О.В. КАЗАРИН

БЕЗОПАСНОСТЬ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ КОМПЬЮТЕРНЫХ СИСТЕМ

МОНОГРАФИЯ

Москва
2003

УДК 621.382.26

Казарин О.В.
Безопасность программного обеспечения компьютерных систем. Монография. – М.: МГУЛ, 2003. – 212 с.

В монографии рассмотрены теоретические и прикладные аспекты проблемы обеспечения безопасности программного обеспечения компьютерных систем различного назначения. Особое внимание уделено моделям и методам создания высокозащищенных и алгоритмически безопасных программ для применения в системах критических приложений.

Монография предназначена для ученых и практиков в области защиты информации, как специальной, так коммерческой и личной. 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.

Табл. 7. Ил. 19. Библ. 70 назв.

Рецензенты: *канд. техн. наук, с.н.с. И.В. Егоркин;*
канд. техн наук, с.н.с. В.Ю. Скиба

ISBN 5-283-01667

ПРЕДИСЛОВИЕ

Современный компьютерный мир представляет собой разнообразную и весьма сложную совокупность вычислительных устройств, систем обработки информации, телекоммуникационных технологий, программного обеспечения и высокоэффективных средств его проектирования. Вся эта многогранная и взаимосвязанная метасистема решает огромный круг проблем в различных областях человеческой деятельности, от простого решения школьных задач на домашнем персональном компьютере до управления сложными технологическими процессами.

Чем сложнее задача автоматизации и чем ответственнее область, в которой используются компьютерные информационные технологии, тем все более и более критичными становятся такие свойства как надежность и безопасность информационных ресурсов, задействованных в процессе сбора, накопления, обработки, передачи и хранения компьютерных данных.

Вредоносные воздействия на информацию в процессе функционирования компьютерных систем (КС) различного назначения осуществляется с целью нарушения ее конфиденциальности, целостности и доступности. Решение задач, связанных с предотвращением воздействия непосредственно на информацию, осуществляется в рамках комплексной проблемы обеспечения безопасности информации и имеет достаточно развитую научно-методическую базу. При этом, рассматривая информацию как активный эксплуатируемый ресурс, можно говорить о том, что процесс обеспечения безопасности информации включает в себя и обеспечение безопасности программного обеспечения КС. Данный аспект обеспечения безопасности информации и средств ее обработки именуется эксплуатационной безопасностью, так как соответствует этапу применения КС. В то же время, в последнее время появились новые проблемы обеспечения безопасности, связанные с информационными технологиями, которые, по мнению ряда зарубежных и отечественных экспертов в области их создания и применения, в значительной степени определяют эффективность создаваемых компьютерных систем.

Мировые исследования последних лет показали, что функциональные и надежностные характеристики КС определяются качеством и надежностью программного обеспечения, входящего в их состав. Кроме проблем качества и надежности программного обеспечения при создании КС фундаментальная проблема его безопасности приобретает все большую актуальность. При этом в рамках данной проблемы на первый план выдвигается безопасность технологий создания программного обеспечения компьютерных систем. Данный аспект проблемы безопасности программных ком-

плексов является сравнительно новым и связан с возможностью внедрения в тело программных средств на этапе их разработки (или модификации в ходе авторского сопровождения) так называемых «программных закладок». В связи с этим все более актуальным становится проблема обеспечения технологической безопасности программного обеспечения КС различного уровня и назначения.

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

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

В главах 2 и 3 рассмотрены современные методы обеспечения технологической и эксплуатационной безопасности программ соответственно. Важное место отводится методам создания алгоритмически безопасного программного обеспечения, позволяющим выявлять и устранять программные дефекты деструктивного характера, как на этапе создания, так и на этапе применения программ.

В гл. 4 рассматриваются вопросы стандартизации в области безопасности программного обеспечения и проведения его сертификационных испытаний. Кроме того, рассматриваются особенности поведения программиста – разработчика, который может осуществлять, в том числе, и действия диверсионного типа.

ГЛАВА 1. ВВЕДЕНИЕ В ТЕОРИЮ ОБЕСПЕЧЕНИЯ БЕЗОПАСНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1. Зачем и от кого нужно защищать программное обеспечение компьютерных систем

Безопасность *программного обеспечения* (ПО) в широком смысле является свойством данного ПО функционировать без проявления различных негативных последствий для конкретной компьютерной системы. Под уровнем безопасности 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 ПО. 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 КС.

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 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 иден-

тификационных признаков программы (в связи с необходимостью получения программы «схожей» с оригиналом). Таким образом, манипуляции подобного рода могут сделать и 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. раздел 4.4). 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;
- 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 и 3).

1.2. Угрозы безопасности программного обеспечения и примеры их реализации в современном компьютерном мире

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.

Наибольшее распространение *компьютерные вирусы* получили с развитием персональных ЭВМ (ПЭВМ) и появлением микропроцессоров фирмы Intel. Это обусловлено тем, что для ПЭВМ наиболее распростра-

ненными операционными системами (ОС) были и используются по настоящее время (в новых версиях) ОС MS-DOS и ОС Windows, которые по многим параметрам открыты и беззащитны к вирусной угрозе. В известных классификациях вирусов [3,59] более 90% от их общего числа составляют именно вирусы для среды этих операционных систем. Для наиболее распространенных современных сетевых и многозадачных операционных систем ряда Windows, OS/2 и клона Unix по сравнению с этим вирусов обнаружено не столь много.

В последние 10-15 лет компьютерные вирусы нанесли значительный ущерб, как отдельным средствам вычислительной техники, так и сложным телекоммуникационным системам различного назначения. Интенсивные проявления вирусных заражений начались примерно в середине 80-х годов. Так, с 1986 по 1989 год было зарегистрировано 450 случаев проникновения через сеть INTERNET компьютерных вирусов в сеть Министерства обороны США DDN, из которых 220 были классифицированы как успешные. Только стоимость операций по выявлению источников вирусных атак в DDN превысила 100 тысяч долларов. Факты попыток проникновения с использованием компьютерных вирусов в информационные банки критических систем в первой половине 80-х были зарегистрированы в различных сетях США, Франции, Великобритании, ФРГ, Израиля, Пакистана и Японии. По мнению исследователей, после заражения одной ЭВМ в сети среднее время заражения следующего узла сети составляет от 10 до 20 минут. При такой интенсивности размножения некоторые вирусы способны за несколько часов вывести из строя всю сеть.

Классическим примером широкомасштабной вирусной угрозы является известный вирус Морриса, выведший 21 ноября 1988 на 24 часа из строя сеть ARPARNET. Промышленная ассоциация компьютерных вирусов (Computer Virus Industry Association - CVIA) выполнила детальный анализ расходов, связанных с действием этого вируса, заразившего 7,3% или 6200 из 85200 компьютеров сети. Пользователи потеряли свыше 8 млн. часов рабочего времени, а операторы и администраторы сети потратили около 1,13 млн. человеко-часов на то, чтобы привести сеть в рабочее состояние. Расходы от потерянной возможности доступа в сеть и средства, затраченные на ее восстановление, составили 98 млн. долларов. К декабрю 1988 г. в Ливерморской лаборатории США (Lawrence Livermore National Laboratories), которая занимается, в том числе, разработкой ядерного оружия 3-го поколения, было зафиксировано не менее 10 попыток проникновения в сеть лаборатории через каналы связи со Стэндфордским университетом, университетом штата Вашингтон и через сеть INTERNET. В результате было поражено 800 компьютеров. В том же году было зафиксировано 200 попыток заражения (из них 150 - успешных) глобальной компьютерной се-

ти NASA. Причем 16 мая в течение 7 часов было заражено 70 ЭВМ, а после заражения в них была создана специальная программа для облегчения проникновения в сеть в будущем. Наиболее характерные исторические примеры проявления компьютерных вирусов и тенденции их роста в настоящее время можно найти в таблице 1.1 и на рис.1.1.

В качестве основных средств вредоносного (деструктивного) воздействия на КС необходимо, наряду с другими средствами информационного воздействия, *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.

