_k состоит из 2^k слов.

Компиляторы, как определено выше, преобразовывают детерминированные программы в завуалированные программы, которые выполняются на вероятностных *RAM*-машинах. Теперь непосредственно обратимся к определениям компилятора, защищающего программное обеспечение.

Оракул спецификации для программы **П** – это оракул, который на запрос x возвращает тройку $(\mathbf{\Pi}(x),t_{\mathbf{\Pi}}(\text{x}),s_{\mathbf{\Pi}}(x))$.

Отметим, что $t_{\mathbf{\Pi}}(x)$ и $s_{\mathbf{\Pi}}(x)$ обозначает время выполнения и пространственные размеры программы **П** на данных x. Далее даются основное определение для задачи защиты программ. В этом определении *ADV* будет рассматриваться и как вмешивающийся, и как невмешивающийся противник.

Определение 5.6. Для данного компилятора C и противника *ADV*, компилятор C *защищает программное обеспечение от противника ADV*, если существует вероятностная оракульная машина M, удовлетворяющая следующим условиям:

- *(M функционирует примерно за то же самое время, как и ADV)* Существует полином $p(\cdot)$ такой, что для каждой строки α время выполнения M на входе $(k',|\alpha|)$ (с учетом доступа к случайному

оракулу) было ограничено $p(k')T$, где T обозначает время выполнения ADV при экспериментировании с $RAM_{k'}$ на входе α;

- *(M с доступом к оракулу спецификации обеспечивает выход почти идентичный выходу ADV после экспериментирования с результатами работы компилятора)* Для каждой программы **П** статистическое расстояние между следующими двумя распределениями вероятностей ограничено $2^{-k'}$.
    1. Распределение выхода машины ADV при экспериментировании с $RAM_{k'}^{f}$ на входе $\mathbf{\Pi}_f$, где $(f,\mathbf{\Pi}f)\leftarrow C(\mathbf{\Pi})$. Отметим, что $RAM_{k'}^{f}$ обозначает интерактивную пару $(CPU_{k'},MEM_{k'})$, где $CPU_{k'}$ имеет доступ к оракулу f. Распределение берется над пространством вероятностей, состоящим из всех возможных выборов функции f и всех возможных результатов выработки случайного бита («подбрасываний монеты») машины ADV с равномерным распределением вероятностей.
    2. Распределение выхода оракульной машины M на входе $(k',O(|\mathbf{\Pi}|))$ и при доступе к оракулу спецификации для **П**. Распределение берется над пространством вероятностей состоящим из всех возможных результатов подбрасываний монеты машины M с равномерным распределением вероятностей.

Компилятор C обеспечивает *слабую* защиту программ, если C защищает программы против любого невмешивающего противника. Компилятор C обеспечивает *доказуемую защиту программ от вмешательства*, если C защищает программы против любого вмешивающего противника.

Далее приведем определения затрат на защиту программ. Необходимо напомнить, что для простоты, мы ограничиваем время выполнения программы **П** следующим условием: $t_{\Pi}(x)>|\mathbf{\Pi}|+|x|$ для всех x.

Определение 5.7. Пусть C - компилятор и $g:\mathbf{N}\rightarrow\mathbf{N}$ – некоторая целочисленная функция. *Затраты компилятора C* на большинстве аргументов g, если для каждой **П**, каждого $x\in\{0,1\}^*$ и каждой случайно выбранной f требуемое время выполнения $RAM_{k'}$ на входе $(\mathbf{\Pi}_f,x)$ и при доступе к оракулу f ограничены сверху $g(T)T$, где $T=t_{\Pi}(x)$.

5.5. Определения забывающей RAM-машины и забывающего моделирования

5.5.1. Модели доступа

Необходимо начать с определения модели доступа как последовательности ячеек памяти, к которым **ЦП** обращается в процессе вычислений. Это определение распространяется также на оракульный **ЦП**.

Определение 5.7. *Модель доступа*, обозначаемая как $A^k(y)$, *детерминированной RAM_k-машины* на входе y – это последовательность $(a_1,...,a_i,...)$ такая, что для каждого i, i-тое сообщение, посланное машиной CPU_k при ее взаимодействии с машиной $MEM_k(y)$, имеет форму $(\cdot,a_i,\cdot)$.

При рассмотрении вероятностных *RAM*-машин, мы определяем случайную величину, которая для каждой возможной функции f принимает модель доступа, соответствующая вычислениям, в которых *RAM*-машина имеет доступ к этой функции. В связи с этим дается следующее определение.

Определение 5.8. *Модель доступа*, обозначаемая как $\widetilde{A}^k(y)$, *вероятностной RAM_k-машины* на входе y – это случайная величина, которая принимает значение модели доступа машины RAM_k на некотором входе y и при доступе к однородно выбранной функции f.

Теперь можно перейти к определению *забывающей RAM-машины*. Мы определяем забывающую *RAM*-машину как вероятностную *RAM*-машину, для которой распределение вероятностей последовательности адресов памяти, к которым осуществляется доступ в процессе выполнения программы, зависит только от времени выполнения и не зависит от конкретного частного входа.

Определение 5.9. Для каждого $k\in\mathbf{N}$ определим *забывающую RAM_k-машину* как *вероятностную RAM_k-машину*, удовлетворяющую следующему условию. Для каждых двух строк y_1 и y_2, если $|\widetilde{A}^k(y_1)|$ и $|\widetilde{A}^k(y_2)|$ *идентично распределены,* тогда также *идентично распределены* $\widetilde{A}^k(y_1)$ и $\widetilde{A}^k(y_2)$.

Интуитивно, последовательность операций доступа к памяти забывающей RAM_k-машины не открывает никакой информации относительно входа за исключением значения времени выполнения на этом входе.

Определения *RAM*-машины и забывающей *RAM*-машины необходимо для того, чтобы дать точное определение *забывающего моделирования* независимой *RAM*-машины посредством забывающей *RAM*-машины. Определение моделирования в данном случае минимально необходимое, - требуется только, чтобы обе машины вычисляли одну и ту же функцию. Кроме того, необходимо потребовать, чтобы входы, имеющие идентичное время выполнения на оригинальной *RAM*-машине, сохраняли бы идентичное время выполнения на забывающей *RAM*-машине. Для простоты, ниже представляется только определение для забывающего моделирования детерминированных *RAM*-машин.

Определение 5.10. Для данных машин, - вероятностной $RAM'_{k'}$, и RAM_k вероятностная машина $RAM'_{k'}$ *моделирует забывающим образом* RAM_k, если выполняются следующие условия:

- вероятностная машина $RAM'_{k'}$ моделирует RAM_k с вероятностью 1. Другими словами, для каждого входа y и каждого выбора оракульной функции f выход оракула $RAM'_{k'}$ на входе y и при доступе к оракулу f равняется выходу RAM_k на входе y;
- вероятностная машина $RAM'_{k'}$ – является забывающей. Необходимо подчеркнуть, что здесь рассматривается модель доступа $RAM'_{k'}$ на фиксированном входе и случайно выбранной оракульной функции.

Случайная величина, представляющая собой время выполнения вероятностной $RAM'_{k'}$ на входе y полностью определена текущим временем RAM_k на входе y.

Следовательно, модель доступа при забывающем моделировании (которая описывается случайной величиной, определенной над выбором случайного оракула) имеет распределение, зависящее только от времени выполнения оригинальной машины. А именно, пусть $\widetilde{A}^{k'}(y)$ обозначает модель доступа при забывающем моделировании RAM_k на входе y. Тогда $\widetilde{A}^{k'}(y_1)$ и $\widetilde{A}^{k'}(y_2)$ идентично распределены, если время выполнения RAM_k на этих входах (т.е., y_1 и y_2) идентично.

Теперь мы обратимся к определению затрат при забывающем моделировании.

Определение 5.11. Для данных вероятностных машин $RAM'_{k'}$ и RAM_k предположим, что вероятностная $RAM'_{k'}$ моделирует забывающим образом вычисления RAM_k и путь g:$\mathbf{N}\rightarrow\mathbf{N}$ - есть некоторая целочисленная функция. Тогда затраты на моделирование являются не больше g, если для каждого y

требуемое время выполнения $RAM'_{k'}$ на входе y ограничено сверху $g(T)T$, где T обозначает время выполнения RAM_k на входе y.

5.5.2. Моделирование с метками времени

В заключение этого подраздела приводится свойство некоторого *RAM*-моделирования, которое будет необходимо для решения задачи доказуемой защиты программ. Это свойство требует, чтобы при восстановлении значения из ячеек памяти, **ЦП** «знал бы» сколько раз содержимое этих ячеек модифицировалось. То есть, для данного адреса **МП** a и общего числа команд (обозначаемого j), выполненных **ЦП,** общее число команд «запомнить в ячейку a» может быть эффективно вычислено алгоритмом $Q(j,a)$. Далее рассматривается только моделирование детерминированных *RAM*-машин.

Определение 5.12. Для данной оракульной машины $RAM'_{k'}$ и машины RAM_k предположим, что оракульная $RAM'_{k'}$ с доступом к оракулу f' моделирует вычисления $RAM'_{k'}$. Тогда моделирование является *моделированием с метками времени*, если существует $O(k')$-временной алгоритм $Q(\cdot,\cdot)$ такой, что выполняется следующее условие. Пусть (i,a,v) – j-тое сообщение, посланное $CPU'_{k'}$ (в процессе повторных выполнений $RAM'_{k'}$). Тогда, число предыдущих сообщений формы («запомнить»,a,$\cdot$), посланных $CPU'_{k'}$, равняется точно (j,a).

Далее отметим, что алгоритм $Q(j,a)$ может вычислить любую версию обращения **ЦП** к ячейке памяти a в раунде j. Обозначим номер такой версии как *версия*(a).

Таким образом, чтобы «знать» номер версии любого адреса в некоторый момент времени, достаточно для **ЦП** сохранить счет числа шагов, которые выполняются. Подчеркнем, что **ЦП** не мог бы позволить себе хранить номер версии всех адресов памяти, так что проставление меток времени важно только для получения доказуемой защиты программ от вмешательства.

5.6. Сведение защиты программ к забывающему моделированию на RAM-машине

Наконец сейчас будет показано, как *свести задачу защиты программ к задаче моделирования RAM-машины на забывающей RAM-машине*. Эта задача заключается в сокрытии модели доступа, полностью игнорируя тот факт, что содержимое памяти и коммуникаций между **ЦП** и **МП** доступно для противника. Мы начинаем со сведения задачи достижения слабой

защиты программ (т.е., защита от невмешивающихся противников) к построению забывающего *RAM*-моделирования. Далее мы сводим задачу защиты программ (доказуемой защиты от вмешательства) к построению забывающего моделирования с метками времени.

Напомним, что противник называется *невмешивающимся*, если все выбранные им входы инициируют выполнение программы на них и он читает содержимое памяти и коммуникаций между **ЦП** и **МП** при таком выполнении. Без потери общности, достаточно рассматривать противников, которые только читают коммуникационные ленты (так как содержимое ячеек памяти определено входом и коммуникациями между **ЦП** и **МП**). При использовании забывающего моделирования универсальной
RAM-машины остается только скрыть содержимое «области значений» в сообщениях, обмениваемых между **ЦП** и **МП**. Это делается посредством шифрования, которое использует случайный оракул.

Теорема 5.1. Пусть $\{RAM_k\}_{k\in \mathbf{N}}$ - вероятностная *RAM*-машина, которая выполняет забывающее моделирование универсальной *RAM*-машины. Кроме того, предположим, что t шагов оригинальной *RAM*-машины моделируются за менее чем $g(t)t$ шагов забывающей *RAM*-машины. Тогда существует компилятор, который защищает программы от невмешивающихся противников с затратами не боле $O(g(t))$.

Доказательство. Информация, доступная невмешивающемуся противнику состоит из сообщений, обмениваемых между **ЦП** и **МП**. Напомним, что сообщения от CPU_k к MEM_k имеют форму (i,a,v), где $i\in\{\textit{выборка}, \textit{сохранить}, \textit{стоп}\}$, $a\in\{1,2,...,2^k\}$ и $v\in\{0,1\}^{O(k)}$, в то время как сообщения от MEM_k к CPU_k имеют форму $v\in\{0,1\}^{O(k)}$. При забывающем моделировании, по определению, «область адресов» (т.е., a) не вскрывает никакой информации относительно входа $y=(\Pi_f,x)$. Просто необходимо устранить возможность, когда «область команд» (т.е., i) «давала бы» какой-либо полезную информацию для противника. Следовательно, осталось только завуалировать содержимое области значений (т.е., v) так, чтобы **ЦП** мог восстанавливать оригинальные значения. Идея состоит в том, чтобы выполнить завуалировать v, используя оракул, доступный **ЦП**.

Для завуалирования машина CPU_k содержит специальный счетчик, обозначаемый *счт*, инициализированный нулевым значением. Мы

модифицируем RAM_k добавлением случайного оракула, обозначаемого f. Понятно, что новый случайный оракул может быть объединен со случайным оракулом, используемым при забывающем моделировании. Всякий раз, когда CPU_k должен сохранять значение в памяти MEM_k, счетчик *счт* увеличивается и значение v шифруется посредством пары ($v \oplus f$(*счт*),*счт*), где $\oplus$ обозначает поразрядную операцию «исключающую или». При восстановлении пары (u,j), завуалированное значение восстанавливается посредством вычисления $u \oplus f(j)$. Подчеркнем, что и завуалирование, и восстановление может быть легко выполнены, когда имеется доступ к оракулу f.

Компилятор C, защищающий программное обеспечение, функционирует следующим образом. На входе параметр k и программы **П**, состоящей из последовательности команд $\pi_1,..,\pi_n$, компилятор однородно выбирает функцию f и множества $\mathbf{П}_f=(\pi_1 \oplus f(2^k+1),2^k+1),...,(\pi_n \oplus f(2^k+n),2^k+n)$.

Так как общее время выполнения машины RAM_k во всех экспериментах, инициированных противником, является не более 2^k, мы никогда не используем тот же самый аргумент f для двух различных операций шифрования. Это следует из того, что завуалирование (которое фактически использует шифр «одноразовый блокнот») является абсолютно безопасным (в информационно-теоретическом смысле), и следовательно, противник не получает никакой информации относительно оригинального содержания области значений. ■

Напомним, что на практике можно заменять случайный оракул на псевдослучайный. Следовательно, результат теоремы будет верен только для противников, ограниченных по времени некоторым полиномом. В этом случае, компилятор на входном параметре k и программе **П** равномерно выбирает псевдослучайную функцию f. Функция f может быть аппаратно реализовано в CPU_k. Таким образом, **ЦП** способен вычислять f на входах длины k и poly(k)-временной противник не может различать поведение этого **ЦП** от **ЦП**, описанного в доказательстве теоремы. Следовательно, любой невмешивающийся противник, который что-либо вычисляет мог бы вычислять то же самое время с доступом только к оракулу спецификаций. Подобные замечания верны и для следующей теоремы.

<u>Теорема 5.2.</u> Пусть $\{RAM_k\}_{k \in \mathbf{N}}$ - вероятностная *RAM*-машина, которая выполняет забывающее моделирование с метками времени универсальной *RAM*-машины. Кроме того, предположим, что t шагов оригинальной

RAM-машины моделируются меньше, чем за $g(t)t$ шагов забывающей *RAM*-машины. Тогда существует компилятор, который защищает программное обеспечение от вмешивающихся противников, с затратами не более $O(g(t))$.

Доказательство. Для защиты от противника, который может изменять содержание сообщений, обмениваемых между **ЦП** и **МП**, используются схемы аутентификации. Без потери общности, ограничимся противником, который только изменяет сообщения в направлении от **МП** к **ЦП**.

Метка аутентификации будет зависеть от значения, которое хранится в фактической ячейке памяти и от количества предыдущих команд «сохранить» в этой ячейке. Интуитивно, такая метка аутентификации предотвращает возможность изменять значения, заменять его значением, хранимым в другой ячейке, или заменять его значением, которое было сохранено ранее в той же самой ячейке.

Центральный процессор CPU_k, рассмотренный в предыдущей теореме, далее модифицируется следующим образом. Модифицированная машина CPU_k имеет доступ к еще одной случайной функции, обозначаемой f. Эта функция может быть объединена с другими. В случае если CPU_k желает сохранить завуалированное значение v в ячейке памяти он сначала определяет текущий номер *верси*(a). Отметим, что номер *версии*(a) может быть вычислен CPU_k в соответствии с определением моделирования с метками времени. Модифицированная машина CPU_k теперь посылает сообщение (сохранить, a, (v,f(a,*версия*(a),v))) вместо сообщения («сохранить»,a,v), посланного первоначально. После получения сообщения (v,t) из **МП** в ответ на запрос («выборка»,a,·), модифицированная машина CPU_k определяет текущее значение номера *версия*(a) и сравнивает t с f(a, *версия*(a),v). В случае, если эти два значения равны CPU_k работает как и прежде. В противном случае, CPU_k немедленно останавливается, предотвращая, таким образом, защиту от вмешательства. Таким образом, попытки изменить сообщения от **МП** к **ЦП** будут обнаружены с очень высокой вероятностью. ■

5.7. Нетривиальное решение задачи забывающего моделирования

5.7.1. Вводные замечания

Отметим, что тривиальное решение для забывающего моделирования *RAM*-машины заключается в полном сканировании фактической памяти RAM_k-машины для каждой операции доступа к виртуальной памяти

(которая должна быть выполнена для оригинальной *RAM*-машины). Далее описывается первое нетривиальное решение [GO,O] задачи забывающего моделирования RAM_k-машины посредством вероятностной $RAM'_{k'}$. Это решение еще называется решением задачи «Квадратного корня» [GO].

Пусть заранее известен объем памяти, обозначаемый *m*, требуемый для соответствующей программы. Ниже мы показываем, как моделировать такую *RAM*-машину посредством забывающей *RAM*-машиной с объемом памяти $m+2\sqrt{m}$ таким образом, что *t* шагов оригинальной *RAM*-машины моделируются за $O(t\sqrt{m})$ шагов на забывающей *RAM*-машине.

Всякий раз, когда мы говорим о доступе к *виртуальном* памяти, мы подразумеваем доступ к памяти, требуемый для оригинальной *RAM*-машины, которая моделируется. Доступ к памяти при забывающем моделировании *RAM*-машины рассматривается как фактический доступ к памяти. Кроме того, без потери общности, будем понимать, что виртуальная операция доступа состоит из обработки содержимого единственной ячейки памяти (т.е., *выборка*(*i*), сопровождаемая командами *сохранить*(*i*,·) для некоторого *i*).

5.7.2. Общее описание алгоритма «Квадратного корня»

Интуитивно, чтобы полностью скрыть виртуальную модель доступа, мы должны скрыть следующее:

- к каким виртуальным ячейкам осуществляется доступ и в каком порядке?
- сколько раз к конкретной виртуальной ячейке осуществляется доступ (в случае, если к ней обращались)?

В первом случае достаточно каким-либо образом «перемешать» память так, чтобы противник не знал, какой фактический адрес памяти соответствует данному виртуальному адресу. Во втором случае, мы должны убедиться, что к любой («перемешанной») локальной памяти осуществляется доступ более одного раза. Высокоуровневые шаги моделирования следующие.

Алгоритм КК

1. Инициализация: Первые $m+\sqrt{m}$ слов при моделировании *RAM*-машины расположены так, чтобы содержать *m* виртуальных адресов (к которым оригинальная *RAM*-машина обращается в процессе выполнения) и $\sqrt{m}$ «фиктивных» слов (см. рис.5.1). Оставшиеся $\sqrt{m}$ слов распределены так, чтобы обслужить

дополнительную («краткосрочную») память. Здесь и далее эта память обозначается *защ*. (см. рис. 5.1).

2. Моделирование шагов RAM-машины: До тех пор пока моделирование *RAM*-машины не остановится, выполнить. (Моделирование выполняется за *периоды*, каждый из которых состоит из $\sqrt{m}$ шагов оригинальной/моделируемой машины). В каждом таком периоде выполняются следующие шаги.

2.1. Случайно переставить содержимое ячеек от 1 *до* $m+\sqrt{m}$. То есть необходимо равномерно выбрать перестановку π над целыми числами от 1 до $m+\sqrt{m}$ и забывающим образом переместить содержимое виртуального слова i в фактическое слово π_i. Необходимо подчеркнуть, что память *защ* (т.е., ячейки от ($m+\sqrt{m}+1$) до ($m+2\sqrt{m}$)) не участвует в этом случайном перемешивании. Таким образом, фактические адреса от 1 до $m+\sqrt{m}$ называются *перемешанной памятью*.

2.2. Моделировать $\sqrt{m}$ *виртуальных операций доступа к памяти оригинальной RAM-машины.* В процессе моделирования мы сохраняем значения виртуальных операций доступа, восстановленных и модифицированных в течение текущего периода в памяти *защ*. Так как размер памяти *защ* равняется числу возможных операций доступа в одном периоде, можно хранить все значения, восстановленные в течение текущего периода в памяти *защ*. Доступ к памяти оригинальной *RAM*-машины, скажем доступ к виртуальному слову i, моделируется следующим образом:

- сканируется вся память *защ* и проверяется, находится ли содержимое виртуального слова i в одном из слов памяти *защ*. Подчеркнем, что здесь мы обращаемся к каждой ячейке памяти *защ* в предопределенном порядке независимо от того, находится ли там виртуальное слово, которое нам надо;
- в случае, если i-тое виртуальное слово не найдено в памяти *защ*, мы восстанавливаем его из фактического слова $\pi(i)$, которое является текущей ячейкой i-того виртуального слова в течение этого периода;
- в противном случае (т.е., в случае, если i-тое виртуальное слово найдено в памяти *защ*), мы получаем доступ к

следующему «фиктивному слову» в перемешанной памяти (например, мы обращаемся к фактическому адресу $\pi(m+j)$, где j - число шагов, моделируемых в текущем периоде);

- в любом случае модифицируемое значение для i-той виртуальной ячейки записано (забывающим образом) в память *зщт* посредством сканирования заново всех слов памяти *зщт*.

3. *Модифицировать перемешанную память.* В конце периода, используются значения, сохраненные в памяти *зщт* для модификации забывающим образом содержимого перемешанной памяти.

Перед тем как приступить к деталям реализации вышеупомянутых шагов, сделаем несколько замечаний относительно того, почему они составляют забывающее моделирование. Далее покажем, как осуществить доступ к памяти на шаге 1 фиксированным образом а, следовательно, независимо от входа и виртуальной модели доступа оригинальной *RAM*-машины. Различают два типа операций доступ к памяти, выполненных на шаге 2: полное сканирование памяти *зщт* (т.е., осуществление доступа к каждому из слов дважды на каждую виртуальную операцию доступа) и

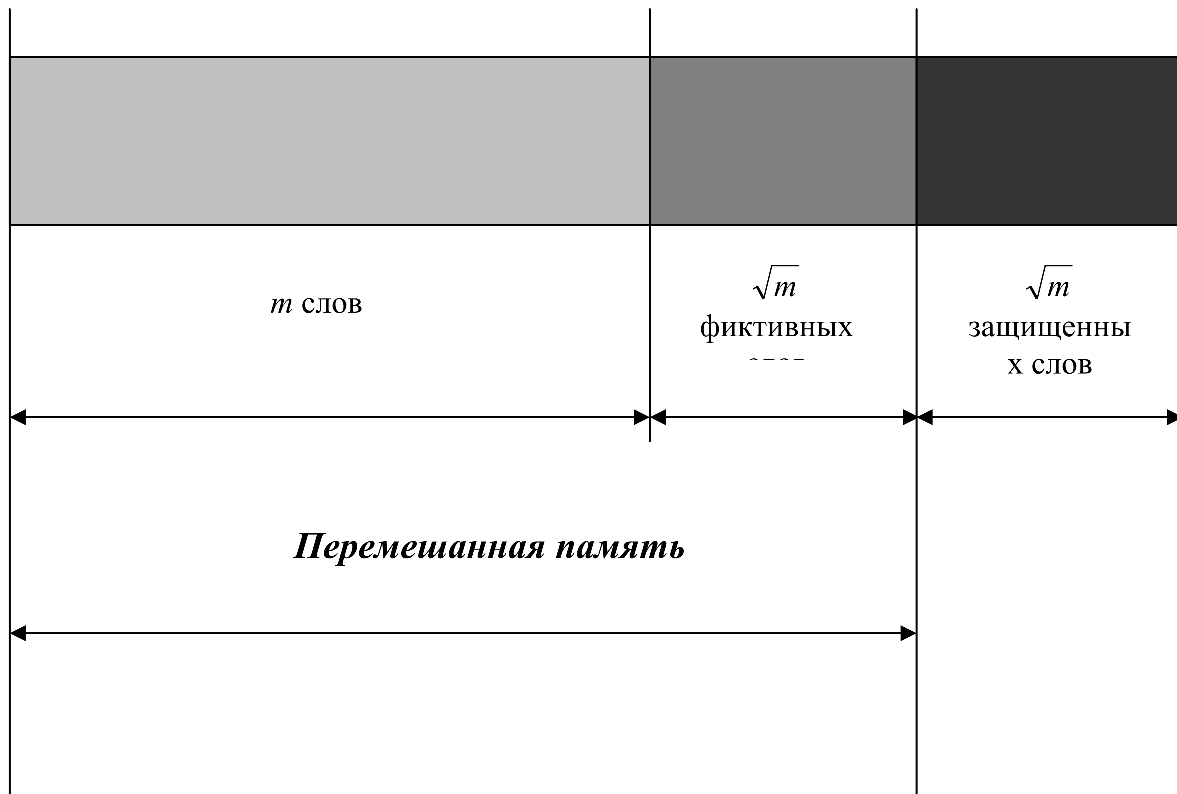


Рис. 5.1. Структура данных для решения задачи «Квадратного корня»

осуществление доступа к $\sqrt{m}$ ячейкам перемешанной памяти во время каждого периода. Для каждых возможных $\sqrt{m}$ виртуальных операций доступа, последние $\sqrt{m}$ фактических операций доступа равномерно распределены среди всех $\sqrt{m}$ подмножеств $\{1,...,m+\sqrt{m}\}$, где распределение вероятностей индуцировано выбором перестановки π. Таким образом, фактический доступ, выполняемый на шаге 2, не открывает никакой информации относительно виртуальных операций доступа, выполняемых в этом шаге. Легко увидеть, что шаг 3 не создает никаких новых трудностей, поскольку он может быть сделан при выполнении операций фактического доступа на шагах 1 и 2 в обратном порядке.

5.7.3. Реализация алгоритма «Квадратного корня»

Сначала, мы покажем, как выбирать и сохранять в **ЦП** случайную перестановку над $\{1,2,...,n\}$, используя $O(\log n)$ памяти и случайный оракул. Идея состоит в том, чтобы использовать оракул для проставления меток случайно выбранных элементов и различные целые числа из множества меток, обозначаемого T_n. Перестановка получается посредством сортировки элементов в соответствии с их метками. Если же необходимо получить вероятность коллизии ε (т.е., для нашего приложения $\varepsilon=2^{-2k}$), достаточно иметь метки, выбранные случайно из множества $T_n=\{1,2,...,n^2/\varepsilon\}$,. Пусть $\tau:\{1,2,...,n\}\rightarrow T_n$ - случайная функция, тривиально созданная случайным оракулом. В этом случае $\pi(i)=k$, тогда и только тогда, когда $\pi(i)$ - наименьший элемент в $\{\tau(j){:}1\leq j\leq n\}$. В нашем случае $n=m+\sqrt{m}$, а именно n элементов состоят из m виртуальных адресов, связанных с целыми числами $1,...,m$ и $\sqrt{m}$ макетов, связанных с $(m+1,..., m+\sqrt{m}\}$.

Теперь мы имеем дело с задачей забывающей сортировки n элементов посредством меток. Определяющее условие состоит в том, что *RAM*-машина, которая выполняет сортировку, может хранить только фиксированное число значений одновременно. Идея состоит в том, чтобы «выполнить» сортирующую сеть Батчера [GO], который позволяет

сортировать n элементов, выполняя $n\lceil \log_2 n \rceil^2$ сравнений. Каждое сравнение «выполняется» посредством осуществления доступа к двум соответствующим словам, чтением их содержания и записью этих значений обратно в необходимом порядке. Последовательность операций доступа к памяти, сгенерированной для этой цели фиксирована и не зависит от входа. Отметим, что забывающая *RAM*-машина может легко вычислять в каждой точке, какое сравнение требуется для реализации следующего. Это следует из простой структуры сети Батчера, которая является однородной относительно логарифмического пространства. Этот алгоритм будет работать, если мы сохраняем метку каждого элемента вместе с самим элементом (виртуальное слово или макет).

Далее мы точно определим, как осуществить доступ к виртуальной ячейке или к макету i. Отметим, что после шага 1 виртуальные ячейки от 1 до m (также как и макеты от m+1 до $m+\sqrt{m}$) сортируются согласно их меток (т.е., $\tau(\cdot)$). Таким образом, фактический доступ в перемешанную память на шаге 2 выполняется двоичным поиском необходимой метки. А именно, предположим, что мы хотим получить доступ к элементу $i\in\{1,...,m+\sqrt{m}\}$. Затем, вместо того, чтобы непосредственно достичь фактической ячейки $\pi(i)$, как предлагается выше, мы выполняем двоичный поиск метки $\pi(i)$. Этот двоичный поиск заканчивается на фактической ячейке $\pi(i)$. Помимо этого другие фактические операции доступа, выполняемые во время поиска, полностью определены $\pi(i)$. Таким образом, эти дополнительные фактические операции доступа не открывают никакой информации противнику.

Далее описываются две альтернативных реализации шага 3. Первый вариант - реверсия модели доступа на шаге 2. Второй вариант – полная сортировка фактической памяти (то есть, все $m+2\sqrt{m}$ слов, включая память *зщт*) дважды, как описано в алгоритме. Первая сортировка выполняется в соответствии с ключом (v,σ), где v - виртуальный адрес (∞ - для макетов) и $\sigma\in\{0,1\}$ указывает, исходит ли это слово из памяти *зщт* или из перемешанной памяти. Таким образом, сортируемый список имеет виртуальные адреса, появляющиеся так, чтобы некоторые из них появляются в двойном экземпляре один за другим (одна версия из памяти *зщт*, а другая из перемешанной памяти). Затем, мы сканируем этот список и для каждого виртуального адреса, появляющегося в дубликате, маркируется второе местонахождение (возникающее из перемешанной

памяти) также как и макет (т.е., ∞). В заключение, мы снова полностью сортируем память, но на сей раз виртуальными адресами. Фактически, эта дополнительная сортировка не нужна, поскольку память будет подвергнута пересортировке когда-либо при следующем выполнении шага 1.

Далее следует детальное описание шага 2. Главная идея при этом моделировании состоит в том, чтобы осуществить доступ к каждой виртуальной ячейке в «перемешанной памяти» только в течение каждого шага периода. Как только осуществится доступ к некоторой виртуальной ячейке, необходимо сохранить версию этой виртуальной ячейки в памяти *защ*. В течение шага 2, *счт* содержит число виртуальных операций доступа, моделируемых в текущем периоде. Переменная *счт* - первоначально содержит 0 и увеличивается, пока достигнет значения $\sqrt{m}$. Булева переменная *found* будет инициироваться, если требуемое значение найдено в памяти *защ*. Когда оригинальная *RAM*-машина осуществляет доступ к i-тому виртуальному слову, забывающая *RAM*-машина работает следующим образом:

- (2a) сканирует полностью память *защ* и ищет виртуальный адрес i. А именно, для j, пробегающему значения от $m+\sqrt{m}+1$ до $m+2\sqrt{m}$, доступ к фактической ячейке памяти j переменная *found* устанавливается в *true* и сохраняется в ЦП, если виртуальный адрес i совпадает с фактической ячейкой j. (Переменная *found* инициализирована в значение *false* до этого сканирования и остается такой же, если виртуальный адрес i не был найден);
- (2b) если *found*=*false*, тогда забывающая *RAM*-машина осуществляет доступ к слову с меткой $\pi(i)$ и сохраняет содержимое в ЦП. Как показано выше, это реализуется посредством двоичного поиска метки $\pi(i)$;
- (2c) если *found*=*true*, тогда забывающая *RAM*-машина осуществляет доступ к слову с меткой $\pi(m+счт)$, которое является макетом. Это также реализуется посредством двоичного поиска метки $\tau(m+счт)$;
- (2d) просматривает полностью память *защ* снова и записывает модифицируемое значение i-того виртуального слова в памяти *защ*. А именно, для $m+\sqrt{m}+1$ до $m+2\sqrt{m}$ доступ к фактической ячейке памяти j запоминается в ее модифицированном значении виртуального адреса i, если адрес j содержит старое значение

виртуального адреса i (т.е., *found*=*true*), либо *found*=*false* и j - первое пустое слово в памяти *зщт*.

- Значение *счт* увеличивается на 1.

Подчеркнем, что невмешивающийся противник не может видеть, сохранил ли ЦП значения или нет и, таким образом, не может различать выполнение шага 2b от выполнения шага 2c. Ясно, что шаги 2a и 2d имеют фиксированную модель доступа и, таким образом, не никакая информация, полезная для такого противника, не вскрывается.

5.7.4. Анализ алгоритма «Квадратного корня»

Как обсуждалось выше, последовательность фактических операций доступа к памяти забывающей *RAM*-машины действительно не открывает никакой информации относительно последовательности виртуальных операций доступа к памяти оригинальной *RAM*-машины. Это действительно так, потому что во время шагов 1, 2a, 2d и 3 фактическая модель доступа фиксирована, в то время как во время шагов 2b и 2c фактические модели доступа неразличимы и «случайны».

Теперь остается вычислить затраты на моделирование (т.е., отношение числа операций доступа, выполненных на забывающей *RAM*-машиной к числу оригинальных операций доступа). Далее мы вычисляем общее число фактических операций доступа, выполняемых на период (т.е., $m+2\sqrt{m}$ виртуальных операций доступа). Число фактических операций доступа на шаге 1 определено числом сравнений в сортирующей сети Батчера, а именно, $O(m\log^2 m)$. То же самое делается на шаге 3. Что касается шага 2, каждая виртуальная операция доступа выполняется за $2\sqrt{m}+\log_2(m+\sqrt{m})=O(\sqrt{m})$ фактических операций доступа. Это составляет амортизационные затраты $O(\log^2 m\sqrt{m})$. Фактически, вышеупомянутый выбор параметров (то есть, размер памяти *зщт*) не оптимален.

При использовании памяти *зщт* размера s, мы получаем амортизационные затраты

$$\frac{O(m\log^2 m)}{s}+(2s+1+\log m),$$

которые минимизированы установкой $s=\Theta(\log m\sqrt{m})$.

5.8. Заключительные замечания

В данной главе был представлен компилятор, который трансформировал *RAM*-программы в эквивалентные программы, которые

предотвращают попытки противника выяснить что-либо относительно этих программ при их выполнении. Перенос был выполнен на уровне команд, а именно операции доступа к памяти для каждой команды заменялись последовательностью избыточных операций доступа к памяти. Понятно, что все формулировки и результаты, показанные выше, применимы к любому другому уровню детализации выполнения программ. Например, на уровне «пролистывания» памяти это означало бы, что мы имеем дело с операциями «получить страницу» и «сохранить страницу», как с атомарными (базовыми) командами доступа. Таким образом, единственная операция «доступ к странице» заменяется последовательностью избыточных операций «доступ к странице». В целом исследуется механизм для забывающего доступа к большому количеству незащищенных ячеек памяти при использовании ограниченного защищенного участка памяти. Применение к защите программ было единственным приложением, обсужденным выше, но возможны также и другие приложения.

Одно из возможных приложений – это управление распределенными базами данных в сети доверенных сайтов, связанных небезопасными каналами. Например, если в сети сайтов нет ни одного, который содержал бы полную базу данных, значит необходимо распределить всю база данных среди этих сайтов. Пользователи соединяются с сайтами так, чтобы можно было восстановить информацию из базы данных таким образом, чтобы не позволить противнику (который контролирует каналы) изучить какая часть базы данных является наиболее используемой или, вообще, узнать модель доступа любого пользователя. В данном случае не требуется скрывать факт, что запрос к базе данных был выполнен некоторым сайтом в некоторое время, - просто надо скрывать любую информацию в отношении фрагмента необходимых данных. Также принимается предположение о том, что запросы пользователей выполняются «один за другим», а не параллельно. Легко увидеть, что забывающее моделирование *RAM*-машины может применяться к этому приложению посредством ассоциирования сайтов с ячейками памяти. Роль центрального процессора будет играть сайт, который в текущий момент времени запрашивает данные из базы и информация о моделировании может циркулировать между сайтами забывающим способом. Отметим, что вышеупомянутое приложение отличается из традиционной задачи анализа трафика.

Другое приложение - это задача контроля структуры данных, которая следует из определений самотестирующихся программ, рассматриваемых выше. В этой конструкции желательно сохранить структуру данных при использовании малого количества доверенной памяти. Большая часть структуры данных может сохраняться в незащищенной памяти, где и надо решать задачу защиты от вмешательства противника. Цель состоит в том, как обеспечить механизм контроля целостности данных, которые необходимо сохранять посредством забывающего моделирования *RAM*-машиной.

ГЛАВА 6. КРИПТОПРОГРАММИРОВАНИЕ

6.1. КРИПТОПРОГРАММИРОВАНИЕ ПОСРЕДСТВОМ ИСПОЛЬЗОВАНИЯ ИНКРЕМЕНТАЛЬНЫХ АЛГОРИТМОВ

Одним из основных инструментов методологии криптопрограммирования являются *инкрементальные криптографические алгоритмы*. Цель *инкрементальной криптографии* заключается в разработке криптографических алгоритмов обработки электронных данных, обладающих следующим принципиальным свойством. Если алгоритм применяется к электронным данным D для достижения каких-либо их защитных свойств, то применение инкрементального алгоритма к данным D, подвергнувшихся модификации – D', должно осуществляться быстрее, чем необходимость заново обработать первоначальный электронный документ. В тех приложениях, когда указанные алгоритмы используют, например, алгоритмы шифрования электронных документов или их цифровой подписи, требование повышения эффективности инкрементальных алгоритмов является основным. Один из основных методов применения инкрементальных алгоритмов заключается в использовании их аутентификационных признаков для антивирусной защиты [BGG].

При обработке электронных документов инкрементальными алгоритмами рассматриваются такие операции обработки данных как «вставка» и «стирание» для символьных строк или «cut» - «вырезание и помещение в буфер» и «paste» - «извлечение из буфера и вставка» для текста. Основная задача здесь заключается в разработке эффективных инкрементальных алгоритмов для схем цифровой подписи и схем аутентификации сообщений, поддерживающих вышеупомянутые операции по модификации электронных данных. Такие алгоритмы должны обладать основным качественным свойством, а именно *свойством защиты от вмешательства*, что, таким образом, и делает их применимыми для *защиты программ от вирусов и других разрушающих программных средств*.

Основные криптографические примитивы, такие как шифрование и цифровая подпись имеют фундаментальную теоретическую базу. Во многих работах (см., например, обзор в работе [КЛП]) были даны базовые

определения их криптографической стойкости, основанные на обобщенных теоретико-сложностных и теоретико-информационных предположениях. Главная проблема, которая остается и затрудняет использование на практике многих доказуемо стойких теоретических криптоконструкций, заключается в их пространственно-временной неэффективности. Инкрементальность, в этом смысле, является новой мерой эффективности, которая является вполне приемлемой во многих алгоритмических приложениях.

Пусть далее рассматривается процессор, защищенный от физического вмешательства, который имеет ограниченное количество безопасной локальной памяти. Необходимо получить доступ к файлам, находящихся на удаленных (возможно небезопасных) носителях, например, хост-станциях или *www*-серверах. Компьютерный вирус может атаковать удаленную станцию, и исследовать и изменять содержание удаленной информационной среды (но при этом он не имеет доступа к безопасной локальной памяти процессора). Для защиты файлов от таких вирусов, процессор вычисляет для каждого файла аутентификационный признак, как функцию от самого файла и ключа, который хранится в безопасной локальной памяти.

Такая организация защиты при внедрении вируса в файл не позволит вирусу вычислять (или получить каким-либо «известным только вирусу» образом) новый аутентификационный признак, а значит при реализации процесса верификации признака, таким образом, обнаружится вторжение вируса. Следует обратить внимание на то, что для корректной верификации аутентификационного признака защищенный процессор должен заново подтвердить подлинность файлов. Очевидно, что наиболее привлекательным способом такой организации защиты от вирусов является модернизация аутентификационного признака быстрее, чем необходимость его вычисления заново. Эта проблема особенно сложна в том случае, когда защищенная локальная память является не достаточно большой для того, чтобы хранить (даже временно) фрагмент файла или когда «слишком ресурсозатратно» ввести в локальную память полный файл.

Таким образом, основная идея инкрементальных алгоритмов, состоит в том, чтобы воспользоваться какими-либо имеющимися преимуществами организации программно-аппаратного процесса вычислений и найти такие способы криптографических преобразований над электронными данными D которые позволят обрабатывать (в целях их защиты) всякий раз

эти данные не заново, а обрабатывать (посредством быстродействующих криптографических преобразований) уже имеющиеся аутентификационные признаки, которые ранее были получены для D. Когда «изменения» в обрабатываемых электронных данных не велики, инкрементальные методы могут дать большие преимущества по эффективности.

6.2. Основные элементы инкрементальной криптографии

6.2.1. Базовые примитивы

Инкрементальность можно рассматривать для любого криптографического примитива. В данном случае рассматриваются два из них – цифровая подпись и шифрование. Инкрементальность далее рассматривается, как правило, для «прямых» преобразований, а именно для генерации подписи и шифрования, но все рассуждения будут верны и для «сопряженных» преобразований, а именно для верификации подписи и дешифрования.

6.2.2. Операции модификации

При рассмотрении проблемы модификации защищаемого файла в терминах применения фиксированного набора основных операций по модификации электронного документа исследуются следующие операции модификации: замена блока в документе другим; вставка нового блока; удаление старого блока. Операции должны быть достаточно «мощны» для демонстрации реальных модификаций таких как: замена, вставка и удаление. Соответственно также рассматриваются операции «cut» и «paste», например, операции разбиения отдельного документа на два, а затем, вставка двух документов в один.

6.2.3. Инкрементальные алгоритмы

Зафиксируем базовое криптографическое преобразование **T** (например, цифровая подпись документа с некоторым ключом). Каждой элементарной операции модификации текста (например, вставки) будет соответствовать инкрементальный алгоритм **I**. На вход этого алгоритма подаются: исходный файл, значения преобразования **T** на нем, описание операций модификации и, возможно, соответствующие ключи или другие параметры. Это позволяет вычислить значение **T** для результирующего файла. Основная проблема здесь заключается в проектировании схем обработки файлов, с включенными в них эффективными

инкрементальными алгоритмами. Предположим, что имеется подпись $\sigma_{\text{стар}}$ для файла $D_{\text{стар}}$ и файл $D'_{\text{стар}}$, измененный посредством вставки в файл $D_{\text{стар}}$ некоторых данных. Необходимо получить новую цифровую подпись путем подписывания строки, состоящей из $\sigma_{\text{стар}}$ и описания операций модификации над документом $D_{\text{стар}}$. Это схема называется *схемой, зависящей от истории*. Могут иметься приложения, когда такие действия могут применяться. В большинстве же случаев это не желательно, так как когда делается большое количество изменений, то затраты на верификацию подписи (а эти затраты пропорциональны числу изменений) резко увеличиваются. В связи с этим размеры подписи растут со временем. Чтобы избежать таких затрат необходимо использовать *схемы, свободные от истории* или *HF-схемы*. Все нижеприведенные схемы являются схемами, свободными от истории.