Таблица 1.1

Год	События, цифры, факты
21.11. 1988	Вирус Морриса на 24 часа вывел из строя сеть ARPANET. Ущерб составил 98 млн. долларов.
1988	Зафиксировано 200 попыток заражения вирусами (150 - успешных) глобальной компьютерной сети NASA. Причем 16 мая в течение 7 часов было заражено 70 ЭВМ.
с 20.03. по 9.09. 1988	Промышленная ассоциация компьютерных вирусов (Computer Virus Industry Association - CVIA) зарегистрировала 61795 случаев заражения вирусами различных информационных систем по всему миру.
1986 - 1989	Зарегистрировано 450 случаев попыток НСД и заражения вирусами (220 - успешные) сети МО США DDN. Длительность цикла проникновения и выборки информации не превышала 1 мин.
1992	В США было заражено чуть более одного из каждых десяти офисных компьютеров (данные для более, чем 60000 ПЭВМ фирм Mac, Atari, Amiga, PC)
1994	Национальная аудиторская служба Великобритании (National Audit Office - NAO) зарегистрировала 562 случая заражения вирусами компьютерных систем британских правительственных организаций, что в 3.5 раза превышает уровень 1993 г.
1995	В космическом центре Джонсона NASA зарегистрировано 52 случая заражения компьютеров Mac и PC вирусам. Время, затраченное на их устранение, составило более 350 часов
1995	Появление макровирусов. Изменение динамики процентного содержания макровирусов в общем числе компьютерных вирусов с 16% в январе 1995 г. до 44% в ноябре 1995 г.
1995	В сети Bitnet (международная академическая сеть) за 2 часа вирус, замаскированный под рождественское поздравление, заразил более 500 тысяч компьютеров по всему миру, при этом сеть IBM прекратила вообще работу на несколько часов.
Декабрь 1996	Компьютерная атака на WebCom (крупнейшего провайдера услуг WWW в США) вывела из строя на 40 часов больше 3000 абонентских пунктов WWW. Атака представляла собой «синхронный поток», которая блокирует функционирование сервера и приводит к «отказу в обслуживании». Поиск маршрута атаки длился 10 часов.

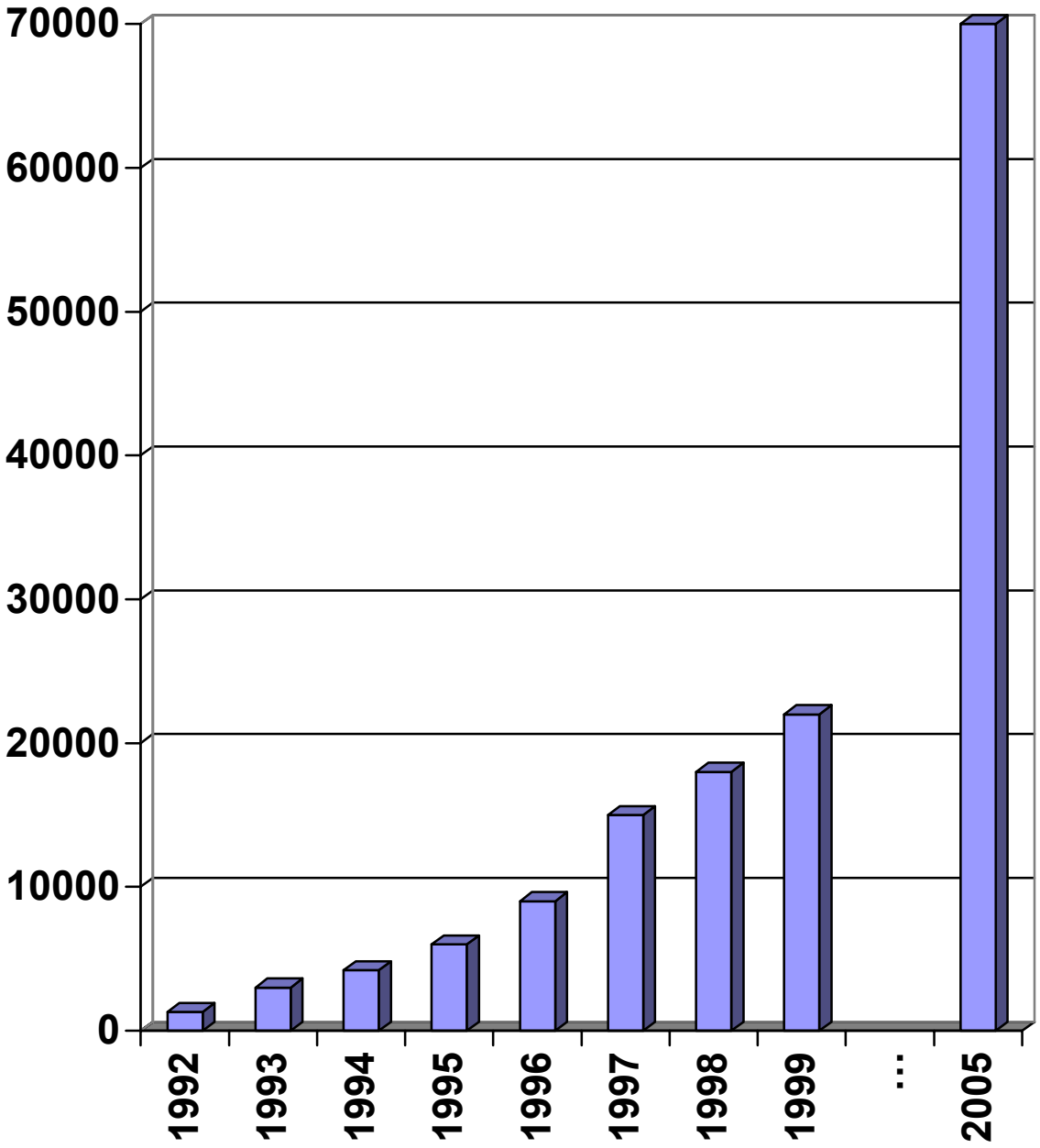

Рис. 1.1. График роста компьютерных вирусов

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 [44]. 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.

1.3. Базовые научные дисциплины, принятая аксиоматика и терминология

1.3.1. Теоретические основы

Развитие теоретических основ создания и применения сложных систем управления существенно зависит от темпов развития информатики - комплекса наук, изучающих закономерности проявления и движения информации, ее источники, информационные потоки, информационные процессы в различных областях деятельности человека, методы и средства накопления, обработки и передачи информации с помощью ЭВМ и других

технических средств. В то же время, информатика является лишь обеспечивающим звеном в общей системе научных основ управления, которые опираются на теоретический фундамент кибернетики. Кибернетический подход к созданию систем управления организационно-технического типа состоит в том, чтобы в необходимой мере были учтены информационные и динамические свойства всех элементов системы, функционирующей на основе принципов обратной связи, для выполнения целевых задач.

Теоретическим базисом информатики является теория информации, которая входит на правах самостоятельного раздела в кибернетику и включает методы математического описания и исследования информационных процессов различной природы, методы передачи, обработки, хранения, извлечения и классификации информации в различных областях деятельности. Информатика включает в себя методы теории информации, но применительно к техническим системам ее обработки.

Создание любой компьютерной системы невозможно без разработки и оптимизации алгоритмического обеспечения. Основой создания алгоритмического обеспечения является теория алгоритмов - раздел математики, изучающий процедуры (алгоритмы) вычислений и математические объекты, которые могут быть определены на базе методов теорий множеств, отношений и функционалов. Причем наиболее важными для решений проблем функционирования программного обеспечения КС являются такие разделы этой теории, как теория вычислительных процедур и теория сложности алгоритмов. Теория вычислительных процедур изучает различные классы автоматов с точки зрения их алгоритмических возможностей, а также процессы вычислений, порождаемые автоматами различных классов. Теория сложности алгоритмов изучает методы получения оценок сложности вычислений различных функций и классов функций. Эти две теории послужили основой создания в рамках информатики раздела, именуемого прикладной теорией алгоритмов и изучающего математические модели дискретных систем, входящих в состав систем управления. Прикладная теория алгоритмов в значительной мере определяет основное содержание процессов исследования функциональных свойств и характеристик элементов систем управления, а также может использоваться для синтеза алгоритмического обеспечения комплексов средств автоматизации, входящих в состав компьютерных систем.