6.2.4. Безопасность

Свойство инкрементальности вводит новые проблемы безопасности [Ba3], а, следовательно, «назревает» необходимость новых определений. Рассмотрим случай схем подписи или аутентификации сообщений. Разумно предположить, что противник не только имеет доступ к предыдущим подписанным версиям фалов, но также способен выдавать команды на модификацию текста в существующих файлах и получать инкрементальные подписи для измененных файлов. Такая атака на основе выбранного сообщения (см. приложение) для инкрементальных алгоритмов подписи может вести к «взлому» используемой оригинальной схемы подписи, которая не может быть взломана при проведении противником атак, когда инкрементальные алгоритмы не используются. Кроме того, в некоторых сценариях, например, при вирусных атаках можно предположить, что противник может вмешиваться не только в содержание существующих документов, но и в соответствующие аутентификационные признаки, полученные посредством применения схемы подписи (или схемы аутентификации сообщений). Соответственно рассматриваются два определения безопасности: базовое, когда необходимо противостоять первому вышеописанному противнику, и более сильное понятие безопасности, когда доказывается стойкость защиты от вмешательств.

6.2.5. Секретность в инкрементальных схемах

Исходя из вышесказанного, появляется новая проблема, которая проявляется в инкрементальном сценарии, а именно - проблема секретности различных версий файлов. Предположим μ - подпись для электронных данных M и μ' является подписью несколько измененных данных M'. Тогда, нам необходимо построить такую инкрементальную схему получения подписи μ', в которой последняя (подпись μ') давала бы как можно меньше информации об оригинальном коде M.

6.3. Методы защиты данных посредством инкрементальных алгоритмов маркирования

6.3.1. Инкрементальная аутентификация

Основные определения и обозначения

Пусть **АУТ**(m) - обычный (оригинальный) алгоритм аутентификации сообщений и **АУТ**$_\alpha(m)$- функция маркирования сообщения m, индуцированная схемой **АУТ** с ключом аутентификации α. Пусть **ВЕР**$_\alpha(m,\beta)$ - соответствующий алгоритм верификации, где β={true, false} – предикат корректности проверки.

Далее будут использоваться деревья поиска и, следовательно, необходимо напомнить, что 2-3-дерево имеет все концевые узлы (листья) на одном и том же самом уровне/высоте (как и в случае сбалансированных двоичных деревьев) и каждая внутренняя вершина имеет или 2, или 3 дочерних узла [АХУ]. В данном случае 2-3-дерево подобно двоичному дереву является упорядоченным деревом и, таким образом, концевые узлы являются упорядоченными. Пусть V_h – определяет множество всех строк длины не больше h, ассоциированных очевидным образом с вершинами сбалансированного 2-3-дерева высоты h. Маркированное дерево может рассматриваться как функция $T: V_h \rightarrow \{0,1\}^*$, которая приписывает аутентификационный признак (АП) каждой вершине.

Пусть совокупный аутентификационный признак файла F получен посредством использования 2-3-дерева аутентификационных признаков для каждого из блоков файла $F=F[1],...,F[l]$ (далее такое дерево будет называться *маркированным деревом*). Каждая вершина w ассоциирована с меткой, которая состоит из АП (аутентифицирующих дочерние узлы) и счетчика, представляющего число узлов в поддереве с корнем w.

Алгоритм маркирования

Алгоритм создания 2-3-дерева аутентификационных признаков (алгоритм маркирования) работает следующим образом.

Алгоритм САП$^{2\text{-}3}$

1. Получить для каждого i, признак $(\alpha,F[i])$**АУТ=(**$T(w)$), где w – i-тый концевой узел.
2. Получить для каждого неконцевого узла w, признак $(\alpha,(L_1,L_2,L_3),\textit{рзм})$**АУТ=(**$T(w)$), где L_i - метка i-того дочернего узла w (в случае, если w имеет только два дочерних узла, то $L_3=\gamma$) и *рзм* - число узлов в поддереве с корнем w).
3. Получить для корня дерева признак $(\alpha,(L_1,L_2,L_3),Id,\textit{счт})$**АУТ=(**$T(\lambda)$), где Id - название документа и *счт* – соответствующее показание счетчика (связанное с этим документом).

Инкрементальный алгоритм маркирования

Предположим, что файл F, аутентифицированный маркированным деревом, подвергается операции замены, то есть j-тый блок файла F заменен блоком $F(\sigma)$. Сначала необходимо проверить, что путь от требуемого текущего значения до корня дерева корректен. Для этого необходимо выполнить следующий алгоритм.

Алгоритм ИАМ$^{2\text{-}3}$

Пусть $u_0,...u_h$ - путь из корня $u_0=\lambda$ к j-тому концевому узлу обозначается как u_h. Тогда:

1. Проверить, что алгоритм **ВЕР**$_\alpha$ при верификации принимает $T(\lambda)$ как корректный АП строки $(\alpha,(L_1,L_2,L_3),Id,\textit{счт})$**АУТ=**$T(\lambda)$, где Id - название документа и *счт* - текущее значение счетчика (связанного с этим документом).
2. Для i=1,...,h-1 проверить, что **ВЕР**$_\alpha$ принимает $T(u_i)$ как корректный АП строки $((L_1,L_2,L_3),\textit{рзм})$, где L_i - метка i-того дочернего узла w (в случае, если w имеет только два дочерних узла, то $L_3=\gamma$) и *рзм* - число узлов в поддереве с корнем w)).
3. Проверить, что **ВЕР**$_\alpha$ принимает $T(u_h)$ как корректный АП блока $F[j]$.
4. Если все эти проверки успешны, тогда совокупный АП файла F получается следующим образом.

   4.1. Установить $T(u_h)$:=АУТ($F(\sigma)$).

4.2. Для $i=h-1,...,1$ установить $T(u_i)$:=АУТ($T(u_i1),T(u_i2),T(u_i3)$).

4.3. Установить $T(\lambda)$:=АУТ(($T(u_i0),T(u_i1),T(u_i1)$),Id,*счт*+1).

Следует подчеркнуть, что значения T на всех других вершинах (то есть, не стоящих на пути $u_0,...,u_h$) остаются неизменяемыми.

Следует также отметить, что предлагаемая инкрементальная схема маркирования имеет дополнительное свойство, заключающееся в том, что она безопасна даже для противника, который может «видеть» как отдельные аутентификационные признаки, так и все маркированное дерево и может даже «вмешиваться» в эти признаки. Для каждого файла, пользователь должен хранить в локальной безопасной памяти ключ x схемы подписи, имя файла и текущее значение счетчика. Всякий раз, когда пользователь хочет проверить целостность файла, он проверяет корректность маркированного дерева открытым образом.

Наиболее эффективным является использование инкрементального алгоритма маркирования для защиты программ, использующих постоянно обновляющие структуры данных, например, файл с исходными данными для программ или итерационно изменяемыми переменными.

6.3.2. Инкрементальное шифрование

Вводные замечания

Будем говорить, что инкрементальное шифрование является *стойким относительно некоторой операции модификации*, если (для данной последовательности шифртекстов $E_1,...,E_t$ над соответствующими открытыми данными D_i, $i=1,...,t$ и при инкрементальном получении каждой последовательности E_i, из предыдущих шифртекстов E_{i-1}) извлечение какой-либо информации об оригинальном документе D_1, также как и его измененных версиях D_2,.., D_t (за исключением того факта, что D_i получено посредством применения этой операции модификации к открытым данным D_{i-1}) *не возможно*. Аналогично, рассмотрим любые две последовательности $A=(A_1,...,A_t)$ и $B=(B_1,...,B_t)$ так, что $A_1,B_1\in\Sigma^l$, где Σ - используемый алфавит. Данные A_i (отн., B_i) получены заменой единственного символа в A_{i-1} (отн., B_{i-1}). Тогда, не должны быть различимы последовательность шифртекстов, полученных посредством применения инкрементального алгоритма I при обработке «командой создания» блока данных A_1 и соответствующих «команд замены» в последовательности A, от последовательности шифртекстов, полученных посредством

применения инкрементального алгоритма I при обработке «команды создания» блока данных B_1 и соответствующих «команд замены» данных B.

Схемы инкрементального шифрования

Предлагаемые решения используют стойкую схему вероятностного шифрования E [GM] (см. также приложение). Предположим, что E может использоваться для шифрования символов из Σ и пар (i,δ), где $\delta\in\Sigma$ и i - целое число не большее, чем длина блоков данных в системе. При использовании E сначала будет описан алгоритм, который шифрует версии блоков данных, в которых осуществляется операции замены. Далее будем рассматривать операцию замены одного символа. Шифрованные версии состоят из двух последовательностей шифртекстов, обозначаемых E_1 и E_2. Первая последовательность E_1 является результатом блочного шифрования некоторого файла D=D[1]...D[l], в то время как вторая E_2, шифрует последовательность изменений, обозначаемую M=M[1]...M[t] , посредством которых текущий документ был получен из D. Тогда инкрементальный алгоритм шифрования заключается в конкатенации шифртекста модифицированного документа с E_2. Каждые l шагов алгоритм восстанавливает модифицированные данные и заново шифрует их, используя блочное шифрование и, таким образом, формирует новую последовательность шифртекстов E_1 (одновременно устанавливая E_2 в нулевую последовательность). Сложность этого алгоритма составляет два блочных шифрования на каждое изменение. Важным моментом является то, что конвейерная обработка является достаточно ресурсозатратной.

Теперь предположим, что нам позволено сохранять промежуточные результаты работы в некоторой безопасной памяти («невидимой противником»). В этом случае, как только длина E_2 достигнет l, мы можем начать подготовку новых шифртекстов для данных, обозначаемых D', которые получаются D применением первых l изменений в M. Для этого необходимо выполнить все требуемые вычисления наряду со следующими l изменениями, в то время как M увеличивается до размера $2l$. Таким образом, мы имеем шифртекст, обозначаемый E' данных D' (конечно, теперь текущие данные другие). При замене E_1 на E' и при исключении первых l изменений в M мы получаем зашифрованную форму текущего документа. Следует подчеркнуть, что в такой модели вычислений все эти операции могут выполняться за константное временя. Наконец необходимо отметить, что алгоритм работает в *периодах*, каждый

состоящий из l изменений. В каждом периоде, алгоритм модифицирует шифртекст для сравнения с изменениями, выполненными в предыдущем периоде.

Выше мы предположили, что пользователь может хранить промежуточные результаты (которые требуют памяти размером $O(l)$) в локальной памяти, невидимой противником. Это предположение нереалистично в некоторых сценариях и несовместимо с вышеприведенными определениями для схем инкрементального шифрования. Далее мы используем вышеупомянутые идеи без этого предположения (идеи берутся из [GO,O], см. предыдущую главу). Для этого нам необходимо шифровать данные, используя три последовательности, обозначаемые E_1,E_2 и E_3. Первые две последовательности E_1 и E_2, является точно такими же, как определенные выше. Их достаточно для дешифрования данных. Дополнительная последовательность E_3 – является «шифрованием рабочей области» и обозначается как $W=W[1]...W[2l]$.

Далее мы опишем операции, выполняемые в одном периоде. В нашем описании, мы не упоминаем явно операции шифрования, и, таким образом, всякий раз, когда мы говорим, что мы устанавливаем элемент последовательности W, это должно означать, что соответствующий шифртекст получен и сохранен.

Сначала, мы устанавливаем элементы последовательности W следующим образом: $W[i]:=(i+D[i])$ для $i \leq l$ и $W[i]:=M[i-l]$ для $l+1 \leq i \leq 2l$. Здесь мы предполагаем, что записи изменений имеют форму (i,δ), где i – номер ячейки памяти и δ символ, который размещен в этой ячейке. Затем, мы сортируем пары из W по их левым элементам в соответствии с так называемыми *ключами сортировки* так, что, если два ключа сортировки равны, тогда соответствующие пары хранятся упорядоченными. Определяющим является то, что сортировка выполняется посредством использования эффективной сети сортировки такой, например, как сеть сортировки Батчера [BGG]. Следует подчеркнуть, что всякий раз, когда две пары сравниваются и переставляются (или нет), тогда они заново шифруются посредством E (так что противник не может «понять» были ли они переставлены или нет). Это гарантирует то, что вся процедура сортировки не дает никакой информация противнику. Как только сортировка завершена, алгоритм просматривает W для нахождения всех мест нахождения элементов с одним и тем же ключом сортировки и

сохраняет последнее (которое является новым), в большом фиктивном значении (l+1,...). Теперь, мы посредством ключа сортировки заново сортируем пары из W и получаем последовательность, в которой первые l элементов содержат модифицированный документ D'. В заключение, мы устанавливаем E_1 для хранения в ней зашифрованного блока данных D' и исключаем первые l элементов последовательности E_2.

При использовании *AKS*-сети сортировки [BGG] вышеприведенная реализация инкрементального алгоритма шифрования требует $O(l\log l)$ шагов (в то время как, если мы используем сеть Батчера, то получаем сумму из $O(l)+(l\log_2 l)$ шагов). Эти шаги могут быть распределены равномерно среди l операций модификации, обеспечивающих желаемую сложность.

Как было упомянуто выше, каждый из рассматриваемых алгоритмов можно адаптировать для реализации схемы инкрементального шифрования для операций вставки/удаления (единственного символа), которая является эффективной в строгом смысле. Для этого длина открытых данных устанавливается в некоторых предопределенных пределах (например, между $l/2$ и $2l$). А именно, схема инкрементального шифрования состоит из трех последовательностей E_1, E_2 и E_3, где E_1 и E_2 – определены выше, а E_3 - шифрование рабочей области для некоторого забывающего моделирующего устройства (см. предыдущую главу). Также как и как выше, алгоритм работает в периодах, состоящих из l изменений каждый. В каждом периоде, инкрементальный алгоритм выполняет l изменений предыдущего периода. Это делается посредством моделирования *RAM*-программы, которая содержит структуру данных, обеспечивающую эффективное выполнение операций вставки/удаления, например, 2-3-дерево.

6.4. Вопросы стойкости инкрементальных схем

Обеспечение безопасности в традиционных схемах подписи и шифрования сводится к исследованию поведения противника, который, не зная секретного ключа, пытается осуществить злоумышленные действия в отношении этих схем. Например, в схемах подписи, в том числе, требуется, чтобы противник, не зная ключа, не мог бы подделывать подписи.

В случае инкрементальной криптографии нас будет интересовать информация о *предыдущих версиях документа*, которая может быть получена законным пользователем при проверке текущего документа

вместе с текущей криптографической формой. То есть, предположим, что мы имеем блок данных D вместе с его обработанной криптографической формой и мы говорим, что D был получен из некоторого другого блока данных, именуемого D' посредством удаления единственного символа. *Абсолютная секретность* должна означать то, что противник ничего не может сказать о местоположении удаленного символа. *Частичная секретность* может означать, что противник не может ничего сообщить о значении удаленного символа, но может иметь некоторую информацию относительно его местоположения.

Абсолютная секретность представляет собой естественный интерес в контексте исследования стойкости схем подписи. Предположим, что мы используем инкрементальную схему подписи для многих пользователей. Желательно, чтобы ни один из этих пользователей не мог узнать что-либо из подписи другого пользователя. Частичная секретность также представляет интерес. Предположим, что абонент **A** имеет некоторую инкрементальную рабочую схему, с помощью которой он подписал некоторый документ. Ясно, что **A** не должен заботиться о том, узнает ли **B**, которому он дал такую инкрементальную рабочую схему, что **A** подписал этот документ, используя подпись к некоторому другому документу, т.е. **B** не может выяснить любые детали относительно предыдущего использования инкрементальной схемы.

Определение абсолютной и частичной секретности может быть легко дано с использованием следующей стандартной парадигмы. А именно, для данного блока данных D и подписи к нему нельзя различить, была ли подпись сделана как реакция на команду создания или как реакция на команду модификации. При определении частичной секретности главным аргументом является то, что единственной высвобождаемой информацией является количество изменений для данного блока D.

Схема инкрементальной аутентификации, рассматриваемая выше, удовлетворяет определению частичной секретности.

6.5. Применение инкрементальных алгоритмов для защиты от вирусов

Предлагаемые схемы аутентификации для защиты от вмешивающихся противников при антивирусной защите могут быть использованы следующим образом.

Пусть каждый защищаемый файл вместе с деревом аутентификационных признаков хранится в общедоступной памяти. При

подходящем выборе сложностных параметров затраты памяти для хранения дерева АП могут быть незначительными по отношению к самому файлу. Например, мы можем разделить файл на блоки длины s^2, где s - длина АП (и соответствующего ключа) базовой схемы аутентификации сообщений). Для L-битного файла, мы получим дерево АП с L/s^2 ветвями, которое может быть закодировано двоичной строкой размером $O(L/s)$. Для каждого файла пользователю необходимо хранить только $O(s)$ битов в локальной защищенной памяти. Эти биты используются для хранения ключа схемы аутентификации, имени файла и текущего счетчика версий.

Как только файл подвергся изменениям, дерево АП (хранящееся в небезопасной памяти) и счетчик версий (хранящийся в локальной защищенной памяти) изменяются в соответствии с вышеописанным инкрементальным алгоритмом аутентификации. В любое время, как только целостность файла подверглась сомнению, можно верифицировать корректность дерева АП очевидным образом.

Базовая схема аутентификации сообщений может быть любой из стандартных. Например, режим генерации имитовставки алгоритма ГОСТ 28147-89 или любые схемы аутентификации, использующие псевдослучайные функции [Ba1].

Кроме того, предлагаемая схема имеет дополнительное свойство безопасности, а именно, даже, если противник может просматривать дерево АП и даже изменять его, схема аутентификации по-прежнему остается безопасной.

Другое направление инкрементальной криптографии заключается в использовании ее методов при защите программ, которая определяется в терминах предыдущего раздела. А именно, такая конструкция состоит из процессора с ограниченным количеством локальной памяти для хранения и доступа к информации, хранящейся в удаленной незащищенной памяти. Моделирование должно быть забывающим в том смысле, что фактическая модель доступа не дает никакой информации об оригинальной модели доступа. Перенос забывающего моделирования *RAM*-программ в инкрементальную схему шифрования очевидно. Роль процессора играет пользователь, в то время как роль удаленной памяти ассоциирована с шифрованием. Схема защиты программ с полилогарифмическими затратами существует [O] и, используя эти результаты, можно показать, что эффективность схемы инкрементального шифрования при защите программ не хуже, чем ее эффективность при защите электронных данных, рассматриваемой в настоящей главе.

Однако идеи, используемые для защиты программ (в смысле предыдущего раздела и [GO,O]) можно адаптировать для получения инкрементальной схемы шифрования для операций вставки/удаления (единственного символа), которые является эффективным в строгом смысле, т.е. как число шагов моделирования на одну оригинальную операцию. Адаптация достигается «конвейерной обработкой», описание которой дано выше.

ГЛАВА 7. МЕТОДЫ И СРЕДСТВА АНАЛИЗА БЕЗОПАСНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

7.1. Общие замечания

Широко известны различные программные средства обнаружения элементов РПС - от простейших антивирусных программ-сканеров до сложных отладчиков и дизассемблеров - анализаторов и именно на базе этих средств и выработался набор методов, которыми осуществляется анализ безопасности ПО.

Авторы работ [ЗШ,ПБП] предлагают разделить методы, используемые для анализа и оценки безопасности ПО, на две категории: контрольно-испытательные и логико-аналитические (см. рис.7.1). В основу данного разделения положены принципиальные различия в точке зрения на исследуемый объект (программу). Контрольно-испытательные методы анализа рассматривают РПС через призму фиксации факта нарушения безопасного состояния системы, а логико-аналитические - через призму доказательства наличия отношения эквивалентности между моделью исследуемой программы и моделью РПС.

В такой классификации тип используемых для анализа средств не принимается во внимание - в этом ее преимущество по сравнению, например, с разделением на статический и динамический анализ.

Комплексная система исследования безопасности ПО должна включать как контрольно-испытательные, так и логико-аналитические методы анализа, используя преимущества каждого их них. С методической точки зрения логико-аналитические методы выглядят более предпочтительными, так как позволяют оценить надежность полученных результатов и проследить последовательность (путем обратных рассуждений) их получения. Однако эти методы пока еще мало развиты и, несомненно, более трудоемки, чем контрольно-испытательные.

7.2. Контрольно-испытательные методы анализа безопасности программного обеспечения

Контрольно-испытательные методы - это методы, в которых критерием безопасности программы служит факт регистрации в ходе

тестирования программы нарушения требований по безопасности, предъявляемых в системе предполагаемого применения исследуемой программы [ЗШ]. Тестирование может проводиться с помощью тестовых запусков, исполнения в

Рис. 7.1. Методы и средства анализа безопасности ПО

виртуальной программной среде, с помощью символического выполнения программы, ее интерпретации и другими методами.

Контрольно-испытательные методы делятся на те, в которых контролируется процесс выполнения программы и те, в которых отслеживаются изменения в операционной среде, к которым приводит запуск программы. Эти методы наиболее распространены, так как они не требуют формального анализа и позволяют использовать имеющиеся технические и программные средства и быстро ведут к созданию готовых методик. В качестве примера можно привести методику пробного запуска в специальной среде с фиксацией попыток нарушения защиты и процедур разграничения доступа.

Рассмотрим формальную постановку задачи анализа безопасности ПО для ее решения с помощью контрольно-испытательных методов.

Пусть задано множество ограничений на функционирование программы, определяющих ее соответствие требованиям по безопасности в системе предполагаемой эксплуатации. Эти ограничения задаются в виде множества предикатов $C=\{c_i(a_1,a_2,...a_m)|i=1,...,N\}$ зависящих от множества аргументов $A=\{a_i|i=1,...,M\}$.

Это множество состоит из двух подмножеств:

- подмножества ограничений на использование ресурсов аппаратуры и операционной системы, например оперативной памяти, процессорного времени, ресурсов ОС, возможностей интерфейса и других ресурсов;
- подмножества ограничений, регламентирующих доступ к объектам, содержащим данные (информацию), то есть областям памяти, файлам и т.д.

Для доказательства того, что исследуемая программа удовлетворяет требованиям по безопасности, предъявляемым на предполагаемом объекте эксплуатации, необходимо доказать, что программа не нарушает ни одного из условий, входящих в C. Для этого необходимо определить множество параметров $P=\{p_i|i=1,...,K\}$, контролируемых при тестовых запусках программы. Параметры, входящие в это множество определяются используемыми системами тестирования. Множество контролируемых параметров должно быть выбрано таким образом, что по множеству измеренных значений параметров P можно было получить множество значений аргументов A.

После проведения T испытаний по вектору полученных значений параметров P_i,i=1,...,T можно построить вектор значений аргументов A_i, i=1,...,T.

Тогда задача анализа безопасности формализуется следующим образом.

Программа не содержит РПС, если для любого ее испытания i=1,...,T множество предикатов $C=\{c_j(a_{1i},a_{2i}...a_{Mi})|j=1,...,N\}$ истинно.

Очевидно, что результат выполнения программы зависит от входных данных, окружения и т.д., поэтому при ограничении ресурсов, необходимых для проведения испытаний, контрольно-испытательные методы не ограничиваются тестовыми запусками и применяют механизмы экстраполяции результатов испытаний, включают в себя методы символического тестирования и другие методы, заимствованные из достаточно проработанной теории верификации (тестирования правильности) программы.

Рассмотрим схему анализа безопасности программы контрольно-испытательным методом (рис.7.2).

Контрольно-испытательные методы анализа безопасности начинаются с определения набора контролируемых параметров среды или программы. Необходимо отметить, что этот набор параметров будет зависеть от используемого аппаратного и программного обеспечения (от операционной системы) и исследуемой программы. Затем необходимо составить программу испытаний, осуществить их и проверить требования к безопасности, предъявляемые к данной программе в предполагаемой среде эксплуатации, на запротоколированных действиях программы и изменениях в операционной среде, а также используя методы экстраполяции результатов и стохастические методы.

Очевидно, что наибольшую трудность здесь представляет определение набора критичных с точки зрения безопасности параметров программы и операционной среды. Они очень сильно зависят от специфики операционной системы и определяются путем экспертных оценок. Кроме того в условиях ограниченных объемов испытаний, заключение о выполнении или невыполнении требований безопасности как правило будет носить вероятностный характер.

7.3. Логико-аналитические методы контроля безопасности программ

При проведении анализа безопасности с помощью логико-аналитических методов (см. рис.7.3) строится модель программы и формально доказывается эквивалентность модели исследуемой программы и модели РПС. В простейшем случае в качестве модели программы может выступать ее битовый образ, в качестве моделей вирусов множество их сигнатур, а доказательство эквивалентности состоит в поиске сигнатур вирусов в программе. Более сложные методы используют формальные модели, основанные на совокупности признаков, свойственных той или иной группе РПС.

Формальная постановка задачи анализа безопасности логико-аналитическими методами может быть сформулирована следующим образом.

Выбирается некоторая система моделирования программ, представленная множеством моделей всех программ - *Z*. В выбранной системе исследуемая программа представляется своей моделью *M*, принадлежащей множеству *Z*. Должно быть задано множество моделей РПС $V=\{v_i|i=1,...,N\}$, полученное либо путем построения моделей всех известных РПС, либо путем порождения множества моделей всех возможных (в рамках данной модели) РПС. Множество *V* является подмножеством множества *Z*. Кроме того, должно быть задано отношение эквивалентности определяющее наличие РПС в модели программы, обозначим его $E(x,y)$. Это отношение выражает тождественность программы *x* и РПС *y*, где *x* - модель программы, *y* - модель РПС, и *y* принадлежит множеству *V*.

Тогда задача анализа безопасности сводится к доказательству того, что модель исследуемой программы *M* принадлежит отношению $E(M,v)$, где *v* принадлежит множеству *V*.

Для проведения логико-аналитического анализа безопасности программы необходимо, во-первых, выбрать способ представления и получения моделей программы и РПС. После этого необходимо построить модель исследуемой программы и попытаться доказать ее принадлежность к отношению эквивалентности, задающему множество РПС.

На основании полученных результатов можно сделать заключение о степени безопасности программы. Ключевыми понятиями здесь являются «способ представления» и «модель программы». Дело в том, что на

компьютерную программу можно смотреть с очень многих точек зрения - это

Рис.7.2. Схема анализа безопасности ПО с помощью контрольно-испытательных методов

Рис. 7.3. Схема анализа безопасности ПО с помощью логико-аналитических методов

и алгоритм, который она реализует, и последовательность команд процессора, и файл, содержащий последовательность байтов и т.д. Все эти понятия образуют иерархию моделей компьютерных программ. Можно выбрать модель любого уровня модели и способ ее представления, необходимо только чтобы модель РПС и программы были заданы одним и тем же способом, с использованием понятий одного уровня. Другой серьезной проблемой является создание формальных моделей программ, или хотя бы определенных классов РПС. Механизм задания отношения между программой и РПС определяется способом представления модели. Наиболее перспективным здесь представляется использование семантических графов и объектно-ориентированных моделей.

В целом полный процесс анализа ПО включает в себя три вида анализа:

- лексический верификационный анализ;
- синтаксический верификационный анализ;
- семантический анализ программ.

Каждый из видов анализа представляет собой законченное исследование программ согласно своей специализации.

Результаты исследования могут иметь как самостоятельное значение, так и коррелироваться с результатами полного процесса анализа.

Лексический верификационный анализ предполагает поиск распознавания и классификацию различных лексем (сигнатур) объекта исследования (программы), представленного в исполняемых кодах. При этом лексемами являются сигнатуры. В данном случае осуществляется поиск сигнатур следующих классов:

- сигнатуры вирусов;
- сигнатуры элементов РПС;
- сигнатуры «подозрительных функций»;
- сигнатуры штатных процедур использования системных ресурсов и внешних устройств.

Поиск сигнатур реализуется с помощью специальных программ-сканеров.

Синтаксический верификационный анализ предполагает поиск, распознавание и классификацию синтаксических структур РПС, а также построение структурно-алгоритмической модели самой программы.

Решение задач поиска и распознавания синтаксических структур РПС имеет самостоятельное значение для верификационного анализа программ,

поскольку позволяет осуществлять поиск элементов РПС, не имеющих сигнатуры. Структурно-алгоритмическая модель программы необходима для реализации следующего вида анализа - семантического.

Семантический анализ предполагает исследование программы изучения смысла составляющих ее функций (процедур) в аспекте операционной среды компьютерной системы. В отличие от предыдущих видов анализа, основанных на статическом исследовании, семантический анализ нацелен на изучение динамики программы - ее взаимодействия с окружающей средой. Процесс исследования осуществляется в виртуальной операционной среде с полным контролем действий программы и отслеживанием алгоритма ее работы по структурно-алгоритмической модели.

Семантический анализ является наиболее эффективным видом анализа, но и самым трудоемким. По этой причине целесообразно сочетать в себе три перечисленных выше вида анализа. Выработанные критерии позволяют разумно сочетать различные виды анализа, существенно сокращая время исследования, не снижая его качества.

7.4. Сравнение логико-аналитических и контрольно-испытательных методов анализа безопасности программ

Для сравнения методов предлагаются следующие признаки: представления предметной области, методы решения проблем неразрешимости легитимности и неперечислимости рабочего пространства, а также надежность получаемых результатов [ПБП]. Надежность методов анализа определяется вероятностью *ошибок первого* и *второго рода*. Под ошибкой первого рода понимается принятие за РПС безопасной программы, а под ошибкой второго рода - объявление программы безопасной, когда на самом деле она содержит РПС.

С методической точки зрения логико-аналитические методы выглядят более предпочтительными, так как основываются на формальном подходе и приближают перспективное решение проблемы связанное с доказательством разрешимости множества РПС. Кроме того, они позволяют создать легко применяемые средства анализа, независящие от анализируемых программ. Однако на данное время любой из этих методов имеет существенный недостаток - исследование безопасности проводится лишь относительно некоторого подмножества РПС.

С практической точки зрения, - с точки зрения обеспечения безопасности КС контрольно-испытательные методы обладают рядом

преимуществ, связанных с их привязкой к конкретной КС и программе, а также с их надежностью в отношении ошибок второго рода. Однако затраты, необходимые для организации процесса тестирования, являются преградой для их применения, за исключением критических компьютерных систем.

Из вышесказанного можно сделать вывод, что ни один из методов не имеет решающего преимущества перед другим. Использование методов той и другой группы должно опираться только на их соответствие решаемой задаче, необходимо применять те методы, которые в данной ситуации наиболее эффективны и оправданы.

Разделение методов, их особенности и преимущества показаны в табл.7.1.

Таким образом, проблема анализа безопасности программного обеспечения в условиях распространения РПС является весьма актуальной. Данная проблема находится в тесной связи с проблемами анализа ПО и его верификацией. Без решения данной проблемы невозможно решить задачу создания защищенных КС, гарантированно являющихся безопасными и целостными.

Для полного решения проблемы анализа безопасности программ необходимо осуществить следующие действия [ПБП].

Таблица 7.1

Методы	***Контрольно-испытательные***	***Логико-аналитические***
Способ представления предметной области	Пространство отношений программы с объектами КС.	Пространство программ.
Принцип поиска РПС	Фиксация установления программой нелегитимности отношения доступа к объектам КС.	Доказательство принадлежности программы к множеству РПС.
Поиск проблемы неразрешимости легитимности отношений	С помощью аппроксимации пространства легитимных отношений для данной программы и КС.	С помощью сведения к проблеме разрешимости множества РПС и анализ безопасности относительно разрешимого подмножества РПС.
Решение проблемы перечислимости рабочего	Статистические и экстраполяционные методы теории верификации и	Не требуется.

Методы	***Контрольно-испытательные***	***Логико-аналитические***
пространства	функционального тестирования.	

Продолжение таблицы 7.1

Ошибки первого рода	Весьма вероятны. Чем строже требования, предъявляемые в заданной КС, тем больше вероятность ошибки.	При строгом доказательстве разрешимости подмножества РПС и корректно определенной характеристической функции исключены.
Ошибки второго рода	Маловероятны. Чем строже требования по безопасности, тем меньше вероятность ошибки.	Неизбежны. Определяются мощностью выбранного разрешимого подмножества РПС.
Преимущества	Не требует теоретической подготовки. Допускает использование имеющихся стандартных программных средств. Устойчивость к ошибкам второго рода. Метод отражает требования конкретных КС.	Опирается на формальные методы. Не требует значительных затрат на этапе применения. Высокая надежность полученных результатов относительно выбранного подмножества РПС. Инвариантность метода по отношению к различным классам программ. Позволяет создавать автоматические простые и доступные средства проверки безопасности.
Недостатки	Проведение испытаний требует существенных затрат времени и других ресурсов. Процесс тестирования требует выделения испытательной КС и должен проводится специалистами.	Подтверждены ошибками второго рода – проверяется лишь часть множества РПС.

1. Создать теоретические основы анализа безопасности ПО, создать словарь предметной области и осуществить в рамках этого словаря формальную постановку задачи анализа безопасности ПО;
2. Создать методы анализа безопасности ПО, используя выбранные формальные определения, доказать их эффективность и реализуемость;
3. Создать конкретные программные средства, реализующие методы анализа безопасности программ в конкретных аппаратно-программных средах;
4. Создать методики применения этих средств и оценить их эффективность.

7.5. Способы тестирования программного обеспечения при испытаниях его на технологическую безопасность

7.5.1. Обобщенные способы анализа программных средств на предмет наличия (отсутствия) элементов разрушающих программных средств

Статистические и динамические способы исследования ПО

Основной особенностью исследования ПО является практическая трудность получения исходных текстов программ. Таким образом, исследователю часто приходится иметь дело с исполняемыми кодами ПО и не очень подробной пользовательской документацией.

Следовательно, одна из важнейших задач тестирования (проверки) программ может быть сформулирована следующим образом: *содержит ли данная программа функцию разрушения (нанесения ущерба)?*

Данная задача сводится, по сути дела, к задаче исследования программы, задаваемой ее объектным или исполняемым кодом. При постановке последней должны разделяться два этапа.

I. Выделение алгоритма программы (или какой-либо интересующей его части) и представление его на языке, удобном для последующего анализа (обычно это язык высокого уровня).

II. Семантический анализ полученного алгоритма для ответа на интересующие вопросы, например, о правильности программы, степени ее надежности, или наличия в ней непротоколированных (недекларированных) функций.

Задача первого этапа решается известными методами дизассемблирования.

Все средства исследования ПО можно разбить на 2 класса: *статические* и *динамические*. Первые оперируют исходным кодом программы как данными и строят ее алгоритм без исполнения, вторые же изучают программу, интерпретируя ее в реальной или виртуальной вычислительной среде. Отсюда следует, что первые являются более универсальными в том смысле, что теоретически могут получить алгоритм всей программы, в том числе и тех блоков, которые никогда не получат управления. Динамические средства могут строить алгоритм программы только на основании конкретной ее трассы, полученной при определенных входных данных. Поэтому задача получения полного алгоритма программы в этом случае эквивалентна построению исчерпывающего набора текстов для подтверждения правильности программы, что практически невозможно, и вообще при динамическом исследовании можно говорить только о построении некоторой части алгоритма.

Два наиболее известных типа программ, предназначенных для исследования ПО, как раз и относятся к разным классам: это отладчик (динамическое средство) и дизассемблер (средство статистического исследования). Если первый широко применяется пользователем для отладки собственных программ и задач построения алгоритма для него вторичны и реализуются самим пользователем, то второй предназначен исключительно для их решения и формирует на выходе ассемблерный текст алгоритма.

Помимо этих двух основных инструментов исследования, можно использовать:

- «дискомпиляторы», генерирующие из исполняемого кода программу на языке высокого уровня;
- «трассировщики», сначала запоминающие каждую инструкцию, проходящую через процессор, а затем переводящие набор инструкций в форму, удобную для статического исследования, автоматически выделяя циклы, подпрограммы и т.п.,
- «следящие системы», запоминающие и анализирующие трассу уже не инструкции, а других характеристик, например вызванных программой прерывания.

Особенности исследования защищенного ПО

Некоторые комплексы программ, особенно импортного производства, могут содержать в себе средства, противодействующие исследованиям (см. главу 12).

В этом случае задача исследования защищенного ПО сводится, в первую очередь, к исследованию и вскрытию самой системы защиты, к анализу ее связи с функциями ПО. Естественно, что система защиты сама может быть защищена на более высоком уровне и так далее, но в любом случае можно начать исследование одним из методов, описанных выше, с самого верхнего уровня. Динамические методы в этом случае оказываются, по крайней мере, не хуже статистических (как будет показано ниже, на самом деле они являются основными), так как система защиты должна получать управление при любом наборе входных данных и ее трасса может быть получена всегда.

Программами, по определению противодействующими их исследованию, являются как средства обеспечения безопасности КС и ее компонентов (системы разграничения доступа, защиты от копирования и т.п.), так и программы, направленные на ее нарушение (компьютерные вирусы, троянские кони и т.п.). Средства противодействия оказываются наиболее важным элементом в таких системах, так как при их отсутствии квалифицированный специалист сможет достаточно быстро разобраться в их логике.

Методы, используемые в ПО для его защиты, базируются на *использовании принципа фон Неймана: программы и данные выглядят и хранятся одинаково, в результате чего программа может модифицировать саму себя*. Этого бывает достаточно для подавления средств статического анализа (поэтому они обычно не пригодны для исследования защищенных комплексов программ). В случае защиты от динамических средств может быть использован тот факт, что изучаемая программа запускается в возмущенной самим средством операционной среде и может это распознать.

Известно, что любую систему защиты можно вскрыть за конечное время. Это следует из того, что ее команды однозначно интерпретируются процессором. При этом время, необходимое для вскрытия хорошей системы защиты, оказывается сравнимым со временем создания защищенной программы заново. Однако не все РПС, особенно вирусы, пишут профессионалы, поэтому часто можно вскрыть или обойти защиту,

найдя ее слабейшее звено. Для этого рекомендуются следующие достаточно универсальные методы.

1. Если программа защищена только от средств статического анализа, она легко изучается динамически, и наоборот.

2. «Метод изменения одного байта» - в момент, когда система защиты сравнивает контрольную информацию (состояние операционной среды, контрольную сумму) с эталонной, простым изменением команды перехода она направляется по нужному пути.

3. Аналогично, результат работы функции, возвращающей текущую контрольную информацию, может быть подменен на эталонное (ожидаемое) значение (например, с помощью перехвата соответствующего прерывания).

4. Когда система защиты расшифровала критичный код, он может быть скопирован в другое место памяти или на диск в момент или вскоре после передачи управления на него. Частный случай – после окончания работы программы весь ее код расшифрован и доступен.

Для исследования высоконадежных профессиональных систем защиты необходимы специальные средства.

Описание способов проведения испытаний, оценки качества и сертификации программных средств

Проведение (организация) тестирования ПО при его проверке на выполнение требований технологической безопасности предполагает определение номенклатуры показателей технологической безопасности ПО, общих методов измерения, испытаний ПО и оценки его качества.

При этом устанавливаются общие правила оценки качества ПО на основе базовых и частных методик его оценки, как в целом, так и по отдельным показателям. При этом частные методики могут разрабатываться как для различных видов (классов) ПО, так и для отдельных программ (комплексов).

При оценке качества ПО каждое его свойство характеризуется показателем в численном выражении. Поскольку в настоящее время показатели технологической безопасности программ только разрабатываются, при оценке данного свойства используются известные показатели качества ПО. Однако технологическая безопасность определяется по отклонениям значений известных показателей от прогнозируемых для каждого типа программ.

Оценка качества ПО представляет собой совокупность операций, включающих в себя: выбор номенклатуры показателей качества оцениваемого ПО; измерение характеристик, сбор и обработку данных по результатам экспериментов (проведения испытаний, тестирования и т.д.); выбор метода (методов) оценивания; расчет оценок значений показателей качества; принятия решения о качестве ПО.

Оценка качества ПО может осуществляться при проведении следующих видов работ:

- определение технических требований к разрабатываемым ПО;
- контроль качества на отдельных этапах разработки ПО;
- испытания и демонстрация работ ПО;
- контроль качества в процессе производства ПО;
- контроль качества при приеме-сдаче и купле-продаже ПО;
- контроль качества и планирование работ при сопровождении ПО;
- принятие решения о снятии ПО с эксплуатации, прекращении производства, разработки.

Состав методического обеспечения проведения испытаний программ

Методическое обеспечение проведения испытаний программных средств включает базовые и частные методики.

А. Базовые методики.

А.1. Методики испытаний:

- методика моделирования программ;
- методика модельных испытаний;
- методика испытаний по требованиям качества;
- методика испытаний по требованиям безопасности.

А.2. Методика оценки показателей качества (ПК):

- методика выбора номенклатуры ПК;
- методика автоматизированной оценки ПК;
- методика экспертной оценки ПК.

А.3. Методики экспертизы ПО:

- методика экспертизы качества ПО;
- методика экспертизы безопасности ПО.

А.4. Методики оценки требований к ПО:

- методика экспертизы требований.

А.5. Методики принятия решений:

- методика расчета относительных оценок ПК;

- методика принятия решения о качестве ПО;
- методика принятия решения о безопасности ПО;
- методика принятия решения о выдаче сертификата качества;
- методика расчета страховых рисков.

Б. Частные методики.

Б.1. По показателям качества:

- методики испытаний, оценки требований и принятия решения о надежности ПО, сопровождаемости, удобстве применения, эффективности, универсальности, корректности и защищенности.

Б.2. По видам ПО:

- методики испытаний, оценки показателей, экспертизы, оценки выполнения требований к отечественным и импортным программным средствам, ОПО, пакетам прикладных программ, системам реального времени, специальному ПО, универсальному ПО, программным средствам защиты информации.

Состав инструментальных средств проведения испытания программ

В состав инструментальных средств проведения испытания программного обеспечения могут входить:

- анализаторы исходных текстов программ;
- анализатор объектных кодов программ;
- программы модельных испытаний;
- программы оценки показателей качества ПО;
- программы оценки показателей безопасности ПО;
- программы экспертизы выполнения требований;
- программы интерфейса эксперта;
- программы принятия решений;
- программы расчета рисков.

Общая номенклатура показателей качества ПО

Номенклатура показателей качества ПО представляет собой систему иерархической структуры, которая отражает логические связи между различными свойствами ПО. Если показатели качества находятся на одном уровне иерархии системы, то свойства, соответствующие этим показателям, соединены логическими связками "И", "ИЛИ". В противном случае либо одно свойство вытекает из другого (логическое следствие), либо свойства логически не зависимы.

Первый уровень системы показателей качества ПО представляет собой совокупность из трех групповых показателей качества, отражающих три комплексных свойства ПО: пригодность, надежность, сопровождаемость.

Описание показателей качества ПО первого уровня приведено в табл. 7.2.

Приведенная номенклатура показателей качества первого уровня является общей для всех программных средств, для нее должно выполняться требование полноты, то есть не допускается внесение дополнительных элементов без переопределения заданных.

Количество последующих уровней иерархии системы показателей качества, состав, количество и содержание показателей на каждом уровне и в целом по системе зависят от того к какой классификационной группировке относится оцениваемое ПО, от области его применения, от вида работ, при которых осуществляется оценивание качества и от совокупности применяемых методов оценки показателей качества.

В методиках и методических материалах по оценке качества различных классов ПО эти характеристики системы показателей качества должны определяться в соответствии с действующими государственными и ведомственными нормативно-техническими документами, устанавливающими номенклатуры показателей качества ПО.

При разработке методов оценки показателей качества и при расчете оценок значений этих показателей необходимо иметь строгое однозначное определение каждого показателя качества ПО.

Каждый показатель качества в системе показателей должен быть определен в виде формального описания следующих четырех определяющих множеств, характеризующих заданный показатель качества.

1). Q_1 - множество проявлений показателя качества.

2). Q_2 - множество значений показателя качества.

3). Q_3 - множество факторов, влияющих на показатель качества.

4). Q_4 - множество критериев показателя качества.

Описание этих множеств приведено в табл.7.3.