Построение сложных программных комплексов для КС предполагает необходимость исследования не только методов повышения качества программного обеспечения, но и методов обеспечения его надежности на всех этапах жизненного цикла КС. Это обстоятельство обуславливает использование при исследовании и разработке КС методов теории управления качеством ПО и методов обеспечения надежности программ. Теория обеспече-

ния качества программного обеспечения изучает методы подтверждения его качества и связанной с ним документации с целью получения гарантии, что создаваемые и используемые программы во всех отношениях соответствует своему назначению. Теория надежности программного обеспечения, являясь разделом теории надежности технических систем, изучает вероятностные и сложностные аспекты способности программ выполнять возложенные на нее функции при поступлении требований на их выполнение.

Совокупность этих теорий, хотя и охватывает различные области организации процессов разработки и эксплуатации компьютерных систем позволяет, тем не менее, структурировать проблему обеспечения безопасности программных комплексов, проводить анализ процессов во всех аспектах функционирования компьютерной системы (морфологическом, функциональном, информационном и прагматическом). Следовательно, возникает необходимость изучения научного направления, именуемого - *теорией обеспечения безопасности программ и их комплексов*, интегрирующей основные научные положения прикладной теории алгоритмов, теории управления качеством программного обеспечения, теории надежности и с единых системных позиций изучающей методы предотвращения случайного или преднамеренного раскрытия, искажения или уничтожения хранимой, обрабатываемой и передаваемой информации в компьютерных системах, а также предотвращения нарушения их функционирования.

1.3.2. 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 про-

граммам 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 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.

1.3.3. 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 безопасности информационного ресурса и получению секретных сведений.

Нарушитель (нарушители) - 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 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 поддержи-

ваемых 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 эксплуатации КС.

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 [36], 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 в зарубежных моделях, так как именно зарубежные программные средства являются наиболее вероятным носителем программных дефектов диверсионного типа.

Графическое представление моделей ЖЦ позволяет наглядно выделить их особенности и некоторые свойства процессов. Первоначально была создана каскадная модель ЖЦ [37], в которой крупные этапы начинались друг за другом с использованием результатов предыдущих работ. Наиболее специфической является спиралевидная модель ЖЦ [36]. 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.

Стандартизация ЖЦ ПО проводится по трем направлениям. Первое направление организуется и стимулируется Международной организацией по стандартизации (ISO - International Standard Organization) и Международной комиссией по электротехнике (IEC - International Electrotechnical Commission). На этом уровне осуществляется стандартизация наиболее общих технологических процессов, имеющих значение для международной кооперации. Второе направление активно развивается в США Институтом инженеров электротехники и радиоэлектроники (IEEE - Institute of Electrotechnical and Electronics Engineers) совместно с Американским национальным институтом стандартизации (American National Standards Institute-ANSI). Стандарты ISO/IEC и ANSI/IEEE в основном имеют рекомендательный характер. Третье направление стимулируется Министерством обороны США (Department of Defense-DOD). Стандарты DOD имеют обязательный характер для фирм, работающих по заказу Министерства обороны США.

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.

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 автоматизации разработки, которые целесообразно выбирать из имеющихся или создавать с учетом объекта разработки и адекватного ему перечня работ.

1.5. Модель угроз и принципы обеспечения безопасности программного обеспечения

Использование при создании программного обеспечения КС сложных операционных систем, инструментальных средств разработки ПО импортного производства увеличивают потенциальную возможность внедрения в программы преднамеренных дефектов диверсионного типа. Помимо этого, при создании целевого 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.

1.5.1. Подход к созданию модели угроз технологической безопасности ПО

Один из возможных подходов к созданию модели технологической безопасности ПО АСУ может основываться на обобщенной концепции технологической безопасности компьютерной инфосферы [19], 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 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.1.2.

Наполнение модели 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.1.3 приведен пример указанной типовой модели для сложных программных комплексов.

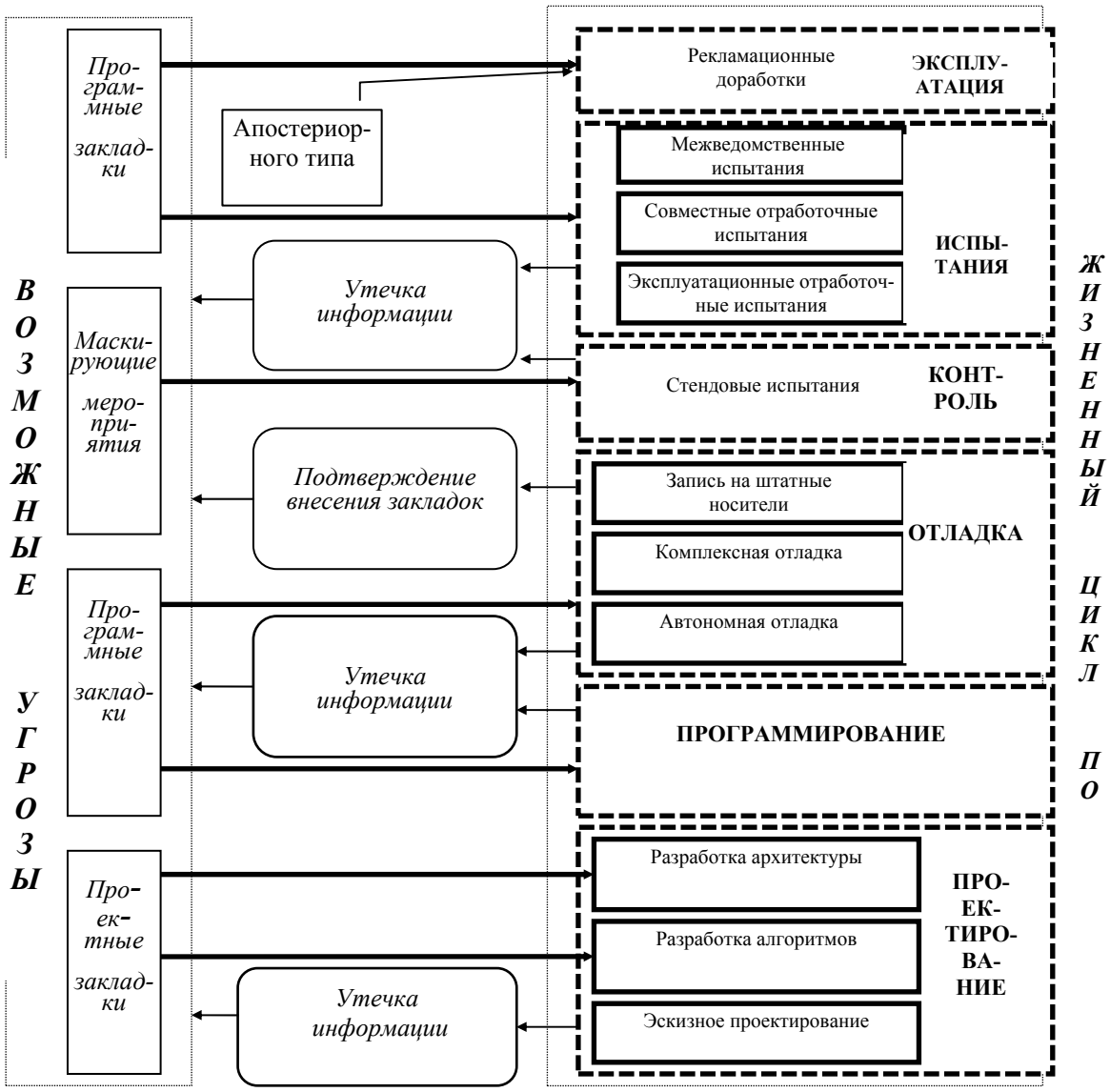

Рис.1.2. Схема угроз технологической безопасности ПО

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 реальными характеристиками.

Поставка вычислительных средств, имеющих высокий уровень экологической опасности.

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.1.3. Пример типовой модели 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.

Организация динамически формируемых команд или параллельных вычислительных процессов.

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.1.3.

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.

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.

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 программной закладки в ветвях программной системы, не проверяемых при контроле.

Формирование «вирусных» программ, не позволяющих выявить их внедрение в программную систему путем контрольного суммирования.

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

Продолжение рис.1.3.

ЭКСПЛУАТАЦИЯ

Сведения о персональном составе контролирующего подразделения и испытываемых программных системах

Внедрение злоумышленников в контролирующее подразделение.