Таблица 7.2

Наименование показателя	***Описание показателя***

«Пригодность»	Комплексное свойство ПО удовлетворять потребность пользователей в обработке данных, выражаемое как степень соответствия функциональной спецификации на ПО потребностям в решении множества задач и выполнении всех функций по обработке данных
«Надежность»	Комплексное свойство ПО выполнять свои функции в соответствии со спецификацией в реальных условиях эксплуатации
«Сопровождаемость»	Комплексное свойство ПО, отражающее способность сохранения или повышения свойства надежности при его эксплуатации, в том числе при изменении спецификации на ПО с целью изменения, свойства пригодности

Таблица 7.3

Обозначение	***Наименование множества***	***Описание множества***
Q_1	Множество проявления показателя качества	Множество объектов реального мира, где может проявиться свойство ПО, соответствующее заданному показателю качества
Q_2	Множество значений показателя качества	Множество значений, которые принимает заданный показатель качества
Q_3	Множество факторов, влияющих на проявления показатель качества	Множество элементов ПО, технических решений, способов и приемов разработки и эксплуатации ПО, его свойств, влияющих на изменение показателя качества
Q_4	Множество критериев показателей качества	Множество результатов влияния ПО на реальный мир, характеризующих проявление свойства ПО,

		соответствующее заданному показателю качества

Способы задания определяющих множеств целесообразно рассматривать при определении введенных трех показателей качества первого уровня. В некоторых методиках (методических материалах) по оценке качества ПО эти показатели могут быть определены иначе.

С помощью определяющих множеств показателей качества ПО путем простых логических рассуждений, математической логики и теории множеств доказывается полнота и непротиворечивость выбираемой номенклатуры показателей качества , обосновывается выбор методов оценки показателей качества ПО и показывается корректность (правильность) проведения расчетов оценок значений показателей качества и принятых решений о качестве ПО.

Выбор номенклатуры показателей качества

Выбор номенклатуры показателей качества конкретного ПО осуществляется из системы показателей качества, установленной действующими нормативно-техническими документами на основе принятых (утвержденных) методик (методических материалов) по оценке качества. Выбор номенклатуры показателей представляет собой следующую последовательность операций:

- определение нормативно-технических документов, устанавливающих систему показателей качества ПО;
- формирование из установленной системы показателей номенклатуры показателей качества оцениваемого ПО по i-ому уровню иерархии и установление логических связей между показателями;
- определение каждого выбранного показателя качества ПО i-го уровня;
- доказательство полноты и непротиворечивости выбранной номенклатуры показателей качества i-го уровня.

Выбор конкретных показателей качества ПО, доказательство полноты и непротиворечивости номенклатуры показателей зависят от области

применения и назначения оцениваемого ПО и от целей и задач оценивания качества ПО и проводятся в соответствии со следующими правилами.

1). Объединение всех множеств Q_1 показателей качества по каждому уровню должны совпадать и соответствовать области применения и назначению оцениваемого ПО.

2). Множество Q_1 показателя качества верхнего уровня должно совпадать или быть вложенным в объединение множеств Q_1 подчиненных показателей всех нижних уровней.

3). Множество Q_2 показателей верхнего уровня должно соответствовать цели оценки качества ПО.

4). Должна существовать измеримая функция, ставящая в соответствие каждому элементу множества Q_2 показателя верхнего уровня элементы множества Q_2, соответствующие показателям нижнего уровня (обратная функция может не существовать).

5). Между элементами множества Q_2 показателей одного уровня должно существовать взаимнооднозначное соответствие.

6). Множество Q_3 показателя i-го уровня должно совпадать с объединением множеств Q_3 подчиненных показателей i-го уровня.

7). Для множеств Q_4 справедливо правило 1.

Расширение и уточнение этих правил устанавливается в методиках (методических материалах) по оценке качества конкретных ПО.

Оценка значений показателей качества ПО

Методы оценки качества ПО можно подразделить на следующие 6 групп.

1). Общий метод оценки качества ПО КС.

2). Измерительные методы.

3). Экспертные методы.

4). Расчетные методы.

5). Методы принятия решений о качестве ПО.

6). Прочие методы.

Общий метод оценки качества ПО КС определяет форму, единую методологию и общие принципы разработки и применения других методов оценки качества ПО. Измерительные методы предназначены для измерения, регистрации, учета, контроля и обработки исходных данных для получения оценок. Экспертные методы предназначены для получения оценок показателей качества в условиях отсутствия исходных данных и (или) высокой трудоемкости их получения и (или) недостоверности их и

(или) сложности расчета оценок другими методами. Расчетные методы представляют собой операции по обработке исходных данных и (или) мнений экспертов в соответствии с заданным алгоритмом с целью получения оценок значений показателей качества ПО КС. Методы принятия решений о качестве ПО служат для выработки заключения о качестве исследуемого ПО на основе полученных оценок в соответствии с целями и задачами оценки его качества.

Как правило, при оценке качества ПО используются комбинация нескольких методов.

Общий метод оценки заданного показателя качества ПО устанавливает общую форму правила получения оценки соответствия элементов множества Q_1 показателя элементам множества Q_2 этого показателя. Если задано конечное множество критериев показателей качества ПО Q_4 в виде числовых характеристик элементов этого множества и заданы весовые значения каждого элемента множества критериев показателя качества на множестве Q_1, то оценка K значения показателя качества ПО по всему множеству Q_1 определяется в виде свертки показателей. Для этого могут использоваться любые известные методы математической статистики.

Измерительные методы оценки показателей качества ПО представляют собой совокупность операций по измерению, регистрации, учету, контролю и расчету характеристик и элементов множеств Q_1, Q_3 и Q_4. Эти методы должны быть ориентированы на получение оценок таких характеристик ПО и результатов его работы, как:

- состав и количество операторов исходного текста;
- время работы ПО;
- число строк комментариев;
- число операторов и операндов;
- число исполненных операторов;
- количество ветвей и маршрутов в программе;
- число точек входа/выхода;
- время реакции;
- объем ввода/вывода;
- количество модулей;
- количество переходов по условию;
- количество циклов;
- количество инструкций эксплуатационной документации;

- количество специфицированных функций;
- количество внутренних/внешних переменных;
- время рестарта;
- объем внутренней/внешней памяти;
- число сбоев, отказов при работе ПО и другие.

Этот список специфичен для конкретных видов ПО. Измерительные методы основаны, как правило, на применении инструментальных средств для получения элементарных характеристик ПО и результатов его работы. Таким инструментальным средством может быть средство вычислительной техники, на котором разрабатывается (испытывается, используется) оцениваемое ПО. Для получения отдельных характеристик может потребоваться создание специальных инструментальных средств. В методиках (методических материалах) по оценке качества ПО должно содержаться описание способов получения исходных данных для расчета оценок показателей качества, их полный перечень и описание методов измерения со ссылкой на необходимые инструментальные средства.

С помощью экспертных методов могут быть получены как исходные данные для расчета оценок значений показателей качества, так и сами оценки.

Определение характеристик показателей качества экспертным методом осуществляется группой экспертов-специалистов, компетентных в решении данной задачи. При этом решение базируется на опыте и интуиции экспертов. При использовании экспертных методов необходимо оценивать компетентность и добросовестность группы экспертов. Определение характеристик показателей качества экспертным методом представляет собой следующую последовательность действий.

1). Подбор и подготовка группы экспертов.

2). Постановка задачи эксперту (экспертам).

3). Контроль работы экспертов.

4). Сбор мнений (оценок) экспертов.

5). Оценка компетентности и добросовестности группы экспертов.

6). Расчет экспертной оценки.

Методики получения экспертных оценок значений показателей качества в целях методологического единства должны основываться на общем методе оценки качества, правилах выбора номенклатуры показателей качества и действующих методических материалах по оценке заданного показателя качества оцениваемого ПО.

Методами принятия решений о качестве ПО могут быть получены обобщенные оценки по группе логически связанных и/или не связанных в номенклатуре показателей качества, а также оценки отдельных показателей качества и оценка качества ПО в целом. Принятие решений о качестве ПО осуществляется после того, как получены необходимые оценки показателей качества. Как правило, этих оценок бывает недостаточно для получения обобщенной оценки о качестве ПО в целом, поэтому принятие решений о качестве ПО осуществляется в условиях неопределенности и риска.

В зависимости от особенностей исследуемого ПО, характера полученных оценок, задач и целей исследования качества ПО степень неопределенности и риска принятия решений о качестве значительно колеблется. Основной причиной применения методов принятия решений о качестве ПО является отсутствие информации для задания меры на соответствующем множестве Q_1. Таким образом, задача принятия решений сводится к задаче определения значимости полученных оценок показателей качества. Процесс принятия решений разделяется на четыре этапа.

I. Определение альтернативных способов принятия решений.

II. Определение степени неопределенности и риска возможных исходов.

III. Ранжирование предпочтений возможных исходов.

IV. Рациональный синтез информации, полученной на первых трех этапах и выработка решения.

После проведения всех испытаний и расчетов, необходимых при принятии решений, полученные в числовом выражении значимости оценок показателей качества ПО должны быть подставлены в виде интегрального выражения.

Среди прочих методов выделяется органолептический метод, который основан на использовании информации о качестве ПО, получаемой в результате восприятия органов чувств человека: зрения, слуха, осязания и т.д. С помощью этого метода может оцениваться качество оформления и упаковки ПО, качество визуализации и звукового оформления функционирования ПО, качество печати документации и т.д. К прочим методам относится комплексный метод оценки качества, основанный на логике умозаключений, определяющих качество ПО путем логических операций (аналогия, следствие, отрицание и т.д.) над характеристиками ПО и результатами его разработки и использования. Наиболее

распространенным среди прочих методов оценки качества ПО является метод сравнения оцениваемого ПО с базовым образцом (эталоном). Этот метод состоит в выборе базового образца ПО и сравнении характеристик оцениваемого ПО с соответствующими характеристиками базового образца ПО и соответствующем пересчете оценок показателей качества базового образца для оцениваемого ПО. Развитие ПО, расширение области их применения обуславливает возможность появления других методов оценки качества ПО. Состав методов оценки качества ПО и их классификация являются открытыми и допускают расширение.

Организационные вопросы проведения испытаний ПО

Оценка качества ПО КС осуществляется специалистами испытательных центров на основании соответствующих планов или договоров, а также специалистами организаций:

- разработчика (в процессе создания ПО);
- заказчика (фондодержателя) при приеме-сдаче ПО (прием в Фонд алгоритмов и программ - ФАП) и при подготовке к разработке ПО;
- изготовителя при производстве ПО;
- пользователя при купле-продаже и эксплуатации ПО, а так же сервисных организаций, сопровождающих ПО.

В результате проведенных работ по оценке качества ПО разработчику (заказчику, пользователю, изготовителю, сервисным организациям) должно выдаваться заключение о качестве оцениваемого ПО (сертификат качества ПО, технические требования к разрабатываемому ПО, техническое задание на разработку (доработку, модификацию) ПО, карта технического уровня ПО, технические условия и т.д.).

Работы по оценке качества ПО должны быть оформлены соответствующим техническим заданием, обеспечены методическими материалами, нормативно-техническими документами и документами, устанавливающими правовые и экономические гарантии для организаций, осуществляющих эти работы и заказывающих эти работы.

Методологические вопросы проведения испытаний ПО

Работы по оценке качества ПО должны осуществляться в соответствии с комплектом нормативно-технических и методологических документов, устанавливающих порядок, правила и методы оценки качества заданного ПО.

Учитывая, что работы по оценке качества ПО представляют собой сложный процесс сбора и обработки разнообразной информации, методики и методологические материалы по оценке качества ПО должны использоваться в комплекте с инструментальными средствами по оценке качества ПО, обеспеченными соответствующими инструктивными материалами. В процессе работ по оценке качества ПО КС должны быть разработаны и использованы ГОСТы, ОСТы, положения и руководящие документы, устанавливающие порядок проведения оценки качества ПО, а также номенклатуры показателей качества ПО, порядок и правила ее выбора, порядок сбора, регистрации, хранения информации о качестве ПО и расчета оценок, нормы трудозатрат и расхода ресурсов на эти работы, классификацию ПО, значения показателей качества базовых образцов ПО, основные термины и определения используемых понятий. Работы по оценке качества ПО так же предусматривают создание и использование комплекта методических документов и инструментальных средств, обеспечивающих выбор номенклатуры показателей качества, их определение, сбор, регистрацию, хранение и обработку исходных данных, выбор методов оценки качества ПО и задание алгоритмов расчета оценок, расчет оценок, принятие решений о качестве ПО, формирование заключения о качестве оцениваемого ПО. Работы по оценке качества ПО можно разделить на два этапа.

I. Создание методов и средств оценки качества оцениваемого ПО.

II. Разработка заключения о качестве ПО.

Первый этап может отсутствовать, если действующих методических документов по оценке качества ПО, обеспеченных инструментальными средствами, достаточно для проведения работ по второму этапу.

7.5.2. Построение программно-аппаратных комплексов для контроля технологической безопасности программ

Состав инструментальных средств контроля безопасности ПО при его разработке

Разработка сложного многофункционального программного обеспечения КС невозможна без создания интегрированной технологии разработки безопасного ПО, позволяющей осуществить комплексную автоматизацию всех этапов его жизненного цикла при гарантированном контроле наличия преднамеренных дефектов. Реализация в рамках единой технологии разработки программ инструментальных средств поддержки создания безопасного ПО направлена на обеспечение промышленного

выпуска программных комплексов с высоким уровнем безопасности и качества, осуществление контроля и экспериментальной оценки программ на наличие дефектов при сохранении высокого уровня производительности труда разработчиков программных комплексов.

В настоящее время уровень развития инструментальных средств разработки программного обеспечения позволяет поддерживать в рамках единой технологии все этапы жизненного цикла программ от их проектирования до кодирования и сопровождения. Однако подобные средства, как правило, реализуют замкнутый цикл работы, предназначены для разработки отдельных программ информационных систем, не позволяют учитывать особенности их целевого применения в контуре КС. Кроме того, в процессе их применения проблематично использование средств контроля технологической безопасности, базовых библиотек стандартных подпрограмм, прошедших сертификацию, а также затруднено получения процедур получения и анализа на безопасность исходного текста готовых программ. Отмеченные недостатки средств разработки программного обеспечения привели к тому, что сложившаяся технология создания сложных комплексов программ (КП) КС, функционирующих в режимах близких к реальному масштабу времени и обеспечивающих выполнение жестких требований к качеству управления, состоит из следующих этапов:

1. Техническое обоснование, системный анализ, проектирование и стратегическое планирование работ по созданию комплекса программ с учетом характеристик всех структур КС на основе систем имитационного моделирования и CASE-средств.
2. Разработка компонентов (получение программного кода) программных комплексов на основе инструментальных средств разработки.
3. Создание базового набора унифицированных модулей, компонуемых под различные целевые задачи в соответствии с алгоритмами управления (администрирования) и взаимосвязанные посредством стандартных интерфейсов.
4. Проведение стендовых и приемо-сдаточных комплексных испытаний разработанных программных изделий на основе тестирования в соответствии с программой и методикой испытаний.

Комплексное решение проблемы информационной безопасности в рамках интегрированной технологии разработки ПО КС связано с выполнением множества организационно-технических мероприятий и

решением сложных научно-прикладных задач. Однако уже сегодня, на 2-м и последующих этапах существующей технологии создания ПО может быть реализован контроль технологической безопасности готовых макетов программ, разработанных как с помощью отечественных, так и зарубежных средств. В качестве прототипа инструментальных средств контроля технологической безопасности ПО могут служить средства автоматизации тестирования программных модулей. Однако современные средства тестирования позволяют оценивать программы лишь на наличие ошибок, допущенных в процессе их разработки. Поэтому необходимо разрабатывать новые инструментальные средства, способные выявлять преднамеренные дефекты в программах. В перспективе такие средства технологического контроля безопасности ПО будут встроены в единую технологию создания программных средств КС.

С учетом методик контроля технологической безопасности создание программных средств контроля процессов разработки и испытаний ПО направлено на совершенствование технологии проведения тестовых экспериментов с макетами общесистемного и специального программного обеспечения, а также с инструментальными средствами разработки программ. Управляющие модули (мониторы), системы управления базой данных, средства отображения и планирования образуют общесистемное программное обеспечение (ОСПО). Набор общесистемных модулей обеспечивает через стандартные интерфейсы подключение и «настройку» специального программного обеспечения (СПО), которое является функциональным наполнением КП, на конкретный технологический цикл обработки информации. Формирование комплексов программ из унифицированных безопасных модулей является эффективным способом технологии сборочного программирования и дает значительную экономию средств при создании КС по типовой архитектуре.

Структурно-функциональная схема инструментальных средств поддержки создания безопасного программного обеспечения на основе предложенных методик представлена на рис.7.4. и включает следующие основные элементы.

1. Средства экспертизы и организации тестирования ПО.
2. Средства проведения тестирования.
3. Средства ликвидации дефектов.
4. Средства обеспечения тестирования.

Средства экспертизы и организации тестирования ПО состоят из следующих компонентов:

- экспертного анализатора контроля безопасности;
- блока прогнозирования участков воздействия дефектов;
- моделей угроз безопасности ПО;
- планировщика тестов контроля технологической безопасности.

Экспертный анализатор контроля безопасности предназначен для структурной декомпозиции тестируемого ПО КС с целью выделения элементов ОСПО, формирования библиотек СПО и определения состава и характеристик инструментальных средств с использованием которых разрабатывалось это программное обеспечение. Функционирование экспертного анализатора осуществляется в интерактивном режиме взаимодействия эксперта-оператора с программными средствами путем последовательного прохождения технологических этапов проверки.

Результатом работы этого элемента инструментальных средств поддержки создания безопасного ПО являются файлы с данными о компонентах разработанных программ и оценками их показателей качества.

Средства организации тестирования программ в интересах проверки их безопасности позволяют спрогнозировать вероятностным образом те участки программ, которые могут быть потенциально подвержены воздействию дефектов. Прогноз предполагаемых «узких мест» воздействия дефектов осуществляется на основе знания существующих моделей угроз безопасности и полученной статистики о выявленных дефектах. Базируясь на информации, собранной об объекте контроля, планировщик тестов контроля технологической безопасности производит формирование перечня тестов, необходимых для проверки соответствующего вида программ, и автоматизированный расчет числа экспериментов для получения достоверных вероятностных показателей безопасности ПО.

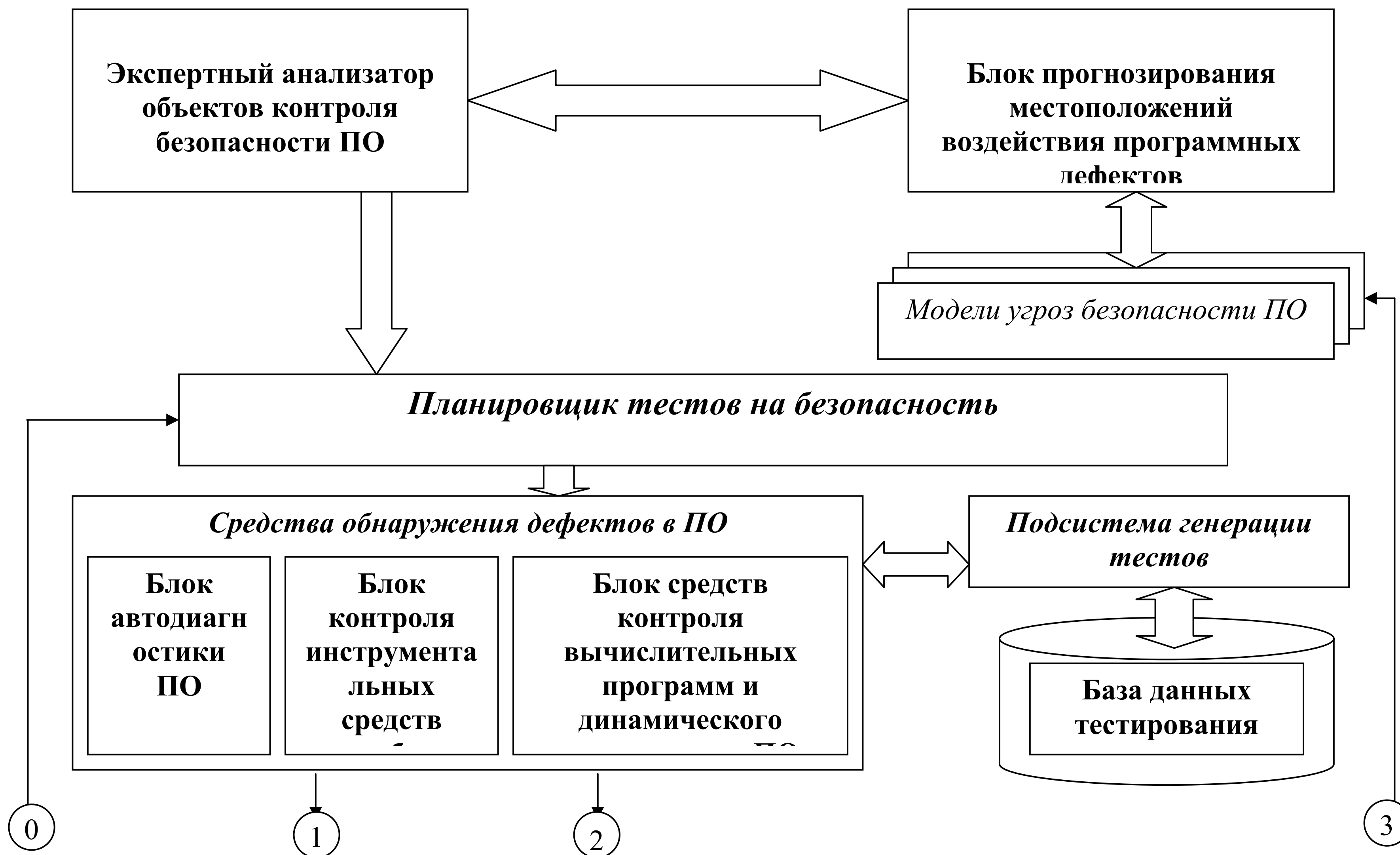
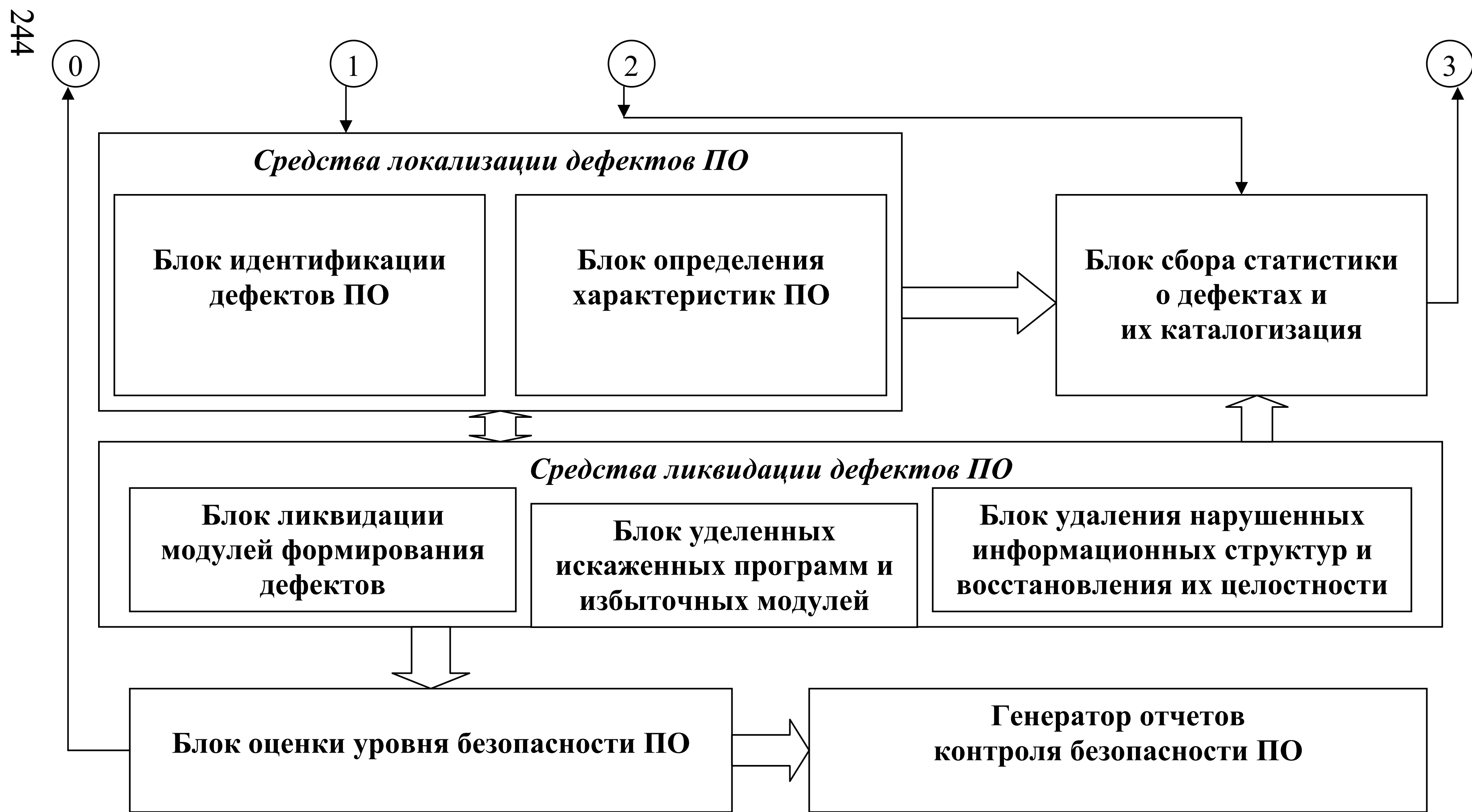


Рис.7.4 Структурно-функциональная схема инструментального комплекса поддержки создания безопасного ПО.



Продолжение рисунка 7.4

Средства проведения тестирования состоят из следующих элементов:

- средств обнаружения дефектов в ПО;
- средств локализации дефектов;
- системы генерации тестов;
- базы данных тестирования.

Средства обнаружения дефектов представляют собой совокупность программных блоков, реализующих методики контроля технологической безопасности ПО и их расширения. В зависимости от вида тестируемых программ осуществляется динамическое или статическое тестирование. Динамическое тестирование заключается в проведении автодиагностики и натурных экспериментов по определению соответствия параметров и алгоритмов управления функционированием программ, предъявленным к ним требованиям. Статическое тестирование представляет собой процесс аналитического моделирования, основанного на автоматизированных вычислениях вероятностей наличия дефектов в программных средствах. В зависимости от назначения ПО обнаружение дефектов производится следующим образом.

В интересах контроля безопасности ОСПО осуществляется:

- выявление критических путей в алгоритмах управления и администрирования,
- регистрация фактов неверной обработки прерываний и несанкционированной передачи межмодульных сообщений,
- определение наличия сбоев в управлении информационно-вычислительным процессом,
- обнаружение случаев изменения приоритетов обработки заявок и нарушения штатной дисциплины их обслуживания,
- контроль нарушения управления прикладными программами,
- выявление недокументированных передач сообщений и изменения характера сообщений,
- обнаружение нарушений целостности структур данных,
- выявление возможностей упрощения алгоритмов программ.

В целях контроля безопасности СПО выполняется:

- сравнение характеристик разработанных программ с параметрами эталона,
- тестирование программных средств в расширенных интервалах входных данных,
- подготовка спецификаций на программы,

- расчет вероятностных характеристик наличия дефектов по выходным данным,
- анализ устойчивости значений вероятностных характеристик,
- выявление случаев искажения параметров программ и результатов их выполнения,
- обнаружение попыток несанкционированного доступа прикладных программ к закрытой для них информации.

Для контроля безопасности инструментальных средств разработки программ предусмотрено:

- выявление случаев генерации избыточного кода для прикладных программ,
- составление спецификаций с учетом целевого применения этих средств,
- построение функциональных диаграмм и таблиц решений,
- определение дефектных участков в структуре программных средств,
- выявление отклонений от документированных штатных функций,
- дублирование и сверка объектного кода путем итерационных трансляций программ,
- сравнительный анализ с другими инструментальными средствами по возможностям отладки.

Средства локализации дефектов, осуществляют их идентификацию в соответствии с принятой на момент тестирования классификацией и определение характеристик программных дефектов, к которым относятся:

- одноразовость и многоразовость (саморепродуцирование) применения;
- самоликвидируемость;
- базируемость (на любых программах, только прикладных, использование комбинаций системных команд);
- изменяемость (неизменяемость) объема программ;
- модифицируемость информационных структур;
- самомодифицируемость.

Рассматриваемые средства имеют возможность выявления факта внесения дефекта на основе детального анализа характеристик программ и сверки их со спецификацией. Кроме того, на этапе локализации дефектов предусматривается, в случае необходимости, дешифрования элементов ПО и информационных структур. При выявлении аппаратных дефектов или

аппаратной реализации РПС формируются рекомендации по рекламации технических средств автоматизации.

Система генерации тестов и база данных тестирования обеспечивают целенаправленный выбор и компоновку набора тестов с учетом вида ПО и прогнозируемого дефекта. В перспективе прорабатывается вопрос создания средств сканирования программ по всем возможным для них угрозам безопасности.

Средства ликвидации дефектов обеспечивают удаление самих дефектов, искаженных программ и испорченной информации и включают в свой состав:

- блок ликвидации модулей формирования дефектов;
- блок удаления искаженных программ и избыточных модулей;
- блок удаления нарушенных информационных структур и восстановления их целостности.

После завершения работы этих средств проводится дополнительная отладка и восстановление программ, корректировка их эталонов, анализ целостности информации и формирование предложений по совершенствованию системы защиты.

Средства обеспечения тестирования состоят из следующих компонентов:

- блока сбора статистики о дефектах и их каталогизации;
- блока оценки уровня безопасности ПО;
- генератора отчетов контроля безопасности ПО.

Блок сбора статистики о дефектах и их каталогизации обеспечивает упорядоченные процедуры накопления данных в интересах уточнения структур тестов и моделей угроз. Блок оценки уровня безопасности ПО обеспечивает контроль значений вероятностных характеристик наличия в программах дефектов путем их сверки с требуемыми значениями. Генератор отчетов контроля безопасности ПО предназначен для формирования и выдачи на средства отображения выходных документов по результатам контроля технологической безопасности ПО.

Таким образом, предложенная структура и состав инструментальных средств поддержки создания безопасного программного обеспечения являются комплексной автоматизированной системой тестирования, средства которой с достаточной полнотой позволяют выявлять, каталогизировать и ликвидировать преднамеренные дефекты. Предложенные средства позволяют гибко сочетать экспертные, натурные и

расчетные методы определения вероятностных характеристик наличия дефектов в программном обеспечении КС.

Структура и принципы построения программно-аппаратных средств контрольно-испытательного стенда испытания технологической безопасности ПО

На современном уровне развития информационных технологий создания сложного ПО КС практическая реализация и использование дополнительных методик и средств комплексного контроля технологической безопасности невозможна без создания экспериментального испытательного стенда.

Целью формирования испытательного стенда контроля технологической безопасности ПО является обеспечение заблаговременного обнаружения и ликвидации преднамеренных и непреднамеренных дефектов в разрабатываемых программах. Технологическая проверка программных средств организуется, как правило, в среде локальной вычислительной сети (ЛВС), в рамках которой имитируются реальные условия их применения и одновременно реализуется тестовый контроль посредством объединения отдельных подсистем через протоколы высокого уровня.

Испытательный стенд должен удовлетворять следующим требованиям:

- обеспечивать аппаратно-программную поддержку применения экспертных, аналитических, натурных и имитационных методов тестирования программ на наличие в них закладок;
- обеспечивать проведение всестороннего анализа работоспособности общесистемного и специального программного обеспечения, инструментальных средств разработки программ в условиях возможного проявления сбоев и разрушающих воздействий;
- предоставлять возможность варьирования различными значениями входного информационного вектора;
- имитировать воздействие различных внешних факторов, создавая, таким образом, условия активизации возможных закладок;
- обеспечивать функционирование широкого класса программных комплексов с возможностью модификации базовой структуры стенда с учетом специфики применения КС, программное обеспечение которой подвергается проверке;

- осуществлять настройку и реконфигурацию предполагаемой модели процесса функционирования программ;
- предоставлять возможность управления тестовым программным обеспечением и моделями угроз;
- обеспечивать взаимозаменяемость отдельных элементов стенда и расширение его новыми компонентами;
- предоставлять значительные по объему вычислительные ресурсы, поддерживать стандартные интерфейсы и протоколы обмена в сетевой программно-технической структуре;
- обеспечивать диспетчеризацию, управление и администрирование информационно-вычислительным процессом в сети ПЭВМ;
- осуществлять сбор, накопление и каталогизацию больших объемов информации о преднамеренных и непреднамеренных дефектах;
- получать исходные данные для подготовки спецификаций и сертификата качества на вновь создаваемые программные изделия.

Анализ существующих контрольно-испытательных стендов показывает, что они предназначены для автономной или комплексной отладки программных комплексов с целью их отладки и выявления незлонамеренных ошибок разработчиков и могут служить основой для создания стенда контроля технологической безопасности ПО. Однако структуры подобных стендов, как правило, не включают в свой состав подсистемы для обнаружения, уничтожения и устранения последствий преднамеренных дефектов в программах.

www.kiev-security.org.ua
BEST rus DOC FOR FULL SECURITY

Поэтому необходимо создание конструктивно новых испытательных стендов, представляющий собой программно-аппаратные комплексы, наполненные моделями внешней и внутренней сред функционирования программ, их реальными макетами, средствами выявления разрушающих воздействий на ПО и контроля целостности информации. В основе создания испытательного стенда можно предложить для реализации технологию распределенной обработки информации «клиент-сервер» и принять следующие принципы его построения.

1. Модульность построения, позволяющую обеспечить гибкую интеграцию и функциональную декомпозицию программно-аппаратных

элементов стенда, формировать унифицированные структурные элементы, а также проводить выборочный и комплексный анализ и тестирование программ на предмет наличия закладок.

2. Структурная универсальность, означающая решение разнообразных задач по выявлению закладок в разнотипном ПО на основе единых средств стенда.

3. «Настраиваемость», под которой подразумевается обеспечение возможности испытаний и отладки программ различных предметных областей, а также гибкость использования информационно согласованных штатных средств стенда для различных условий тестирования.

4. Расширяемость и открытость, означающие возможность дальнейшего развития и модификации стенда, в том числе независимых относительно отдельных его элементов.

5. Унифицируемость, означающая единство среды испытаний, общность средств и протоколов их взаимодействия для всех режимов тестирования и видов объектов контроля.

6. Защищенность, под которой понимается изолированность штатных программно-аппаратной среды стенда от деструктивных воздействий со стороны испытываемых программ.

Исходя из предложенных выше принципов, структура испытательного стенда контроля программного обеспечения на технологическую безопасность, может состоять из следующих основных элементов:

- сектора макетов комплексов программ КС;
- автоматизированного рабочего места (АРМ) обнаружения и ликвидации дефектов;
- АРМ экспертного тестирования;
- сектора имитации моделей информационных угроз;
- сектора планирования и анализа результатов тестового контроля;
- сектора проектного анализа объектов контроля безопасности;
- сервера эталонов КП;
- сервера тестов;
- АРМ администратора стенда.

Сектор макетов комплексов программ КС обеспечивает среду для размещения программных средств в интересах проведения автономных и комплексных испытаний их функционирования в условиях, близких к реальным в режимах одномашинной отладки программных модулей, а также отработки методов сетевого взаимодействия компонентов КП.

АРМ обнаружения и ликвидации дефектов представляет собой программно-аппаратную реализацию средств динамического тестирования готового программного изделия в экстремальных режимах эксплуатации с фиксацией любых отклонений от штатного функционирования и устранением выявленных дефектов. Данное автоматизированное рабочее место в процессе своей работы выполняет роль концентратора, который объединяет потоки информации от абонентов сети стенда, и коммутатора, обеспечивающего логическую коммутацию информационных запросов к серверам и другим абонентам ЛВС.

АРМ экспертного тестирования предназначено для статического тестирования вероятностных характеристик наличия закладок в программах и оценки показателей их качества независимыми экспертами по специальным методикам и в соответствии с ГОСТ 28195-89.

Сектор имитации моделей информационных угроз включает в свой состав имитатор разрушающих воздействий и критических нагрузок, а также сервер моделей угроз. Такой состав сектора обеспечивает не только выявление закладок в программах, но и проверку их работоспособности совместно со своими средствами защиты при имитации разрушающих программных воздействий, значительных информационных нагрузок и ресурсных ограничений.

Сектор планирования и анализа результатов тестового контроля является интегрированной системой планирования экспериментальных исследований на стенде, в том числе управления вычислительными ресурсами, анализа результатов тестового контроля технологической безопасности ПО, генерации отчетных материалов и отображения их с применением компьютерной графики. Важнейшей функцией средств анализа результатов является подготовка исходных данных и формирование сертификата качества и безопасности программного изделия на основе спецификаций, прошедших аудиторскую проверку.

Сектор проектного анализа объектов контроля безопасности осуществляет контроль соответствия структуры разработанного программного комплекса и его прогнозируемых показателей проектным материалам на него. Ввиду того, что структурные ошибки этапа проектирования являются доминирующими в общем множестве ошибок и приводят к наиболее тяжелым последствиям, считается целесообразным наличие рассматриваемого сектора в составе стенда. Кроме того, на основе сравнения характеристик вновь создаваемого программного комплекса с информацией об уже реализованных средствах, находящейся в сервере

эталонов КП, можно решить вопрос контроля выполнения требований по стандартизации, унификации и типизации структур, модулей и интерфейсов ПО КС.

В идеале, формы усовершенствования макетов комплексов программ должны быть следующими: отдельные программные модули, схемный и функциональный эквивалент, прототип комплекса, отработанный комплекс. При соблюдении такой взаимосвязанной последовательности преобразований макета, впервые разрабатываемого программного комплекса, появляется возможность в конечном продукте аккумулировать весь позитивный опыт разработки и контроля безопасности ПО.

По некоторым оценкам при повторном использовании программы-макета для создания комплекса программ, аналогичного первоначальному, результативность промышленного производства прикладных программ повышается более чем в 10 раз.

Серверы эталонов КП и тестов, соответственно, обеспечивают хранение, накопление и выдачу по запросам «клиентов» сети стенда информации и «упакованных» тестов в интересах поддержания целенаправленного процесса технологического контроля программ.

АРМ администратора стенда реализует следующие общесистемные функции:

- многоуровневая защита и изоляция элементов сети стенда и циркулирующей в нем информации от несанкционированного доступа как от абонентов сети, так и тестируемого ПО;
- управление сетевым доступом;
- конфигурирование и реконфигурирование архитектуры стенда;
- координация и диспетчеризация работы стенда, восстановление процесса его функционирования в случае возникновения отказов и сбоев;
- защита от компьютерных вирусов и проведение необходимых профилактических процедур.

Предполагается, что на самом стенде используется только категорированная вычислительная техника и сертифицированное общее и специальное программное обеспечение, производится контроль операционной среды, исключен запуск непроверенных программ вне установленной технологии контроля разрабатываемых программных средств.

В качестве базовых средств реализации предложенного испытательного стенда контроля ПО на технологическую безопасность целесообразно использовать программно-аппаратные платформы, которые по своим функциональным возможностям и вычислительной мощности превосходят как минимум на порядок (если такое возможно) тестируемые программные и/или аппаратные средства. К основным достоинствам операционной среды для такого стенда в сравнении с другими операционными системами можно отнести следующие:

- высокое быстродействие, качество и значительный объем средств тестирования и разработки ПО;
- наличие встроенных высокоуровневых протоколов обмена, обеспечивающих взаимодействие «клиент-сервер»;
- распространенность, тиражируемость, надежность и отработанность пользователями российского рынка программных продуктов фирмы разработчика (например, фирмы Microsoft);
- наличие возможности надстройки средств ОС индивидуальными средствами защиты от несанкционированного доступа,
- приемлемая стоимость;
- экспериментально проверенная согласованность по протоколам обмена с другими распространенными ОС;

Таким образом, предложенные принципы построения и структура испытательного стенда контроля программного обеспечения на технологическую безопасность ориентированы, в отличие от существующих структур стенда, на комплексное экспертно-аналитическое, имитационное и натурное тестирование ПО КС с целью гарантированного выявления преднамеренных и непреднамеренных дефектов в программах на основе использования современной сетевой технологии обработки информации «клиент-сервер».

7.6. Метод расчета вероятности наличия РПС на этапе испытаний программного обеспечения вычислительных задач

7.6.1. Постановка задачи

С точки зрения технологической безопасности тестирование должно позволять не только декларировать факт отсутствия в проверенных частях программных комплексов РПС, но и получать количественные характеристики, чаще всего вероятностные, существования таких дефектов

в непроверенных программных компонентах. Данное утверждение справедливо для больших комплексов программ, полностью проверить все логические ветки которых не представляется возможным. К таким программам, в частности, можно отнести комплексы управления сложным технологическим процессом.

При исследовании сложных комплексов программ возникают существенные трудности использования известных методов детерминированного тестирования, связанные с необходимостью генерации входных наборов данных и расчета эталонных выходных данных. Использование метода стохастического тестирования упрощает генерацию входных данных, однако необходимость расчета эталонов сохраняется. Существенным при этом является то, что принципиально невозможно создать единый алгоритм расчета эталонов для различных тестируемых программ, что особенно характерно для программ, реализующих вычислительные алгоритмы.

Сущность вероятностного тестирования заключается в следующем. Исследуемая программа реализуется на наборах входных данных, представляющих собой случайные величины, распределенные по некоторому закону $F(x)$. Для некоторого множества контрольных точек определяются вероятностные характеристики (ВХ) случайных величин, являющихся для этих точек выходными данными. Полученные ВХ сравниваются с эталонными ВХ, рассчитанными для данного закона распределения входных величин по заданному в спецификациях программы алгоритму, который данная программа реализует. В зависимости от степени совпадения экспериментально определенных ВХ с эталонными делается вывод об отсутствии в программе дефектов, преднамеренно внесенных на этапе ее создания. Необходимо отметить, что данный метод позволяет выявлять любые дефекты программы, в том числе и случайные ошибки. Однако использование стохастического тестирования наиболее целесообразно для тех участков сложных программных комплексов, для которых детерминированные методы требуют существенных по объему затрат на подготовку тестовых наборов данных. В то же время применение на этапе отладки программ более простых методов позволяет практически ликвидировать вероятность проявления случайных ошибок после завершения отладки и представления программного обеспечения на испытания.

Область применения метода вероятностного тестирования программ определяется в основном границами применимости математического

аппарата, используемого для расчета эталонных вероятностных характеристик. Для программ, реализующих вычислительные функции, задача расчета вероятности наличия в программе РПС формулируется следующим образом [ЕУ].

Дано: алгоритм **А**, подлежащий реализации программой **П**, и требуемая достоверность результатов тестирования P_0 (вероятность наличия РПС в нетестируемых ветвях программы при заданном числе испытаний).

Требуется определить.

- последовательность законов распределения $F_1(x),...,F_n(x)$, $j=1,...,\gamma$ входных величин $X=\{x_j\}$, при которой с вероятностью $P_{пр}$ гарантируется отсутствие в тестируемой программе РПС; при этом с вероятностью P_0 такие дефекты могут иметь место в нетестированных участках программы;
- множество контрольных точек ($КТ_i$), $i=1,...,k$, в которых определяется экспериментальное распределение выходных величин;
- множество G_i вероятностных характеристик, снимаемых с заданного множества КТ;
- множество величин L_i таких, что если существует i, что ($|G_{iэкс}-G_{iэт}|>L_i$), то программа содержит дефекты с вероятностью P_0 или не содержит их с вероятностью $P_{пр}$.

Для решения данной задачи необходимо использовать методику, основанную на модификации метода вероятностного тестирования и позволяющую последовательно решить следующие частные задачи: определить множество информативных характеристик G_i случайных величин, снимаемых с некоторого множества $КТ_i$ исследуемой программы; определить критерии принятия решения о наличии дефектов в программе **П**, обеспечивающих заданную достоверность такого решения.

7.6.2. Обоснование состава множества информативных характеристик

Выбор информативных ВХ случайных величин G_i должен производиться с учетом двух основных факторов:

- выбранные ВХ должны существенно изменять свои значения при наличии в программе РПС;
- ВХ должны относительно легко вычисляться при экспериментальных исследованиях программы.

Поскольку информативные характеристики должны реагировать на наличие в программе закладок, изменяющих основной алгоритм функционирования или инициирующих изменение исходных данных (промежуточных или конечных результатов), следовательно, необходимо определить класс функций, которые получаются из исходной под воздействием программной закладки. К сожалению, РПС сами нуждаются в дополнительном исследовании и классификации, могут искажать реализуемую функцию в таком широком диапазоне, что однозначно предсказать ее искаженный вид просто невозможно. Поэтому для количественной оценки информативности той или иной ВХ целесообразно хотя бы приблизительно определить ожидаемый вариант воздействия дефекта на исследуемую программу.

С точки зрения удобства экспериментального вычисления наиболее простой характеристикой является значение функции распределения в одной точке. Вычисление этой характеристики сводится к подсчету значений выходной величины, попавших в заданный интервал. Вычисление этой ВХ для тех контрольных точек программы, где критерием перехода на ту или иную ветвь является значение функции, сводится к подсчету числа прохождений по этим ветвям. Например, экспериментальные исследования программ, входящих в специальное ПО, реализующего комбинационные алгоритмы выбора предпочтений, показали, что для программ такого класса частота прохождения различных ветвей при заданном законе распределения входных данных является достаточно устойчивой информативной характеристикой. Если при этом еще фиксировать временные интервалы прохождения различных путей программы, которые могут существенно отличаться друг от друга, то время выполнения также может использоваться в качестве информативной характеристики.

www.kiev-security.org.ua
BEST rus DOC FOR FULL SECURITY

Для вычислительных программ, обладающих достаточно простой ациклической структурой, но реализующих сложные вычислительные функции, например, вычисления полиномов различной степени в приближенных расчетах, в качестве вероятностных характеристик могут использоваться начальные моменты законов распределения входных данных

$$m_k^* = \frac{1}{n}\sum_{i=1}^{n}(y_i)^k$$

где y_i - значения входной величины при i-том испытании (прогоне программы);

m_k^* - начальный момент, полученный при проверке программы;

n - число прогонов программы.

7.6.3. Алгоритмы приближенных вычислений вероятностных характеристик наличия в программах РПС

В основу алгоритмов приближенных вычислений ВХ положен принцип расчета ВХ по функциям распределения выходных и промежуточных величин. При этом законы их распределения вычисляются как распределения функции от случайных аргументов [ЕУ].

Задача функционального преобразования непрерывных случайных величин формируется следующим образом.

Дано: совместная плотность распределения вероятностей $w_n(x_1,...,x_n)$ непрерывных случайных величин $\varepsilon_1,...,\varepsilon_n$ и совокупность функций $f_1,...,f_m$ от n переменных. С помощью этих функций определены m случайных величин $h_1=f_1(x_1,...,x_n),...,h_m=f_m(x_1,...,x_n)$, где x_i – значения случайных величин ε_i.

Необходимо: определить закон распределения каждой полученной случайной величины h_j и их совместную плотность $W_m(y_1,...,y_m)$, где y_i - значения случайных величин h_j.

Решение этой задачи точными методами [КК] даже для одномерного случая возможно только при жестких ограничениях на вид функции и закон распределения аргумента. Например, применение метода обратной функции требует вычисления на каждом участке монотонности $f(x)$ обратной функции и производной от обратной функции.

Вычисление $W(y)$ методом характеристической функции [КК] ограничено таким набором $w(x)$ и $f(x)$, для которых можно вычислить

характеристическую функцию в явном виде, а по характеристической функции вычислить $W(y)$.

В связи с этим целесообразно воспользоваться приближенным методом, сущность которого заключается в вычислении некоторых характеристик закона распределения и по ним восстановлении всего закона распределения. В качестве таких характеристик можно взять начальные моменты закона распределения:

$$m_k(h)=\int_{-\infty}^{\infty} \ldots \int_{-\infty}^{\infty} f(x_1,\ldots,x_n)^k w(x_1,\ldots,x_n)\, dx_1 \ldots dx_n$$

или для одномерного случая $h=f(x)$

$$m_k(h)=\int_{-\infty}^{\infty} f(x)^k w(x)\, dx$$

при условии, что этот интеграл сходится абсолютно [КК].

Поскольку данный методический подход возможен практически для любых вычислительных алгоритмов, то для иллюстрации его реализуемости можно ограничиться классом функций, представимых конечным степенным рядом. В этом случае если $f(x)=\sum_{i=1}^{N} a_i x^i$ (общий вид), то определение первых $t=r/N$ моментов случайных величин $h=f(k)$ выполняется по следующему алгоритму (r – число первых начальных моментов закона распределения $w(x)$, принимающих значения $m_1(\varepsilon),\ldots,m_r(\varepsilon)$).

Алгоритм А

А$_1$. $i{:=}1$.

А$_2$. Вычислить значения b_j, $j=1,\ldots,N$ полинома $f(x)^i$ путем перемножения $f(x)^i$ и $f(x)^{i-1}$: если $f(x)=\sum_{l=0}^{N} a_l x^l$ и $f(x)^{i-1}=\sum_{p=0}^{N(i-1)} z_p x^p$, то $b_j=\sum_{q=0}^{N(i-1)} a_q z_{j-q}$.

А$_3$. Вычислить $m_i(h) = \sum_{j=0}^{N_i} b_j m_j(\varepsilon)$.

А$_4$. $i := i+1$.

А$_5$. Если $i \le r/N$, то переход на п.А$_2$. В противном случае алгоритм завершается.

Кроме рассмотренного, возможно применение алгоритмов реализующих методы вычисления моментов функции от случайных величин с использованием моментопроизводящих или кумулянтных функций [КК].

Задача вычисления закона распределения $F(y)$ в заданной точке y_0 по L моментам формулируется следующим образом.

Дано: m_i, $i=1,...,L$ - начальные моменты $F(y)$.

Необходимо: определить значения $\sup F(y_0)$ и $\inf F(y_0)$.

Метод вычисления $\sup F(y_0)$ и $\inf F(y_0)$ по известным начальным моментам $F(y)$ описан в [Че]. Алгоритм вычисления $\sup F(y_0)$ и $\inf F(y_0)$ для $L=2k-1$, $k=1,2...$, и известных a и b – конечных значений y, меньше и больше которых соответственно значения функции принимать не могут, реализуют данный метод с некоторыми модификациями.

Алгоритм Б

Б$_1$. Сформировать ряд «подходящих» дробей к интегралу

$$\int_a^b \frac{dF(x)}{z-y} = \frac{1}{z} + \frac{m_1}{z^2} + ... + \frac{m_L}{z^{L+1}}$$

Б$_2$. Преобразовать «подходящую дробь» в непрерывную вида

$$\cfrac{1}{\alpha_1 z + \beta_1 - \cfrac{1}{\alpha_2 z + \beta_2 - \cfrac{1}{\dots\dots\dots\cfrac{1}{\alpha_{L-1} z + \beta_{L-1}}}}}.$$

Б$_3$. Привести непрерывную дробь к дробно рациональному виду $\varphi_L(z)/\psi_L(z)$.

Б$_4$. Выполнить пункты Б$_2$ и Б$_3$ для L=L-1 и вычислить $\varphi_{L-1}(z)/\psi_{L-1}(z)$.

Б$_5$. Определить функцию $A(z) = \dfrac{\varphi_L(z)z - \varphi_{L-1}(z)}{\psi_L(z)z - \psi_{L-1}(z)} = \dfrac{\phi_0(z)}{\phi_1(z)}$.

Б$_6$.Определить вещественные корни полинома $\phi_1(z)$.

Б$_7$. Вычисление интеграла $\int_a^{y_0} dF(y)$ с помощью вычетов. При этом inf $F(y_0)$ будет равно сумме вычетов $\phi_0(z)/\phi_1(z)$ для всех y,y_0, а sup $F(y_0)$, будет равно сумме inf $F(y_0)$ и очередного вычета. Среднее значение $F_{ср}(y_0)$=(sup $F(y_0)$+inf $F(y_0)$)/2 и значение δ=(sup $F(y_0)$+inf $F(y_0)$)/2, определяющее точность восстановления функции распределения, зависят от m_i, i=1,...,L и y_0. Однако $\delta \le 1/L+1$.

Таким образом, с помощью алгоритмов **А** и **Б** можно с заданной точностью рассчитать вероятностные характеристики исследуемой программы.

7.6.4. Обоснование критериев принятия решения о наличии в программе РПС

Задача выбора критериев наличия в исследуемой программе РПС в общем виде, формулируется следующим образом.

Дано:

- множество G_i вероятностных характеристик случайных величин, снимаемых с заданного множества контрольных точек;
- эталонные значения этих ВХ G_i^* и их значения, полученные в результате n испытаний (прогонов) программы.

Необходимо: определить множество L_i таких, что если существует $i(|G_i - G_i^*| > L_i)$, то делается вывод о наличии в исследуемой программе РПС с вероятностью P_0.

Если в качестве информативной характеристики программы выбраны значения закона распределения выходной величины в точке y_0, то задача

определения решающих правил о наличии программных закладок может быть уточнена и записана в следующем виде.

Дано: $P=F(y_0)$; $q=1-P=P(y>y_0)$; задано число прогонов программы n и значения доверительной вероятности P_0.

Необходимо: определить значение доверительного интервала L частоты появления события $\{A\text{-}y_j<y_0\}$, где y_j - j-е значение выходной величины.

Для независимых испытаний частота появления события $A\text{-}y_i<y_0$ является случайной величиной, распределенной по биномиальному закону с математическим ожиданием P и дисперсией $D=Pq/n$. Вероятность появления k событий при n испытаниях в этом случае рассчитывается по формуле $Pk=C_n^k P^k q^{n-k}$.

В качестве доверительного интервала $[P_1^*,P_2^*]$ целесообразно выбирать наименьший интервал, вероятность попадания за границы которого больше $(1-P)/2$. Границы доверительных интервалов для различных значений P, P_0 и n сведены в таблицы [Ве], что существенно облегчает задачу инженерного анализа результатов тестирования при контроле технологической безопасности программного обеспечения. С увеличением n биномиальное распределение будет стремиться к нормальному с теми же математическим ожиданием и дисперсией. При этом для вероятностного тестирования необходимо выбрать такие значения y_0, чтобы $P\approx 0{,}5$, что позволяет заменять биномиальное распределение нормальным с максимальной точностью. Доверительный интервал в этом случае определяется по формулам $P(|P-P^*|<L_{экс})=P_0$, $L_{экс}=\arg Z^*((1+P_0)/2)$, где $\arg Z^*$ - функция, обратная нормальной функции распределения Z^*, полученная по таблицам.

С учетом того, что значение $F(y_0)$ вычисляется с точностью δ, доверительный интервал $L=L_{экс}+\delta$, то есть если при проведении испытаний значений P^* будет отличаться от аналогично рассчитанного на величину, большую, чем $L_{экс}+\delta$, то принимается решение о наличии в исследуемой программе РПС с вероятностью P_0.

Аналогичным образом доверительный интервал может быть определен и для случая, когда в качестве информативной характеристики программы используется математическое ожидание выходной случайной величины: $m_1^*(y_0)=\dfrac{1}{n}\sum_{j=1}^{n} y_j$.

Так как y_j представляет собой случайные величины с одинаковым законом распределения, то законы распределения и их суммы стремятся к нормальному с математическим ожиданием $m_1(y)$ и дисперсией $D(y)/n$.

В этом случае доверительный интервал определяется по формуле $L_{экс}=\sigma \arg Z^*((1+P_0)/2)$, $\sigma = \sqrt{D}$.

Пользуясь методами математической статистики, можно аналогичным образом построить доверительные интервалы и для других ВХ. При этом факторы, влияющие на достоверность определения вероятности наличия в программном обеспечении РПС, можно разбить на три основные группы.

1. *Точность аналитического вычисления ВХ.* Если в качестве метода вычислений использовать метод моментов, то ошибки будут вызваны точностью представления реализуемой функции степенным рядом и ограниченным числом начальных моментов. Если функция представляется конечным степенным рядом или ошибка разложения в ряд достаточно мала, то можно считать, что точность вычисления ВХ будет зависеть от качества моментов. При заданной допустимой ошибке δ вычисления закона распределения требуемое число моментов может быть достаточно просто рассчитано.

2. *Ограниченность числа испытаний (прогонов) программы.* При известных значениях доверительного интервала с помощью методов статистики можно определить необходимое число испытаний, обеспечивающее достоверность результата не меньше P_0.

3. *Закон распределения входных случайных величин.* Для заданного закона распределения аргумента $w(x)$ функции $f_1(x)$ и $f_2(x)$ будут неразличимы, если для каждой точки y

$$F(y)=\sum_i \int_{\Delta_i} w(x)\,dx=\sum_j \int_{\Delta_j} w(x)\,dx$$

или если задаться допустимой точностью вычисления δ функции распределения:

$$\left|\sum_i \int_{\Delta_i} w(x)\,dx-\sum_j \int_{\Delta_j} w(x)\,dx\right|\le\delta,$$

где Δ_i - интервалы аргумента, где $f_1(x)<y$;

Δ_j - интервалы аргумента, где $f_2(x)<y$.

Проверяя функции при различных законах распределения аргумента, можно сократить множество неразличимых функций. Для каждого класса

допустимых функций можно найти такой набор законов распределения аргументов, который обеспечивает минимизацию области неразличимых функций.

7.7. Подходы к исследованию сложных программных комплексов

7.7.1. Общие замечания

Анализ программного обеспечения компонентов КС и процессов его создания показывает [Е1,ЕП,ИБС,Лип0,Лип1,Лип3,Лип4], что эффективность контроля технологической безопасности готовых программ может быть повышена за счет учета специфических особенностей структуры программ и, в частности, параметров, характеризующих сложность проектирования и функционирования готовых программ. К числу параметров, влияющих на сложность проектирования ПО, в первую очередь относятся [Ма]:

- размер программ, выраженный числом команд и программных модулей в комплексе;
- количество обрабатываемых переменных и объем памяти для размещения базы данных;
- ограничения на длительность разработки и на количество специалистов, участвующих в создании комплекса программ.

Кроме того, для сложных комплексов управляющих программ (КУП) проверка отсутствия РПС не всегда возможна вследствие большого числа технологических исходных данных и т.д. В этих условиях целесообразно в первую очередь определить наиболее критичные с точки зрения выполнения целевых функций пути (ветви) программ, которые должны быть подвергнуты наиболее полному тестированию. Критичные пути определяются путем анализа следующих компонентов:

- сложности программных модулей, которая характеризуется трудоемкостью создания оформленного комплекса программ и может быть оценена с учетом внутренней структуры и преобразования переменных в каждом модуле, а также интегрально по некоторым внешним статистическим характеристикам модулей;
- сложности структуры комплекса (или группы программ и связей между модулями по передачам управления и обмену

информацией), определяемой числом связей при взаимодействии модулей, структурой и регулярностью межмодульных связей;

- сложности структуры данных, которую можно оценить количеством и структурой глобальных и обменных переменных, регулярностью их размещения в массивах, а также сложностью доступа к этим переменным.

Информация об усредненных значениях и статистических законах распределения характеристик этих компонентов накапливается с целью дальнейшего использования при исследовании программ, аналогичных по своему функциональному назначению и принципам разработки, а также:

- при классификации программ на основе количественных критериев для выбора рациональных методов и средств обеспечения их технологической безопасности;
- для обоснования правил структурного построения, сегментации больших программ с целью упрощения контроля их технологической безопасности;
- для выбора рациональных методов построения и применения технологически безопасных средств автоматизации программирования и отладки.

Определение критических путей программных комплексов осуществляется в процессе последовательного выполнения следующих работ, связанных с получением информации о сложностных показателях исследуемых программ:

- обработки исходных текстов и характеристик оттранслированных программ, а также расчета абсолютных и относительных численных характеристик объема модулей программ, числа используемых переменных и констант;
- структурного контроля программы, расчета числа маршрутов и циклов, а также количества передач управления в ациклических маршрутах;
- построения иерархической схемы связей программных модулей, расчета количества вызываемых и вызывающих данный модуль программ, определения интенсивности использования переменных на запись и чтение.

При исследовании управляющих программ в КС различного назначения целесообразно использовать следующую классификацию программных модулей:

- диспетчерские программы (верхний уровень иерархии);
- функциональные модули, выполняющие основные задачи обработки информации и принятия решений (средние уровни иерархии);
- стандартные модули, используемые многократно для выполнения достаточно простых процедур и технологических операций по обработке данных (нижние уровни иерархии).

7.7.2. Анализ характеристик программных модулей с помощью управляющего графа

Для количественной оценки сложности программных модулей используется графовая модель, представляющая собой направленный граф с циклами, вершинами которого являются линейные участки программы, а дугами - управляющие связи между линейными участками. Анализируя такой граф, можно обнаружить неисполняемые участки программы, получить все циклы (явные, заданные операторами цикла, и неявные, построенные с помощью условных операторов), обнаружить входы «сбоку» в структурированные конструкции (например, вход в тело цикла, минуя заголовок), наличие нескольких входов и выходов из процедур, отступления от стандартов программирования и т.д., что существенно важно при анализе технологической безопасности программ и, в частности, при определении критичных путей их выполнения.

Количественная оценка сложности программы при помощи управляющего графа вычисляется [Лип0] как сумма $V=e-n+2p$, где e – число дуг, n - число вершин, p - число связных компонент графа. Оценкой сложности можно пользоваться и при проектировании программ: если сложность превосходит некоторое критическое значение, то программу целесообразно разбить на более мелкие модули для упрощения ее понимания и отладки. В качестве критической сложности обычно принимают значение 10 единиц.

Основные алгоритмы, используемые на данном этапе, связаны с преобразованием управляющего графа, его анализом и построением возможных путей реализации программы. При этом преобразование графа основывается на понятии структурного дерева программы, вершинам которого соответствуют структурированные конструкции программы (условные операторы, циклы и последовательные операторы), вершины управляющего графа и неструктурируемые конструкции.

Неструктурируемой конструкцией графа называется его компонент, не содержащий других конструкций с одним входом и одним выходом и не принадлежащий множеству структурированных конструкций.

Вычисление управляющего графа обычно осуществляется при трансляции программы, но, к сожалению, большинство компиляторов не выдает его программисту в качестве результата своей работы.

При использовании ассемблеров управляющий граф легко может быть восстановлен по машинному коду [Лип0],а для языков высокого уровня необходимо создавать нетривиальные программы, сканирующие исходный текст и выделяющие управляющие конструкции языка.

Однако при разработке методики определения критических путей управляющих программ с позиции обеспечения технологической безопасности можно сделать допущение, что управляющий граф программы известен.

Алгоритм построения структурного дерева программы (алгоритм **A**) сводится к реализации следующей конструкции. Построение структурного дерева программы осуществляется параллельно со сверткой управляющего графа, когда выделенная конструкция заменяется одной вершиной. Для краткости управляющий граф на очередном этапе свертки будем называть графом программы. Таким образом, граф программы будет последовательно преобразовываться в $G_0, G_1, \ldots$, где G_0 - исходный управляющий граф последовательности - граф, состоящий из одной вершины.

Алгоритм А

A_1. В структурном дереве образуются вершины простейшего типа, которым соответствуют вершины исходного управляющего графа.

A_2. В графе программы ищется структурированная конструкция. Если такой нет, то выполняется переход к шагу A_4.

A_3. В дереве образуется вершина структурированной конструкции. Эта конструкция сворачивается в вершину, тип которой указывается. Выполняется переход к шагу A_2.

A_4. Если в графе осталась только одна вершина, то работа алгоритма заканчивается.

A_5. В графе выделяется неструктурируемая конструкция и в структурном дереве образуется вершина соответствующего типа. Конструкция в графе сворачивается, после чего выполняется переход к шагу A_2.

Построение структурного дерева выполняется от элементов, представляющих вершины управляющего графа, к элементу, представляющему всю программу целиком. Используя понятие структурного дерева программы, можно рассчитать существенную сложность программы EV: $EV = 1 + \sum_{i \in NS} (V(i) - 1) S(V(i)\text{-}1)$, где NS - множество вершин неструктурируемого типа в структурном дереве; $V(i)$ - топологическая сложность конструкции, соответствующей i-й вершине дерева.

Вычисление $V(i)$ производится на шаге A_5 алгоритма по формуле $V(i)=e(i)-n(i)+2$, где $e(i)$ - количество дуг, а $n(i)$ - количество вершин рассматриваемой конструкции.

Таким образом, построение структурного дерева программы при реализации алгоритма **А** позволяет получить численные оценки сложности, которые в дальнейшем используются при определении критических путей, требующих наиболее тщательного обследования с позиций технологической безопасности программ.

7.7.3. Определение характеристик взаимосвязи модулей и структурной сложности программ с учетом полного числа связей

Характеристики иерархической структуры и взаимосвязей модулей между собой в процессе решения задач КУП определяются на основе:

- анализа блок-схемы комплекса с выделением глобальных переменных, используемых в программах;
- формирования групп программ, обращающихся к анализируемой программе, к заданной зоне глобальных переменных или к конкретной переменной;
- вычисления числа переменных в программных модулях, количества условных переходов и циклов в программах.

В процессе анализа блок-схемы комплекса программ определяется количество уровней иерархии структуры комплекса (r), распределение

числа программ по уровням $P_1(r)$ и распределение среднего количества команд в программах различных уровней $P_2(r)$.

Для получения этих оценок могут использоваться спецификации программ, предоставляемые разработчиками.

Взаимосвязь модулей комплекса программ между собой характеризуются распределением числа вызываемых программ $P_3(r)$ и распределением числа программ, обращающихся к анализируемой программе - $P_4(r)$. Подсчет сложности взаимодействия модулей выполняется с учетом всех управляющих и информационных связей путем интегрирования характеристик модулей. Для каждого i-го модуля число путей входа в него или число модулей, откуда он вызывается, характеризуется величиной a_{ij} входных управляющих связей. Аналогично число выходных управляющих связей можно представить как число модулей b_{ij}, вызываемых из данного. Наименьшая связность реализуется, когда i-й модуль сопрягается только с одним j-м программным модулем и имеет одну входную (a_{ij}=1) и одну выходную (b_{ij}=1) связи. В этом случае модуль не вызывает другие модули, а только возвращает управление после завершения своей функции. В результате i-тый модуль находится на нижнем уровне в данной ветви комплекса программ, так как он не вызывает других модулей.

В общем случае сложность управляющих связей для i-того модуля можно представить суммой: $C_i = \sum_j (\alpha_{ij} + \beta_{ij})$. Тогда сложность связей группы программ имеет вид: $C = \sum_i C_i = \sum_i \sum_j (\alpha_{ij} + \beta_{ij})$, где суммирование идет по всем модулям, входящим в группу.

С точки зрения технологической безопасности показатели сложности целесообразно определять для всех вершин управляющего графа программы, вне зависимости от структурированности обозначаемых этой вершиной компонентов.

Информационные связи между модулями анализировать сложнее, чем управляющие. Это обусловлено тем, что возможны, с одной стороны, информационные связи между модулями, непосредственно не связанными по управлению, а с другой стороны - тем, что каждая информационная связь может характеризоваться различным числом и типом переменных, участвующих в обмене. В первом приближении информационная связь i-того и j-того модулей может быть оценена количеством глобальных и

обменных переменных с одинаковыми именами, используемых в планируемой паре модулей. Степень зависимости каждого модуля от остальных можно характеризовать количеством типов (индекс k) переменных a_{ij} на входе, необходимых для нормального функционирования данного модуля: $A_i = \sum_k a_{ik}$.

Количество результирующих переменных, подготавливаемых i-м модулем, характеризует степень информационной зависимости от него всех остальных модулей в комплексе: $B_i = \sum_k b_{ik}$, где k - количество типов переменных на входе. При этом, если некоторая k-тая результирующая переменная используется несколькими модулями, то количество информационных связей равно соответственно сумме числа модулей, использующих каждую k-тую переменную.

Переменные, которые используются i-тым модулем не модифицируясь, является только входными и не учитываются как выходные связи.

Таким образом, количественно охарактеризовать информационные связи i-того модуля можно следующим образом: $B_i = \sum_k (a_{ik} + b_{ik})$=$A_i$+$B_i$.

Полное количество информационных связей в группе программ равно сумме связей модулей $B = \sum_i B_i = \sum_i \sum_k (a_{ik} + b_{ik})$=$A_i$+$B_i$.

Каждая информационная связь учитывается в модулях, имеющих переменную на входе. В результате некоторая k-тая глобальная переменная может участвовать в нескольких информационных связях программ, способных использовать эту переменную на входе, и в выходных связях только тех программ, которые модифицируют значение данной переменной. Таким образом, сложность информационных связей через k-тую глобальную переменную можно представить выражением: $B_k = \sum_i (a_{ik} + b_{ik})$.

Обычно каждая глобальная переменная используется двумя или более программами, а модифицируется чаще всего одной, т.е. B_k=3-4. С точки зрения обеспечения технологической безопасности программ необходима тщательная проверка условий модификации переменных, так как именно в

эти моменты может изменяться режим работы программы за счет использования значений глобальной переменной в качестве управляющих элементов.

Обменные переменные обеспечивают взаимодействие только двух программ и сложность их связей по каждой переменной минимальная (B_k=2). Кроме того, обменные переменные упрощают связи за счет обеспечения информационного взаимодействия модулей, непосредственно связанных по управлению.

Информационные и управляющие связи могут быть описаны матрицами связи между модулями. Для каждого номера (имени) программы определяется число глобальных и обменных переменных, по которым эта программа взаимодействует с j-той программой. Такая матрица может формироваться по спецификациям модулей или по паспортам оттранслированных программ. Она позволяет выявить модули, характеризующиеся наиболее сложными информационными связями, и рассчитать суммарную сложность информационных связей комплекса или выделенных групп программ. Показатели сложности используются в дальнейшем в качестве весовых коэффициентов при определении критических путей управляющих программ.

7.7.4. Построение критических путей, подлежащих обязательному тестированию

Выделение критических путей управляющих программ имеет целью определение полного множества маршрутов их выполнения, в которых имеется возможность изменения режимов работы комплекса, его блокирование, модификация выходных результатов и выполнения других операций, способных оказать наиболее значимое влияние на реализацию целевых задач комплекса. Обычно критические пути управляющих программ проходят через модули с наиболее сложными связями, так как скрытое функционирование РПС в простых модулях достаточно легко обнаруживается, что обусловливает стремление противника внедрять программные закладки в наиболее сильносвязанные модули. Данный вывод подтверждается также тем, что помещение программных закладок в критические пути дает возможность реализовать различные виды воздействия на процессы функционирования комплекса программ в зависимости от поступивших по инициативе противника или логически заложенных априорно управляющих воздействий. Кроме того, в данном

случае снижается вероятность обнаружения программных закладок на этапах испытания комплексов программ. Следовательно, с точки зрения технологической безопасности управляющих программ в первую очередь необходимо тестировать именно те модули, которые входят в критические пути выполнения целевых задач комплексов.

Алгоритм построения маршрутов тестирования критических путей основан на использовании управляющего графа и структурного дерева комплекса. При этом необходимо построить маршруты тестирования от входа до выхода (если существуют несколько входов и выходов, что характерно для большинства управляющих программ, то такие маршруты строятся для каждого входа и каждого выхода), покрывающие в совокупности каждую ветвь модулей с наибольшей сложностью связей хотя бы один раз. Алгоритм является модификацией известного алгоритма структурного построения тестов [Ма], причем введенные дополнения ориентированы на специфические особенности выявления РПС в сложных программах.

Алгоритм Б

$Б_1$. В управляющем графе компоненты (вершины – линейные участки и дуги - управляющие связи между линейными участками), относящиеся к одному модулю, условно объединяются в макровершину, которой присваивается весовой коэффициент, соответствующий численному значению сложности информационных связей данного модуля.

$Б_2$. Отмечаются все входы и выходы управляющего графа и фиксируется первый вход.

$Б_3$. Если все входы рассмотрены, то выполняется переход к шагу $Б_9$, иначе выполняется переход к шагу $Б_4$.

$Б_4$. Строится очередной путь, причем первой вершиной пути назначается зафиксированный вход.

$Б_5$. Если текущая вершина пути не выходная, то необходимо перейти к шагу $Б_6$, в противном случае - к шагу $Б_4$ (путь построен).

$Б_6$. Если у последней вершины пути выходят необработанные дуги, то выполняется переход к шагу $Б_7$, в противном случае - к шагу $Б_8$.

$Б_7$. В текущий путь включается необработанная дуга с началом в последней (текущей) вершине пути. Если таких дуг несколько, то выполняется переход к шагу $Б_5$.

$Б_8$. Строится кратчайший путь из последней (текущей) вершины до ближайшей вершины допустимого множества, в которое включаются выходные вершины графа, если в текущем пути уже есть хотя бы одна дуга, а также все вершины, из которых выходят некоторые дуги, если ни одна из вершин не пройдена дважды. Если такой кратчайший путь существует, то он присоединяется к текущему пути, после чего делается переход к шагу $Б_5$. В противном случае фиксируется следующий вход, если это возможно, и делается переход к шагу $Б_3$.

$Б_9$. Участкам пути, входящим в макровершины, присваиваются весовые коэффициенты в соответствии с весами макровершин. Для каждого полного пути определяется весовой коэффициент пути, равный сумме весов входящих в него участков.

$Б_{10}$. Вычисляется среднее значение весовых коэффициентов путей тестирования, после чего составляется множество критических путей, весовые коэффициенты которых выше среднего. Алгоритм заканчивается.

Необходимо сделать некоторые пояснения к алгоритму **Б**:

- при построении кратчайшего пути до ближайшей вершины из допустимого множества используется модификация известного алгоритма Дейкстры (достаточно сравнивать вершину с минимальной временной пометкой на принадлежность допустимому множеству) [Кн];
- при выполнении шага $Б_8$ ограничения на допустимое множество накладываются в соответствии с двумя соображениями: во-первых, желательно получить не слишком длинные пути, так как весьма вероятно, что они окажутся нереализуемыми, а во-вторых, желательно получить как можно больше путей, однако при этом необходимо контролировать выполнение условия, чтобы число путей не превысило топологическую сложность рассматриваемого участка программы (топологическая сложность определяется при построении структурного дерева программы по алгоритму **Б**);

- если в программе имеются циклы с известным числом повторений, например, циклы с заголовком типа арифметической прогрессии, то необходимо в соответствующем месте каждого пути, проходящего через этот цикл, вставить требуемое число повторений той части этого пути, которая связана с прохождением цикла.

Исходя из всего выше сказанного, можно утверждать, что рассмотренный метод позволяет определить наиболее критичные с точки зрения контроля технологической безопасности управляющих программ маршруты их выполнения, что существенно увеличивает вероятность обнаружения дефектов диверсионного типа (при их наличии) в наиболее ответственных участках программы. Кроме того, данный метод позволяет выявить неисполняемые части программы, вычислить все явные и скрытые циклы, обнаружить метки, на которые нет передачи управления, что значительно облегчает задачу отладки комплекса программ при его иерархическом построении.

Таким образом, предложенные методы могут использоваться для получения численных оценок вероятности наличия РПС в исследуемом программном обеспечении вычислительных задач. При этом необходимо четко представлять, что это лишь вероятностные оценки наличия, а не указание места, типа и принципа действия программных закладок. Кроме того, использование предложенного метода будет всегда иметь некоторую постоянную погрешность, обусловленную наличием даже в отлаженном программном комплексе ошибок, допущенных программистами и проектировщиками и не выявленных на этапе отладки. В то же время данный метод наиболее эффективен в том случае, если он используется для анализа программного обеспечения вычислительных задач в конкретной предметной области. В этом случае появляется возможность выделения базового множества типовых вычислительных программ и разработки для них программ расчета эталонных вероятностных характеристик, что позволит минимизировать относительные затраты на разработку таких программ и свести их к величине, обратно пропорциональной количеству исследуемых программных комплексов.

ГЛАВА 8. МЕТОДЫ ОБЕСПЕЧЕНИЯ НАДЕЖНОСТИ ПРОГРАММ ДЛЯ КОНТРОЛЯ ИХ ТЕХНОЛОГИЧЕСКОЙ БЕЗОПАСНОСТИ

8.1. Исходные данные, определения и условия

При исследовании методов и средств оценки уровня технологической безопасности программных комплексов учитываются факторы, имеющие, в том числе, чисто случайный характер. Следовательно, показатели, связанные с оцениванием безопасности ПО лучше всего выражать вероятностной мерой, а для их вычисления *можно использовать вероятностные модели надежности ПО* [Гл,Го,ИКБ,Лип0,ТЛН], которые при осуществлении замены условия надежности функционирования программ на условие их безопасности можно использовать для этих целей.

В общем случае систему программного обеспечения можно рассматривать как подсистему некоторой компьютерной системы. Преимущества подобного представления с точки зрения обеспечения заданных характеристик КС, гарантирующих правильное функционирование программ при наличии отказов программного обеспечения, позволяют перейти к определению понятий отказа и дефекта программ и собственно определению надежности программного обеспечения.

В общем случае *отказы* программного обеспечения определяются как отклонения от правильного хода выполнения программы вследствие ошибок, допущенных в процессе преобразования исходного алгоритма в действующую программу, а *ошибка* – это регистрируемый пользователем факт неудовлетворенности качеством программы и определяется как причина дефекта системы программного обеспечения; и наоборот, *дефект* рассматривается как проявление допущенной ранее ошибки [ТЛН].

На основании вышеприведенного определения отказа, можно определить *надежность программного обеспечения* как вероятность того, что отказ программного обеспечения, вызывающий отклонение получаемого выхода от требуемого за допустимые пределы, не произойдет при определенных условиях внешней среды в течение заданного периода наблюдения. Более подробно это означает следующее.

Во-первых, следует иметь в виду, что не все отказы приводят к уменьшению надежности программного обеспечения, а только те, которые вызывают отклонение выхода от требуемого за допустимые пределы.

Во-вторых, под определенными условиями внешней среды следует понимать описание входных данных и состояние вычислительной системы в процессе выполнения программы. Состояние вычислительной системы описывается в основном объемом оперативной памяти и зависит от требований к программному обеспечению в части его способности нормально функционировать при наличии отказов. Под способностью подразумеваются свойства программного обеспечения, которые закладываются при его проектировании (например, возможность смены программ в памяти; возможность возобновления работ с некоторых контрольных точек и т.д.). В общем случае работа в условиях внешней среды, не предусмотренных техническим заданием и проектом программного обеспечения, приведет к снижению надежности последнего.

Заданный период наблюдений, как правило, представляет собой время, необходимое для выполнения поставленной задачи. Выделение определенного интервала наблюдений для оценки качества программного обеспечения, по-видимому, целесообразно в случае систем реального времени, в которых непредсказуемыми являются число прогонов любой из действующих программ, состояние, базы данных и моменты начала выполнения той или иной программы. В условиях так называемого пакетного режима работы, т.е. когда состояние системы, включая базу данных, достоверно известно, в качестве периода наблюдений следует выбрать рабочий цикл, или прогон. В общем случае каждый раз перед повторным выполнением программы необходимо либо восстанавливать состояние памяти, либо осуществлять серию последовательных прогонов программы, при которых определенным образом последовательно изменяется состояние базы данных.

Под определенными *условиями внешней среды* следует понимать описание входных данных и состояние вычислительного процесса в момент выполнения программы при испытаниях. Под заданным периодом функционирования понимается время, необходимое для выполнения поставленной задачи. Выделение определенного интервала времени целесообразно в случае систем реального времени, в которых неопределенными являются количество прогонов любой из действующих программ, состояние баз данных и моменты выполнения той или иной программы. В условиях, когда состояние программы достоверно известно в качестве периода наблюдений следует выбрать рабочий цикл или *прогон*. В любом случае перед каждым повторным выполнением программы необходимо либо восстанавливать состояние памяти, либо осуществлять

серию последовательных прогонов, при котором последовательным образом изменяется состояние базы данных.

Введем следующие обозначения. Пусть **П** - машинная программа, которая может быть определена как описание некоторой вычислимой функции F на множестве E всех значений наборов входных данных, таких что каждый элемент E_i множества E представляет собой набор значений данных, необходимый для выполнения прогона программы: $E=(E_i{:}i=1,2,...,N)$;

Интуитивное определение безопасности ПО может быть уточнено в статистическом смысле на основе следующих простых соображений:

- выполнение программы **П** приводит к получению для каждого E_i определенного значения функции $F(E_i)$;
- множество E определяет все возможные вычисления в программе **П**, то есть каждому набору входных данных E_i соответствует прогон программы **П**, и наоборот, каждому прогону соответствует некоторый набор входных данных E_i;
- наличие дефектов в программе **П** приводит к тому, что ей на самом деле соответствует функция F', отличная от заданной функции F;
- для некоторого E_i отклонение выхода $F'(E_i)$, полученного в результате выполнения программы не должно превышать некоторый установленный уровень безопасности программного обеспечения $S(E_i)$, то есть безопасность обеспечивается при соблюдении ограничения: $F'(E_i) \leq S(E_i)$.

Вопрос о том, приводит ли некоторое отклонение выхода к нарушению условия безопасности, должен решаться в каждом конкретном случае отдельно, поскольку все определяется конкретными особенностями поведения системы после нарушения ее работы.

8.2. Краткий анализ существующих моделей надежности программного обеспечения

8.2.1. Определение модели надежности программного обеспечения

Модель надежности программного обеспечения - это, как правило, математическая модель, построенная для оценки зависимости надежности программного обеспечения от некоторых определенных параметров. Например, параметрами, связанными с некоторой ветвью программы на подмножестве наборов входных данных, с помощью которых эта ветвь контролируется. Другими такими параметрами являются частоты ошибок,

которые позволяют оценить качество систем реального времени, функционирующих в непрерывном режиме, и, в то же время получать косвенную информацию о надежности ПО.

При разработке эмпирических моделей надежности программ предполагается, что связь между надежностью и другими параметрами является статической. С помощью подобного подхода обычно пытаются количественно оценить те характеристики ПО, которые свидетельствуют либо о высокой, либо о низкой его надежности. Иначе говоря, при разработке эмпирической модели надежности ПО стремятся иметь дело с такими параметрами, соответствующее изменение значений которых должно приводить к повышению надежности ПО.

Рассмотрим некоторые модели надежности программного обеспечения, которые в той или иной степени могут представлять интерес в контексте исследуемой работы.

8.2.2. Модель Шумана

Целесообразность применения модели Шумана [ТЛН] для оценки надежности программного обеспечения зависят от принятых допущений и условий, наиболее существенным из которых является условие существования программы-исследователя системы. Остальные допущения и условия носят статический характер и не связаны с какими-либо специфическими свойствами программного обеспечения. По существу они сводятся к следующему:

- предполагается, что в начальный момент компоновки программных средств в системе имеется E_t, ошибок. С этого момента начинается отсчет времени отладки t, которое включает затраты времени на выявление ошибок с помощью тестов, на контрольные проверки и т.п. при этом время исправного функционирования системы не учитывается;
- предполагается, что значение функции частоты отказов $m(t)$ пропорционально числу ошибок, оставшихся в программном обеспечении после израсходования времени t на отладку исследуемой программы.

Программа-исследователь системы должна снабжать испытуемые программные средства входными данными, отражающими реальные условия функционирования. Такие данные называются *функциональным разрезом* и определяются главным образом через распределение вероятностей значений входных переменных.

8.2.3. Модель Джелинского-Моранды

Данная модель [ТЛН], предназначенная в основном для прогнозирования надежности программного обеспечения, представляет собой частный случай модели Шумана. При разработке этой модели предполагалось, что значения интервалов времени отладки (в смысле модели Шумана) между обнаружением двух ошибок имеют экспоненциальное распределение с частотой ошибок (или интенсивностью отказов), пропорциональной числу невыявленных дефектов. Каждая обнаруженная в программе ошибка немедленно устраняется и, следовательно, число оставшихся ошибок уменьшается на единицу.

Модификация модели Джелинского-Моранды была предложена Шиком и Волвертоном. В основе модели Шика-Волвертона лежит предположение, согласно которому частота ошибок пропорциональна не только количеству ошибок в программах, но также и времени отладки, т.е. вероятность обнаружения ошибок с течением времени возрастает.

8.2.4. Другие модели надежности программного обеспечения

Модели Джелинского-Моранды и Шика-Волвертона могут быть обобщены, если допустить возможность возникновения на рассматриваемом интервале более одной ошибки и считать, что исправление ошибок производится лишь после истечения интервала времени, на котором они возникли.

В модели Вейса [ТЛН] помимо предположения об экспоненциальном распределении интервала времени между отказами считается, что существует M причин возможных отказов в начале процесса разработки программного обеспечения. Таким образом, рассмотренная модель может быть полезной для оценки надежности ПО, если принять предположение, что каждая из M причин (источников) отказов представляет собой ошибки некоторого типа, многократно встречающиеся в ПО, например «недостаточен выделенный объем памяти для массива данных».

Для ошибок некоторых типов, например таких, как в описанном случае, существует высокая вероятность того, что при однократном их обнаружении и устранении возможность появления такой ошибки в будущем будет исключена. Для ошибок других типов однократное их обнаружение приводит к исключению подобных ошибок только из относительно небольшой части мест в программном обеспечении. Эта часть ошибок в описанной модели характеризуется вероятностью p_c,

которая, вообще говоря, различна для каждого источника ошибок и должна быть предварительно оценена на основе фактических данных.

Как и во всех моделях для прогнозирования надежности функционирования технических устройств, в случае моделей надежности ПО временной интервал должен выбираться из соображений обеспечения представительности исходных данных. Это означает, что тестирование программ даже в течение длительного промежутка времени с использованием лишь ограниченной области значений входных данных даст несмещенные оценки надежности только в том случае, если в реальных условиях вероятность появления значений входных данных из непроверенной области близка к нулю.

Модели, предложенные Коркореном и др. [ТЛН] заслуживают внимания по следующим причинам. Во-первых, они содержат изменяющиеся вероятности отказов для различных источников ошибок и соответственно разные вероятности их исправления. Во-вторых, они относительно просты с математической точки зрения и допускают простую интерпретацию данных об ошибках в ПО. В этих моделях не используется такой параметр, как время тестирования, а учитывается только результат N испытаний, в которых наблюдается N_i ошибок i-го типа (относящихся к i-тому источнику отказов).

8.2.5. Анализ моделей надежности программ

В моделях Шумана и Джелинского-Моранды в качестве независимой переменной фигурируют время и надежность, вычисляемая по формуле $R(t)=e^{-ht}$.

Значение интенсивности отказов h предполагается постоянным в течение всего периода функционирования системы и изменяется только при обнаружении и устранении ошибок, после чего время t снова отсчитывается от нуля. Поскольку эта формула может быть получена как частный случай рассматриваемой ниже модели Нельсона при некоторых допущениях целесообразно определить условия, при которых верны модели Шумана и Джелинского-Моранды. Эти условия таковы:

- время t должно интерпретироваться как суммарное время работы программы относительно некоторого определенного начального момента времени;
- время t должно быть больше средней длительности выполнения одного прогона программы Δt;

- наборы входных данных для последовательных прогонов программы должны выбираться случайным образом в соответствии с законом распределения, приближенно отражающим реальные условия функционирования, относительно которых производится оценка надежности.

Авторы и той и другой модели попытались расширить их в предположении, что интенсивность отказов пропорциональна числу оставшихся в программе ошибок, а затем применить эти модели к тестированию программ.