Вербовка сотрудников контролирующего подразделения.

Сбор информации о 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.1.3.

1.5.2. 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.1.2.

Таблица 1.2

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 шифрования; скачки электропитания на технических средствах; старение носителей информации; разрушение информации под воздействием физических факторов (аварии 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»); при анализе 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; вывод из строя элементов физических средств защиты информации 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); визуальный канал; подслушивающие устройства; дистанционное фотографирование и т.п.	помехи; отключение электропитания; намеренный вызов случайных факторов.

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.

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 конструктива 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.1.4.

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.

Рис.1.4. Неформализованное описание модели угроз безопасности ПО на этапе исследований попыток несанкционированных действий в отношении информационных ресурсов КС.

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.

1.5.3. 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 конфиденциальный характер. Необходимо обеспечить постоянный, комплексный и действенный контроль за деятельностью разработчиков и пользователей компонентов. 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 безопасности 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 информационной безопасности ПО способности к диагностированию, опережающей нейтрализации, оперативному и эффективному устранению возникающих угроз в условиях резких изменений обстановки информационной борьбы.

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 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 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 АСУ 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 и ограничения доступа к эталонам программных средств, недопущение внесения изменений в них.

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.

ГЛАВА 2. ОБЕСПЕЧЕНИЕ ТЕХНОЛОГИЧЕСКОЙ БЕЗОПАСНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. Формальные методы доказательства правильности программ и их спецификаций