В моделях Вейса и Коркорэна [ТЛН] были предложены формулы для определения степени повышения надежности программы в процессе тестирования, причем в этих моделях была предпринята попытка учесть эффект существования в программе нескольких источников ошибок.

Перечисленные модели нашли ограниченное применение, так как в них слабо учитывались свойства программ, режимы функционирования и стратегии испытаний. Эти модели основывались главным образом на таких общих закономерностях теории вероятностей, как экспоненциальная зависимость надежности от числа испытаний, и на весьма упрощенных представлениях относительно последствий программных ошибок. Хотя в этих моделях экспоненциальный характер зависимости надежности установлен достаточно точно, другие свойства программного обеспечения в них учитываются в лучшем случае весьма приближенно.

8.3. Описание модели Нельсона

8.3.1. Обоснование выбора модели Нельсона

Модель Нельсона была разработана с учетом основных свойств машинных программ и использует методы теории вероятностей лишь в тех случаях, когда невозможно получение полной информации о том или ином факторе, например при ответе на вопрос, какой набор входных данных следует взять при следующем прогоне программы. Все приближения, принятые модели, четко определены, и известны границы их применимости. Поскольку в основу модели Нельсона положены свойства программного обеспечения, она допускает развитие за счет более детального описания других аспектов надежности. Некоторые из полученных обобщений модели [ТЛН] могут рассматриваться в контексте исследования проблемы безопасности программ и рассматриваются ниже. Вследствие отмеченных особенностей модели ее можно рассматривать в

целом как математическую теорию надежности программного обеспечения, а не как простую модель надежности.

8.3.2. Общее описание модели

Описание метода Нельсона приведем в соответствии с работой [ТЛН]. Для описания введем сначала следующие обозначения.

Совокупность действий, включающая ввод E_i, выполнение программы **П**, которое заканчивается получением результата $F'(E_i)$ называется прогоном программы **П**. Необходимо также отметить, что значения входных переменных, образующие E_i, не должны все одновременно подаваться на вход программ **П**. Таким образом, вероятность P того, что прогон программы приведет к обнаружению дефекта, равна вероятности того, что набор данных E_i, используемый в данном прогоне, принадлежит множеству E_e. Если обозначить через n_e число различных наборов значений входных данных, содержащихся в E_e, то $P=n_e/N$ - есть вероятность того, что прогон программы на наборе входных данных E_i, случайно выбранном из E среди равновероятных, закончится обнаружением дефекта. При этом $R=1-P$ - есть вероятность того, что прогон программы **П** на наборе входных данных E_i, случайно выбранном из E среди априорно равновероятных, приведет к получению приемлемого результата.

Однако в процесс функционирования программы выбор входных данных из E обычно осуществляется не с одинаковыми априорными вероятностями, а диктуется определенными условиями работы. Эти условия характеризуются некоторым распределением вероятностей p_i, того, что будет выбран набор входных данных E_i. Распределение P может быть определено через p_i с помощью величины y_i, которая принимает значение 0, если прогон программы на наборе E_i заканчивается вычислением приемлемого значения функции, и значением 1, если этот прогон заканчивается обнаружением дефекта. Поэтому $P=\sum_{i=1}^{N} p_i y_i$ - есть вероятность того, что прогон программы на наборе входных данных E_i, выбранных случайно с распределением вероятностей p_i, закончится обнаружением дефекта. При этом $R=1-P$ есть вероятность того, что прогон программы **П** на наборе входных данных E_i, выбранных случайно с распределением вероятностей p_i, приведет к получению приемлемого результата.

Так как R - вероятность того, что единичный прогон программы не закончится отказом на наборе входных данных, выбранных в соответствии с распределением p_i, то вероятность успешного выполнения n прогонов этой программы при независимом для каждого прогона выборе входных данных в соответствии с распределением P будет равна $R(n)=R=(1-P)^n$.

Таким образом, можно дать следующее математическое определение надежности машинной программы: *надежность программы - это вероятность безотказного выполнения n прогонов программы.* Поэтому прогон является единичным испытанием программы.

На практике обычно выбор входных данных для каждого прогона нельзя считать независимым. Исключение составляют лишь такие последовательности прогонов, которые определяются возрастающими значениями и некоторой входной переменной или некоторым порядком установленных процедур, как в случае программ, работающих в реальном масштабе времени.

С учетом введенного определения функциональный разрез должен быть переопределен в терминах вероятностей p_{ji} выбора E_i в качестве входных данных при j-том прогоне из некоторой последовательности прогонов. Тогда вероятность того, что j-тый прогон закончится отказом, может быть записана в виде $P_i=\sum_{i=1}^{N} p_{ji} y_i$.

Надежность *R(n)* программы **П** равна вероятности того, что в последовательности из n прогонов ни один из них не закончится отказом:

$$R(n)=(1-P_i)(1-P_2)...(1-P_n)=\prod_{j=1}^{n}(1-P_j)$$

Эта формула может быть представлена в виде $R(n)=e^{\sum_{j=1}^{n}\ln(1-P_j)}$.

Некоторые из свойств функции $R(n)$ могут стать более зримыми при следующих допущениях:

для $P_j<<1$

$$R(n)=e^{-\sum_{j=1}^{n}P_j},$$

если $P_j=P$ для всех j, то

$$R(n)=e^{-P_n} .$$

С помощью соответствующих замен переменных и подстановок можно выразить функцию $R(n)$ через время функционирования t. Обозначим через Δt_j время выполнения j-того прогона, через $\sum_{i=1}^{j} \Delta t_i$ - суммарное время выполнения первых j прогонов программы n примем, что $h(t_j) = -\frac{\ln(1-P_j)}{\Delta t_j}$. Тогда $R(n) = e^{-\sum_{j=1}^{n} \Delta t_j h(t_j)}$.

Если величина Δt_j стремится к нулю с ростом n, то сумма в показателе экспоненты становится интегралом и формула для надежности программного обеспечения принимает вид

Эта формула хорошо известна в теории надежности технических устройств. В случае P_j<<1 функция $h(t_j)$ может быть интерпретирована как функция интенсивности отказов, которая, будучи умноженной на Δt_j, дает условную вероятность появления отказов и интервале $(t_j, t_j+\Delta t_j)$ при отсутствии отказов до момента t_j.

8.3.3. Оценка надежности программного обеспечения

Надежность машинной программы может быть оценена путем прогона программы на n наборах входных данных и расчета значения оценки R' по формуле $R'=1-n_e'/n$, где n_e - число наборов входных данных, при которых произошли рабочие отказы.

Если выборка включает n наборов входных данных из E и осуществляется в соответствии с распределением вероятностей p_i, то расчетное значение R' будет представлять собой несмещенную оценку надежности R в том смысле, что для p_i<<1 ожидаемое значение на всем множестве наборов входных данных в пределах выборки равно R. Это может быть показано, если ввести *характеристику выборки* z_{ij}, которая определяется так, что $z_{ij}=1$, если E_i входит в выборку j и $z_{ij}=0$, в противном случае.

Число различных наборов входных данных n_j, входящих в некоторую выборку j, может быть меньше, чем n, так как некоторые наборы E_i могут быть включены в нее более одного раза. Поэтому $\sum_{i=1}^{N} z_{ji} = n_j$.

Однако в большинстве случаев число возможных наборов входных данных N настолько больше размера выборки, что повторный выбор

каких-либо наборов маловероятен. Если s - вероятность того, что взята выборка j и M – количество возможных выборок, то $\sum_{i=1}^{M} z_{ij} s_i = 1-(1-p_i)^n$.

Так как R' можно представить в виде $R'=\frac{1}{n}\sum_{i=1}^{N}(1-y_i)z_{ij}$, то математическое ожидание R' равно

$$M(R')=\sum_{i=1}^{M} s_j R'_i = \frac{1}{n}\sum_{j=1}^{M} s_j \sum_{i=1}^{N}(1-y_i)z_{ij} = \frac{1}{n}\sum_{i=1}^{N}(1-y_i)\sum_{j=1}^{M} s_j z_{ij} =$$

$$=\frac{1}{n}\sum_{i=1}^{N}(1-y_i)[1-(1-p_i)^n]=\sum_{i=1}^{N}(1-y_i)p_i .$$

При p<<1 значение $M(R')$=1-P=R.

Для получения оценки R должен быть определен функциональный разрез p_i программы. На практике этот разрез определяется путем разбиения всего пространства значений входных переменных на подпространства и нахождения вероятностей того, что выбранный набор входных данных будет принадлежать конкретному подпространству. Определение этих вероятностей основано на оценке вероятностей появления тех или иных входов в реальных условиях функционирования, относительно которых оценивается надежность программы.

Как только вероятности p_i определены указанным образом, случайная выборка из n наборов входных данных, распределенных в соответствии с p_i, может быть получена с помощью некоторого датчика случайных чисел. После реализации n испытаний программы для одних входных наборов данных результаты окажутся правильными, а для других могут быть зафиксированы отказы. При этом процесс испытаний не должен прекращаться, а ошибки не должны исправляться до завершения всех n прогонов; на основании полученных данных может быть вычислена оценка надежности R'.

Для дальнейшего рассмотрения нам будут необходимы следующие определения и обозначения, связывающие структурные характеристики программ с их безопасностью. Структурными характеристиками программы **П** являются множество ветвей L_j (j=1,...,n), подмножества входных наборов данных G_j, соответствующие ветвям L_j, множества сегментов Seg$_j$, из которых состоят отдельные ветви, совокупность

операторов ветвления, которые обеспечивают переход от одного сегмента к другому при движении по отдельной ветви программы.

8.4. Оценка технологической безопасности программ на базе метода Нельсона

В данном разделе будем считать, что безопасность программного обеспечения - это вероятность того, что преднамеренные программные дефекты, вызывающие критическое поведение управляемой КС, будут обнаружены при определенных условиями внешней среды и в течение заданного периода наблюдения при испытаниях.

Перед тем как перейти непосредственно к методу оценки, необходимо сделать несколько замечаний. Следует заметить, что реальные условия испытаний программ всегда существенно отличаются от тех, которые требуются для представительного измерения уровня безопасности ПО. Так, например, тестовые прогоны выполняются на входных наборах данных, выбранных не совсем случайным, а выбранных некоторым определенным образом: обычно выбор производится так, чтобы соответствующую ошибку можно было найти как можно быстрее. При этом выбор основывается на опыте и интуиции испытателей, либо осуществляется с учетом функциональных возможностей, которые должна обеспечивать программа, или возможностей соответствующих методик испытаний. Поэтому контрольные примеры, как правило, не являются представительными с точки зрения моделирования реальных условий работы программы и далее описывается процедура грубой оценки величины R, предусматривающая использование результатов испытаний и включающая следующие шаги:

1. Определение множества E входных массивов.
2. Выделение в E подмножеств G_j, связанных с отдельными ветвями программы.
3. Определение для каждого G_j в предполагаемых условиях функционирования значений вероятности P_j.
4. Определение подмножества G_j для каждого входного набора данных, используемого в контрольных примерах.
5. Выявление проверенных пар и непроверенных в ходе испытаний сегментов и пар сегментов.
6. Определение для каждого j величины $P'=a_jP_j$, где a_j определяется в соответствии со следующими правилами [ТЛН].

- a_j=0,99, если подмножество G_j включает более одного контрольного примера;
- a_j=0,95, если подмножество G_j включает ровно один контрольный пример;
- a_j=0,90, если подмножество G_j не включает ни одного контрольного примера, но в процессе проверки программы были найдены все сегменты и все сегментные пары ветви L_j;
- a_j=0,80, если в ходе испытаний были опробованы все сегменты, но не все сегментные пары;
- a_j=0,80-0,20m, если m сегментов ($1 \le m \le 4$) ветви L_j не были опробованы в ходе испытаний;
- a_j=0, если более чем 4 сегмента не были опробованы в процессе испытаний.

7. Вычисление грубой оценки R'' осуществляется по формуле $R'' = \sum_{j=1}^{k} P'_i$, где k представляет собой общее число ветвей программы.

Приведенные выше параметры a_j были определены интуитивно [ТЛН] на основе анализа теоретических результатов исследования и экспериментальных результатов тестирования различных программ. Для того, чтобы получить более точные оценки величины R'' необходимо провести измерения с использованием подходящего метода формирования выборки.

Оценка технологической безопасности ПО осуществляется посредством проверки условия $R'' \le S''$, где S'' установленная нормативными документами граница безопасности ПО. Отметим также, что для систем критических приложений такая граница должна быть достаточно высокой, то есть стремится к 1.

ГЛАВА 9. ПОДХОДЫ К ЗАЩИТЕ РАЗРАБАТЫВАЕМЫХ ПРОГРАММ ОТ АВТОМАТИЧЕСКОЙ ГЕНЕРАЦИИ ИНСТРУМЕНТАЛЬНЫМИ СРЕДСТВАМИ ПРОГРАММНЫХ ЗАКЛАДОК

9.1. Способы внедрения РПС посредством инструментальных средств

Современный этап развития технологии создания программных комплексов характеризуется ориентацией практически всех специализированных организаций на применение методов проблемно-ориентированного программирования. При этом повышение качества программ основывается на использовании мощных инструментальных средств разработки и отладки программ. Использование для автоматизации процессов производства в области программотехники инструментальных средств автоматизации программирования значительно осложняет проблему выявления и устранения РПС. Это обусловлено тем, что программист практически не имеет возможности контролировать непосредственно создаваемые программы, так как работает на уровне логических конструкций языковых средств.

Если целью атаки является нанесение как можно большего вреда, то заманчивой целью для нарушителя, пытающегося внедрить РПС, являются программы, которые используют много различных пользователей, например, компиляторы [РПС]. Для того чтобы понять, как это можно сделать рассмотрим следующую упрощенную структуру компилятора, которая дает представление об общих принципах его работы:

```
compile:
  get (line);
  translate(line);
```

Согласно этой структуры компилятор сначала «получает строку», а затем транслирует ее. Конечно, настоящий компилятор устроен намного сложнее, чем эта схема, но этой иллюстрации отдано предпочтение потому, что она в виде некой модели разъясняет фазы лексического анализа трансляции компилятора. Целью РПС будет поиск новых текстовых участков во входных программах, которые будут транслироваться, и вставление в эти участки различного кода. В примере, представленном ниже, компилятор ищет текстовый участок «read_pwd(p)», наличие которого в функции входа в данную компьютерную систему

известно, как мы предполагаем, нападающей программе. Когда этот участок будет найден, компилятор не будет транслировать «read_pwd(p)», а вместо этого буде транслировать вставку из РПС, которая может устанавливать «лазейку», которая потом позволит злоумышленнику легко получить доступ к системе. Измененный код компилятора следующий:

```
compile;
  get (line)
  if line=«readpwd(p)» then
     translate (destructive means insertion);
  else
     translate(line);
  fi;
```

В этом измененном коде, компилятор получает строку, ищет нужный код текст, и если находит, то транслирует код РПС. Код РПС может включать в себя простую проверку пароля «черного входа» (например, может признаваться правильный пароль «12345» для любого пользователя). Это особенно опасно, поскольку код источника больше не отражает того, что находится в объектном коде и просмотр кода источника (не смотря на то, что проверяется и компилятор) никогда не позволит уловить эту атаку (см рис.9.1).

Заметим, что на рис.9.1 исходный текст или исполняемый код, включающий только те выражения, которые предлагались его разработчиком назван чистым, а код, содержащий РПС, - грязным. Далее заметим, что если компилируется атакованный компилятор и грязный исполняемый код устанавливается код в какой-либо рабочий каталог (так обычно и бывает), то компилятор с внедренным РПС может быть обнаружен, только если кто-нибудь вернется к источнику компилятора и проверит его (что редко случается). Но настоящий источник может быть восстановлен злоумышленником после компилирования грязного источника и создания грязного исполняемого компилятора. Это, вообще говоря, потом поможет восстановить настоящий исполняемый компилятор при рекомпиляции источника, но это редкий случай.

Более изощренный метод внедрения РПС заключается в реализации следующего алгоритма [To].

```
compile;
  get (line)
  if line=«pattern_1» then
     translate (destructive means insertion_N1);
```

```
if line=«pattern_2» then
   translate (destructive means insertion_N2);
else
   translate(line);
fi;
```

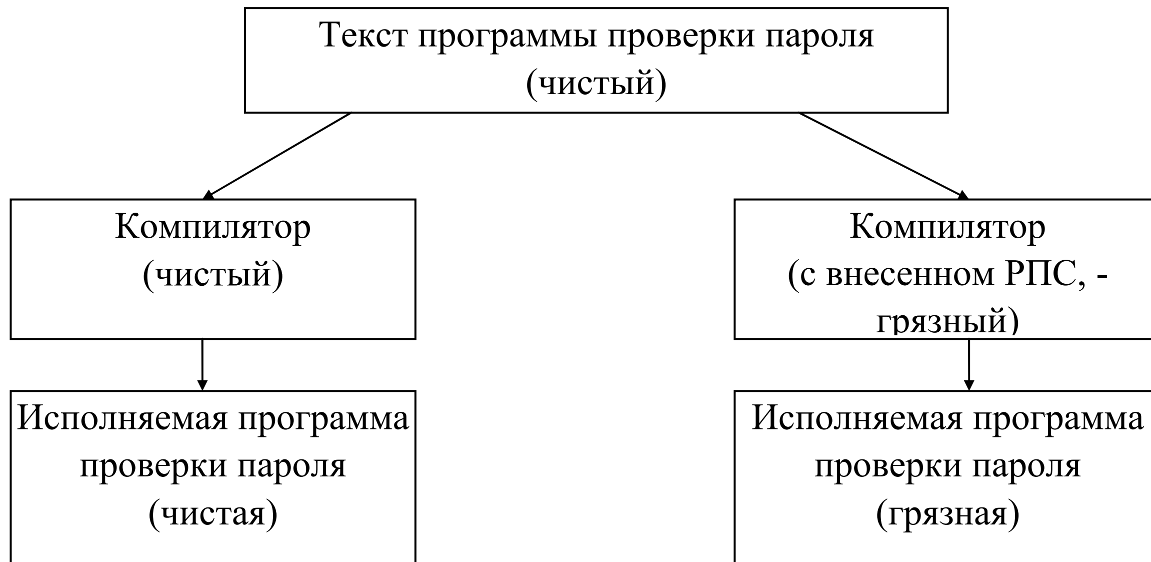


Рис.9.1. Работа компилятора с привнесенным РПС

Сущность метода заключается в простом добавлении еще одного РПС к уже существующему. Второй образец настроен на сам компилятор. Заменяющим текстом может служить самовоспроизводящаяся программа [To], которая вставляет оба РПС в компилятор. В этом случае необходима также фаза обучения. Сначала компилируется модифицированный исходный текст нормальным компилятором, когда получается готовая программа с РПС. Затем устанавливается эта готовая программа как официальный компилятор. Теперь можно удалить РПС из исходного текста компилятора, а новый компилятор будет воспроизводить РПС при каждой перекомпиляции. Команда login (например для ОС Unix), естественно, будет оставаться с РПС без всякого следа в каких-либо исходных текстах.

Таким образом, по мнению лауреата премии Тьюринга К. Томпсона [To]: «Нельзя доверять программе, которую вы не написали

полностью сами. Сколько бы вы не исследовали и не верифицировали исходный текст - это не защитит вас от троянской программы. Для демонстрации атаки такого рода я выбрал компилятор Си. Я мог бы выбрать любую программу, обрабатывающую другие программы - ассемблер, загрузчик или даже микропрограмму, зашитую в аппаратуру. Чем ниже уровень программы, тем труднее и труднее обнаруживать подобные «жучки». Мастерски встроенный «жучок» в микропрограмме будет почти невозможно обнаружить».

9.2. Возможные методы защиты программ от потенциально опасных инструментальных средств

Кроме трансляторов, компиляторов и интерпретаторов и целый ряд других инструментальных средств автоматизации программирования могут иметь встроенные средства автоматического генерирования программных закладок, которые включаются в текст производимой программы в ключевых местах, например, перед анализом условий прерывания, в блоках-переключателях, в блоках управления памятью и т.п. Следовательно, одной из задач обеспечения технологической безопасности программ является диагностический контроль инструментальных средств, позволяющий убедиться в отсутствии в них средств генерации преднамеренных программных дефектов.

Решение данной задачи может основываться на применении функциональных диаграмм [Гл,Ма], позволяющих строить тестовые последовательности по спецификации программной продукции. При этом спецификация исследуемых инструментальных средств разбивается на сегменты приемлемой сложности, в каждом из которых выделяются входные данные (команды инициализации операций) и события, представляющие собой реакцию программных средств на поступившую команду. Каждая команда может рассматриваться как отдельное входное условие или класс эквивалентности входных условий. Реакция программных средств наступает в виде следствия одного или нескольких входных условий. Для удобства анализа программных средств все команды и события получают уникальные номера, а связь команд с событиями определяется матрицей из нулей и единиц. На основании анализа семантики спецификаций инструментальных средств устанавливаются функциональные связи между командами и событиями, которые описываются булевыми функциями в базисе «И,ИЛИ,НЕ». В более сложных случаях могут использоваться расширения

этого базиса за счет функций «И-НЕ», «ИЛИ-НЕ». Ограничения на возможные комбинации входных команд также задаются на основе анализа спецификаций.

Наиболее сложным и слабо формализованным этапом построения тестов диагностического контроля технологической безопасности инструментальных средств является этап нахождения по функциональным диаграммам множества наборов входных условий (тестовых наборов), удовлетворяющих входным ограничениям, при которых имеет место декларированная в спецификации комбинация событий. Описываемый ниже метод [ЕПУ] предназначен для упорядочения данной процедуры, а также для предотвращения возможности частичного пропуска функциональных участков тестируемых программ, слабо отраженных в спецификациях.

Сущность метода заключается в предоставлении функциональной диаграммы в виде логической сети без обратных связей, а также использовании специально введенных операций «прямого продвижения» и «обратного продвижения», часто применяемых в технической диагностике [Н]. В данном случае в логической сети, задающей функциональную диаграмму, входные переменные соответствуют командам, а выходные - следствиям. Все логические элементы сети соответствуют функциональным связям. В дальнейшем для упрощения вместо термина «логическая сеть», задающая функциональную диаграмму, будет использоваться просто «функциональная диаграмма».

Для упорядочения операций с функциональными диаграммами вводится понятие *ранга функциональных связей*. Выходная функциональная связь (ФС) относится к рангу $r=1$. К рангу ($i+1$) относится ФС, выходы которых являются входами ФС ранга i. Ранжирование ФС завершается после определения рангов для всех связей. Необходимо отметить, что если в функциональной диаграмме выход некоторой ФС разветвляется, то может оказаться, что ФС, уже отнесенная к некоторому рангу, в соответствии с описанным правилом должна быть отнесена к другому, большему рангу. В этом случае ФС относится к большему рангу. Максимальное значение ранга ФС называется рангом функциональной диаграммы.

Операция «прямого продвижения» позволяет определить комбинацию событий, вызываемую заданной комбинацией команд. Операция «обратного продвижения» заключается в определении множества команд, вызывающих заданную совокупность событий. При описании этих

операций используется понятие «вырожденное покрытие» булевой функции, описывающей ФС.

Вырожденное покрытие (*ВП*) булевой функции представляет собой сжатую таблицу истинности данной функции. При этом вырожденное покрытие булевой функции f от n аргументов есть совокупность (n+1)-разрядных строк, называемых кубами, в которых первые слева n разрядов представляют набор значений аргументов, а (n+1)-й разряд - значение булевой функции на этом наборе. В отличие от обычной таблицы истинности булевой функции, в которой аргументы принимают значения 0 или 1, аргументы в ВП могут принимать значение x (неопределенное). Если значение аргумента y равно x в некотором кубе ВП, то это значит, что значение булевой функции на наборах аргументов, задаваемых кубом, не зависит от значения y, т.е. y может принимать как значение 0 так и значение 1 и задавать поэтому две строки таблицы истинности булевой функции.

Функциональная диаграмма характеризуется вектором состояния P, размерность которого равна сумме числа ФС и числа входных условий. Каждый разряд вектора P представляет значение либо команды, либо ФС и может принимать одно из 3-х значений: 0, 1 или x. Считается, что значение разряда вектора P определено, если в нем записан 0 или 1.

Операционная схема метода состоит из этапов, каждый их которых реализуется с использованием собственного алгоритма, наиболее значимыми из которых являются следующие:

- алгоритм построения вырожденного покрытия функциональной диаграммы (алгоритм **А**);
- алгоритм определения множества комбинаций входных команд для заданной совокупности событий (алгоритм **Б**);
- алгоритм вычисления эталонных значений событий для заданных тестовых наборов команд (алгоритм **В**).

Алгоритм А

Вырожденное покрытие функциональной диаграммы строится из ВП булевых функций, описывающих ФС. Алгоритм определения ВП булевой функции f состоит из следующих процедур.

1. Любым известным методом [Н] находится минимальная дизъюнктивная нормальная форма (МДНФ) функции f и ее отрицания.

2. Находятся кубы ВП, соответствующие значениям f=1. Для этого каждый простой импликанте ставится в соответствие куб, значения разрядов которого формируются по следующим правилам:

2.1. Если переменная входит в импликанту без отрицания, то в кубе, в котором f=1, ей соответствует 1, если же переменная входит с отрицанием, то значение 0;

2.2. Если переменная не входит в импликанту, то в кубе ей соответствует x.

3. Находятся кубы ВП, соответствующие значениям f=0. Для этого к МДНФ отрицания функции f применяют правила, определенные пунктом 2 алгоритма **А**.

Вырожденные покрытия булевой функции, описывающих ФС и применяемых при построении функциональной диаграммы имеют вид, показанный в табл.9.1.

Таблица 9.1

Функции							
И			ИЛИ			НЕ	
y_1	y_2	f	y_1	y_2	f	y	f
0	x	0	1	x	1	0	1
x	0	0	x	1	1	1	0
1	1	1	0	0	0	-	-

Вырожденное покрытие функциональной диаграммы строится по следующей процедуре.

1. Функциональные связи, входящие в функциональную диаграмму, нумеруются по следующим правилам:

а) номера 1,...,a присваиваются входным командам в произвольном порядке;

б) номера a+t,...,b, t≥1 присваиваются функциональным связям так, что номер выходного условия ФС (события) всегда больше номера ее любого входного условия (команды).

2. Строится таблица, имеющая d столбцов (d - количество ФС).

3. Начиная с ФС с наименьшим номером, в таблицу заносятся вырожденные покрытия булевых функций, описывающих ФС с учетом принятой нумерации.

Для реализации алгоритма **Б** выполняется операция «обратного продвижения», служащая для определения множества комбинаций входных условий (команд), при реализации которых имеет место заданная комбинация событий, соответствующая вектору P состояния функциональной диаграммы. При этом в векторе P должны быть определены все события, являющиеся следствиями из заданной комбинации команд, а остальные разряды вектора P должны быть равны x. На комбинацию событий может быть наложено ограничение $M(y_1,y_2)$, запрещающее одновременное равенство единице y_1 и y_2.

Продвижение от событий к командам в соответствии с вектором P заключается в выполнении алгоритма **Б**. Исходными данными при этом являются: вектор состояния функциональной диаграммы (совокупность значений реакций исследуемых инструментальных средств на входные команды) и вырожденное покрытие функциональной диаграммы.

Алгоритм Б

1. Начальная установка номера шага j:=0.

2. Из вектора P_j состояния ФД выбирается ФС S_h с наибольшем номером, ранее не рассматривавшаяся, значение которой в P_j определено (0 или 1).

3. Если S_h есть входное условие ФД, то алгоритм завершается, т.е. значения входных условий, определенных в P_j, являются искомыми.

В противном случае выполняется п.4.

4. Из таблицы вырожденных покрытий ФС выбираются кубы $K_h^{(1)}$,...,$K_h^{(k)}$, в которых значения ФС совпадают со значениями, определенными в P_j, а значения входных условий ФС не противоречат их значениям в P_j. Противоречие имеет место тогда и только тогда, когда значение входного условия в кубе $K_n^{(i)}$ равно 0(1), а в P_j противоположное - 1(0). Если h больше наименьшего номера ФС, значение которого определено в P_j, но из вырожденного покрытия ФС S_h нельзя выбрать ни одного куба, удовлетворяющего указанным условиям, то P_j не реализуем, т.е.

не существует комбинации входных условий, при которой значение следствий соответствует P_j.

5. Находится множество векторов $P_j^{(1)},...,P_{j+1}^{(k)}$ за счет поразрядного пересечения расширенных кубов $K_h^{(1)},...,K_h^{(k)}$ вырожденного покрытия S_h с вектором P_j: $P_{j+1}^{(i)} = P_j P\{K_h^{(i)}\}$, $i=1,...,k$, причем операция П выполняется по следующим правилам: выбранные кубы расширяются за счет присвоения всем неопределенным ФС значения x, 1П1=1, 0П0=0, 1Пx=1, 0Пx=xП0=0, 1П0=0П1=d.

6. $j:=j+1$; пункты 2-5 выполняются для каждого из найденных векторов $P_g^{(i)}$, $i=1,...,k$.

В результате работы этого алгоритма заполняются соответствующие строки таблицы решений [Го]. Для получения полной таблицы решений необходимо выполнить алгоритм для всех комбинаций событий, включаемых в таблицу 9.1.

Рассматриваемые комбинации входных команд могут содержать некоторые неопределенные значения (x). Предположим, что таких значений l. Тогда существует 2^l полностью определенных комбинаций входных команд, причем доопределение комбинаций для получения тестовых наборов должно осуществляться с учетом ограничений, полученных при анализе спецификаций исследуемых на технологическую безопасность инструментальных средств. Возможны следующие ограничения на входные условия:

- $E(x_1,...,x_n)$ - запрещены комбинации входных условий, в которых более одной переменной равны 1;
- $I(x_1,...,x_n)$ - запрещена комбинация входных условий, в которых все $x_i=0$;
- $O(x_1,x_2)$ - одно и только одно из значений x_1 и x_2 должно иметь место, т.е. запрещены комбинации ($x_1=1$, $x_2=1$) или ($x_1=0$, $x_2=0$);
- $R(x_1,x_2)$ - запрещена комбинация ($x_1=0$, $x_2=0$).

Возможны комбинации ограничений, например, $E(x_1,...,x_n)$ и $R(x_i,x_j)$, $(i,j)=1,...,n$. В общем случае ограничение может быть задано указанием произвольного множества запрещенных комбинаций команд (входных условий). Учет ограничений может быть осуществлен следующими способами.

Первый заключается в генерации комбинаций, удовлетворяющих ограничениям, второй основан на генерации всех возможных вариантов

доопределения с последующим отбором тех вариантов, которые удовлетворяют системам ограничений. Исследования показали, что в алгоритмическом и программном отношениях первый способ оказывается значительно сложнее второго.

Особенно возрастает сложность программ тестирования в связи с необходимостью генерации тестовых наборов при наличии комбинации ограничений. Кроме того, при этом надо определить, все ли допустимые тестовые наборы построены, а также после генерации очередного тестового набора проверить, не содержится ли он в уже сформированном множестве.

Второй способ лишен этих недостатков. Действительно, для проверки соответствия комбинации некоторому ограничению строится относительно простая подпрограмма. При наличии некоторых ограничений, наложенных на одни и те же входные условия, каждый вариант доопределения комбинации входных условий последовательно и в произвольном порядке проверяется на выполнение каждого из наложенных ограничений, в то время как при генерации допустимого тестового набора эту последовательность приходится учитывать. Отпадает необходимость и проверки наличия очередного тестового набора, удовлетворяющего ограничениям в уже сформированном множестве, так как этот набор в принципе не может совпадать ни с одним из наборов, уже сформированных ранее, потому что эти наборы не генерируются, а отбираются из попарно различных вариантов доопределения комбинаций. Недостатком второго способа является необходимость перебора всех вариантов доопределения комбинаций входных условий. Учитывая, что подпрограммы учета входных условий для типовых участков инструментальных средств автоматизации программирования достаточно просты, то при использовании современных средств вычислительной техники этот недостаток не практике несущественен.

Алгоритм **В** производит вычисление эталонных значений реакций исследуемых инструментальных средств на заданный тестовый набор входных условий осуществляется с использованием операции прямого продвижения вектора P к выходам функциональной диаграммы. В данном алгоритме X_T определяет тестовый набор входных условий, а значения остальных разрядов вектора P равны x. Необходимо заметить, что в векторе P можно было бы определить те значения реакций на входные команды, при которых определялся тестовый набор X_T. Однако, если в векторе P значения этих реакций положить равными x и выполнить

продвижение к выходам, то появляется дополнительная возможность проверки правильности вычисления X_T. Кроме того, комбинация входных условий, доопределенная с учетом ограничений, как правило, порождает комбинации реакций, в которых наряду с определенными значениями, соответствующими набору команд X_T, определены и значения других реакций исследуемого участка программных средств.

Задача, решаемая данным алгоритмом, формируется следующим образом.

Дано:

- вырожденное покрытие функциональной диаграммы инструментальных средств (или их относительно автономного участка), исследуемых на отсутствие блоков формирования программных закладок диверсионного типа;
- вектор $P(0)$ состояния функциональной диаграммы, соответствующий тестовому набору X_T.

Требуется: найти для исследуемых средств значения реакций, соответствующие вектору $P(0)$.

Решение данной задачи осуществляется следующим образом.

Алгоритм В

1. Начальная установка номера шага алгоритма j:=1.
2. В векторе P_{j-1} определяется функциональная связь S_g с минимальным номером, значение которой не определено, а хотя бы одно входное условие этой ФС имеет значение 0 или 1. Так как в соответствии с принятым правилом нумерации номер ФС всегда больше номера любого ее входного условия, то $g \geq V+j$, где V - число входных условий ФС. Поэтому при выполнении этой операции следует анализировать значения ФС с номерами $V+j, V+j+1, ..., V+N-1$, где N - число ФС.
3. Из вырожденного покрытия ФС S_g выбирается куб K_{hg}, значения компонентов которого не противоречат значениям соответствующих компонентов вектора состояния P_{j-1}. Противоречие отсутствует, если для всех компонентов куба K_{hg} выполняются следующие условия:
   3.1. Если в кубе компонент а имеет значение x, то в векторе $P(j-1)$ значение компонента а может быть 0, 1 или x;

3.2. Если в кубе K_{hg} компонент a имеет значение d, d=0,1, то и в векторе $P_{j\text{-}1}$ этот компонент должен иметь значение d.

4. Если не выбрано ни одного куба, то P_j=$P_{j\text{-}1}$ и осуществляется переход к п.6, в противном случае – к п.5.

5. Формируется вектор P_j состояния ФД, разряды которого образуются с помощью операций П поразрядного пересечения (см. п.5 алгоритма Б) компонентов выбранного куба K_{gh} с соответствующими компонентами вектора $P_{j\text{-}1}$ и заменой значений и этих компонентов результатами операции перечисления.

6. Если g<N+V, то осуществляется переход к п.7. В противном случае алгоритм завершается.

7. j:=j+1, переход к п.2.

Таким образом, рассмотренный метод построения тестов проверки технологической безопасности исследуемых инструментальных средств и алгоритмы его реализации позволяют сформировать и проанализировать таблицы решений для полного тестового набора. В то же время, создание самих тестов представляет собой самостоятельную задачу, связанную с необходимостью разработки некоторого программного обеспечения, которое само по себе должно соответствовать логическим схемам проведения проверок.

ГЛАВА 10. МЕТОДЫ ИДЕНТИФИКАЦИИ ПРОГРАММ И ИХ ХАРАКТЕРИСТИК

10.1. Идентификация программ по внутренним характеристикам

Обеспечение безопасности программ, когда их исходные тексты попадают в руки злоумышленников, которые стремятся привнести в код программы РПС до того как программы подвергнутся компиляции может заключаться в использовании методов идентификации программ и их характеристик.

При установления степени подобия исходной и исследуемой программы целесообразнее всего выбрать критерий, который насколько это возможно, не зависит от маскировок, вносимых в исходный текст программ нарушителем. Для этого необходимо выбрать параметры, характеризующие собственно программу и связанные с такими ее свойствами, которые трудно изменить и которые сохраняются в машинном коде программы. К таким параметрам может относиться, например, распределение операторов по тексту программы [ЗПО], которое сложно изменить нарушителю, не искажая назначения программы. Такие изменения требуют глубокого понимания текста программы и логики вносимых изменений, что сопряжено с огромной работой по преобразованию программы.

Сначала рассмотрим вопросы анализа подобия последовательностей операторов в программе, поскольку этот подход не чувствителен к поверхностной маскировке, которую мог бы попытаться внести нарушитель, изменяя некоторые атрибуты программы, например, имена переменных, нумерацию строк и т.п. Для этого необходимо написать программу - анализатор, которая будет тестировать исследуемую программу и выделять операторы, накапливая их в файле как данные, отражающие порядок их использования. Введем для последовательности операторов программы с номером n обозначение seq n. Тогда последовательность операторов для программ 1 и 2 будет обозначаться seq1 и seq2 соответственно. Одна из характеристик последовательности операторов - частота появления отдельного оператора. Анализ последовательности операторов оказывается эффективным в тех случаях, когда нарушитель изменяет или перемещает отдельные части программы, добавляет дополнительные операторы или погружает скопированную

программу в некоторый модуль. При таких манипуляциях значительные участки последовательности операторов сохраняются неизменными, так как попытка изменить их равносильна переписыванию программы с сопутствующей трудоемкой операцией отладки. Рассмотрим распределение частот появления операторов в программе. Если программа скопирована целиком, но при этом замаскирована, число появлений каждого оператора в копии будет аналогично числу появлений в оригинале. Нарушитель может изменить некоторые операторы и добавить новые, но в целом процент изменений в программе, вероятно, будет мал, и распределение частот появления операторов ожидается одинаковым как для копии, так и для оригинала.

В то же время, если программа (или отдельный программный модуль) включена в большую программу необходимо рассматривать другую характеристику, связанную с сохранением структуры последовательности операторов и определяемую некоторой функцией. Такое подобие структур может быть выражено как максимум взаимной корреляцией функций двух программ, положение которого зависит от размещения модуля в программе. Интересен вопрос, будет ли заимствованная программа, откомпилированная в машинный код, обеспечивать достаточное значение корреляционной функции, чтобы выделить модуль, включенный в состав программы, а также будет ли взаимная корреляционная функция машинного кода соответствовать взаимной корреляционной функции исходной программы на языке высокого уровня.

Другая характеристика программы - автокорреляционная функция, определяющая меру соответствия, с которой одни и те же последовательности операторов повторяются в самой программе. По всей видимости, корреляционная функция должна быть чувствительна к добавлению, удалению или перемещению операторов, чем гистограмма частот появления операторов в программе, поскольку при сокращении последовательности значение корреляционной функции может существенно уменьшаться.

10.2. Способы оценки подобия целевой и исследуемой программ с точки зрения наличия программных дефектов

Частоту появления операторов в программе можно изобразить в виде гистограммы. Для этого достаточно написать подпрограмму, которая будет подсчитывать каждое появление операторов в последовательности зафиксированных операторов программы. На рис.10.1 приведена

гистограмма операторов для информационно - поисковой системы (ИПС) [ЗПО]; на абсциссе графика отмечено количество возможных операторов интерпретатора языка, используемого в этих текстах. Было выявлено [ЗПО], что некоторые операторы очень часто встречаются в программах различного назначения, некоторые редко, а некоторые вообще не встречаются. Для сравнения на рис.10.2 приведена гистограмма для программы редактирования текстов (РТ), которая отличается от гистограммы на рис.10.1.

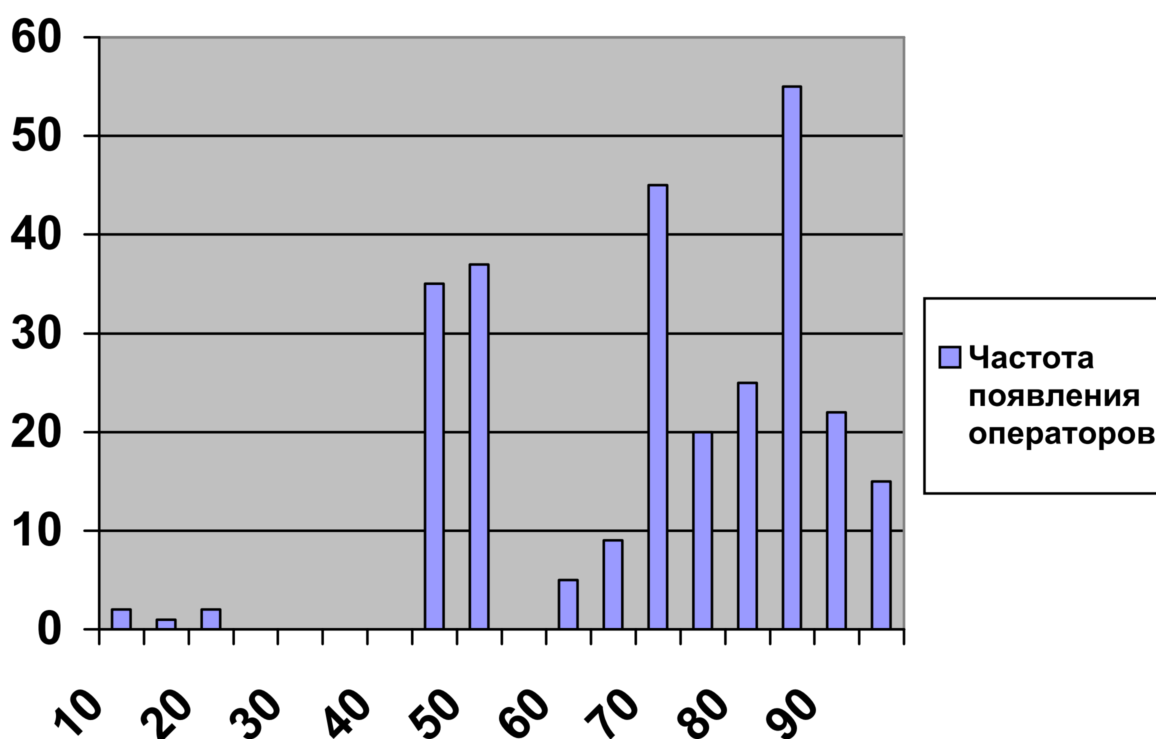


Рис. 10.1. Гистограмма частоты появления операторов в программе для поисково-информационной системы

Повторяемость некоторых операторов делает степень подобия программ визуально видимой, особенно для программ с одинаковым функциональным назначением, поскольку целевая направленность часто определяет и выбор операторов. Глаз плохо различает относительные значения амплитуд на различных гистограммах, поэтому удобнее изображать частоту появления операторов в одной программе в зависимости от частоты появления в другой. В этом случае для программ

одного вида подобия точки будут размещаться на биссектрисе первого квадранта под углом 45°. Если программы существенно различаются по частоте вхождения операторов в последовательности операторов, тогда точки будут иметь существенный разброс.