Традиционные методы анализа ПО связаны с доказательством правильности программ (верификация 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 [12].

```
integer r, dd;
  r:=a; dd:=d;
  while dd≤r do dd:=2*dd;
  while dd≠d do dd:=2*dd;
        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:

$$dd_n=d*2^n \qquad (2.1.1)$$

Кроме того, поскольку $d>0$, можно сделать вывод, что для любого конечного значения r отношение $dd_r>r$ 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

$$0 \leq r < dd \qquad (2.1.2)$$

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) следует, что

$$0 \leq r < d \qquad (2.1.3)$$

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

$$dd \equiv 0 (\mathrm{mod}\ d) \qquad (2.1.4)$$

Это следует, например, из того, что возможные значения dd имеют вид (см. (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

$$a \equiv r (\mathrm{mod}\ d) \qquad (2.1.5)$$

Оно выполняется после начальных присваиваний.

Повторяемый оператор первого заголовка (dd:=2*dd) сохраняет отношение (2.1.5), и поэтому весь первый цикл сохраняет отношение (2.1.5).

Повторяемый оператор из второго цикла состоит из двух операторов. Первый (dd:=2*dd) сохраняет инвариантность (2.1.5); второй тоже сохраняет отношение (2.1.5), так как он либо не изменяет значение r, либо уменьшает r на текущее dd, а эта операция в силу (2.1.4) также сохраняет справедливость отношения (2.1.5). Таким образом, весь повторяемый оператор второго цикла сохраняет отношение (2.1.5).

Объединяя отношения (2.1.3) и (2.1.5), получаем, что окончательное значение r удовлетворяет условиям $0 \le r < d$ и $a \equiv r \pmod d$, то есть r - наименьший неотрицательный остаток от деления a на d.

И хотя методы доказательства правильности программ существенно ограничены для практического использования, тем не менее есть области, где данные методы могут найти прикладное значение. Следующий пример характеризует это.

Большинство известных алгоритмов электронной цифровой подписи (например, [10,70]) 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 [36]) требует высокого быстродействия рабочих алгоритмов криптографических схем.

Представление чисел

Пусть A, N, e - три целых положительных числа многократной точности, причем $A<N$. Тогда для любого e при вычислении $A^e \pmod N \equiv C$, результат редукции $C \in \{1, N-1\}$. Если e представить n-разрядным двоичным вектором, то всю операцию возведения в степень можно свести к чередованию операций вида $A*B \bmod N$ и $B*B \bmod N$, где $0<B<N-1$ [36, 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 использует 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, ибо 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 $r=10$ [36, 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 [36, 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 $r=2^{\lambda}$.

*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 [66]. 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 [64] 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.2.1, а на рис.2.2 представлен псевдокод процедуры 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 малым. 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*]) $abs(P_i)<N$, $r^{m-1}\le N\le r^m$.

Доказательство теоремы 2.1. Доказательство теоремы проведем методом индукции.

Если *k*=abs(*p*_short) DIV *n*_short, где DIV - целочисленное деление, то

*p*_short=$(k+\delta)$**n*_short, (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

$\delta>0.5$ (2.1.7)

На *i*-том шаге алгоритм вычисляет:

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

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 $Pi=-P'+k*N$ (см. алгоритм), то согласно (2.1.7)

$P_i=\delta*N$, если $\delta\le0.5$ (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)

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. 2.1. Псевдокод алгоритма модулярного умножения $A*B$ modulo N

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

Вариант B:

$p\_short>0$ (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):

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

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

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 [36, стр.291-295]). Чем больше основание системы счисления r, тем ниже вероятность того, что пробное частное k от деления первых цифр больших чисел не будет соответствовать действительному частному.

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

2.2. Методы и средства анализа безопасности программного обеспечения

Широко известны различные средства программного обеспечения 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 [17,45] предлагают разделить методы, используемые для анализа и оценки безопасности ПО, на две категории: контрольно-испытательные и логико-аналитические (см. рис.2.3). 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, т.к. 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.

2.2.1. 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 [17]. Тестирование может проводиться с помощью тестовых запусков, исполнения в виртуальной программной среде, с помощью символического выполнения программы, ее интерпретации и другими методами.

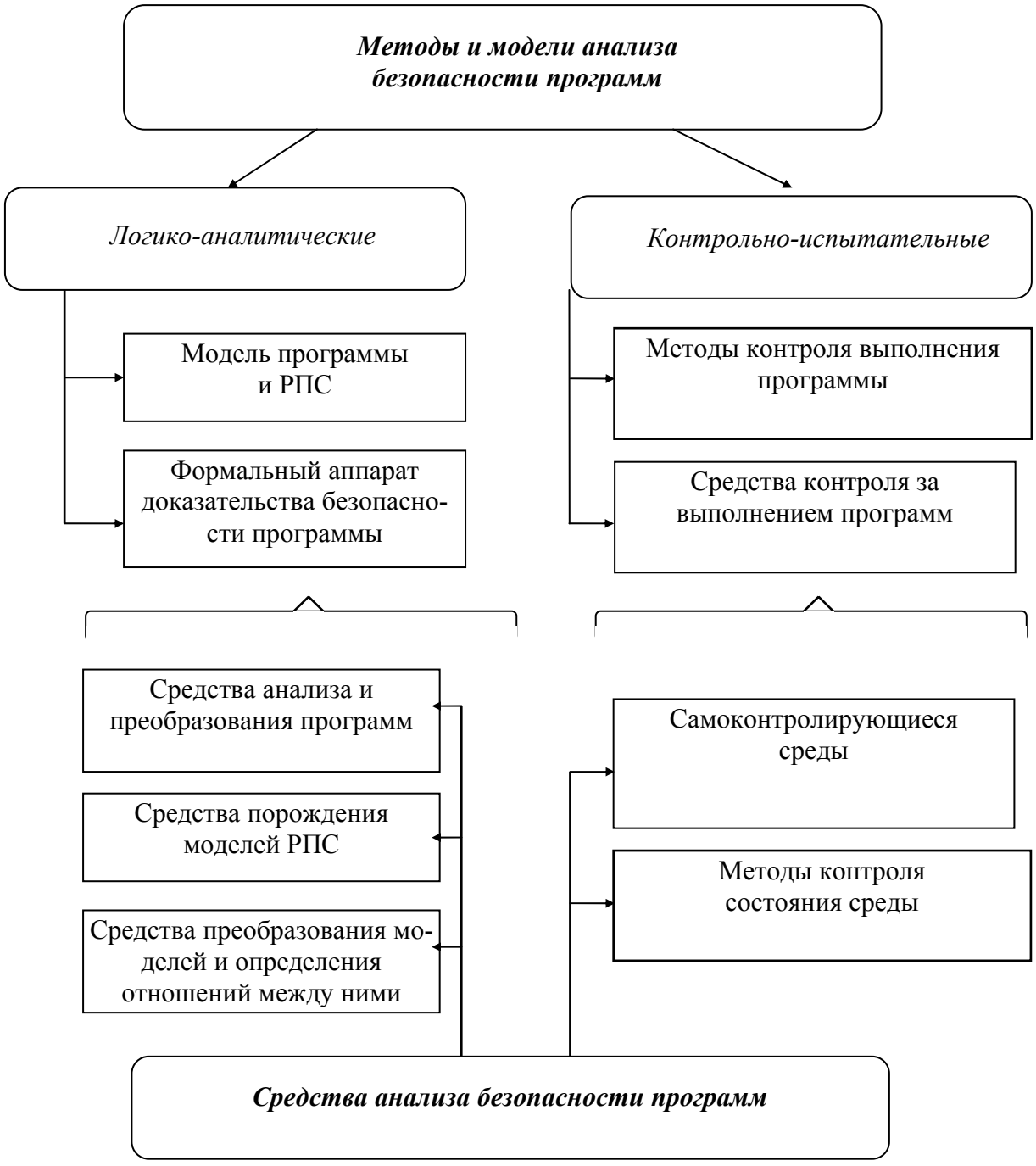

Рис. 2.3. Методы и средства анализа безопасности ПО

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, 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 систем защиты и разграничения доступа. Рассмотрим формальную постановку задачи анализа безопасности ПО для решения ее 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.2.4).

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.

2.2.2. Логико-аналитические методы контроля безопасности программ

При проведении анализа безопасности с помощью логико-аналитических методов (см. рис.2.5) 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.

Рис.2.4. Схема анализа безопасности ПО с помощью контрольно-испытательных методов

Рис. 2.5. Схема анализа безопасности ПО с помощью логико-аналитических методов

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 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 объекта исследования (программа), представленного в исполняемых кодах. При этом лексемами являются сигнатуры. В данном случае осуществляется поиск сигнатур следующих классов:

- сигнатуры вирусов;
- сигнатуры элементов РПС;
- сигнатуры (лексемы) «подозрительных функций»;
- сигнатуры штатных процедур использования системных ресурсов и внешних устройств.

Поиск лексем (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 методика сочетает в себе три перечисленных выше анализа. Выработанные критерии позволяют разумно сочетать различные виды анализа, существенно сокращая время исследования, не снижая его качества.

2.3. Методы обеспечения надежности программ для контроля их технологической безопасности

При исследовании методов и средств оценки уровня технологической безопасности программных комплексов учитываются факторы, имеющие, как правило, чисто случайный характер. Следовательно, показатели, свя-

занные с оцениванием безопасности ПО лучше всего выражать вероятностной мерой, а для их вычисления можно использовать вероятностные модели надежности ПО [56], которые при осуществлении замены условия правильности функционирования программ на условие их безопасности можно использовать для этих целей.

Исходные данные, определения и условия

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.

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.

Интуитивное определение безопасности ПО может быть уточнено в статистическом смысле на основе следующих простых соображений:

- машинная программа p 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: $E=(E_i:i=1,2,...,N)$;
- выполнение программы p приводит к получению для каждого Ei определенного значения функции $F(E_i)$;
- множество E определяет все возможные вычисления в программе p, то есть каждому набору входных данных E_i соответствует прогон программы p, и наоборот, каждому прогону соответствует некоторый набор входных данных E_i;
- наличие дефектов в программе p 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 уровень безопасности программного обеспечения $S(E_i)$, то есть безопасность обеспечивается при соблюдении ограничения: $F'(E_i)$, $S(E_i)$. (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).

Совокупность действий, включающая ввод E_i, выполнение программы p, которое заканчивается получением результата $F'(E_i)$ называется прогоном программы p. Необходимо также отметить, что значения входных переменных, образующие E_i, не должны все одновременно подаваться на вход программ p. 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 p 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 p на наборе входных данных E_i, выбранных случайно с распределением вероятностей p_i, приведет к получению приемлемого результата.

Введем также определения и обозначения, связывающие структурные характеристики программ с их безопасностью. Структурными характеристиками программы p 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.

Оценка технологической безопасности программ на базе метода Нельсона

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 [56].
    - 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 [56] 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.

2.4. Методы создания алгоритмически безопасных процедур

2.4.1. Постановка задачи

Основной отличительной особенностью подхода, связанного с созданием алгоритмически безопасного ПО является то, что начало процесса обеспечения безопасности программ при их разработке можно перенести на более ранние этапы жизненного цикла программного обеспечения, например, на этапы, предшествующие этапу испытания программ, тем самым увеличив общее время на внесение в программы защитных функций. Здесь уместно процитировать слова Э.В. Дейкстра, одного из основоположников современной методологии программирования, сказанные им еще в 1972 году ([14], стр.41): «В настоящее время общепринятой техникой является составление программы, а затем ее тестирование. Однако тестирование программ может быть очень эффективным способом демонстрации наличия ошибок, но оно безнадежно неадекватно для доказательства их отсутствия... Не следует сначала писать программу, а потом доказывать ее правильность, поскольку в этом случае требование найти доказательство только увеличит тяготы бедного программиста». Эти слова как нельзя лучше подходят и к современной проблематике, связанной, в данном случае, не столько с разработкой правильных, сколько безопасных программ. Иными словами, ключом к созданию безопасного программного обеспече-

ния является стремление защищаться от дефектов с самого начала жизненного цикла программ, а не после факта их создания.

Кроме того, следует отметить, что одно из основных достоинств методов создания алгоритмически безопасного программного обеспечения заключается в том, что данные методы позволяют защищать программное обеспечение, как на этапе разработки, так и на этапе его эксплуатации.

В то же время, необходимо сказать, что основная сложность в разработке безопасных программ указанного типа заключается в трудности нахождения эффективных алгоритмов их функционирования, которые являлись к тому же доказуемо корректными. И, предположительно, проблема неразрешимости доказательства безопасности для сложных программ при использовании методов их тестирования на этапе испытаний остается, в случае применения методов создания алгоритмически безопасного программного обеспечения, по-прежнему неразрешимой, что будет видно из дальнейших рассуждений.

Одним из главных методических вопросов создания безопасного программного обеспечения является постановка задачи (формулировка проблемы) ее разработки. Предположим, некоторому разработчику (коллективу разработчиков) предписывается разработать программу *P* для некоторого объекта автоматизации. При этом последствия некорректного функционирования программы таковы, что могут привести к неким «нежелательным» или даже катастрофическим последствиям для объекта автоматизации. Исходя из гипотезы, что в общем случае проблема доказательства безопасности для сложных комплексов программ является неразрешимой, неформальная качественная постановка задачи разработки указанных программ может быть тогда сформулирована следующим образом.

Постановка 1. «При некоторых условиях и ограничениях необходимо разработать программу *P*, которая корректно вычисляет результат *почти* для всех своих входных значений».

Таким образом, разработчик констатирует факт, что лишь для незначительной (возможно достаточно малой) доли входных значений программа может выдать некорректное выходное значение в результате необнаруженного программного дефекта или активизации необнаруженной программной закладки.

Понятно, что в некоторых случаях возникает определенная необходимость получения каких-либо более точных количественных оценок степени защиты созданной программы от внутренних дефектов, с целью получения определенных гарантий надежного функционирования объекта автоматизации.

При использовании контрольно-испытательных методов анализа безопасности программ, когда анализу подвергается только исполняемый

код программы [17,45] можно получить (как правило, экспертным путем) либо приближенные количественные характеристики обнаружения дефектов в контролируемых программах, либо стратификационные характеристики ненарушения программой некоторых условий безопасности. В этом случае постановка 1 задачи разработки программы *P* принципиально не меняется. (Естественно не рассматриваются простейшие случаи, когда при тестировании возможен контроль результата программы при полном переборе всех входных значений, при всех ограничениях и допущениях и т.п.).

Постановка же задачи разработки безопасного программного обеспечения за счет алгоритмически безопасных процедур меняется с точки зрения получения точных количественных (в данном случае вероятностных) характеристик кардинальным образом.

Постановка 2. «При некоторых условиях и ограничениях необходимо разработать программу *P**, которая корректно вычисляет результат *для всех* своих входных значений *с пренебрежимо малой* вероятностью ошибки».

На примере постановок 3, 4 и 5, 6 можно также проследить существенную разницу в парадигме разработки безопасных программ. Постановки 3 и 4 в отличие от постановок 1 и 2 учитывают не «содержание» наборов входных значений программы *P*, а их распределения вероятностей, а постановки 5 и 6 являются в некотором смысле обобщенными и учитывают заданный структурный критерий тестирования.

Постановка 3. «При некоторых условиях и ограничениях необходимо разработать программу *P*, которая некорректно вычисляет результат *лишь для некоторого* частного распределения вероятностей своих входных величин».

Постановка 4. «При некоторых условиях и ограничениях необходимо разработать программу *P**, которая корректно вычисляет результат *для любого* распределения вероятностей своих входных величин с пренебрежимо малой вероятностью ошибки».

Постановка 5. «При некоторых условиях и ограничениях необходимо разработать программу *P*, которая корректно вычисляет результат *почти* на всех тестах полной системы тестов программы относительно заданного структурного критерия тестирования».

Постановка 6. “При некоторых условиях и ограничениях необходимо разработать программу *P**, которая корректно с пренебрежимо малой вероятностью ошибки вычисляет результат на всех тестах полной системы тестов относительно заданного структурного критерия тестирования».

Одним из принципиальных условий решения задачи в постановке 2, 4 и 6 является наличие некоего свойства алгоритмической трансформации, позволяющего переходить от традиционного пути создания программ, ко-

торые будут затем проверяться на наличие дефектов, к априорно безопасным программам. К числу таких примеров можно отнести самокорректирующиеся и самотестирующиеся программы [23,26,31,33,62,63] (п.2.6.1.), обладающие свойством случайной самосводимости и программы, созданные на базе методов конфиденциальных вычислений [24,25,32] (п.2.6.2), начало изучения которых было положено в работах [61,67]. Рассмотренные до этого методы анализа безопасности ПО связаны с попытками обезопасить фактически уже разработанное программное обеспечение от действий злоумышленника. Это означает, что разработка безопасного ПО возможна за счет создания средств противодействия программным закладкам для продуктов, созданных на основе существующих информационных технологий создания программного обеспечения. То есть, только после факта разработки программ начинается верификационный анализ, тестирование или контроль их на технологическую безопасность. В этом смысле проблема обеспечения технологической безопасности программного обеспечения более близка к фундаментальной проблеме его надежности.

В то же время, данные методы создания безопасного ПО характеризуются рядом существенных недостатков, к числу которых можно отнести: невозможность перекрытия для большинства программных комплексов всего спектра тестовых наборов исходных данных; необходимость создания высоконадежных механизмов тестирования программ, например, механизмов экстраполяции результатов испытаний; существенные ресурсозатраты на проведение испытаний и т.п.

Поэтому в последнее время появилась насущная необходимость в создании новых информационных технологий разработки ПО, исходно ориентированных на создание безопасных программных продуктов относительного заданного класса. 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 (необязательно связанные с внедрением апостериорных 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 два направления, неформальное введение в которые дается ниже.

2.4.2. Методы создания самотестирующихся и самокорректирующихся программ для решения вычислительных задач

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 *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, затраченное на вызовы P должно составлять константный мультипликативный фактор от времени выполнения P. Таким образом, самотестирование должно лишь незначительно замедлять время выполнения программы P.

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 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 $(\varepsilon_1,\varepsilon_2)$-с*амотестирующейся 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-β.

- если $err(P,f,D_n)\geq\varepsilon_2$, тогда программа T_f^P выдаст на выходе «сбой» с вероятностью не менее 1-β.

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-β.

($\varepsilon_1,\varepsilon_2,\varepsilon$)-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.

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 [33]. 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 де-

фектов. Следует в то же время отметить, что предположительно предлагаемый метод можно использовать для программ, вычисляющих функции особого вида, а именно функции, обладающие свойством случайной самосводимости.

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 $Y = f(X)$.

Метод верификации расчетных программ на основе ST-пары функций.

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***

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.

Вызвать программу P для вычисления значения $Y_0^* = f\left(X^*\right)$.

Вызвать c раз программу P для вычисления множества значений $\left\{f\left(a_1^*\right), \ldots, f\left(a_c^*\right)\right\}$.

Определить значения $Y_1^* = g_c\left(f\left(a_1^*\right), \ldots, f\left(a_c^*\right)\right)$.

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 $T = \sum_{i=1}^{c} t_i + t_x + t_g + t_{h^{-1}} < T_P(X) \cdot \left(1 + c + K_{gh}(X, c)\right)$,

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 ко 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.

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 результатов работы [62] 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 [33], которая реализует функцию дискретного возведения в степень (размерность переменных и констант не превышает 64 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:

$g_2(a_1, a_2) = \left[f_{AM}(a_1) \cdot f_{AM}(a_2) \right] (\text{mod } M)$ и $h_2(a_1, a_2) = a_1 + a_2$;

$g_3^1(a_1, a_2, a_3) = \left[f_{AM}(a_1) \cdot f_{AM}(a_2) \cdot f_{AM}(a_3) \right] (\text{mod } M)$ 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 авторы в данной работе опускают).

Исследование возможности обнаружения предложенным методом преднамеренно внесенных ошибок 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 [33]. 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.

Метод создания самокорректирующейся процедуры вычисления теоретико - числовой функции дискретного экспоненцирования

Обозначения и определения для функции дискретного возведения в степень вида $g^x \text{modulo} M$. 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. Тогда равновероятный случайный выбор элемента a из множества Ω будет обозначаться как $a \in_R \Omega$. Оракульной программой, в данном случае, является программа вычисления функции $g^x \text{modulo} M$, по отношению к исследуемым самотестирующейся и самокорректирующейся программам.

Алгоритм $A^x \text{modulo } N$ можно вычислить многими способами [34,64], один из наиболее общеизвестных и применяемых, - 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. 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.

Псевдокод алгоритма A^xmodulo$N$