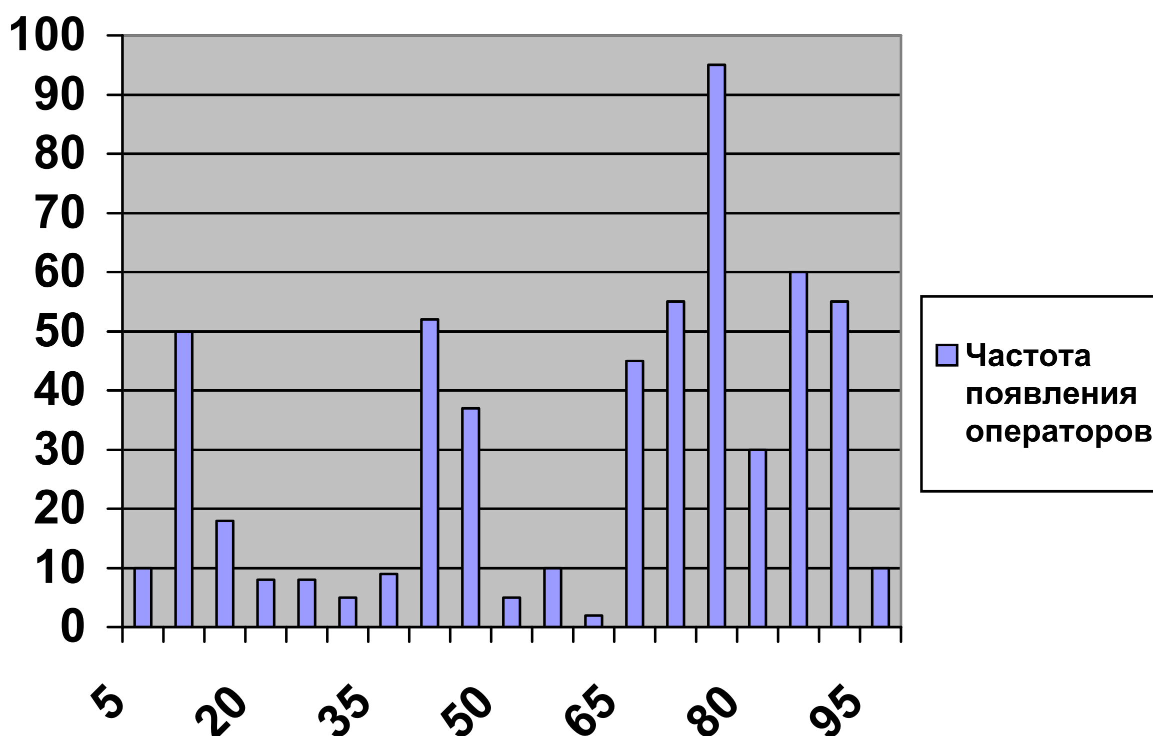


Рис. 10.2. Гистограмма частоты появления операторов в программе для редакторов текстов

Визуальное восприятие можно выразить математически, используя понятие корреляции. Простую меру оценки подобия можно получить, подсчитывая для каждого оператора с номером n среднее значение частоты его появления D_n в каждой программе. Математически эта мера подобия A_n для одного оператора записывается в виде

$$A_n = \frac{S_n}{2} - D_n = \frac{f_n^{(1)} + f_n^{(2)}}{2} - \left| f_n^{(1)} - f_n^{(2)} \right|, \qquad (10.1)$$

где $f_n^{(1)}$, $f_n^{(2)}$ - частоты появления оператора с номером n в программах 1 и 2 соответственно; S_n средняя сумма частот появления операторов с номером n в обеих программах; D_n - разность между частотами появления оператора с номером n в программах 1 и 2.

Мера для всей последовательности операторов получается путем суммирования по всем операторам и нормировки (чтобы снять зависимость от длины программы, сумму делят на полное число операторов в обеих программах). Такая мера подобия $A(1,2)$ для программ 1 и 2 имеет вид:

$$A(1,2)=\frac{\sum_{n=1}^{N} S_n-2D_n}{\sum_{n=1}^{N} S_n}, \qquad (10.2)$$

где N - число операторов в языке.

Если частота появления операторов в обеих программах одинакова, мера подобия $A(1,2)$=1, если операторы, присутствующие в программах, образуют пересекающие множества A(1,2)=-1. Для рассмотренных последовательностей операторов мера подобия равна A(ИПС,РТ)=-0,8. Статистическая формула для корреляции имеет вид:

$$C(1,2)=\frac{\sum_{n=1}^{N}(f_n^{(1)}-\overline{f^{(1)}})(f_n^{(2)}-\overline{f^{(2)}})}{\left[\sum_{n=1}^{N}(f_n^{(1)}-\overline{f^{(1)}})^2\cdot\sum_{n=1}^{N}(f_n^{(2)}-\overline{f^{(2)}})^2\right]^{1/2}},$$

где $\overline{f^{(1)}}$, $\overline{f^{(2)}}$ - средние значения частот появления всех операторов в программах 1 и 2 соответственно.

Значения коэффициентов корреляции, вычисленных по формуле (10.2) всегда находятся в пределах от 0 до 1. Если программы имеют почти один и тот же вид подобия, то коэффициент корреляции близок к 1, в случае вычисления коэффициента корреляции для программ ИПС и РТ, коэффициент корреляции равен C(ИПС,РТ)=0,56.

Вклад в значение коэффициента корреляции частоты появления операторов зависит от популярности применения некоторых операторов в программах разного типа. Этот вклад приводит к большему значению коэффициента корреляции по сравнению с мерой подобия. Это означает, что пороговое значение коэффициента корреляции при оценке подобия программ должно быть увеличено или, наоборот, соответствующие измерения должны быть скорректированы на величину, учитывающую степень подобия программ.

Распределение частот появления операторов наиболее полезно при сравнении двух программ, поскольку не зависит от порядка следования программ. Однако оно мало пригодно, когда программа встроена в программный продукт в сочетании с другими программными модулями,

частотный спектр операторов которых может его поглотить. Для выявления присутствия модулей в большой программе более удобны коэффициенты взаимной корреляции.

Взаимная корреляция может быть использована для оценки взаимосвязи операторов в различных точках программы. Сравнивая последовательности операторов двух программ, можно достаточно просто проверить взаимную корреляцию операторов по их местоположению в этих последовательностях. Каждый раз, когда в последовательностях встречаются одинаковые операторы, это событие фиксируется и их общее число суммируется. В том случае, когда одна программа короче другой, более короткая продлевается циклическим повтором. Если обе программы идентичны, отношение числа зафиксированных операторов к общему числу операторов в программе равно 1.

Прямая корреляция между элементами множества не является оптимальной мерой, поскольку фоновая корреляция, обусловленная случайностью, может оказаться очень высокой, и поэтому необходимо переходить от анализа отдельных элементов к анализу групп элементов. Улучшенную корреляцию можно получить, если рассматривать группу элементов. Поэтому при поиске в некоторой последовательности операторов совпадающих элементов следует проверять, является следующий элемент совпадающим, и если это так, то совпадение фиксируется и этот процесс продолжается до окончания сравниваемой последовательности. Если последовательность имеет длину n, объем выборки, отнесенный к первому элементу, увеличивается с 1 до n.

Расчет корреляции (которая в данном случае называется взвешенной взаимной корреляцией) продолжается путем перехода к следующему (второму) элементу последовательности. В этом случае соответствующая выборка увеличивается с 2 до n-1.

Затруднения, связанные с использованием метода простой корреляции для последовательностей машинных команд, состоят в определении длины последовательности, поскольку длина должна быть различна для разных операторов высокого уровня. Метод взвешенной корреляции, когда устанавливается высокий порог повторяемости, решает эту проблему, поскольку, если и теперь отмечаются совпадающие последовательности, то весьма вероятно, что последовательность действительно выявляет совпадающие операторы языка высокого уровня, а случайные совпадения, имеющие корреляцию ниже установленного порога, во внимание не принимаются. Выше мы предполагали, что программы обработаны одним

и тем же компилятором. В том случае, когда компилятор не известен, возможно, следует провести тестирование с различными компиляторами, пока корреляция не будет выявлена.

Для анализа корреляции внутри самой программы вводится автокорреляционная функция. Для этого необходимо воспользоваться подпрограммами для определения взаимных корреляций. Если в качестве двух тестовых использовать одну и ту же последовательность, то автокорреляция представляет значительный интерес, поскольку дает некоторую числовую характеристику программы. По всей вероятности автокорреляционные функции различного типа можно использовать и при тестировании программ на технологическую безопасность, когда разработанную программу еще не с чем сравнивать на подобие с целью обнаружения программных дефектов.

Таким образом, программы имеют целую иерархию структур, которые могут быть выявлены, измерены и использованы в качестве характеристик последовательности данных. При этом в ходе тестирования, измерения не должны зависеть от типа данных, хотя данные, имеющие структуру программы, должны обладать специфическими параметрами, позволяющими указать меру распознавания программы. Поэтому указанные методы позволяют в определенной мере выявить те изменения в программе, которые вносятся нарушителем либо в результате преднамеренной маскировки, либо преобразованием некоторых функций программы, либо включением модуля, характеристики которого отличаются от характеристик программы, а также позволяют оценить степень обеспечения безопасности программ при внесении программных закладок.

ГЛАВА 11. МЕТОДЫ И СРЕДСТВА ЗАЩИТЫ ПРОГРАММ ОТ КОМПЬЮТЕРНЫХ ВИРУСОВ

11.1. Общая характеристика и классификация компьютерных вирусов

Под *компьютерным вирусом* (или просто *вирусом*) понимается автономно функционирующая программа, обладающая способностью к самостоятельному внедрению в тела других программ и последующему самовоспроизведению и самораспространению в информационно-вычислительных сетях и отдельных ЭВМ [Без,Бел,ЗМС,ФДК,Со]. Предшественниками вирусов принято считать так называемые *троянские программы*, тела которых содержат скрытые последовательности команд (модули), выполняющие действия, наносящие вред пользователям. Наиболее распространенной разновидностью троянских программ являются широко известные программы массового применения (редакторы, игры, трансляторы и т.д.), в которые встроены так называемые «логические бомбы», срабатывающие по наступлении некоторого события. Следует отметить, что троянские программы не являются саморазмножающимися.

Принципиальное отличие вируса от троянской программы состоит в том, что вирус после его активизации существует самостоятельно (автономно) и в процессе своего функционирования заражает (инфицирует) программы путем включения (имплантации) в них своего текста. Таким образом, компьютерный вирус можно рассматривать как своеобразный «генератор троянских программ». Программы, зараженные вирусом, называются *вирусоносителями*.

Заражение программы, как правило, выполняется таким образом, чтобы вирус получил управление раньше самой программы. Для этого он либо встраивается в начало программы, либо имплантируется в ее тело так, что первой командой зараженной программы является безусловный переход на компьютерный вирус, текст которого заканчивается аналогичной командой безусловного перехода на команду вирусоносителя, бывшую первой до заражения. Получив управление, вирус выбирает следующий файл, заражает его, возможно, выполняет какие-либо другие действия, после чего отдает управление вирусоносителю.

«Первичное» заражение происходит в процессе поступления инфицированных программ из памяти одной машины в память другой, причем в качестве средства перемещения этих программ могут

использоваться как носители информации (дискеты, CD-ROM, CD-RW, флэш-память и т.п.), так и каналы вычислительных сетей. Вирусы, использующие для размножения сетевые средства, принято называть *сетевыми*.

Цикл жизни вируса обычно включает следующие периоды: внедрение, инкубационный, репликации (саморазмножения) и проявления. В течение инкубационного периода вирус пассивен, что усложняет задачу его поиска и нейтрализации. На этапе проявления вирус выполняет свойственные ему целевые функции, например необратимую коррекцию информации в компьютере или на магнитных носителях.

Физическая структура компьютерного вируса достаточно проста. Он состоит из головы и, возможно, хвоста. Под головой вируса понимается его компонента, получающая управление первой. Хвост - это часть вируса, расположенная в тексте зараженной программы отдельно от головы. Вирусы, состоящие из одной головы, называют *несегментированными*, тогда как вирусы, содержащие голову и хвост - *сегментированными*.

Наиболее существенные признаки компьютерных вирусов позволяют провести следующую их классификацию.

I. *По режиму функционирования:*

- *резидентные вирусы* - вирусы, которые после активизации постоянно находятся в оперативной памяти компьютера и контролируют доступ к его ресурсам;
- *транзитные вирусы* - вирусы, которые выполняются только в момент запуска зараженной программы.

II. *По объекту внедрения:*

- *файловые вирусы* - вирусы, заражающие файлы с программами;
- *загрузочные (бутовые) вирусы* - вирусы, заражающие программы, хранящиеся в системных областях дисков.

В свою очередь файловые вирусы подразделяются на вирусы, заражающие:

- исполняемые файлы;
- командные файлы и файлы конфигурации;
- составляемые на макроязыках программирования, или файлы, содержащие макросы (*макровирусы*);
- файлы с драйверами устройств;

- файлы с библиотеками исходных, объектных, загрузочных и оверлейных модулей, библиотеками динамической компоновки и т.п.

Загрузочные вирусы подразделяются на вирусы, заражающие:

- системный загрузчик, расположенный в загрузочном секторе дискет и логических дисков;
- внесистемный загрузчик, расположенный в загрузочном секторе жестких дисков.

III. *По степени и способу маскировки:*

- вирусы, не использующие средств маскировки;
- *stealth-вирусы* - вирусы, пытающиеся быть невидимыми на основе контроля доступа к зараженным элементам данных;
- *вирусы-мутанты (MtE-вирусы)* - вирусы, содержащие в себе алгоритмы шифрования, обеспечивающие различие разных копий вируса.

В свою очередь, MtE-вирусы делятся на

- обычные вирусы-мутанты, в разных копиях которых различаются только зашифрованные тела, а дешифрованные тела вирусов совпадают;
- полиморфные вирусы, в разных копиях которых различаются не только зашифрованные тела, но и их дешифрованные тела.

Наиболее распространенные типы вирусов характеризуются следующими основными особенностями.

Файловый транзитный вирус целиком размещается в исполняемом файле, в связи с чем он активизируется только в случае активизации вирусоносителя, а по выполнении необходимых действий возвращает управление самой программе. При этом выбор очередного файла для заражения осуществляется вирусом посредством поиска по каталогу. *Файловый резидентный вирус* отличается от нерезидентного логической структурой и общим алгоритмом функционирования. Резидентный вирус состоит из так называемого инсталлятора и программ обработки прерываний. Инсталлятор получает управление при активизации вирусоносителя и инфицирует оперативную память путем размещения в ней управляющей части вируса и замены адресов в элементах вектора прерываний на адреса своих программ, обрабатывающих эти прерывания. На так называемой фазе слежения, следующей за описанной фазой инсталляции, при возникновении какого-либо прерывания управление

получает соответствующая подпрограмма вируса. В связи с существенно более универсальной по сравнению с нерезидентными вирусами общей схемой функционирования, резидентные вирусы могут реализовывать самые разные способы инфицирования.

Наиболее распространенными способами являются инфицирование запускаемых программ, а также файлов при их открытии или чтении. Отличительной особенностью последних является инфицирование загрузочного сектора (бут-сектора) магнитного носителя. Голова *бутового вируса* всегда находится в бут-секторе (единственном для гибких дисков и одном из двух - для жестких), а хвост - в любой другой области носителя. Наиболее безопасным для вируса способом считается размещение хвоста в так называемых псевдосбойных кластерах, логически исключенных из числа доступных для использования. Существенно, что хвост бутового вируса всегда содержит копию оригинального (исходного) бут-сектора. Механизм инфицирования, реализуемый бутовыми вирусами, например, при загрузке MS DOS, таков. При загрузке операционной системы с инфицированного диска вирус, в силу своего положения на нем (независимо от того, с дискеты или с винчестера производится загрузка), получает управление и копирует себя в оперативную память. Затем он модифицирует вектор прерываний таким образом, чтобы прерывание по обращению к диску обрабатывались собственным обработчиком прерываний вируса, и запускает загрузчик операционной системы. Благодаря перехвату прерываний бутовые вирусы могут реализовывать столь же широкий набор способов инфицирования и целевых функций, сколь и файловые резидентные вирусы.

Stealth-вирусы пользуются слабой защищенностью некоторых операционных систем и заменяют некоторые их компоненты (драйверы дисков, прерывания) таким образом, что вирус становится невидимым (прозрачным) для других программ. Для этого заменяются функции DOS таким образом, что для зараженного файла подставляются его оригинальная копия и содержание, каким они были до заражения.

Полиморфные вирусы содержат алгоритм порождения дешифрованных тел вирусов, непохожих друг на друга. При этом в алгоритмах дешифрования могут встречаться обращения практически ко всем командам процессора Intel и даже использоваться некоторые специфические особенности его реального режима функционирования.

Макровирусы распространяются под управлением прикладных программ, что делает их независимыми от операционной системы.

Подавляющее число макровирусов функционируют под управлением системы Microsoft Word for Windows. В то же время, известны макровирусы, работающие под управлением таких приложений как Microsoft Word for Windows, Microsoft Exel for Windows, Lotus Ami Pro, Lotus 1-2-3, Lotus Notes, в операционных системах фирм Microsoft и Apple [Бел].

Сетевые вирусы, называемые также *автономными репликативными программами*, или, для краткости, *репликаторами*, используют для размножения средства сетевых операционных систем. Наиболее просто реализуется размножение в тех случаях, когда сетевыми протоколами предусмотрен обмен программами. Однако, размножение возможно и в тех случаях, когда указанные протоколы ориентированы только на обмен сообщениями. Классическим примером реализации процесса размножения с использованием только стандартных средств электронной почты является уже упоминаемый репликатор Морриса [ФДК]. Текст репликатора передается от одной ЭВМ к другой как обычное сообщение, постепенно заполняющее буфер, выделенный в оперативной памяти ЭВМ-адресата. В результате переполнения буфера, инициированного передачей, адрес возврата в программу, вызвавшую программу приема сообщения, замещается на адрес самого буфера, где к моменту возврата уже находится текст вируса.

Тем самым вирус получает управление и начинает функционировать на ЭВМ-адресате.

«Лазейки», подобные описанной выше и обусловленные особенностями реализации тех или иных функций в программном обеспечении, являются объективной предпосылкой для создания и внедрения репликаторов злоумышленниками. Эффекты, вызываемые вирусами в процессе реализации ими целевых функций, принято делить на следующие группы:

- искажение информации в файлах, либо в таблице размещения файлов (FAT-таблице), которое может привести к разрушению файловой системы в целом;
- имитация сбоев аппаратных средств;
- создание звуковых и визуальных эффектов, включая, например, отображение сообщений, вводящих оператора в заблуждение или затрудняющих его работу;

- инициирование ошибок в программах пользователей или операционной системе.

Теоретически возможно создание «*вирусных червей*» - разрушающих программ, которые незаметно перемещаются между узлами вычислительной сети, не нанося никакого вреда до тех пор, пока не доберутся до целевого узла. В нем программа размещается и перестает размножаться.

Поскольку в будущем следует ожидать появления все более и более скрытых форм компьютерных вирусов, уничтожение очагов инфекции в локальных и глобальных сетях не станет проще. Время компьютерных вирусов «общего назначения» уходит в прошлое.

11.2. Общая характеристика средств нейтрализации компьютерных вирусов

Наиболее распространенным средством нейтрализации компьютерных вирусов являются *антивирусные программы (антивирусы).* Антивирусы, исходя из реализованного в них подхода к выявлению и нейтрализации вирусов, принято делить на следующие группы:

- детекторы;
- фаги;
- вакцины;
- прививки;
- ревизоры;
- мониторы.

Детекторы обеспечивают выявление вирусов посредством просмотра исполняемых файлов и поиска так называемых сигнатур - устойчивых последовательностей байтов, имеющихся в телах известных вирусов. Наличие сигнатуры в каком-либо файле свидетельствует о его заражении соответствующим вирусом. Антивирус, обеспечивающий возможность поиска различных сигнатур, называют *полидетектором*.

Фаги выполняют функции, свойственные детекторам, но, кроме того, «излечивают» инфицированные программы посредством «выкусывания» вирусов из их тел. По аналогии с полидетекторами, фаги, ориентированные на нейтрализацию различных вирусов, именуют *полифагами*.

В отличие от детекторов и фагов, *вакцины* по своему принципу действия подобны вирусам. Вакцина имплантируется в защищаемую

программу и запоминает ряд количественных и структурных характеристик последней. Если вакцинированная программа не была к моменту вакцинации инфицированной, то при первом же после заражения запуске произойдет следующее. Активизация вирусоносителя приведет к получению управления вирусом, который, выполнив свои целевые функции, передаст управление вакцинированной программе. В последней, в свою очередь, сначала управление получит вакцина, которая выполнит проверку соответствия запомненных ею характеристик аналогичным характеристикам, полученным в текущий момент. Если указанные наборы характеристик не совпадают, то делается вывод об изменении текста вакцинированной программы вирусом. Характеристиками, используемыми вакцинами, могут быть длина программы, ее контрольная сумма и т.д.

Принцип действия *прививок* основан на учете того обстоятельства, что любой вирус, как правило, помечает инфицируемые программы каким-либо признаком с тем, чтобы не выполнять их повторное заражение. В ином случае имело бы место многократное инфицирование, сопровождаемое существенным и поэтому легко обнаруживаемым увеличением объема зараженных программ. Прививка, не внося никаких других изменений в текст защищаемой программы, помечает ее тем же признаком, что и вирус, который, таким образом, после активизации и проверки наличия указанного признака, считает ее инфицированной и «оставляет в покое».

Ревизоры обеспечивают слежение за состоянием файловой системы, используя для этого подход, аналогичный реализованному в вакцинах. Программа-ревизор в процессе своего функционирования выполняет применительно к каждому исполняемому файлу сравнение его текущих характеристик с аналогичными характеристиками, полученными в ходе предшествующего просмотра файлов. Если при этом обнаруживается, что, согласно имеющейся системной информации, файл с момента предшествующего просмотра не обновлялся пользователем, а сравниваемые наборы характеристик не совпадают, то файл считается инфицированным. Характеристики исполняемых файлов, получаемые в ходе очередного просмотра, запоминаются в отдельном файле (файлах), в связи с чем, увеличение длин исполняемых файлов, имеющего место при вакцинации, в данном случае не происходит. Другое отличие ревизоров от вакцин состоит в том, что каждый просмотр исполняемых файлов ревизором требует его повторного запуска.

Монитор представляет собой резидентную программу, обеспечивающую перехват потенциально опасных прерываний, характерных для вирусов, и запрашивающую у пользователей подтверждение на выполнение операций, следующих за прерыванием. В случае запрета или отсутствия подтверждения монитор блокирует выполнение пользовательской программы.

Антивирусы рассмотренных типов существенно повышают вирусозащищенность отдельных ПЭВМ и вычислительных сетей в целом, однако, в связи со свойственными им ограничениями, естественно, не являются панацеей. В работе [СМ] приведены основные недостатки при использовании антивирусов.

В связи с этим необходима реализация альтернативных подходов к нейтрализации вирусов: создание операционных систем, обладающих высокой вирусозащищенностью по сравнению с наиболее «вирусодружественной» MS DOS, MS Windows, разработка аппаратных средств защиты от вирусов и соблюдение технологии защиты от вирусов.

11.3. Классификация методов защиты от компьютерных вирусов

Проблему защиты от вирусов необходимо рассматривать в общем контексте проблемы защиты информации от несанкционированного доступа и технологической и эксплуатационной безопасности ПО в целом. Основной принцип, который должен быть положен в основу разработки технологии защиты от вирусов, состоит в создании многоуровневой распределенной системы защиты, включающей:

- регламентацию проведения работ на ПЭВМ;
- применение программных средств защиты;
- использование специальных аппаратных средств защиты.

При этом количество уровней защиты зависит от ценности информации, которая обрабатывается на ПЭВМ.

Для защиты от компьютерных вирусов в настоящее время используются следующие методы.

Архивирование. Заключается в копировании системных областей магнитных дисков и ежедневном ведении архивов измененных файлов. Архивирование является одним из основных методов защиты от вирусов. Остальные методы защиты дополняют его, но не могут заменить полностью.

Входной контроль. Проверка всех поступающих программ детекторами, а также проверка длин и контрольных сумм вновь поступающих программ на соответствие значениям, указанным в документации. Большинство известных файловых и бутовых вирусов можно выявить на этапе входного контроля. Для этой цели используется *батарея детекторов* (несколько последовательно запускаемых программ). Набор детекторов достаточно широк, и постоянно пополняется по мере появления новых вирусов. Однако при этом могут быть обнаружены не все вирусы, а только распознаваемые детектором. Следующим элементом входного контроля является контекстный поиск в файлах слов и сообщений, которые могут принадлежать вирусу (например, Virus, COMMAND.COM, Kill и т.д.). Подозрительным является отсутствие в последних 2-3 килобайтах файла текстовых строк - это может быть признаком вируса, который шифрует свое тело.

Рассмотренный контроль может быть выполнен с помощью специальной программы, которая работает с базой данных «подозрительных» слов и сообщений, и формирует список файлов для дальнейшего анализа. После проведенного анализа новые программы рекомендуется несколько дней эксплуатировать в карантинном режиме. При этом целесообразно использовать ускорение календаря, т.е. изменять текущую дату при повторных запусках программы. Это позволяет обнаружить вирусы, срабатывающие в определенные дни недели (пятница, 13 число месяца, воскресенье и т.д.).

Профилактика. Для профилактики заражения необходимо организовать раздельное хранение (на разных магнитных носителях) вновь поступающих и ранее эксплуатировавшихся программ, минимизация периодов доступности дискет для записи, разделение общих магнитных носителей между конкретными пользователями.

Ревизия. Анализ вновь полученных программ специальными средствами (детекторами), контроль целостности перед считыванием информации, а также периодический контроль состояния системных файлов.

Карантин. Каждая новая программа проверяется на известные типы вирусов в течение определенного промежутка времени. Для этих целей целесообразно выделить специальную ПЭВМ, на которой не проводятся другие работы. В случае невозможности выделения ПЭВМ для карантина программного обеспечения, для этой цели используется машина,

отключенная от локальной сети и не содержащая особо ценной информации.

Сегментация. Предполагает разбиение магнитного диска на ряд логических томов (разделов), часть из которых имеет статус READ_ONLY (только чтение). В данных разделах хранятся выполняемые программы и системные файлы. Базы данных должны храниться в других секторах, отдельно от выполняемых программ. Важным профилактическим средством в борьбе с файловыми вирусами является исключение значительной части загрузочных модулей из сферы их досягаемости. Этот метод называется сегментацией и основан на разделении магнитного диска (винчестера) с помощью специального драйвера, обеспечивающего присвоение отдельным логическим томам атрибута READ_ONLY (только чтение), а также поддерживающего схемы парольного доступа. При этом в защищенные от записи разделы диска помещаются исполняемые программы и системные утилиты, а также системы управления базами данных и трансляторы, т.е. компоненты ПО, наиболее подверженные опасности заражения. В качестве такого драйвера целесообразно использовать программы типа ADVANCED DISK MANAGER (программа для форматирования и подготовки жесткого диска), которая не только позволяет разбить диск на разделы, но и организовать доступ к ним с помощью паролей. Количество используемых логических томов и их размеры зависят от решаемых задач и объема винчестера. Рекомендуется использовать 3 - 4 логических тома, причем на системном диске, с которого выполняется загрузка, следует оставить минимальное количество файлов (системные файлы, командный процессор, а также программы - ловушки).

Фильтрация. Заключается в использовании программ - сторожей, для обнаружения попыток выполнить несанкционированные действия.

Вакцинация. Специальная обработка файлов и дисков, имитирующая сочетание условий, которые используются некоторым типом вируса для определения, заражена уже программа или нет.

Автоконтроль целостности. Заключается в использовании специальных алгоритмов, позволяющих после запуска программы определить, были ли внесены изменения в ее файл.

Терапия. Предполагает дезактивацию конкретного вируса в зараженных программах специальными программами (фагами). Программы-фаги «выкусывают» вирус из зараженной программы и пытаются восстановить ее код в исходное состояние (состояние до

момента заражения). В общем случае технологическая схема защиты может состоять из следующих этапов:

- входной контроль новых программ;
- сегментация информации на магнитном диске;
- защита операционной системы от заражения;
- систематический контроль целостности информации.

Необходимо отметить, что не следует стремиться обеспечить глобальную защиту всех файлов, имеющихся на диске. Это существенно затрудняет работу, снижает производительность системы и, в конечном счете, ухудшает защиту из-за частой работы в открытом режиме. Анализ показывает, что только 20-30% файлов должно быть защищено от записи.

При защите операционной системы от вирусов необходимо правильное размещение ее и ряда утилит, которое может гарантировать, что после начальной загрузки операционная система еще не заражена резидентным файловым вирусом. Это обеспечивается при размещении командного процессора на защищенном от записи диске, с которого после начальной загрузки выполняется копирование на виртуальный (электронный) диск. В этом случае при вирусной атаке будет заражен дубль командного процессора на виртуальном диске. При повторной загрузке информация на виртуальном диске уничтожается, поэтому распространение вируса через командный процессор становится невозможным.

Кроме того, для защиты операционной системы может применяться нестандартный командный процессор (например, командный процессор 4DOS, разработанный фирмой J.P.Software), который более устойчив к заражению. Размещение рабочей копии командного процессора на виртуальном диске позволяет использовать его в качестве программы-ловушки. Для этого может использоваться специальная программа, которая периодически контролирует целостность командного процессора, и информирует о ее нарушении. Это позволяет организовать раннее обнаружение факта вирусной атаки.

В качестве альтернативы MS DOS было разработано несколько операционных систем, которые являются более устойчивыми к заражению. Из них следует отметить DR DOS и Hi DOS. Любая из этих систем более «вирусоустойчива», чем MS DOS. При этом, чем сложнее и опаснее вирус, тем меньше вероятность, что он будет работать на альтернативной операционной системе.

Анализ рассмотренных методов и средств защиты показывает, что эффективная защита может быть обеспечена при комплексном использовании различных средств в рамках единой операционной среды. Для этого необходимо разработать интегрированный программный комплекс, поддерживающий рассмотренную технологию защиты. В состав программного комплекса должны входить следующие компоненты.

- *Семейство (батарея) детекторов.* Детекторы, включенные в семейство, должны запускаться из операционной среды комплекса. При этом должна быть обеспечена возможность подключения к семейству новых детекторов, а также указание параметров их запуска из диалоговой среды. С помощью данной компоненты может быть организована проверка ПО на этапе входного контроля.
- *Программа-ловушка вирусов.* Данная программа порождается в процессе функционирования комплекса, т.е. не хранится на диске, поэтому оригинал не может быть заражен. В процессе тестирования ПЭВМ программа - ловушка неоднократно выполняется, изменяя при этом текущую дату и время (организует ускоренный календарь). Наряду с этим программа-ловушка при каждом запуске контролирует свою целостность (размер, контрольную сумму, дату и время создания). В случае обнаружения заражения программный комплекс переходит в режим анализа зараженной программы - ловушки и пытается определить тип вируса.
- *Программа для вакцинации.* Предназначена для изменения среды функционирования вирусов таким образом, чтобы они теряли способность к размножению. Известно, что ряд вирусов помечает зараженные файлы для предотвращения повторного заражения. Используя это свойство возможно создание программы, которая обрабатывала бы файлы таким образом, чтобы вирус считал, что они уже заражены.
- *База данных о вирусах и их характеристиках.* Предполагается, что в базе данных будет храниться информация о существующих вирусах, их особенностях и сигнатурах, а также рекомендуемая стратегия лечения. Информация из БД может использоваться при анализе зараженной программы-ловушки, а также на этапе входного контроля ПО. Кроме того, на основе информации, хранящейся в БД, можно выработать рекомендации по

использованию наиболее эффективных детекторов и фагов для лечения от конкретного типа вируса.

- *Резидентные средства защиты.* Эти средства могут резидентно разместиться в памяти и постоянно контролировать целостность системных файлов и командного процессора. Проверка может выполняться по прерываниям от таймера или при выполнении операций чтения и записи в файл.

ГЛАВА 12. МЕТОДЫ ЗАЩИТЫ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ОТ ИССЛЕДОВАНИЯ

12.1. Классификация средств исследования программ

12.1.1. Вводная часть

В этом подразделе мы будем исходить из предположения, что на этапе разработки программная закладка была обнаружена и устранена, либо ее вообще не было. Для привнесения программных закладок, например в этом случае необходимо взять готовый исполняемый модуль, дизассемблировать его и после внесения закладки подвергнуть повторной компиляции. Другой способ заключается в незаконном получении текстов исходных программ, их анализе, внесении программных дефектов и дальнейшей замене оригинальных программ на программы с приобретенными закладками. И, наконец, может осуществляться полная замена прикладной исполняемой программы на исполняемую программу нарушителя, что впрочем, требует от последнего необходимость иметь точные и полные знания целевого назначения и конечных результатов прикладной программы.

Все средства исследования ПО можно разбить на 2 класса: статические и динамические. Первые оперируют исходным кодом программы как данными и строят ее алгоритм без исполнения, вторые же изучают программу, интерпретируя ее в реальной или виртуальной вычислительной среде. Отсюда следует, что первые являются более универсальными в том смысле, что теоретически могут получить алгоритм всей программы, в том числе и тех блоков, которые никогда не получат управления. Динамические средства могут строить алгоритм программы только на основании конкретной ее трассы, полученной при определенных входных данных. Поэтому задача получения полного алгоритма программы в этом случае эквивалентна построению исчерпывающего набора текстов для подтверждения правильности программы, что практически невозможно, и вообще при динамическом исследовании можно говорить только о построении некоторой части алгоритма.

Два наиболее известных типа программ, предназначенных для исследования ПО, как раз и относятся к разным классам: это отладчик (динамическое средство) и дизассемблер (средство статистического исследования). Если первый широко применяется пользователем для отладки собственных программ и задачи построения алгоритма для него

вторичны и реализуются самим пользователем, то второй предназначен исключительно для их решения и формирует на выходе ассемблерный текст алгоритма.

Помимо этих двух основных инструментов исследования, можно использовать:

- «дискомпиляторы», программы, генерирующие из исполняемого кода программу на языке высокого уровня;
- «трассировщики», сначала запоминающие каждую инструкцию, проходящую через процессор, а затем переводящие набор инструкций в форму, удобную для статического исследования, автоматически выделяя циклы, подпрограммы и т.п.;
- «следящие системы», запоминающие и анализирующие трассу уже не инструкции, а других характеристик, например вызванных программой прерываний.

12.1.2. Методы защиты программ от исследования

Для защиты программ от исследования необходимо применять методы защиты от исследования файла с исполняемым кодом программы, хранящемся на внешнем носителе, а также методы защиты исполняемого кода, загружаемого в оперативную память для выполнения этой программы.

В первом случае защита может быть основана на шифровании конфиденциальной части программы, а во втором - на блокировании доступа к исполняемому коду программы в оперативной памяти со стороны отладчиков [ЗШ]. Кроме того, перед завершением работы защищаемой программы должен обнуляться весь ее код в оперативной памяти. Это предотвратит возможность несанкционированного копирования из оперативной памяти дешифрованного исполняемого кода после выполнения защищаемой программы.

Таким образом, защищаемая от исследования программа должна включать следующие компоненты:

- инициализатор;
- зашифрованную конфиденциальную часть;
- деструктор (деинициализатор).

Инициализатор должен обеспечивать выполнение следующих функций:

- сохранение параметров операционной среды функционирования (векторов прерываний, содержимого регистров процессора и т.д.);
- запрет всех внутренних и внешних прерываний, обработка которых не может быть запротоколирована в защищаемой программе;
- загрузка в оперативную память и дешифрование кода конфиденциальной части программы;
- передача управления конфиденциальной части программы.

Конфиденциальная часть программы предназначена для выполнения основных целевых функций программы и защищается шифрованием для предупреждения внесения в нее программной закладки.

Деструктор после выполнения конфиденциальной части программы должен выполнить следующие действия:

- обнуление конфиденциального кода программы в оперативной памяти;
- восстановление параметров операционной системы (векторов прерываний, содержимого регистров процессора и т.д.), которые были установлены до запрета неконтролируемых прерываний;
- выполнение операций, которые невозможно было выполнить при запрете неконтролируемых прерываний;
- освобождение всех незадействованных ресурсов компьютера и завершение работы программы.

Для большей надежности инициализатор может быть частично зашифрован и по мере выполнения может дешифровать сам себя. Дешифроваться по мере выполнения может и конфиденциальная часть программы. Такое дешифрование называется динамическим дешифрованием исполняемого кода. В этом случае очередные участки программ перед непосредственным исполнением расшифровываются, а после исполнения сразу уничтожаются.

Для повышения эффективности защиты программ от исследования необходимо внесение в программу дополнительных функций безопасности, направленных на защиту от трассировки. К таким функциям можно отнести:

- периодический подсчет контрольной суммы области оперативной памяти, занимаемой защищаемым исходным кодом; сравнение текущей контрольной суммы с предварительно сформированной эталонной и принятие необходимых мер в случае несовпадения;

- проверку количества занимаемой защищаемой программой оперативной памяти;
- сравнение с объемом, к которому программа адаптирована, и принятие необходимых мер в случае несоответствия;
- контроль времени выполнения отдельных частей программы;
- блокировку клавиатуры на время отработки особо критичных алгоритмов.

Для защиты программ от исследования с помощью дизассемблеров можно использовать и такой способ, как усложнение структуры самой программы с целью запутывания злоумышленника, который дизассемблирует эту программу. Например, можно использовать разные сегменты адреса для обращения к одной и той же области памяти. В этом случае злоумышленнику будет трудно догадаться, что на самом деле программа работает с одной и той же областью памяти.

12.1.3. Анализ программ на этапе их эксплуатации

В данном разделе будут рассмотрены методы исследования программ нарушителем с помощью дизассемблеров и отладчиков на этапе эксплуатации программ с целью внесения в них РПС. То есть задача защиты в отличие от задач защиты, рассмотренных в предыдущем подразделе, здесь решается «с точностью до наоборот». Этот подраздел необходим для понимания процесса исследования программ, но не специалистами в области защиты ПО (см. главу 7), которые пытаются обнаружить и устранить программные дефекты, а потенциальными нарушителями, которые пытаются внести в исследуемую программу РПС. Т.е. цель, объекты и субъекты обеспечения безопасности ПО здесь разные.

Основная схема анализа нарушителем исполняемого кода, в данном случае, может состоять из следующих этапов [ПБП]:

- выделение чистого кода, то есть удаление кода, отвечающего за защиту этой программы от несанкционированного запуска, копирования и т.п. и преобразования остального кода в стандартный правильно интерпретируемый дизассемблером;
- лексический анализ;
- дизассемблирование;
- семантический анализ;
- перевод в форму, удобную для следующего этапа (в том числе и перевод на язык высокого уровня);

- синтаксический анализ.

После снятия нарушителем защиты осуществляется поиск сигнатур (лексем) РПС. Примеры сигнатур РПС приведены в работе [ПБП]. Окончание этапа дизассемблирования предшествует синтаксическому анализу, то есть процессу отождествлению лексем, найденных во входной цепочке, одной из языковых конструкций, задаваемых грамматикой языка, то есть синтаксический анализ исполняемого кода программ состоит в отождествлении сигнатур, найденных на этапе лексического анализа, одному из видов РПС.

www.kiev-security.org.ua
BEST rus DOC FOR FULL SECURITY

При синтаксическом анализе могут встретиться следующие трудности (хотя - это проблема нарушителя):

- могут быть не распознаны некоторые лексемы. Это следует из того, что макроассемблерные конструкции могут быть представлены бесконечным числом регулярных ассемблерных выражений;
- порядок следования лексем может быть известен с некоторой вероятностью или вообще не известен;
- грамматика языка может пополняться, так как могут возникать новые типы РПС или механизмы их работы.

Таким образом, окончательное заключение возможности внесения РПС можно дать только на этапе семантического анализа, а задачу этого этапа можно конкретизировать как свертку терминальных символов в нетерминалы как можно более высокого уровня там, где входная цепочка задана строго.

Так как семантический анализ удобнее вести на языке высокого уровня далее проводится этап перевода ассемблерного текста в текст на языке более высокого уровня, например, на специализированном языке макроассемблера, который нацелен на выделение макроконструкций, используемых в РПС.

На этапе семантического анализа дается окончательный ответ на вопрос о возможности внесения во входной исполняемый код РПС, и если да, то какого типа. При этом используется вся информация, полученная на всех предыдущих этапах. Однако нарушителю-исследователю необходимо учитывать, что эта информация может считаться правильной лишь с

некоторой вероятностью, причем не исключены вообще ложные факты, или умозаключения исследователя.

12.2. Способы защиты программ от исследования

Способы защиты от исследования можно разделить на четыре класса [ПАС].

1. *Способ, сущность которого заключается в оказании влияния на процесс функционирования отладочного средства через общие программные или аппаратные ресурсы.* В данном случае наиболее известны:

- использование аппаратных особенностей микропроцессора (особенности очередности выборки команд, особенности выполнения команд и т.д.);
- использование общего программного ресурса (например, общего стека) и разрушение данных или кода отладчика, принадлежащих общему ресурсу, либо проверка использования общего ресурса только защищаемой программой (например, определение стека в области, критичной для выполнения защищаемой программы);
- переадресация обработчиков отладочных событий (прерываний) от отладочного средства к защищаемой программе.

Выделение трех групп защитных действий в данном классе не случайно, поскольку объективно существуют общие аппаратные ресурсы отладчика, и защищаемая программа в случае однопроцессорного вычислителя выполняются на одном и том же процессоре), общие программные ресурсы (поскольку и отладчик, и защищаемая программа выполняются в одной и той же операционной среде), наконец, отладчик создает специфичные ресурсы, существенные для его собственной работы (например адресует себе отладочные прерывания).

2. *Влияние на работу отладочного средства путем использован~ особенностей его аппаратной или программной среды*. Например:

- перемещения фрагментов кода или данных с помощью контроллер прямого доступа к памяти;
- влияния на процесс регенерации оперативной памяти (на некотором участке кода регенерация памяти отключается, а затем опять включается, - при нормальной работе никаких изменений нет, при медленном выполнении программы отладчиком она «зависает»);
- перехода микропроцессора в защищенный режим.

3. *Влияние на работу отладчика через органы управления или/и устройства отображения информации*.

Выдаваемая отладочными средствами информация анализируется человеком. Следовательно, дополнительный способ защиты от отладки это нарушение процесса общения оператора и отладчика, а именно искажение или блокирование вводимой с клавиатуры и выводимой на терминал информации.

4. *Использование принципиальных особенностей работы управляемого человеком отладчика*. В данном случае защита от исследования состоит в навязывании для анализа избыточно большого объема кода (как правило, за счет циклического исполнения некоторого его участка).

Рассмотрим данный метод подробнее. Пусть имеется некоторое полноцикловое преобразование из N состояний t: $T=t_1,t_2,...,t_N$ (Например, обычный двоичный счетчик либо, рекуррента). Значение N выбирается не слишком большим. Например, для k-битового счетчика $N=2^k$. Участок кода, защищаемый от изучения, динамически преобразуется (шифруется) с использованием криптографически стойкого алгоритма на ключе t, который выбирается случайно и равновероятно из множества состояний T.

Работа механизма защиты от исследования выглядит следующим образом. Программа полноциклового преобразования начинает работу с детерминированного или случайного значения. На установленном значении производится дешифрование зашифрованного участка кода. Правильность дешифрования проверяется подсчетом значения хэш-кода расшифрованного участка программного кода с использованием элементов, связанных с отладкой (стек, отладочные прерывания и др.). Хэш-функция должна с вероятностью 1 определять правильность дешифрования (для этого значение хэш-кода должно быть не менее k).

Предположим, что полноцикловое преобразование стартует с первого значения. Тогда при нормальном выполнении программы (скорость работы высокая) будет совершено i циклов алгоритма, после чего защищенный участок будет корректно исполнен. При работе отладчика, управляемого человеком, скорость выполнения программы на несколько порядков ниже, поэтому для достижения необходимого значения 1 будет затрачено значительное время.

Для численной оценки данного метода введем следующие значения. Предположим, что i в среднем равно $N/2$. Пусть w_0 - время выполнения цикла алгоритма (установка текущего значения, дешифрование, проверка

правильности дешифрования) в штатном режиме функционирования (без отладки); w_1 - время выполнения того же цикла в режиме отладки; z — предельное время задержки при штатной работе защищенной программы. Тогда $N=z/w_0$. Затраты времени злоумышленника исчисляются средней величиной $T_{зл}=Nw_1/2$. Для приблизительных расчетов $w_1/w_0 \approx 10000$.