```
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.
```

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.

Построение самотестирующейся/самокорректирующейся программной пары для функции дискретного экспоненцирования.

Сначала рассмотрим следующие 4 алгоритма (см. рис.2.6 - 2.9). Для доказательства полноты и безопасности указанной пары доказывается следующая теорема.

Теорема 2.2. 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 следующие две леммы.

Лемма 2.1. 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 modulo M=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. 2.6. 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. 2.7. 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. 2.8. 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. 2.9. 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 $\varepsilon \le 1/8$, то в одном цикле вероятность $\mathrm{Prob}[R_k = f_{M,g}(x)] \ge 3/4$. Чтобы вероятность корректного ответа была не менее чем $1-\beta$, исходя из оценки Чернова [62], необходимо выполнить не менее $12\ln(1/\beta)$ циклов ■.

Лемма 2.2. Программа S_T_exp(n,M,q,g,β) является (1/288,1/8)-самотестирующейся программой, которая контролирует результат вычисления значения функции g^x modulo M с любым модулем M.

Доказательство. Полнота теста линейной состоятельности доказывается аналогично доказательству полноты в лемме й, 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 1 для того, чтобы вероятность корректных ответов R_l и R_e в обоих тестах была не более 1-β 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 (алгоритмы данной программы 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 (при использовании в вышеуказанных алгоритмах параметров с их независимым, равновероятным и случайным выбором), что программа вида S_T(·) является (ε/36,ε)-самотестирующейся программой. Отсюда 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^xmodulo N в нем используется операция ABmodulo 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:

$$[P_M(a_1,b_1)+P_M(a_1,b_2)+P_M(a_2,b_1)+P_M(a_2,b_2)](\mathrm{mod}\ M).$$

Алгоритм вычисления A^xmodulo N выполняется для c=2. Однако, декомпозиция x, как следует из свойства самосводимости функции A^xmodulo 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.

Применение самотестирующихся и самокорректирующихся программ

Применение 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.

Активными исследованиями в области создания самотестирующихся и самокорректирующихся программ в вычислительной математике ученые и практики начали заниматься с начала 90-х годов. В этот период были разработаны самотестирующиеся программы для ряда теоретико-числовых и теоретико-групповых задач, для решения задач с матрицами, полиномами, линейными уравнениями, рекуррентными соотношениями и аппроксимирующими функциями.

Основная идея использования идей самотестирования в криптографии заключается в неформальном 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.

К другим направлениям в теории и практике самотестирования можно отнести построение псевдослучайных генераторов, в котором центральным звеном является конструкция, которую можно рассматривать как самокорректирующую программу для решения задачи, эквивалентной проблеме дискретных логарифмов, а также разработку самотестирующихся программ для параллельных вычислений, где используется идея самотестирования для построения схемы константной глубины для проверки мажоритарных функций.

2.4.3. Создание безопасного программного обеспечения на базе методов теории конфиденциальных вычислений

Постановка задачи

Задачу конфиденциальных вычислений, 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.

Общие замечания по проблематике конфиденциальных вычислений

Задача конфиденциального вычисления была впервые сформулирована А.Яо для случая с двумя участниками в 1982 г. В 1987 г. О. Голдрайх, С. Микали и А. Вигдерсон показали, 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 сво-

их 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 [67] предложены определения для многосторонних конфиденциальных вычислений при наличии защищенных каналов и в присутствии адаптивных противников.

В работе [61], 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. Автор привел первый асинхронный протокол византийских соглашений с оптимальной устойчивостью, где противник может «подкупать» $\lceil n/3 \rceil$-1 из n участников вычислений.

Определение односторонней функции, описание используемых примитивов, схем и протоколов

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. (см, например, определения в работе [6]). 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.

Далее приводится формальное определение односторонней функции в терминах теории сложности вычислений. Для этого введем следующее обозначение. Пусть $a\in_R M$ обозначает, что элемент a случайно, с равномерным распределением вероятностей и независимо от других событий выбран из всех элементов множества M.

Функция f называется *односторонней*, если существует полиномиальная машина Тьюринга T, которая на любом входе $x\in\Sigma$ вычисляет $f(x)$ и для любого полиномиального алгоритма A справедливо следующее.

Пусть $x\in_R\Sigma_n$ и слово $y=f(x)$ подано на вход алгоритма A. Тогда для любого полинома p и для всех достаточно больших n вероятность $\mathrm{Prob}(f(A(y))=y)<1/p(n)$.

Вероятность берется по выбору значения x из Σ_n и, если A - это вероятностная машина Тьюринга, - по вырабатываемым ею случайным величинам.

(n,t)-Пороговые схемы. Используемая в данном разделе (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 $P_1,...,P_n$, (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:

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

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.

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.

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

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

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

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

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

$\mathrm{Prob}(s^*=s \mid (n,t,t^*)\mathbf{РзПр}=(\text{Да},s^*)$ & **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.

Широковещательный примитив (Br-субпротокол). 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. Для Br-протокола не требуется, чтобы несбоящие процессоры завершали протокол, в том случае, если дилер нечестен.

Br-протокол

Код для дилера (по входу m):

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

Код для процессоров:

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

Предложение 2.1. *Br*-протокол является *t*-усточивым широковещательным протоколом для противников, которые могут приостанавливать отправку сообщений.

Доказательство. Если дилер честен, то все несбоящие процессоры получают сообщение (*сбщ*,*m*) и, таким образом, завершают протокол с выходом *m*. Если несбоящий процессор завершил протокол с выходом *m*, то он посылает сообщение (*эхо*,*m*) ко всем другим процессорам и, таким образом, каждый несбоящий процессор завершит протокол с выходом *m*.

Ниже описывается $\lfloor (n\text{-}1)/3 \rfloor$-устойчивый широковещательный протокол, который именуется **BB**. 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 процессорам.

Код для процессора P_i*:*

2. После получения сообщения (*сбщ*,*m*), послать сообщение (*эхо*,*m*) ко всем процессорам.

3. После получения *n*-1сообщений (*сбщ*,*m′*), которые согласованы со значением *m′* послать сообщение (*гот*,*m′*) ко всем процессорам.

4. После получения *t*+1 сообщений (*гот*,*m′*), которые согласованы со значением *m′*, послать (*гот*,*m′*) ко всем процессорам.

5. После получения *n*-1сообщений (*сбщ*,*m′*), которые согласованы со значением *m′*, послать сообщение (*OK*,*m′*) ко всем процессорам и принять *m′* как распространяемое сообщение.

Протокол византийского соглашения (BA-субпротокол). Введем следующее определение. При *византийских соглашениях* 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 $d=x$.

Обобщенные модели вычислений для сети синхронно и асинхронно взаимодействующих процессоров

Принимаемая модель создания программного обеспечения на базе методов конфиденциального вычисления функции позволяет единообразно трактовать как ошибки, возникающие, например, в результате сбоев технических средств, так и ошибки, возникающие за счет привнесения в вычислительный процесс преднамеренных программных дефектов. Следует

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

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

Модель сбоев и модель 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. В то же время гарантируется, 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.

Модель взаимодействия

Взаимодействие процессоров может осуществляться посредством конфиденциальных и 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 участниками вычислений в r-том раунде в установленной модели взаимодействия.

Синхронная модель вычислений

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 сообще-

ния 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 $\boldsymbol{\Pi}$. *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 входа, все распространяемые им сообщения, все сообщения, полученные i по конфиденциальным каналам связи, и все случайные параметрами, сгенерированные процессором i во время работы сети.

Идеальный и реальный сценарии

Для доказуемо конфиденциального вычисления вводятся понятия идеального и реального сценария [10]. В идеальном сценарии дополнительно вводится *достоверный процессор* (ТР-процессор). Процессоры конфиденциально посылают свои входы ТР-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 нечестных процессоров. После чего те, вместе с корректными входами честных 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, изучать по полученному значению функции входы честных процессоров.