В ряде способов защиты от отладки идентификация отладчика и направление его по ложному пути происходят одновременно, в одном и том же фрагменте кода (так, при определении стека в области кода защищаемой программы при работе отладчика, использующего тот же стек, код программы будет разрушен). В других случаях ложный путь в работе программы формируется искусственно. Часто для этого используют динамическое преобразование программы (шифрование) во время ее исполнения.

Способ динамического преобразования заключается в следующем: первоначально в оперативную память загружается фрагмент кода, содержание части команд которого не соответствует тем командам, которые данный фрагмент в действительности выполняет; затем этот фрагмент по некоторому закону преобразуется, превращаясь в исполняемые команды, которые затем и выполняются.

Преобразование кода программы во время ее выполнения может преследовать три основные цели:

- противодействие файловому дизассемблированию программы;
- противодействие работе отладчику;
- противодействие считыванию кода программы в файл из оперативной памяти.

Перечислим основные способы организации преобразования кода программы [ПАС]:

1. Замещение фрагмента кода функцией от находящейся на данном месте команды и некоторых данных.
2. Определение стека в области кода и перемещение фрагментов кода с использованием стековых команд.
3. Преобразование кода в зависимости от содержания предыдущего фрагмента кода или некоторых условий, полученных при работе предыдущего фрагмента.
4. Преобразование кода в зависимости от (внешней к программе) информации.

5. Преобразование кода, совмещенное с действиями, характерными для работы отладочных средств.

Первый способ заключается в том, что по некоторому адресу в коде программы располагается, например, побитовая разность между реальными командами программы и некоторой хаотической информацией, которая располагается в области данных. Непосредственно перед выполнением данного участка программы происходит суммирование хаотической информации с содержанием области кода и в ней образуются реальные команды.

Второй способ состоит в перемещении фрагментов кода программы в определенное место или наложении их на уже выполненные команды при помощи стековых операций.

Третий способ служит для защиты от модификаций кода программы и определения точек останова в программе. Он состоит в том, что преобразование следующего фрагмента кода происходит на основе функции, существенно зависящей от каждого байта или слова предыдущего фрагмента или нескольких фрагментов кода программы. Такую функцию называют обычно контрольной суммой участка кода программы. Особенности данного способа является то, что процесс преобразования должен соответственно зависеть от посчитанной контрольной суммы (подсчитанного значения хэш-кода) и должен содержать в явном виде операций сравнения.

Четвертый способ заключается в преобразовании кода программы на основе некоторой внешней информации, например считанной с ключевой дискеты некопируемой метки, машинно-зависимой информации или ключа пользователя. Это позволит исключить анализ программы, не имеющего ключевого носителя или размещенной на другом компьютере, где машино-зависимая информация иная.

Пятый способ состоит в том, что вместо адресов отладочных прерываний помещается ссылка на процедуру преобразования кода программы. При этом либо блокируется работа отладчика, либо неверно преобразуется в исполняемые команды код программы.

Важной задачей защиты программ от исследований является защита от трассировки программы по заданному событию (своего рода выборочное исследование). В качестве защиты от трассировки по заданному событию (прерыванию) можно выделить три основных способов.

1. Пассивная защита - запрещение работы при переопределении обработчиков событий относительно заранее известного адреса.
2. Активная защита первого типа - замыкание цепочек обработки событий минуя программы трассировки.
3. Активная защита второго типа - программирование функций, исполняемых обработчиками событий, другими способами, не связанными вызовом штатных обработчиков или обработчиков событий, которые текущий момент не трассируются.

Например, для защиты от трассировки по дисковым прерываниям для ОС MS DOS при чтении некопируемой метки с дискеты или винчестера можно использовать следующие приемы:

- работа с ключевой меткой путем прямого программирования контроллера гибкого диска (активная защита второго типа);
- определение одного из неиспользуемых прерываний для работы с диском (активная защита первого типа);
- прямой вызов соответствующих функций в ПЗУ (BIOS) после восстановления различными способами их физического адреса (активная защита первого типа);
- определение факта переопределения адреса прерывания на другую программу и невыполнение в этом случае дисковых операций (пассивная защита).

При операциях с жестким диском, как правило, используется прерывание int 13h. Для предотвращения трассировки программы по заданному прерыванию (в данном случае прерыванию int 13h) можно также использовать указанные выше способы, а именно:

- переопределение исходного прерывания в BIOS на неиспользуемый вектор прерывания;
- прямой вызов функций BIOS.

При опасности трассировки по событиям операционной среды могут быть выделены следующие действия программ:

- определение факта замены обработчиков событий на собственные функции (в частности, для защиты от отладчиков);
- файловые операции, связанные со считываниями различных счетчиков или паролей, вычисление контрольных сумм и значений хэш-кодов;

- файловые операции, связанные со считыванием заголовков и другой существенно важной информации в исполняемых файлах или загружаемых библиотеках.

12.3. Способы встраивания защитных механизмов в программное обеспечение

Встраивание защитных механизмов можно выполнить следующими основными способами:

- вставкой фрагмента проверочного кода в исполняемый файл;
- преобразованием исполняемого файла к неисполняемому виду (шифрование, архивация с неизвестным параметром и т.д.) и применением для загрузки не средств операционной среды, а некоторой программы, в теле которой и осуществляются необходимые проверки;
- вставкой проверочного механизма в исходный код на этапе разработки и отладки программного продукта;
- комбинированием указанных методов.

Применительно к конкретной реализации защитных механизмов для конкретной вычислительной архитектуры можно говорить о защитном фрагменте в исполняемом или исходном коде. К процессу и результату встраивания защитных механизмов можно предъявить следующие требования:

- высокая трудоемкость обнаружения защитного фрагмента при статическом исследовании (особенно актуальна при встраивании в исходный код программного продукта);
- высокая трудоемкость обнаружения защитного фрагмента при динамическом исследовании (при отладке и трассировке по внешним событиям);
- высокая трудоемкость обхода или редуцирования защитного файла.

Возможность встраивания защитных фрагментов в исполняемый код обусловлена типовой архитектурой исполняемых модулей различных операционных сред, содержащих, как правило, адрес точки входа в исполняемый модуль. В этом случае добавление защитного фрагмента происходит следующим образом. Защитный фрагмент добавляется к началу или концу исполняемого файла, точка входа корректируется таким образом, чтобы при загрузке управление передалось дополнительному защитному фрагменту, а в составе защитного фрагмента

предусматривается процедура возврата к оригинальной точке входа. Достаточно часто оригинальный исполняемый файл подвергается преобразованию. В этом случае перед возвратом управления оригинальной точке входа производится преобразование образа оперативной памяти загруженного исполняемого файла к исходному виду.

В случае дополнения динамических библиотек возможна коррекция указанным образом отдельных функций.

Существенным недостатком рассмотренного метода является его легкая обнаруживаемость и в случае отсутствия преобразования оригинального кода исполняемого файла – легкая возможность обхода защитного фрагмента путем восстановления оригинальной точки входа.

12.4. Обфускация программ

12.4.1. Вводные замечания

В данном подразделе кратко затрагиваются вопросы, связанные с активно развивающимися теорией и практикой обфускации программ. Неформально говоря, под *обфускацией программ* здесь понимается преобразование программ с целью максимального затруднения их анализа и модификации, при сохранении, в то же время, их функциональных возможностей. Известные на сегодня методы обфускации, как правило, носят эмпирический характер и слабо обоснованы теоретически. В работе [CTL] приведена классификация методов обфускации, а в работе [BGI] предпринята, скорее всего, первая попытка формализации и теоретического обоснования задачи обфускации программ.

Неформально, *обфускатор* – это (эффективный, вероятностный) «компилятор», который в качестве входа имеет программу P и производит новую программу $\boldsymbol{O}(P)$, которая имеет те же самые функциональные возможности, как и P, но, в то же время, программа $\boldsymbol{O}(P)$ является «*неясной*», («*непонятной*») для противника (наблюдателя, постороннего лица) в некотором заранее определенном смысле. Обфускаторы, если будет доказано, что они существуют, могут применяться для защиты программного обеспечения и, кроме того, могут иметь широкую область криптографических и теоретико-сложностных приложений. Существование обфускаторов для этих приложений должно быть основано на формализации определения понятия «неясности» [BGI].

12.4.2. Теоретические основания

Пусть обфускатор – это (эффективный, вероятностный) «компилятор», который по входу программы (схеме) P выдает новую программу $\boldsymbol{O}(P)$, удовлетворяющую следующим условиям:

1. *Условие функциональности.* Обфусктор $\boldsymbol{O}(P)$ должен вычислять ту же самую функцию, что и программа P.
2. *Условие (свойство) «виртуального черного ящика».* «Все, что может быть вычислено эффективным образом из обфускатора $\boldsymbol{O}(P)$, может быть эффективно вычислено при оракульном доступе к программе P».

Одна из интерпретаций определения, приведенного выше, неформально можно объяснить так, что: во-первых, должна сохраняться функциональная эквивалентность P и $\boldsymbol{O}(P)$ и, во-вторых, все что противник мог бы получить при доступе к *обфусцирующей программе*, он мог бы с таким же успехом получить при доступе к некоторому «черному ящику».

Основной результат работы [BGI] заключается в том, что даже при очень слабой формализации, полная обфускация программ является теоретически невозможной. Это доказывается посредством построения (посредством односторонних функций) семейства функций F, которые являются *необфусцирующими* в том смысле, что существует свойство $\pi: F \rightarrow \{0,1\}$ такое, что:

- по данной программе (схеме), которая вычисляет функцию $f \in F$, значение $\pi(f)$ может быть также эффективно вычислено;
- по данному оракульному доступу к (случайно выбранной) функции $f \in F$, никакой эффективный алгоритм не может вычислять $\pi(f)$ лучше, чем случайным угадыванием.

Таким образом, при таком определении семейства функций F, не существует способов обфускации программ при вычислении F, даже если обфускация предназначена для сокрытия лишь одного бита информации об этой функции (т.е. $\pi(f)$) и даже если обфускатор имеет неограниченное время вычислений.

Приводятся примеры некоторых, казалось бы потенциальных приложений обфускаторов для программ, реализующих схемы цифровой подписи, схемы шифрования, и псевдослучайные функции, однако обфускация для которых не возможна.

Этот результат о теоретической невозможности полной обфускации расширен различными путями, включая обфускаторы, которые: (a) не

обязательно вычисляют за полиномиальное время, (б) только приблизительно сохраняют функциональные возможности (*аппроксимирующие обфускаторы*) и (в) должны работать только для очень ограниченных моделей вычислений.

12.4.3. Практические вопросы построения обфускаторов

Рассмотрим пример исходного псевдокода некоторой программы перед обфускацией (см. рис.12.1.). Можно попытаться понять, каков результат обфускации приведенной программы (см. рис.12.2). Для этого надо узнать, что за классы принимают участие в работе программы, попробовать понять их назначение. В большинстве случаев - это достаточно большой труд.

```
private void CalcPayroll(SpecialList employeeGroup)
    {
      while(employeeGroup.HasMore())
      {
        employee = employeeGroup.GetNext(true);
        employee.UpdateSalary(); DistributeCheck(employee);
      }
    }
```

Рис. 12.1. Псевдокод программы до обфускации

```
private void _1(_1 _2)
    {
     while(_2._1())
     {
       _1 = _2._1(true);
       _1._1();
       _1(_1);
     }
    }
```

Рис. 12.2. Псевдокод программы после обфускации

Таким образом, задача обфускации заключается, как это видно из рис. 12.1.-12.2., в затруднении для понимания и анализа исходного кода программы, запутывании и устранении логических связей в этом коде.

Основные функции при обфускации программ, например, для *.Net-сборок* могут заключаться в следующем. Сначала анализируются метаданные, так как не все члены сборки обфускатор может обработать. Например, обфускатору не стоит заменять имена конструкторов типа данных – это может привести к нежелательным последствиям. Хотя в некоторых случаях это возможно.

Когда список членов сборки для обфускации готов, обфускатор присваивает им новые имена, базируясь на определенном алгоритме. Одни обфускаторы присваивают имена такой же длины, что и были, но лишенные прежнего смысла, другие базируются на нумерации всех членов и обфусцируют их соответствия номеру члена сборки, третьи – базируются на обработке уникального идентификатора члена сборки, четвертые – стараются минимизировать длину имени и часто используют одно и то же имя среди членов, что дает, скорее всего, максимальный эффект при обфускации.

После этого производится запись данных обратно в сборку или генерация новой сборки, ее оптимизация – так как многие из обфускаторов способны удалять ненужную информацию из сборки (отладочную информацию, неиспользуемые поля, классы и т.п.). Таким образом, сборка после обфускации готова и мы получаем обфуцированную программу.

ГЛАВА 13. МЕТОДЫ И СРЕДСТВА ОБЕСПЕЧЕНИЯ ЦЕЛОСТНОСТИ И ДОСТОВЕРНОСТИ ИСПОЛЬЗУЕМОГО ПРОГРАММНОГО КОДА

13.1. Методы защиты программ от несанкционированных изменений

Решение проблемы обеспечения целостности и достоверности электронных данных включает в себя решение, по крайней мере, трех основных взаимосвязанных задач: подтверждения их авторства и подлинности, а также контроль целостности данных. Решение этих трех задач в случае защиты программного обеспечения вытекает из необходимости защищать программы от следующих злоумышленных действий:

- РПС могут быть внедрены в авторскую программу или эта программа может быть полностью заменена на программу-носитель РПС;
- могут быть изменены характеристики (атрибуты) программы;
- злоумышленник может выдать себя за настоящего владельца программы;
- законный владелец программы может отказаться от факта правообладания ею.

Наиболее эффективными методами защиты от подобных злоумышленных действий предоставляют криптографические методы защиты. Это обусловлено тем, что хорошо известные способы контроля целостности программ, основанные на контрольной сумме, продольном контроле и контроле на четность, как правило, представляют собой довольно простые способы защиты от внесения изменений в код программ. Так как область значений, например, контрольной суммы сильно ограничена, а значения функции контроля на четность вообще представляются одним-двумя битами, то для опытного нарушителя не составляет труда найти следующую коллизию: $f(k_1)=f(k_2)$, где k_1 - код программы без внесенной нарушителем закладки, а k_2 - с внесенной программным закладкой и f - функция контроля. В этом случае значения функции для разных аргументов совпадают при тестировании и, следовательно, РПС обнаружено не будет.

Для установления подлинности (неизменности) программ необходимо использовать более сложные методы, такие как аутентификация кода программ, с использованием криптографических способов, которые

обнаруживают следы, остающиеся после внесения преднамеренных искажений.

В первом случае аутентифицируемой программе ставится в соответствие некоторый аутентификатор, который получен при помощи стойкой криптографической функции. Такой функцией может быть криптографически стойкая хэш-функция (например, функция ГОСТ Р 34.11-94) или функция электронной цифровой подписи (например, функция изГОСТ Р 34.10-94). И в том, и в другом случае аргументами функции может быть не только код аутентифицируемой программы, но и время и дата аутентификации, идентификатор программиста и/или предприятия - разработчика ПО, какой-либо случайный параметр и т.п. Может использоваться также любой симметричный шифр (например, DES или ГОСТ 28147-89) в режиме генерации имитовставки. Однако это требует наличия секретного ключа при верификации программ на целостность, что бывает не всегда удобно и безопасно. В то время как при использовании метода цифровой подписи при верификации необходимо иметь только некоторую общедоступную информацию, в данном случае открытый ключ подписи. То есть контроль целостности ПО может осуществить любое заинтересованное лицо, имеющее доступ к открытым ключам используемой схемы цифровой подписи.

Подробные справочные данные и технические детали криптографических методов, применяемых, в том числе, и для защиты ПО рассмотрены в приложении.

Можно еще более усложнить действия злоумышленника по нарушению целостности целевых программ, используя схемы подписи с верификацией по запросу [Ка5,ка14]. В этом случае тестирование программ по ассоциированным с ними аутентификаторам можно осуществить только в присутствии лица, сгенерировавшего эту подпись, то есть в присутствии разработчика программ или представителей предприятия-изготовителя программного обеспечения. В этом случае, если даже злоумышленник и получил для данной программы некий аутентификатор, то ее обладатель может убедиться в достоверности программы только в присутствии специалистов-разработчиков, которые немедленно обнаружат нарушения целостности кода программы и (или) его подлинности.

Комбинация схем подписи и интерактивных систем доказательств позволила создать схемы подписи с верификацией по запросу. Такие

схемы используются для обеспечения целостности и достоверности программно обеспечения.

13.2. Схема подписи с верификацией по запросу

В работах Д. Шаума (см., например, [Ка5,Ка14]) впервые была предложена схема подписи с верификацией по запросу, в которой абонент **V** не может осуществить верификацию подписи без участия абонента **S**. Такие схемы могут эффективно использоваться в том случае, когда фирма - изготовитель поставляет потребителю некоторый информационный продукт (например, программное обеспечение) с проставленной на нем подписью указанного вида [КУ10]. Однако проверить эту подпись, которая гарантирует подлинность программы или отсутствие ее модификаций, можно только уплатив за нее. После факта оплаты фирма - изготовитель дает разрешение на верификацию корректности полученных программ.

Схема состоит из трех этапов (протоколов), к которым относятся непосредственно этап генерации подписи, этап верификации подписи с обязательным участием подписывающего (протокол верификации) и этап оспаривания, если подпись или целостность подписанных сообщений подверглась сомнению (отвергающий протокол).

Схема ПВЗ

Пусть каждый пользователь **S** имеет один открытый ключ P и два секретных ключа S_1 и S_2. Ключ S_1 всегда остается в секрете, - он необходим для генерации подписи. Ключ S_2 может быть открыт для того, чтобы конвертировать схему подписи с верификацией по запросу в обычную схему электронной цифровой подписи.

Вместе с обозначениями секретного и открытого ключей $x \in_R Z_q$ и $y \in_R Z^*_p$ (взятых из отечественного стандарта на электронную цифровую подпись) введем также обозначения $S_1=x$ и $S_2=u$, $u \in_R Z_q$, а также открытый ключ $P=(g,y,w)$, где $w \equiv g^u (\mathrm{mod}\, p)$. Открытый ключ P публикуется в открытом сертифицированном справочнике.

Протокол ГП

Подпись кода m вычисляется следующим образом. Выбирается $k \in_R Z_q$ и вычисляется $r \equiv g^k (\mathrm{mod}\, p)$. Затем вычисляется $s \equiv [xr+mku] (\mathrm{mod}\, q)$. Пара (r,s) является подписью для

кода m. Подпись считается корректной тогда и только тогда, когда $r^u \equiv g^{sw} y^{-rw} (\bmod p)$, где $w \equiv m^{-1} (\bmod q)$.

Проверка подписи (с участием подписывающего) осуществляется посредством следующего интерактивного протокола.

Протокол верификации ПВ

Абонент **V** вычисляет $\gamma \equiv g^{sw} y^{-rw} (\bmod p)$ и просит абонента S доказать, что пара (r,s) есть его подпись под кодом m. Эта задача эквивалентна доказательству того, что дискретный логарифм γ по основанию r равен (по модулю p) дискретному логарифму w по основанию g, то есть, что $\log_g^{(p)} w \equiv \log_r^{(p)} \gamma$. Для этого:

1. Абонент **V** выбирает $a,b \in_R Z_q$, вычисляет $\delta \equiv r^a g^b (\bmod p)$ и посылает δ абоненту **S**.

2. Абонент **S** выбирает $t \in_R Z_q$, вычисляет $h_1 \equiv \delta g^t (\bmod p)$, $h_2 \equiv h_1^u (\bmod p)$ и посылает h_1 и h_2 абоненту **V**.

3. Абонент **V** высылает параметры a и b.

4. Если $\delta \equiv r^a g^b (\bmod p)$, то абонент **S** посылает **V** параметр t; в противном случае - останавливается.

5. Абонент **V** проверяет выполнение равенств $h_1 \equiv r^a g^{b+t} (\bmod p)$ и $h_2 \equiv \gamma^a w^{b+t} (\bmod p)$.

Если проверка завершена успешно, то подпись принимается как корректная.

Протокол ОП

В отвергающем протоколе **S** доказывает, что $\log_g^{(p)} w \neq \log_r^{(p)} \gamma$. Следующие шаги выполняются в цикле l раз.

1. Абонент **V** выбирает $d,e \in_R Z_q$, $d \neq 1$, $\beta \in_R \{0,1\}$. Вычисляет $a \equiv g^e (\bmod p)$, $b \equiv w^e (\bmod p)$, если $\beta=0$ и $a \equiv r^e (\bmod p)$, $b \equiv \gamma^e (\bmod p)$, если $\beta=1$. Посылает **S** значения a, b, d.

2. Абонент **S** проверяет соотношение $a^u (\bmod p) \equiv b$. Если оно выполняется, то $\alpha=0$, в противном случае $\alpha=1$. Выбирает $R \in_R Z_q$, вычисляет $c \equiv d^\alpha g^R (\bmod p)$ и посылает **V** значение c.

3. Абонент **V** посылает абоненту **S** значение e.

4. Абонент **S** проверяет, что выполняются соотношения из следующих двух их пар: $a \equiv g^e (\bmod p)$, $b \equiv w^e (\bmod p)$ и

$a \equiv r^e (\mathrm{mod}\, p)$, $b \equiv \gamma^e (\mathrm{mod}\, p)$. Если да, то посылает **V** значение R. Иначе останавливается.

5. Абонент **V** проверяет, что $d^{\beta} g^{R} (\mathrm{mod}\, p) \equiv c$.

Если во всех l циклах проверка в п.5 выполнена успешно, то абонент **V** принимает доказательства.

Таблица 13.1. Протокол верификации является интерактивным протоколом доказательств с абсолютно нулевым разглашением.

Доказательство. Требуется доказать, что вышеприведенный протокол удовлетворяет трем свойствам: полноты, корректности и нулевого разглашения [Ка5, Ка14].

Полнота означает, что если оба участника (**V** и **S**) следуют протоколу и (r,s) - корректная подпись для сообщения m, то **V** примет доказательство с вероятностью близкой к 1. Из описания протокола верификации очевидно, что абонент **S** всегда может надлежащим образом ответить на запросы абонента **V**, то есть доказательство будет принято с вероятностью 1.

Корректность означает, что если **V** действует согласно протоколу, то какие действия не предпринимал бы **S**, он может обмануть **V** лишь с пренебрежимо малой вероятностью. Здесь под обманом понимается попытка **S** доказать, что $\log_g^{(p)} w \equiv \log_r^{(p)} \gamma$, когда на самом деле эти логарифмы не равны.

Предположим, что $\log_g^{(p)} w \neq \log_r^{(p)} \gamma$. Ясно, что для каждого a существует единственное значение b, то которое дает данный запрос δ. Поэтому δ не содержит в себе никакой информации об a. Если S смог бы найти h_1, h_2, t_1 и t_2 такие, что

$$h_1 \equiv r^{a_1} g^{b_1+t_1} \equiv r^{a_2} g^{b_2+t_2} (\mathrm{mod}\, p)$$

и

$$h_2 \equiv \gamma^{a_1} w^{b_1+t_1} \equiv \gamma^{a_2} w^{b_2+t_1} (\mathrm{mod}\, p),$$

где $a_1 \neq a_2$, то тогда выполнялось бы соотношение

$$\log_g^{(p)} r \equiv [(a_1-a_2)^{-1}((b_2-b_1)+(t_2-t_1))](\mathrm{mod}\, q) \equiv \log_w^{(p)} \gamma.$$

Отсюда, очевидно, следует, что $\log_g^{(p)} w \equiv \log_r^{(p)} \gamma$. В самом деле, пусть $\log_w^{(p)} \gamma \equiv \log_g^{(p)} r \equiv \lambda$. Тогда

$$\gamma \equiv w^{\lambda} \equiv g^{\lambda \log_g^{(p)} w} \equiv r^{\log_g^{(p)} w} (\bmod p),$$

что противоречит предположению. Следовательно, какие бы h_1, h_2, t_1 и t_2 не выбрал S, проверка, которую проводит **V**, может быть выполнена только для одного значения a. Отсюда вероятность обмана не превосходит $1/q$. Отметим, что протокол верификации является безусловно стойким для абонента **V**, то есть доказательство корректности не зависит ни от каких предположений о вычислительной мощности доказывающего (**S**).

Свойство нулевого разглашения означает, что в результате выполнения протокола абонент **V** не получает никакой полезной для себя информации (например, о секретных ключах, используемых **S**). Для доказательства нулевого разглашения необходимо для любого возможного проверяющего V^* построить моделирующую машину M_{V^*}, которая является вероятностной машиной Тьюринга, работает за полиномиальное в среднем время и создает на выходе (без участия **S**) такое же распределение случайных величин, которое возникает у V^* в результате выполнения протокола. В нашем случае, случайные величины, которые «видит» $\mathbf{V}^*$, - это h_1, h_2, и t. Необходимо доказать, что протокол верификации является доказательством с абсолютно нулевым разглашением, то есть моделирующая машина создает распределение случайных величин (h_1,h_2,t), которое в точности совпадает с их распределением, возникающим при выполнении протокола. Моделирующая машина M_{V^*} использует в своей работе $\mathbf{V}^*$ в качестве «черного ящика».

Моделирующая машина

1. Запоминает состояние машины $\mathbf{V}^*$, то есть содержимое всех ее лент, внутреннее состояние и позиции головок на лентах. Затем получает от $\mathbf{V}^*$ значение δ и после этого снова запоминает состояние машины $\mathbf{V}^*$.

2. Выбирает $\eta \in_{\mathrm{R}} Z_q$ и вычисляет $h_1' \equiv g^{\eta} (\bmod p)$ и $h_2'' \equiv r^{\eta} (\bmod p)$.

3. Получает от $\mathbf{V}^*$ значения a и b. Если $\delta \neq r^a g^b (\bmod p)$, то M_{V^*} останавливается.

4. Машина M_{V^*} «отматывает» $\mathbf{V}^*$ на состояние, которое было запомнено в конце шага 1. Выбирает $t \in_{\mathrm{R}} Z_q$ и вычисляет $h_1 \equiv r^a g^{b+t} (\bmod p)$ и $h_2 \equiv \gamma^a w^{b+t} (\bmod p)$.

5. Машина M_{V^*} передает $\mathbf{V}^*$ h_1, h_2 и получает ответ (a',b'). Возможны два варианта:

5.1. $a=a'$, $b=b'$. В этом случае моделирование закончено и M_{V^*} записывает на выходную ленту тройку (h_1,h_2,t) и останавливается.

5.2. $a\neq a'$ или $b\neq b'$. Отсюда следует, что $\delta \equiv r^a g^b \equiv r^{a'} g^{b'} (\bmod p)$. Предположим, что $b\neq b'$. Из этого следует, что $a\neq a'$. Следовательно, M_{V^*} может вычислить $r \equiv g^{(b'-b)/(a-a')} (\bmod p)$. Отсюда $(b'-b)/(a-a')=l$ - дискретный логарифм r по основанию g.

6. Машина M_{V^*} «отматывает» $\mathbf{V}^*$ на состояние, которое было заполнено в начале шага 1. Получает от $\mathbf{V}^*$ значение δ.

7. Выбирает $\eta\in_R Z_q$ вычисляет $h_1 \equiv g^{\eta} (\bmod p)$ и $h_2 \equiv w^{\eta} (\bmod p)$ и передает их V^*.

8. Получает от $\mathbf{V}^*$ значения a и b. Если $\delta \neq r^a g^b (\bmod p)$, то M_{V^*} останавливается. В противном случае вычисляет $t=[\eta-al-b](\bmod q)$, выдает (h_1,h_2,t) на выходную ленту и останавливается.

К пп. 7 и 8 необходимо сделать следующее пояснение. Поскольку $\log_g^{(p)} w \equiv \log_r^{(p)} \gamma$, из этого следует, что $\log_w^{(p)} \gamma \equiv \log_g^{(p)} r$. Отсюда

$$h_2 \equiv w^{\eta} \equiv w^{b+t} w^{al} \equiv w^{b+t} \gamma^a (\bmod p).$$

Из описания моделирующей машины M_{V^*} очевидно, что она работает за полиномиальное время. Величины (h_1,h_2,t) на шаге 5.1 выбираются в точности как в протоколе и поэтому имеют такое же распределение вероятностей. Кроме того, значения (h_1,h_2), выбираемые на шаге 7, имеют такое же распределение, как и в протоколе. Чтобы показать что и t имеет одинаковое распределение, достаточно заметить, что машина $\mathbf{V}^*$ не может по h_1 и h_2 определить, с кем она имеет дело - с **S** или M_{V^*}. Поэтому, даже если бы $\mathbf{V}^*$ могла каким-либо «хитрым» образом строить a и b с учетом (h_1,h_2), распределение вероятностей величин a и b в обоих случаях одинаковы. Но значение t определяется однозначно четверкой величин a, b, h_1, h_2, при условии выполнения проверки на шаге 5 протокола. ■

Таблица 13.2. Отвергающий протокол является протоколом доказательства с абсолютно нулевым разглашением.

Доказательство. Полнота протокола очевидна. Если абоненты **S** и **V** следуют протоколу, тогда абонент **V** всегда примет доказательства абонента **S**.

Для доказательства корректности прежде всего заметим, что если $\log_g^{(p)} w \equiv \log_r^{(p)} \gamma$, то a и b, выбираемые абонентом V на шаге 1, не несут в себе никакой информации о значении β. Поэтому, если S может "открыть" c, сгенерированное им на шаге 2, лишь единственным образом (то есть выдать на шаге 4 единственное значение R, соответствующее данному α), то проверка на шаге 5 будет выполнена с вероятностью 1/2 в одном цикле и с вероятностью $1/2^l$ во всех l циклах.

Если же **S** может сгенерировать c таким образом, что с вероятностью, которая не является пренебрежимо малой, он может на шаге 4 "открыть" оба значения α, то есть найти R_1 и R_2 такие, что $c \equiv dg^{R_1} (\bmod p)$ и $c \equiv g^{R_2} (\bmod p)$, то алгоритм, который использует **S** для этой цели, можно использовать для вычисления дискретных логарифмов: $\log_g d = R_2 - R_1$. Так как при случайном выборе значения d логарифм $\log_g d$ может быть вычислен с вероятностью, которая не является пренебрежимо малой, это противоречит гипотезе о трудности вычисления дискретных логарифмов.

Далее доказывается, что отвергающий протокол является доказательством с абсолютно нулевым разглашением. Для этого необходимо для всякого возможного проверяющего $\mathbf{V}^*$ построить моделирующую машину M_{V^*}, которая создает на выходе (без участия **S**) такое же распределение случайных величин (в данном случае, c и R), какое возникает у $\mathbf{V}^*$ в результате выполнения протокола.

Моделирующая машина

Следующие шаги выполняются в цикле l раз.

1. Машина M_{V^*} запоминает состояние машины $\mathbf{V}^*$.

2.Получает от $\mathbf{V}^*$ значения a, b и d.

3.Выбирает $\alpha \in_R \{0,1\}$, $R \in_R Z_q$ и вычисляет $c \equiv d^{\alpha} g^{R} (\bmod p)$. Посылает $\mathbf{V}^*$ значение c.

4.Получает от $\mathbf{V}^*$ значение e.

5.Проверяет, было ли «угадано» на шаге 2 значение α (это значение было «угадано», если $a \equiv g^e (\bmod p)$, $b \equiv w^e (\bmod p)$ и

α=0, либо $a \equiv r^e (\mathrm{mod}\, p)$, $b \equiv \gamma^e (\mathrm{mod}\, p)$ и α=1). Если да, то записывает на входную ленту значение (c,R). В противном случае «отматывает» $\mathbf{V}^*$ на то состояние, которое было запомнено на шаге 1, и переходит на шаг 2.

Легко видеть, что распределения случайных величин (c,R), возникающее в процессе выполнения протокола и создаваемые моделирующей машиной M_{V^*}, одинаковы, поскольку R в обоих случаях - чисто случайная величина, а величина c записывается на выходную ленту машины M_{V^*} только тогда, когда α совпало с β.

Поскольку значение α выбирается машиной M_{V^*} на шаге 3 случайным образом, а c не дает $\mathbf{V}^*$ никакой информации о значении α, на каждой итерации α будет угадано с вероятностью 1/2. Отсюда следует, что машина M_{V^*} работает за полиномиальное в среднем время. ■

В работе [Ка14] показано, как строить схемы *конвертируемой и селективно конвертируемой подписи с верификацией по запросу* на основе отечественного стандарта ГОСТ Р 34.10-94. В таких схемах открытие определенного секретного параметра некоторой схемы подписи с верификацией по запросу позволяет трансформировать последнюю в обычную схему цифровой подписи. При этом открытие секретного параметра в конвертируемой схеме подписи с верификацией по запросу дает возможность верифицировать все имеющиеся и сгенерированные в дальнейшем подписи, в то время как в селективно конвертируемых схемах подписи с верификацией по запросу можно верифицировать лишь какую-либо одну подпись.

13.3. Примеры применения схемы подписи с верификацией по запросу

Предположим фирма-изготовитель программного обеспечения распространяет свою продукцию посредством электронной почты. На каждом экземпляре ПО проставляется некоторая подпись (тип подписи пока не определен, но в любом случае подпись представляет собой значение функции, аргументами которой обязательно являются подписываемые данные и секретный ключ подписывающего). Эта подпись гарантирует, что программы являются подлинными (то есть, разработаны фирмой-изготовителем) и не были модифицированы. Однако

верифицировать эту подпись, можно только уплатив за программный продукт. В этом случае фирма - изготовитель во взаимодействии с потребителем устанавливают корректность подписи, а значит и подлинность проданных программ.

Селективно конвертируемая схема подписи с верификацией по запросу можно использовать следующим образом. В рассмотренном ранее сценарии для схемы подписи с верификацией по запросу (ПВЗ) предположим, что фирме - изготовителю программного обеспечения необходимо, чтобы некоторые из сотрудников этой организации могли бы верифицировать ПВЗ, однако это надо сделать так, чтобы в дальнейшем они не могли генерировать подпись с использованием полученной информации о некоторых секретных параметрах схемы ПВЗ. Возможна ситуация, когда фирма - изготовитель по каким - либо причинам прекращает свою деятельность. Тогда она открывает для пользователей свою секретную информацию для верификации подписи с тем, чтобы программы фирмы можно еще было бы безопасно использовать. Предположим также следующий сценарий. Та же фирма изготовила новую версию программного обеспечения, в то время как старая версия морально устарела и ее нет смысла использовать в дальнейшем. Однако, фирма желает убедить потребителей в своей надежности и в том, что все это время они использовали достоверные программы. В этом случае фирма публикует свой некоторый секретный параметр, позволяющий пользователям убедиться в оригинальности программ.

Особенности реализации подобных сценариев, а также злоумышленные действия в отношении предлагаемых схем защиты, рассмотрены в работах [КУ10,Ка5,Ка14].

Напомним, что все необходимые сведения об алгоритмах криптографического преобразования данных, в том числе и в целях защиты программ, а также другие сведения из криптографии и криптоанализа, приведены в приложении.

ГЛАВА 14. ОСНОВНЫЕ ПОДХОДЫ К ЗАЩИТЕ ПРОГРАММ ОТ НЕСАНКЦИОНИРОВАННОГО КОПИРОВАНИЯ

14.1. Основные функции средств защиты от копирования

При защите программ от несанкционированного копирования применяются методы, которые позволяют привносить в защищаемую программу функции привязки процесса выполнения кода программы только на тех ЭВМ, на которые они были инсталлированы.

Инсталлированная программа для защиты от копирования при каждом запуске должна выполнять следующие действия:

- анализ аппаратно-программной среды компьютера, на котором она запущена, формирование на основе этого анализа текущих характеристик своей среды выполнения;
- проверка подлинности среды выполнения путем сравнения ее текущих характеристик с эталонными, хранящимися на винчестере;
- блокирование дальнейшей работы программы при несовпадении текущих характеристик с эталонными.

Этап проверки подлинности среды является одним из самых уязвимых с точки зрения защиты. Можно детально не разбираться с логикой защиты, а немного «подправить» результат сравнения, и защита будет снята.

При выполнении процесса проверки подлинности среды возможны три варианта: с использованием множества операторов сравнения того, что есть, с тем, что должно быть, с использованием механизма генерации исполняемых команд в зависимости от результатов работы защитного механизма и с использованием арифметических операций. При использовании механизма генерации исполняемых команд в первом байте хранится исходная ключевая контрольная сумма BIOS, во второй байт записывается подсчитанная контрольная сумма в процессе выполнения задачи. Затем осуществляется вычитание из значения первого байта значение второго байта, а полученный результат добавляется к каждой ячейки оперативной памяти в области операционной системы. Понятно, что если суммы не совпадут, то операционная система функционировать не будет. При использовании арифметических операций осуществляется преобразование над данными арифметического характера в зависимости от результатов работы защитного механизма.

Для снятия защиты от копирования применяют два основных метода: статический и динамический [РД].

Статические методы предусматривают анализ текстов защищенных программ в естественном или преобразованном виде. Динамические методы предусматривают слежение за выполнением программы с помощью специальных средств снятия защиты от копирования.

14.2. Основные методы защиты от копирования

14.2.1. Криптографические методы

Для защиты инсталлируемой программы от копирования при помощи криптографических методов инсталлятор программы должен выполнить следующие функции:

- анализ аппаратно-программной среды компьютера, на котором должна будет выполняться инсталлируемая программа, и формирование на основе этого анализа эталонных характеристик среды выполнения программы;
- запись криптографически преобразованных эталонных характеристик аппаратно-программной среды компьютера на винчестер.

Преобразованные эталонные характеристики аппаратно-программной среды могут быть занесены в следующие области жесткого диска:

- в любые места области данных (в созданный для этого отдельный файл, в отдельные кластеры, которые должны помечаться затем в FAT как зарезервированные под операционную систему или дефектные);
- в зарезервированные сектора системной области винчестера;
- непосредственно в файлы размещения защищаемой программной системы, например, в файл настройки ее параметров функционирования.

Можно выделить два основных метода защиты от копирования с использованием криптографических приемов:

- с использованием односторонней функции;
- с использованием шифрования (которое также может использовать односторонние функции).

Односторонние функции это функции, для которых при любом x из области определения легко вычислить $f(x)$, однако почти для всех y из ее области значений, найти $y=f(x)$ вычислительно трудно (см. приложение).

Если эталонные характеристики программно-аппаратной среды представить в виде аргумента односторонней функции x, то y - есть «образ» этих характеристик, который хранится на винчестере и по значению которого вычислительно невозможно получить сами характеристики. Примером такой односторонней функции может служить функция дискретного экспоненцирования, описанная ранее, с размерностью операндов не менее 512 битов.

При шифровании эталонные характеристики шифруются по ключу, совпадающему с этими текущими характеристиками, а текущие характеристики среды выполнения программы для сравнения с эталонными также зашифровываются, но по ключу, совпадающему с этими текущими характеристиками. Таким образом, при сравнении эталонные и текущие характеристики находятся в зашифрованном виде и будут совпадать только в том случае, если исходные эталонные характеристики совпадают с исходными текущими.

14.2.2. Метод привязки к идентификатору

В случае если характеристики аппаратно-программной среды отсутствуют в явном виде или их определение значительно замедляет запуск программ или снижает удобство их использования, то для защиты программ от несанкционированного копирования можно использовать методов привязки к идентификатору, формируемому инсталлятором. Суть данного метода заключается в том, что на винчестере при инсталляции защищаемой от копирования программы формируется уникальный идентификатор, наличие которого затем проверяется инсталлированной программой при каждом ее запуске. При отсутствии или несовпадении этого идентификатора программа блокирует свое дальнейшее выполнение.

Основным требованием к записанному на винчестер уникальному идентификатору является требование, согласно которому данный идентификатор не должен копироваться стандартным способом. Для этого идентификатор целесообразно записывать в следующие области жесткого диска:

- в отдельные кластеры области данных, которые должны помечаться затем в FAT как зарезервированные под операционную систему или как дефектные;
- в зарезервированные сектора системной области винчестера.

Некопируемый стандартным образом идентификатор может помещаться на дискету, к которой должна будет обращаться при каждом

своем запуске программа. Такую дискету называют ключевой. Кроме того, защищаемая от копирования программа может быть привязана и к уникальным характеристикам ключевой дискеты. Следует учитывать, что при использовании ключевой дискеты значительно увеличивается неудобство пользователя, так как он всегда должен вставлять в дисковод эту дискету перед запуском защищаемой от копирования программы.

14.2.3. Методы, основанные на работе с переходами и стеком

Данные методы основаны на включение в тело программы переходов по динамически изменяемым адресам и прерываниям, а также самогенерирующихся команд (например, команд, полученных с помощью сложения и вычитания). Кроме того, вместо команды безусловного перехода (JMP) может использоваться возврат из подпрограммы (RET). Предварительно в стек записывается адрес перехода, который в процессе работы программы модифицируется непосредственно в стеке.

При работе со стеком, стек определяется непосредственно в области исполняемых команд, что приводит к затиранию при работе со стеком. Этот способ применяется, когда не требуется повторное исполнение кода программы. Таким же способом можно генерировать исполняемые команды до начала вычислительного процесса.

14.2.4. Манипуляции с кодом программы

При манипуляциях с кодом программы можно привести два следующих способа:

- включение в тело программы «пустых» модулей;
- изменение защищаемой программы.

Первый способ заключается во включении в тело программы модулей, на которые имитируется передача управления, но реально никогда не осуществляется. Эти модули содержат большое количество команд, не имеющих никакого отношения к логике работы программы. Но «ненужность» этих программ не должна быть очевидна потенциальному злоумышленнику.

Второй способ заключается в изменении начала защищаемой программы таким образом, чтобы стандартный дизассемблер не смог ее правильно дизассемблировать. Например, такие программы, как Nota и Copylock, внедряя защитный механизм в защищаемый файл, полностью модифицируют исходный заголовок EXE-файла.

Все перечисленные методы были, в основном направлены на противодействия статическим способам снятия защиты от копирования. В следующем подразделе рассмотрим методы противодействия динамическим способам снятия защиты.

14.2.5. Методы противодействия динамическим способам снятия защиты программ от копирования

Набор методов противодействия динамическим способам снятия защиты программ от копирования включает следующие методы [РД]:

- периодический подсчет контрольной суммы, занимаемой образом задачи области оперативной памяти, в процессе выполнения. Это позволяет:
  - заметить изменения, внесенные в загрузочный модуль;
  - в случае если программу пытаются «раздеть», выявить контрольные точки, установленные отладчиком;
- проверка количества свободной памяти и сравнение и с тем объемом, к которому задача «привыкла» или «приучена». Это действия позволит застраховаться от слишком грубой слежки за программой с помощью резидентных модулей;
- проверка содержимого незадействованных для решения защищаемой программы областей памяти, которые не попадают под общее распределение оперативной памяти, доступной для программиста, что позволяет добиться «монопольного» режима работы программы;
- проверка содержимого векторов прерываний (особенно 13h и 21h) на наличие тех значений, к которым задача «приучена». Иногда бывает полезным сравнение первых команд операционной системы, отрабатывающих этим прерывания, с теми командами, которые там должны быть. Вместе с предварительной очисткой оперативной памяти проверка векторов прерываний и их принудительное восстановление позволяет избавиться от большинства присутствующих в памяти резидентных программ;
- переустановка векторов прерываний. Содержимое некоторых векторов прерываний (например, 13h и 21h) копируется в область свободных векторов. Соответственно изменяются и обращения к прерываниям. При этом слежение за известными векторами не даст желаемого результата. Например, самыми первыми исполняемыми командами программы копируется содержимое вектора 21h (4

байта) в вектор 60h, а вместо команд int 21h в программе везде записывается команда int 60h. В результате в явном виде в тексте программы нет ни одной команды работы с прерыванием 21h;

- постоянное чередование команд разрешения и запрещения прерывания, что затрудняет установку отладчиком контрольных точек;
- Контроль времени выполнения отдельных частей программы, что позволяет выявить «остановы» в теле исполняемого модуля.

Многие перечисленные защитные средства могут быть реализованы исключительно на языке Ассемблер. Одна из основных отличительных особенностей этого языка заключается в том, что для него не существует ограничений в области работы со стеком, регистрами, памятью, портами ввода/вывода и т.п.

ГЛАВА 15. ПРАВОВАЯ И ОРГАНИЗАЦИОННАЯ ПОДДЕРЖКА ПРОЦЕССОВ РАЗРАБОТКИ И ПРИМЕНЕНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ЧЕЛОВЕЧЕСКИЙ ФАКТОР

15.1. СТАНДАРТЫ И ДРУГИЕ НОРМАТИВНЫЕ ДОКУМЕНТЫ, РЕГЛАМЕНТИРУЮЩИЕ ЗАЩИЩЕННОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ И ОБРАБАТЫВАЕМОЙ ИНФОРМАЦИИ

15.1.1. Международные стандарты в области информационной безопасности

Общие вопросы

За рубежом разработка стандартов проводится непрерывно, последовательно публикуются проекты и версии стандартов на разных стадиях согласования и утверждения. Некоторые стандарты поэтапно углубляются и детализируются в виде совокупности взаимосвязанных по концепциям и структуре групп стандартов.

Принято считать, что неотъемлемой частью общего процесса стандартизации информационных технологий (ИТ) является разработка стандартов, связанных с проблемой безопасности ИТ, которая приобрела большую актуальность в связи с тенденциями все большей взаимной интеграции прикладных задач, построения их на базе распределенной обработки данных, систем телекоммуникаций, технологий обмена электронными данными.

Разработка стандартов для открытых систем, в том числе и стандартов в области безопасности ИТ, осуществляется рядом специализированных международных организаций и консорциумов таких, как, например, ISO, IEC, ITU-T, IEEE, IAB, WOS, ECMA, X/Open, OSF, OMG и др. [Су].

Значительная работа по стандартизации вопросов безопасности ИТ проводится специализированными организациями и на национальном уровне. Все это позволило к настоящему времени сформировать достаточно обширную методическую базу, в виде международных, национальных и отраслевых стандартов, а также нормативных и руководящих материалов, регламентирующих деятельность в области безопасности ИТ.

Архитектура безопасности Взаимосвязи открытых систем

Большинство современных сложных информационных структур,

лежащих в качестве основы существующих АС проектируются с учетом идеологии Эталонной модели (ЭМ) Взаимосвязи открытых систем (ВОС), которая позволяет оконечному пользователю сети (или его прикладным процессам) получить доступ к информационно-вычислительным ресурсам значительно легче, чем это было раньше. Вместе с тем концепция открытости систем создает ряд трудностей в организации защиты информации в системах и сетях. Требование защиты ресурсов сети от НСД является обязательным при проектировании и реализации большинства современных информационно-вычислительных сетей, соответствующих ЭМ ВОС.

В 1986 г. рядом международных организаций была принята *Архитектура безопасности ВОС (АБ ВОС)*. В архитектуре ВОС выделяют семь уровней иерархии: физический, канальный, сетевой, транспортный, сеансовый, представительный и прикладной. Однако в АБ ВОС предусмотрена реализация механизмов защиты в основном на пяти уровнях. Для защиты информации на физическом и канальном уровне обычно вводится такой механизм защиты, как линейное шифрование. Канальное шифрование обеспечивает закрытие физических каналов связи с помощью специальных шифраторов. Однако применение только канального шифрования не обеспечивает полного закрытия информации при ее передаче по сети, так как на узлах коммутации пакетов информация будет находиться в открытом виде. Поэтому НСД нарушителя к аппаратуре одного узла ведет к раскрытию всего потока сообщений, проходящих через этот узел. В том случае, когда устанавливается виртуальное соединение между двумя абонентами сети и коммуникации, в данном случае, проходят по незащищенным элементам сети, необходимо сквозное шифрование, когда закрывается информационная часть сообщения, а заголовки сообщений не шифруются. Это позволяет свободно управлять потоками зашифрованных сообщений. Сквозное шифрование организуется на сетевом и/или транспортном уровнях согласно ЭМ ВОС. На прикладном уровне реализуется большинство механизмов защиты, необходимых для полного решения проблем обеспечения безопасности данных в ИВС.

АБ ВОС устанавливает следующие службы безопасности (см. табл.15.1).

- обеспечения целостности данных (с установлением соединения, без установления соединения и для выборочных полей сообщений);

- обеспечения конфиденциальности данных (с установлением соединения, без установления соединения и для выборочных полей сообщений);

Таблица 15.1

Назначение службы	***Номер службы***	***Процедура защиты***	***Номер уровня***
Аутентификация:			
Одноуровневых	1	Шифрование, цифровая подпись	3,4
объектов		Обеспечение аутентификации	3,4,7
		Шифрование	3,4
источника данных	2	Цифровая подпись	3,4,7
Контроль доступа	3	Управление доступом	3,4,7
Засекречивание:			
соединения	4	Шифрование	1-4,6,7
		Управление трафиком	3
в режиме без соединения	5	Шифрование	2-4,6,7
		Управление трафиком	3
выборочных полей	6	Шифрование	6,7
потока данных		Шифрование	1,6
	7	Заполнение потока	3,7
		Управление трафиком	3
Обеспечение целостности:			
соединения с	8	Шифрование, обеспечение	4,7
восстановлением		целостности данных	
	9	Шифрование, обеспечение	3,4,7
соединения без		целостности данных	
восстановления	10	Шифрование, обеспечение	7
		целостности данных	
выборочных полей	11	Шифрование	3,4,7
		Цифровая подпись	4
без установления		Обеспечение целостности	3,4,7
соединения	12	данных	
		Шифрование	7
		Цифровая подпись	4,7
выборочных полей без		Обеспечение целостности	7
соединения		данных	

Обеспечение невозможности отказа от факта:			
отправки	13	Цифровая подпись, обеспечение целостности данных, подтверждение характеристик данных	7
доставки	14	Цифровая подпись, обеспечение целостности данных, подтверждение характеристик данных	7

- контроля доступа;
- аутентификации (одноуровневых объектов и источника данных);
- обеспечения конфиденциальности трафика;
- обеспечения невозможности отказа от факта отправки сообщения абонентом - передатчиком и приема сообщения абонентом - приемником.

Состояние международной нормативно-методической базы

С целью систематизации анализа текущего состояния международной нормативно-методической базы в области безопасности ИТ необходимо использовать некоторую классификацию направлений стандартизации. В общем случае, можно выделить следующие направления.

1. Общие принципы управления информационной безопасностью.
2. Модели безопасности ИТ.
3. Методы и механизмы безопасности ИТ (такие, как, например: методы аутентификации, управления ключами и т.п.).
4. Криптографические алгоритмы.
5. Методы оценки безопасности информационных систем.
6. Безопасность *EDI*-технологий.
7. Безопасность межсетевых взаимодействий (межсетевые экраны).
8. Сертификация и аттестация объектов стандартизации.

Стандартизация вопросов управления информационной безопасностью

Анализ проблемы защиты информации в информационных системах требует, как правило, комплексного подхода, использующего общеметодологические концептуальные решения, которые позволяют определить необходимый системообразующей контекст для редуцирования общей задачи управления безопасностью ИТ к решению частных задач. Поэтому в настоящее время возрастает роль стандартов и регламентирующих материалов общеметодологического назначения.

На роль такого документа претендует, находящийся в стадии утверждения проект международного стандарта ISO/IEC DTR 13335-1,2,3 – «Информационная технология. Руководство по управлению безопасностью информационных технологий». Данный документ содержит:

- определения важнейших понятий, непосредственно связанных с проблемой управления безопасностью ИТ;
- определения важных архитектурных решений по созданию систем управления безопасностью ИТ (СУБ ИТ), в том числе, определение состава элементов, задач, механизмов и методов СУБ ИТ;
- описание типового жизненного цикла и принципов функционирования СУБ ИТ;
- описание принципов формирования политики (методики) управления безопасностью ИТ;
- методику анализа исходных данных для построения СУБ ИТ, в частности методику идентификации и анализа состава объектов защиты, уязвимых мест информационной системы, угроз безопасности и рисков и др.;
- методику выбора соответствующих мер защиты и оценки остаточного риска;
- принципы построения организационного обеспечения управления в СУБ ИТ и пр.

Стандартизация моделей безопасности ИТ

С целью обеспечения большей обоснованности программно-технических решений при построении СУБ ИТ, а также определения ее степени гарантированности, необходимо использование возможно более точных описательных моделей как на общесистемном (архитектурном) уровне, так и на уровне отдельных аспектов и средств СУБ ИТ.

Построение моделей позволяет структурировать и конкретизировать исследуемые объекты, устранить неоднозначности в их понимании, разбить решаемую задачу на подзадачи, и, в конечном итоге, выработать необходимые решения.

Можно выделить следующие международные стандарты и другие документы, в которых определяются основные модели безопасности ИТ:

- ISO/IEC 7498-2-89 - «Информационные технологии. Взаимосвязь открытых системы. Базовая эталонная модель. Часть 2. Архитектура информационной безопасности» (кратко описанная выше);
- ISO/IEC DTR 10181-1 – «Информационные технологии. Взаимосвязь открытых систем. Основы защиты информации для открытых систем. Часть 1. Общее описание основ защиты информации ВОС»;

- ISO/IEC DTR 10745 – «Информационные технологии. Взаимосвязь открытых систем. Модель защиты информации верхних уровней»;
- ISO/IEC DTR 11586-1 – «Информационные технологии. Взаимосвязь открытых систем. Общие функции защиты верхних уровней. Часть 1. Общее описание, модели и нотация»;
- ISO/IEC DTR 13335-1 – «Информационные технологии. Руководство по управлению безопасностью информационных технологий. Часть 1. Концепции и модели безопасности информационных технологий».

Стандартизация методов и механизмов безопасности ИТ

На определенном этапе задача защиты информационных технологий разбивается на частные подзадачи, такие как обеспечение конфиденциальности, целостности и доступности. Для этих подзадач должны вырабатываться конкретные решения по организации взаимодействия объектов и субъектов информационных систем. К таким решениям относятся методы:

- аутентификации субъектов и объектов информационного взаимодействия, предназначенные для предоставления взаимодействующим сторонам возможности удостовериться, что противоположная сторона действительно является тем, за кого себя выдает;
- шифрования информации, предназначенные для защиты информации в случае перехвата ее третьими лицами;
- контроля целостности, предназначенные для обеспечения того, чтобы информация не была искажена или подменена;
- управления доступом, предназначенные для разграничения доступа к информации различных пользователей;
- повышения надежности и отказоустойчивости функционирования системы, предназначенные для обеспечения гарантий выполнения информационной системой целевых функций;
- управления ключами, предназначенные для организации создания, распространения и использования ключей субъектов и объектов информационной системы, с целью создания необходимого базиса для процедур аутентификации, шифрования, контроля подлинности и управления доступом.

Организации по стандартизации уделяют большое внимание разработке типовых решений для указанных выше аспектов безопасности. К ним, в первую очередь отнесем следующие международные стандарты:

- ISO/IEC 9798-91 – «Информационные технологии. Защита информации. Аутентификация объекта».

  Часть 1. Модель.

  Часть 2. Механизмы, использующие симметричные криптографические алгоритмы.

  Часть 3. Аутентификация на базе алгоритмов с открытыми ключами.

  Часть 4. Механизмы, использующие криптографическую контрольную функцию.

  Часть 5. Механизмы, использующие алгоритмы с нулевым разглашением.
- ISO/IEC 09594-8-88 – «Взаимосвязь открытых систем. Справочник. Часть 8. Основы аутентификации»;
- ISO/IEC 11577-94 – «Информационные технологии. Передача данных и обмен информацией между системами. Взаимосвязь открытых систем. Протокол защиты информации на сетевом уровне»;
- ISO/IEC DTR 10736 – «Информационные технологии. Передача данных и обмен информацией между системами. Протокол защиты информации на транспортном уровне»;
- ISO/IEC CD 13888 – «Механизмы предотвращения отрицания».

  Часть 1. Общая модель.

  Часть 2. Использование симметричных методов.

  Часть 3. Использование асимметричных методов;
- ISO/IEC 8732-88 – «Банковское дело. Управление ключами»;
- ISO/IEC 11568-94 – «Банковское дело. Управление ключами».

  Часть 1. Введение. Управление ключами.

  Часть 2. Методы управления ключами для симметричных шифров.

  Часть 3. Жизненный цикл ключа для симметричных шифров;
- ISO/IEC 11166-94 – «Банковское дело. Управление ключами посредством асимметричного алгоритма».

  Часть 1. Принципы процедуры и форматы.

Часть 2. Принятые алгоритмы, использующие криптосистему RSA;

- ISO/IEC DIS 13492 – «Банковское дело. Управление ключами, относящимися к элементам данных»;
- ISO/IEC CD 11770 – «Информационные технологии. Защита информации. Управление ключами».

Часть 1. Общие положения.

Часть 2. Механизмы, использующие симметричные методы.

Часть 3. Механизмы, использующие асимметричные методы;

- ISO/IEC DTR 10181- «Информационные технологии. Взаимосвязь открытых систем. Основы защиты информации для открытых систем».

Часть 1. Общее описание основ защиты информации в ВОС.

Часть 2. Основы аутентификации.

Часть 3. Управление доступом.

Часть 4. Безотказность получения.

Часть 5. Конфиденциальность.

Часть 6. Целостность.

Часть 7. Основы проверки защиты.

К этому же уровню следует отнести стандарты, описывающие интерфейсы механизмов безопасности ИТ:

- ISO/IEC 10164-7-92. «Информационные технологии. Взаимосвязь открытых систем. Административное управление системы. Часть 7. Функции уведомления о нарушениях информационной безопасности».
- ISO/IEC DTR 11586. «Информационные технологии. Взаимосвязь открытых систем. Общие функции защиты верхних уровней».

Часть 1. Общее описание, модели и нотация.

Часть 2. Определение услуг сервисного элемента обмена информацией защиты.

Часть 3. Спецификация протокола сервисного элемента обмена информацией защиты.

Часть 4. Спецификация синтаксиса защищенной передачи.

В стандартах этого уровня, как правило, не указываются конкретные криптографические алгоритмы, а декларируется, что может быть использован любой криптоалгоритм, при этом подразумевалось использование определенных зарубежных криптографических алгоритмов.

Поэтому в ряде случаев при использовании некоторых стандартов может потребоваться их адаптация к отечественным криптоалгоритмам.

Стандартизация международных криптографических алгоритмов

Организация ISO стандартизировала ряд криптографических алгоритмов в таких международных стандартах, как, например:

- ISO/IEC 10126-2-91 – «Банковское дело. Процедуры шифрования сообщения. Часть 2. Алгоритм DEA»;
- ISO/IEC 8732-87 – «Информационные технологии. Защита информации. Режимы использования 64-битного блочного алгоритма»;
- ISO/IEC 10116-91- «Банковское дело. Режимы работы *n*-битного блочного алгоритма шифрования»;
- ISO/IEC 10118-1,2-88 – «Информационные технологии. Шифрование данных. Хэш-функция для цифровой подписи»;
- ISO/IEC CD 10118-3,4 – «Информационные технологии. Защита информации. Функции хэширования»;
- ISO/IEC 9796-91 – «Информационные технологии. Схема электронной подписи, при которой производится восстановление сообщения»;
- ISO/IEC CD 14888 – «Информационные технологии. Защита информации. Цифровая подпись с добавлением».

Однако широкое внедрение этих алгоритмов представляется малореальным, поскольку техническая политика крупных индустриальных государств, в т.ч. и Российской Федерации, направлена, как правило, на использование собственных криптоалгоритмов.

15.1.2. Отечественная нормативно-правовая база, под действие которой подпадают АС различного назначения

Стандартизация в области защиты информации от НСД

К основным стандартам и нормативным техническим документам по безопасности информации, в первую очередь, относится комплект руководящих документов Гостехкомиссии России (1998 г), которые в соответствии с Законом «О стандартизации» можно отнести к отраслевым стандартам, в том числе «Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности средств вычислительной техники»», «Автоматизированные системы.

Защита от несанкционированного доступа к информации. Классификация автоматизированных систем и требования по защите информации», «Временное положение по организации разработки, изготовления и эксплуатации программных и технических средств защиты информации от несанкционированного доступа в автоматизированных системах и средствах вычислительной техники», «Средства вычислительной техники. Межсетевые экраны. Защита от несанкционированного доступа к информации. Показатели защищенности от НСД к информации», ГОСТ Р 50739-95 «Средства вычислительной техники. Защита от несанкционированного доступа к информации. Общие технические требования».

Особенности защиты программ нашли свое отражение в следующих руководящих документах: РД «Защита информации от несанкционированного доступа к информации. Программное обеспечение средств защиты информации. Классификация по уровню контролю отсутствия недекларированных возможностей» и проект РД «Антивирусные средства. Показатели защищенности и требования по защите от вирусов».

В первом документе устанавливается классификация программного обеспечения автоматизированных систем и средств вычислительной техники по уровню контролю отсутствия в нем недекларированных возможностей, где уровень гарантированности определяется набором требований, предъявляемых к составу, объему и содержанию документации представляемой заявителем для проведения испытаний программ и к содержанию испытаний.

Во втором документе устанавливается классификация средств антивирусной защиты по уровню обеспечения защиты от воздействия программ-вирусов на базе перечня показателей защищенности и совокупности описывающих их требований.

Кроме того, следующие нормативные документы, так или иначе косвенно регламентируют отдельные вопросы обеспечения безопасности ПО:

- ГОСТ 28195-89. Оценка качества программных средств. Общие положения;
- ГОСТ 21552-84. Средства вычислительной техники. ОТТ, приемка методы испытаний, маркировка, упаковка, транспортировка и хранение;

- ГОСТ ВД 21552-84. Средства вычислительной техники. ОТТ, приемка методы испытаний, маркировка, упаковка, транспортировка и хранение;
- ТУ на конкретный вид продукции (ПО).

Стандартизация отечественных криптографических алгоритмов

Отечественные стандарты [ГОСТ1-ГОСТ4] описывают криптографические алгоритмы, достаточные для решения большинства прикладных задач:

- ГОСТ 28147-89 «Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования» описывает три алгоритма шифрования данных (из них один - так называемый «режим простой замены» - является служебным, два других – «режим гаммирования» и «режим гаммирования с обратной связью» - предназначены для шифрования целевых данных) и алгоритм выработки криптографической контрольной суммы (имитовставки), предназначенной для контроля целостности информации;
- ГОСТ Р 34.10-94 «Информационная технология. Криптографическая защита информации. Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма»;
- ГОСТ Р 34.11-94 «Информационная технология. Криптографическая защита информации. Функция хэширования»$
- ГОСТ Р 34.10-2002 «Информационная технология. Криптографическая защита информации. Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма».

Алгоритмы электронной цифровой подписи и хэширования данных часто связаны друг с другом, когда описываются процедуры удостоверения авторства и подлинности информации (электронных документов).

15.2. Сертификационные испытания программных средств

До получения готового программного изделия оценить его показатели качества можно лишь вероятностным образом на макроуровне рассмотрения структуры программного комплекса. Поэтому возникает

насущная потребность в организации специального этапа в процессе создания ПО необходимого для *подтверждения соответствия показателям качества реального программного изделия заданным к нему требованиям*. Причем контроль выполнения этих требований должен осуществляться с учетом предполагаемых условий применения при форсированных нагрузках и тестировании всех установленных режимов. В рамках создания современных информационных технологий решение задач испытания ПО и получения документального подтверждения требуемых показателей качества программ объединяется в рамках процесса сертификации.

Сертификация программного обеспечения представляет собой процесс испытаний программ в нагруженных режимах применения, подтверждающий соответствие показателей качества программного изделия требованиям установленным в нормативно-технических документах на него и обеспечивающий документальную гарантию использования программного средства при соблюдении заданных ограничений.

Сертификация программного обеспечения КС возможна при выполнении следующих условий:

- разработке шкалы показателей качества с учетом специфики целевого использования программных средств и набора их функциональных характеристик;
- каталогизации программ, как объекта сертификации на основе опыта их эксплуатации;
- создании специализированного центра сертификации, выполняющего роль «третейской» организации контроля качества;
- разработке нормативно-технической базы, регламентирующей сертификацию программного обеспечения;
- разработке эталонов программных средств, которые удовлетворяют требованиям технических заданий на разработку разнотипных программных комплексов;
- разработке специальной технологии испытаний, определяющей объем и содержание сертификационных испытаний;
- реализации комплекса тестового программного обеспечения для широкого спектра программных изделий.

В процессе сертификации сложного ПО следует выделить два аспекта: *методический* и *технологический*. Методический аспект связан с

разработкой комплекса методик сертификации программного обеспечения с учетом специфики его применения, а технологический с автоматизацией процесса применения методического аппарата.

Следует отметить, что по некоторым оценкам до 70% общих затрат на создание и внедрение сложных программных комплексов приходится на реализацию процесса их сертификации. Причем значительная доля этих затрат относится к организации аппаратно-программной платформы, моделирующих средств и тестового обеспечения стенда сертификации.

Кроме того, важнейшим вопросом создания качественных программных изделий является обеспечение технологической безопасности ПО на этапе сертификационных стендовых испытаний. Недостаточный уровень развития современных информационных технологий разработки ПО, доминирующее использование зарубежных инструментальных средств и применение разработчиками программ лишь средств защиты от непреднамеренных дефектов обуславливают существенные, принципиально новые изменения технологии создания программ в этих условиях. Поэтому, *одной из задач сертификации на современном уровне развития информационных технологий становится выявление преднамеренных программных дефектов*.

Технологическая безопасность на этапе сертификационных испытаний характеризуется усилением мер контроля, так как в настоящее время предполагается, что вероятность внедрения закладок на окончательных фазах разработки программ выше, чем на начальных фазах в связи со снижением вероятности их обнаружения при последовательном технологическом контроле. В связи с этим завершающей процедурой тестового контроля и испытаний программ должна стать сертификация ПО по требованиям безопасности с выпуском сертификата на соответствие этого изделия требованиям технического задания. В условиях существующих технологий создания ПО сертификация программ является наиболее дешевым и быстро реализуемым способом «фильтрации» компьютерных систем от низкокачественных, не отвечающих условиям безопасности программных средств.

Сертификационные испытания программных средств, в том числе защищенных программных средств и программных средств контроля защищенности проводятся в государственных и отраслевых сертификационных центрах (испытательных лабораториях).

Право на проведение сертификационных испытаний защищенных средств вычислительной техники, в том числе программных средств

предоставляется Гостехкомиссией России по согласованию с Госстандартом России предприятиям-разработчикам защищенных СВТ, специализированным организациям ведомств, разрабатывающих защищенные СВТ, в том числе программные средства.

В соответствии с «Положением о сертификации продукции по требованиям безопасности информации» и «Положением о сертификации средств защиты информации» (см., например, [Зак]) по результатам сертификационных испытаний оформляется акт, а разработчику выдается сертификат, заверенный в Гостехкомиссии России и дающий право на использование и распространение этих средств как защищенных.

Средства, получившие сертификат, включаются в номенклатуру защищенных СВТ, в том числе программных средств.

Разработанные программные средства после их приемки представляются для регистрации в специализированный фонд Государственного фонда алгоритмов и программ.

Практические аспекты проведения сертификационных испытаний ПО приведены в приложении.

15.3. Безопасность программного обеспечения и человеческий фактор. Психология программирования.

15.3.1. Человеческий фактор

Преднамеренные и непреднамеренные нарушения безопасности программного обеспечения компьютерных систем большинство отечественных и зарубежных специалистов связывают с деятельностью человека. При этом технические сбои аппаратных средств КС, ошибки программного обеспечения и т.п. часто рассматриваются лишь как второстепенные факторы, связанные с проявлением угроз безопасности.

С некоторой степенью условности злоумышленников в данном случае можно разделить на два основных класса:

- *злоумышленники-любители (будем называть их хакерами);*
- *злоумышленники-профессионалы.*

Хакеры - это люди, увлеченные компьютерной и телекоммуникационной техникой, имеющие хорошие навыки в программировании и довольно любознательные. Их деятельность в большинстве случаев не приносит особого вреда компьютерным системам.

Ко второму классу можно отнести отечественные, зарубежные и международные криминальные сообщества и группы, а также

правительственные организации и службы, которые осуществляют свою деятельность в рамках концепции «информационной войны». К этому же классу можно отнести и сотрудников самих предприятий и фирм, ведущих разработку и эксплуатацию программного обеспечения.

Хакеры и группы хакеров

Хакеры часто образуют небольшие группы. Иногда эти группы периодически собираются, а в больших городах хакеры и группы хакеров встречаются регулярно. Но основная форма взаимодействия осуществляется через Интернет, а ранее - через электронные доски BBS. Как правило, каждая группа хакеров имеет свой определенный (часто критический) взгляд на другие группы. Хакеры часто прячут свои изобретения от хакеров других групп и даже от соперников в своей группе.

Существуют несколько типов хакеров. Это хакеры, которые:

- стремятся проникнуть во множество различных компьютерных систем (маловероятно, что такой хакер объявится снова после успешного проникновения в систему);
- получают удовольствие, оставляя явный след того, что он проник в систему;
- желают воспользоваться оборудованием, к которому ему запрещен доступ;
- охотятся за конфиденциальной информацией;
- собираются модифицировать определенный элемент данных, например баланс банка, криминальную запись или экзаменационные оценки;
- пытаются нанести ущерб «вскрытой» (обезоруженной) системе.

Группы хакеров, с некоторой степенью условности, можно разделить на следующие:

- группы хакеров, которые «получают удовольствие» от вторжения и исследования больших ЭВМ, которые используются в различных государственных учреждениях;
- группы хакеров, которые специализируются на телефонной системе;
- группы хакеров - коллекционеров кодов, - это хакеры, запускающие перехватчики кода, которые ищут карты вызовов (calling card) и номера PBX (private branch exchange - частная телефонная станция с выходом в общую сеть);
- группы хакеров, которые специализируются на операциях в финансово-кредитной сфере (Они используют компьютеры для кражи денег, вычисления номеров кредитных карточек и другой ценной информации, а затем продают свои услуги и методы другим, включая членов организованных преступных группировок. Эти хакеры могут скупать у коллекционеров кодов номера PBX и продавать их за 200-500$, и подобно другим видам информации неоднократно. Архивы кредитных бюро, информационные срезы баз данных уголовных архивов правоохранительных органов и баз данных других государственных учреждений также представляют для них большой интерес. Хакеры в этих группах, как правило, не находят взаимопонимания с другими хакерами);
- группы хакеров, которые специализируются на сборе и торговле пиратским программным обеспечением.

Типовой потрет хакера

Ниже приводится два обобщенных портрета хакера, один составлен по данным работы [CM] и характеризует скорее зарубежных хакеров-любителей, в то время как второй - это обобщенный портрет отечественного злонамеренного хакера, составленный Экспертно-криминалистическим центром МВД России [Сы].

В первом случае отмечается, что многие хакеры обладают следующими особенностями [CM]:

- *мужчина*: большинство хакеров - мужчины, как и большинство программистов;
- *молодой*: большинству хакеров от 14 до 21 года, и они учатся в институте или колледже. Когда хакеры выходят в деловой мир в качестве программистов, их программные проекты источают

большую часть их излишней энергии, и корпоративная обстановка начинает менять их жизненную позицию. Возраст компьютерных преступников показан на рис. 15.1 [CM];

- *сообразительный*: хакеры часто имеют коэффициент интеллекта выше среднего. Не смотря на свой своеобразный талант, большинство из них в школе или колледже не были хорошими учениками. Например, большинство программистов пишут плохую документацию и плохо владеют языком;
- *концентрирован на понимании, предсказании и управлении*: эти три условия составляет основу компетенции, мастерства и самооценки и стремительные технологические сдвиги и рост разнообразного аппаратного и программного обеспечения всегда будут вызовом для хакеров;
- *увлечен компьютерами*: для многих пользователей компьютер - это необходимый рабочий инструмент. Для хакера же - это «удивительная игрушка» и объект интенсивного исследования и понимания;
- *отсутствие преступных намерений*: по данным [CM] лишь в 10% рассмотренных случаев компьютерной преступности нарушения, совершаемые хакерами, привели к разрушению данных компьютерных систем. В связи с этим можно предположить, что менее 1% всех хакеров являются злоумышленниками.

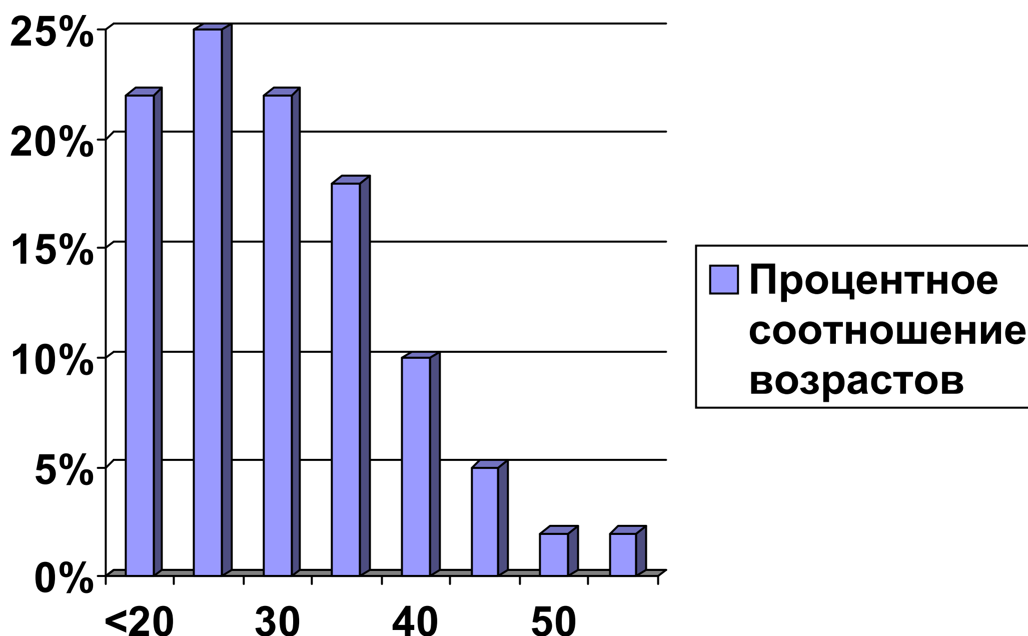


Рис. 15.1. Возрастное распределение обнаруженных компьютерных преступников

Обобщенный портрет отечественного хакера выглядит следующим образом [Сы]: это мужчина в возрасте от 15 до 45 лет, либо имеющий многолетний опыт работы на компьютере, либо почти не обладающий таким опытом; в прошлом к уголовной ответственности не привлекался; является яркой, мыслящей личностью, способной принимать ответственные решения; хороший, добросовестный работник; по характеру нетерпимый к насмешкам и к потере своего социального статуса в рамках окружающей его группы людей; любит уединенную работу; приходит на службу первым и уходит последним; часто задерживается на работе после окончании рабочего дня и очень редко использует отпуска и отгулы.

Криминальные сообщества и группы, сценарий взлома компьютерной системы

В связи со стремительным ростом информационных технологий и разнообразных компьютерных и телекоммуникационных средств и систем, наблюдается экспоненциальный рост как количества компьютерных атак, так и объем нанесенного от них ущерба. Анализ показывает, что такая тенденция постоянно сохраняется [Ка1].

За последнее время в нашей стране не отмечено ни одного компьютерного преступления, которое было бы совершено одиночкой [Сы]. Более того, известны случаи, когда организованными преступными группировками нанимались бригады из десятков хакеров. Им предоставлялись отдельные охраняемые помещения, оборудованные самыми передовыми компьютерными средствами и системами для проникновения в компьютерные сети коммерческих банков.

Специалисты правоохранительных органов России неоднократно отмечали тот факт, что большинство компьютерных преступлений в банковской сфере совершается при непосредственном участии самих служащих коммерческих банков. Результаты исследований, проведенных с привлечением банковского персонала, показывают, что доля таких преступлений приближается к отметке 70%. При осуществлении попытки хищения 2 млрд. рублей из филиала одного крупного коммерческого банка преступники оформили проводку фиктивного платежа с помощью удаленного доступа к компьютеру через модем, введя пароль и

идентификационные данные, которые им передали сообщники из состава персонала этого филиала. Далее эти деньги были переведены в соседний банк, где преступники попытались снять их со счета, оформив поддельное платежное поручение [Сы].

По данным Экспертно-криминалистического центра МВД России принципиальный сценарий взлома защитных механизмов банковской компьютерной системы представляется следующим. Компьютерные злоумышленники-профессионалы обычно работают только после тщательной предварительной подготовки. Они снимают квартиру на подставное лицо в доме, в котором не проживают сотрудники ФСБ, МВД или МГТС. Подкупают сотрудников банка, знакомых с деталями электронных платежей и паролями, и работников телефонной станции, чтобы обезопасить себя на случай поступления запроса от службы безопасности банка. Нанимают охрану из бывших сотрудников МВД. Чаще всего взлом банковской компьютерной системы осуществляется рано утром, когда дежурный службы безопасности теряет свою бдительность, а вызов помощи затруднен.

Злоумышленники в профессиональных коллективах программистов-разработчиков

Согласно существующей статистики в коллективах людей занятых той или иной деятельностью, как правило, только около 85% являются вполне лояльными (честными), а остальные 15% делятся примерно так: 5% - могут совершить что-нибудь противоправное, если, по их представлениям, вероятность заслуженного наказания мала; 5% - готовы рискнуть на противоправные действия, даже если шансы быть уличенным и наказанным складываются 50 на 50; 5% - готовы пойти на противозаконный поступок, даже если они почти уверены в том, что будут уличены и наказаны. Такая статистика в той или иной мере может быть применима к коллективам, участвующим в разработке и эксплуатации информационно-технических составляющих компьютерных систем.

Таким образом, можно предположить, что не менее 5% персонала, участвующего в разработке и эксплуатации программных комплексов, способны осуществить действия криминального характера из корыстных побуждений либо под влиянием каких-нибудь иных обстоятельств.

По другим данным [CM] считается, что от 80 до 90% компьютерных нарушений являются внутренними, в частности считается, что на каждого

«... подлого хакера приходится один обозленный и восемь небрежных работников, и все они могут производить разрушения изнутри».

15.3.2. Информационная война

В настоящее время за рубежом в рамках создания новейших оборонных технологий и видов оружия активно проводятся работы по созданию так называемых *средств нелетального воздействия* [Ка1]. Эти средства позволяют без нанесения разрушающих ударов (например, современным оружием массового поражения) по живой силе и технике вероятного противника выводить из строя и/или блокировать его вооружение и военную технику, а также нарушать заданные стратегии управления войсками.

Одним из новых видов оружия нелетального воздействия является *информационное оружие*, представляющее собой совокупность средств поражающего воздействия на информационный ресурс противника. Воздействию информационным оружием могут быть подвержены, прежде всего, компьютерные и телекоммуникационные системы противника. При этом *центральными объектами воздействия* являются *программное обеспечение*, структуры данных, средства вычислительной техники и обработки информации, а также каналы связи.

Появление информационного оружия приводит к изменению сущности и характера современных войн и появлению нового вида вооруженного конфликта - *информационная война*.

Несомненным является то, что информационная война, включающая *информационную борьбу в мирное и военное время*, изменит и характер военной доктрины ведущих государств мира. Многими зарубежными странами привносится в доктрину концепция выигрывать войны, сохраняя жизни своих солдат, за счет технического превосходства.

Ввиду того, что в мировой практике нет прецедента ведения широкомасштабной информационной войны, а имеются лишь некоторые прогнозы и зафиксированы отдельные случаи применения информационного оружия в ходе вооруженных конфликтов и в процессе деятельности крупных государственных и коммерческих организаций [Ка1], анализ содержания информационной войны за рубежом возможен по отдельным публикациям, так как, по некоторым данным информация по этой проблеме за рубежом строго засекречена.

Анализ современных методов ведения информационной борьбы позволяет сделать вывод о том, что к прогнозируемым формам информационной войны можно отнести следующие:

- глобальная информационная война;
- информационные операции;
- преднамеренное изменение замысла стратегической и тактической операции;
- дезорганизация жизненно важных для страны систем;
- нарушение телекоммуникационных систем;
- обнуление счетов в международной банковской системе;
- уничтожение (искажение) баз данных и знаний важнейших государственных и военных объектов.

К методам и средствам информационной борьбы в настоящее время относят:

- воздействие боевых компьютерных вирусов и преднамеренных дефектов диверсионного типа;
- несанкционированный доступ к информации;
- проявление непреднамеренных ошибок ПО и операторов компьютерных систем (в т.ч. за счет использования средств информационно-психологического воздействия на личный состав);
- воздействие радиоэлектронными излучениями;
- физические разрушения систем обработки информации.

Таким образом, в большинстве развитых стран мира в рамках концепции информационной войны разрабатывается совокупность разнородных средств, которые можно отнести к информационному оружию. Такие средства могут использоваться в совокупности с другими боевыми средствами во всех возможных формах ведения информационной войны. Кроме существовавших ранее средств поражающего воздействия в настоящее время разрабатываются принципиально новые средства информационной борьбы, а именно бое вые компьютерные вирусы и преднамеренные программные дефекты диверсионного типа.

15.3.3. Психология программирования

При создании высокоэффективных, надежных и безопасных программ (программных комплексов), отвечающих самым современным требованиям к их разработке, эксплуатации и модернизации необходимо не только умело пользоваться предоставляемой вычислительной и

программной базой современных компьютеров, но и учитывать интуицию и опыт разработчиков языков программирования и прикладных систем. Помимо этого, целесообразно дополнять процесс разработки программ экспериментальными исследованиями, которые основываются на применении концепции психологии мышления при исследовании проблем вычислительной математики и информатики. Такой союз вычислительных, информационных систем и программирования принято называть психологией программирования.

Психология программирования - это наука о действиях человека, имеющего дело с вычислительными и информационными ресурсами автоматизированных систем, в которой знания о возможностях и способностях человека как разработчика данных систем могут быть углублены с помощью методов экспериментальной психологии, анализа процессов мышления и восприятия, методов социальной, индивидуальной и производственной психологии.

К целям психологии программирования наряду с улучшением использования компьютера, основанного на глубоком знании свойств мышления человека, относится и определение, как правило, экспериментальным путем, склонностей и способностей программиста как личности. Особенности личности играют критическую роль в определении (исследовании) рабочего стиля отдельного программиста, а также особенностей его поведения в коллективе разработчиков программного обеспечения. Ниже приводится список характеристик личности и их предполагаемых связей с программированием. При этом особое внимание уделяется тем личным качествам программиста, которые могут, в той или иной степени, оказать влияние на надежность и безопасность разрабатываемого им программного обеспечения.

Внутренняя/внешняя управляемость. Личности с выраженной внутренней управляемостью стараются подчинять себе обстоятельства и убеждены в способности сделать это, а также в способности повлиять на свое окружение и управлять событиями. Личности с внешней управляемостью (наиболее уязвимы с точки зрения обеспечения безопасности программного обеспечения) чувствуют себя жертвами не зависящих от них обстоятельств и легко позволяют другим доминировать над ними.

Высокая/низкая мотивация. Личности с высокой степенью мотивации способны разрабатывать очень сложные и сравнительно надежные программы. Руководители, способные повысить уровень мотивации, в то

же время, могут стимулировать своих сотрудников к созданию программ с высоким уровнем их безопасности.

Умение быть точным. На завершающих этапах составления программ необходимо особое внимание уделять подробностям и готовности проверить и учесть каждую деталь. Это позволит повысить вероятность обнаружения программных дефектов как привнесенных в программу самим программистом (когда нарушитель может ими воспользоваться в своих целях), так и другими программистами (в случае, если некоторые из них могут быть нарушителями) при создании сложных программных комплексов коллективом разработчиков.

Кроме того, психология программирования изучает, с точки зрения особенностей создания безопасного программного обеспечения, такие характеристики качества личности как исполнительность, терпимость к неопределенности, эгоизм, степень увлеченности, склонность к риску, самооценку программиста и личные отношения в коллективе.

Корпоративная этика

Особый психологический настрой и моральные стимулы программисту может создать особые корпоративные условия его деятельности, в частности различные моральные обязательства, оформленные в виде кодексов чести. Ниже приводится «Кодекс чести пользователя компьютера» [CM].

- *Обещаю не использовать компьютер в ущерб другим людям.*
- *Обещаю не вмешиваться в работу компьютера других людей.*
- *Обещаю «не совать нос» в компьютерные файлы других людей.*
- *Обещаю не использовать компьютер для воровства.*
- *Обещаю не использовать компьютер для лжесвидетельства.*
- *Обещаю не копировать и не использовать чужие программы, которые были оплачены не мною.*
- *Обещаю не использовать компьютерные ресурсы других людей без разрешения и соответствующей компенсации.*
- *Обещаю не присваивать результаты интеллектуального труда других людей.*

- *Обещаю думать об общественных последствиях разрабатываемых мною программ или систем.*
- *Обещаю всегда использовать компьютер с наибольшей пользой для живущих ныне и будущих поколений.*

ЗАКЛЮЧЕНИЕ

Количество и уровень деструктивности угроз безопасности для программных комплексов компьютерных систем как со стороны внешних, так и со стороны внутренних источников угроз постоянно возрастает. Это объясняется стремительным развитием компьютерных и телекоммуникационных средств, глобальных информационных систем, необходимостью разработки для них сложного программного обеспечения с применением современных средств автоматизации процесса проектирования программ. Кроме того, это объясняется значительным или даже резким повышением в последнее время активности деятельности хакеров и групп хакеров, атакующих компьютерные системы, криминальных групп компьютерных взломщиков, различных специальных подразделений и служб, осуществляющих свою деятельность в области создания средств воздействия на критически уязвимые объекты информатизации.

В настоящей работе излагаются теоретические и научно-практические основы обеспечения безопасности программного обеспечения компьютерных систем, рассматриваются методы и средства защиты программного обеспечения от воздействия разрушающих программных средств на различных этапах его жизненного цикла, а также защита от несанкционированных копирования, распространения и использования.

Значительная часть материала посвящена выявлению и анализу угроз безопасности программного обеспечения, рассмотрению основ формирования моделей угроз и их вербальному описанию, методов и средств обеспечения технологической и эксплуатационной безопасности программ.

Необходимой составляющей проблемы обеспечения информационной безопасности программного обеспечения является общегосударственная система стандартов и других нормативных и методических документов по безопасности информации, а также международные стандарты и рекомендации по управлению качеством программного обеспечения (ознакомиться с большей их частью можно в главе 15 и приложении), которые позволяет предъявить к создаваемым и эксплуатируемым программных комплексам требуемый уровень реализации защитных функций.

Изложенный материал, по мнению автора, позволит при изучении технологии проектирования защищенного программного обеспечения

избежать многих ошибок, которые могут существенно повлиять на качество проекта и эффективность конечной системы в целом при ее реализации на объектах информатизации различного назначения.

Файл взят с сайта - http://www.natahaus.ru/

где есть ещё множество интересных и редких книг, программ и прочих вещей.
Данный файл представлен исключительно в ознакомительных целях.
Уважаемый читатель!
Если вы скопируете его,
Вы должны незамедлительно удалить его
 сразу после ознакомления с содержанием.
Копируя и сохраняя его Вы принимаете на себя всю ответственность, согласно действующему международному законодательству .
Все авторские права на данный файл сохраняются за правообладателем.
Любое коммерческое и иное использование кроме предварительного ознакомления запрещено.
Публикация данного документа не преследует за собой никакой коммерческой выгоды. Но такие документы способствуют быстрейшему профессиональному и духовному росту читателей и являются рекламой бумажных изданий таких документов.

Все авторские права сохраняются за правообладателем.
Если Вы являетесь автором данного документа и хотите дополнить его или изменить, уточнить реквизиты автора или опубликовать другие документы, пожалуйста, свяжитесь с нами по e-mail - мы будем рады услышать ваши пожелания.