В реальном сценарии не существует TP-процессора, и все участники вычислений 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.

Асинхронная модель вычислений

В данном подразделе рассматривается полностью асинхронная сеть из n процессоров, которые соединены открытыми или конфиденциальными каналами. 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. Каналы, являющиеся конфиденциальными или открытыми, рассматриваются как *частные планировщики* (PD_j). Информация, известная частному планировщику, есть не что иное как сообщения, посланные от источника сообщений их адресату. В данной модели вычислений каждый их шах рассматривается как *раунд при асинхронных вычислениях*. Идеальный и реальный сценарии в асинхронной модели

Сценарий в асинхронной модели с TP-участником заключается в добавлении TP-процессора в существующую асинхронную сеть при наличии t потенциальных сбоев (сбоящих процессоров). При этом честные процес-

соры, также как и TP-процессор не могут ожидать наличия более, чем *n-t* входов для вычислений с целью получения их выхода, так как *t* процессоров (сбоящие процессоры) могут никогда не присоединиться к вычислениям.

Асинхронный TP-сценарий

В начале вычислений процессоры посылают свои входы TP-процессору. В то же время, существует планировщик *D*, который доставляет сообщения участников некоторому базовому подмножеству процессоров, мощностью не меньше *n-t*, обозначаемому как *C* и являющемуся независимым от входов честных процессоров. TP-процессор по получению входов - аргументов функции (возможно некорректных) из множества *C*, предопределенно оценивает значение вычисляемой функции, основываясь на *C* и входах участников из *C*. (Здесь для корректности может использоваться следующее предопределенное оценивание: установить входы из *C* в 0 и вычислить данную функцию). Затем TP-процессор посылает значение оценочной функции обратно участникам совместно с базовым множеством *C*. Наконец честные процессоры вычислений выдают то, что они получили от TP-участника. Нечестные участники 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 TP-сценарии.

Далее в соответствии с работой [61] сделаем попытку формализовать понятия полноты и безопасности протокола асинхронных конфиденциальных вычислений.

Безопасность асинхронных вычислений

Для удобства мы унифицируем коалицию нечестных процессоров и планировщика. Это не увеличит мощность противника в TP-сценарии, так как и нечестные участники, 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 $|B|$-vector $\vec{z} = z_1 ... z_{|B|}$, 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 $f_C(\vec{x})\cong f(\vec{x}/_{(\bar{C},\bar{0})})$

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

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

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

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

где r объединенный случайный вход нечестных 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 $\pi(\vec{x},B,D)=\pi_1(\vec{x},B,D)\ldots\pi_n(\vec{x},B,D)$.

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

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

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

Конфиденциальное вычисление функции

Общие положения. Для некоторых задач, решаемых в рамках методологии конфиденциальных вычислений, достаточно введения определения *конфиденциального вычисления функции* [67].

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 (т.е. 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 содержится в вычисленном значении функции.

По аналогии с идеальным и реальным сценариями, приведенными выше, можно ввести понятия «реальное и идеальное вычисление функции f» [67].

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 соответственно и r^* - заданный неким произвольным образом раунд выполнения протокола $\boldsymbol{P}$.

Пусть функцию f можно представить как композицию d функций (функций от двух аргументов-векторов) $g_1\circ...\circ g_\eta\circ g_{\eta+1}\circ...\circ g_d$:

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

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

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

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

Введем понятие моделирующего устройства $\boldsymbol{M}$. Здесь можно проследить некоторые аналогии с моделирующей машиной в интерактивных системах доказательств с нулевым разглашением (см., например, [27,29]).

Пусть ${}_{\text{р}}\Theta^{\text{Прот}}$ - распределение вероятностей над множеством историй (трафика T и случайных параметров) нечестных участников во время выполнения протокола $\boldsymbol{P}$.

Моделирующее устройство, взаимодействующее с нечестными участниками, осуществляет свое функционирование в рамках идеального сценария. Моделирующее устройство $\boldsymbol{M}$ создает распределение вероятностей параметров взаимодействия ${}_{\text{и}}\Theta^{\text{Прот}}$ между $\boldsymbol{M}$ и нечестными участниками.

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

Условие корректности. Для всех несбоящих процессоров P_i функция

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

вычисляется с вероятностью ошибки близкой к 0.

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

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

Проверяемые схемы разделения секрета как конфиденциальное вычисление функции

Для вышеприведенной проверяемой схемы **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 может осуществляться 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 восстанавливаемый секрет, обозначается как **ПРСК**.

Пусть сеть N состоит из n процессоров $P_1,P_2,...,P_{n-1},P_n$, 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 $|R|=\nu$. 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:$(S^n \otimes R^n \otimes W) \to W$, а функция h:$W \to S$.

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 участников.

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 участники протоколов **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, проверить и восстановить секрет. Свойство конфиденциальности вытекает из свойства конфиденциальности функции.

Схема ПРСК

Предлагаемая схема ПРСК, 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 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. 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 *Pi* 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 участнику 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 честных процессоров, так как полином имеет степень t+1 и ему необходимо для интерполяции t +2 точки.

Доказательство безопасности схемы проверяемого разделения секрета

Теорема 2.3. 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, $i\le n$, 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, который 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 2.3. 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 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 2.4. 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 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 боль-

шого K, является экспоненциально малым.

Конфиденциальность. Для доказательства теоремы необходимо доказать следующие две леммы.

Лемма 2.5. 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 A: 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 B: 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 многочлен 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, [6,27,29]) 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 вы-

числений. 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 2.6. 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 ***M*** сводится к следующему.

Описание работы моделирующего устройства *M*

1. В раунде , ***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 ***M*** будет изучать истинный секрет s. Если такой процессор является «подкупленным» противником **Прот** в конце этого раунда (или в следующих раундах), то и ***M***, и **Прот** узнают истинную долю s_l и ***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 ***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 способен создать неразличимые распределения.

9. 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 2.2 – 2.5 доказательство теоремы 2.3 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. Демонстрация собственно протоколов конфиденциальных вычислений, решающих те или иные теоретические или прикладные задачи, выходит за рамки настоящей работы.

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.2.10). 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.

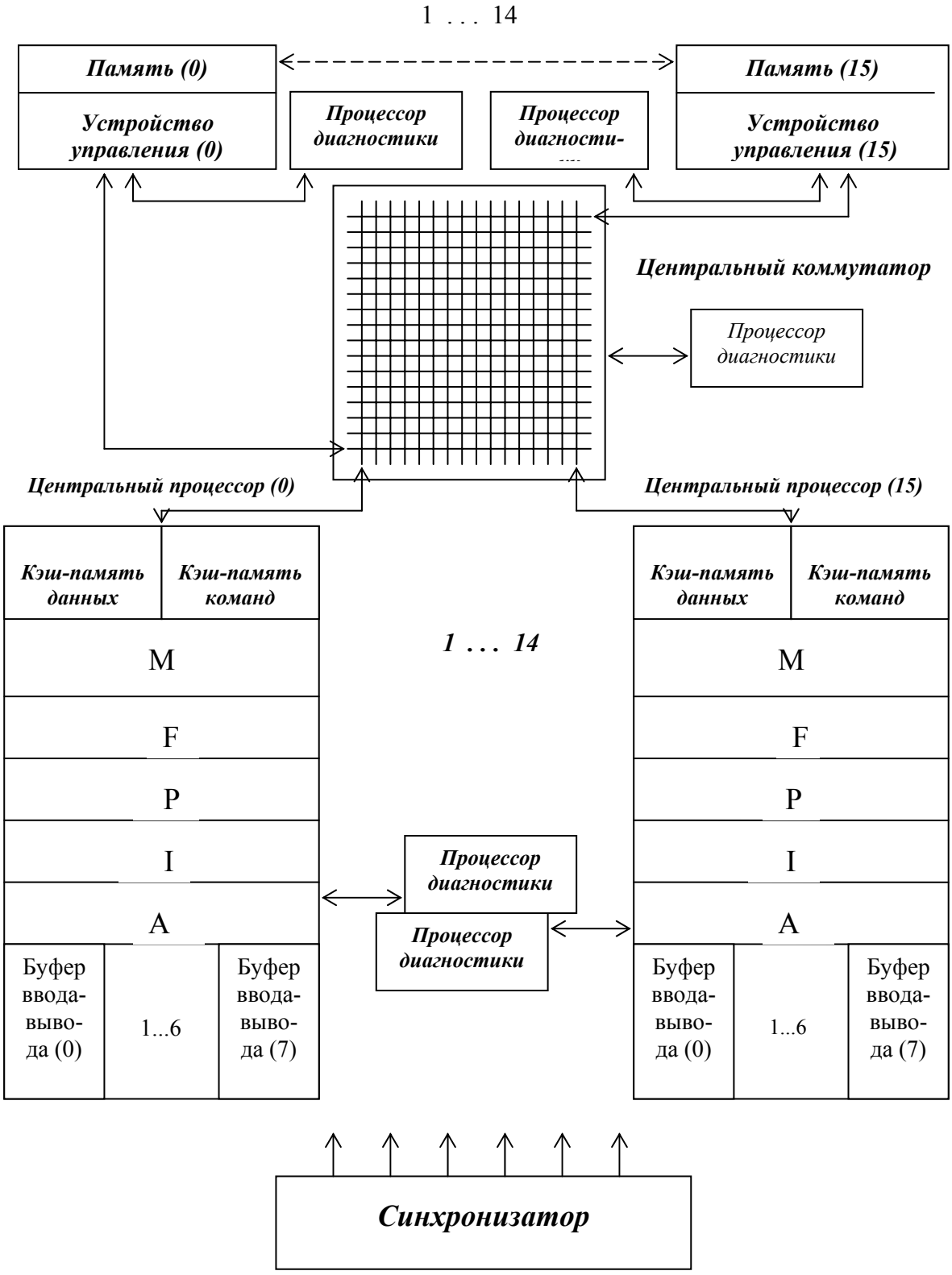

Рис. 2.10. RL-прототип модели синхронных конфиденциальных вычислений

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 их перераспределение 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 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.

1. Протоколы византийского соглашения (определение дано выше).

2. *Протоколы разделения секрета* (определение дано выше).

3. *Протоколы электронного голосования.* В таком протоколе должно быть обеспечено три основных условия:

- обеспечения правильности подачи и подсчета голосов;

- обеспечения тайного голосования избирателей;
- обеспечения невозможности срыва выборов или искажения их результатов.

4. *Протоколы выработки общей случайной строки*. Протокол позволяет безопасным образом группе участников, часть из которых являются нечестными, выработать с равномерным распределением вероятностей общую для всех случайную строку. Такой протокол является часто используемым вычислительным примитивом для построения более сложных протоколов распределенных вычислений.

Методы конфиденциальных вычислений могут позволить для многопроцессорных систем проектировать высокозащищенную программно-аппаратную среду для использования в автоматизированных системах критически уязвимых объектов информатизации.

2.4.4. Криптопрограммирование

Криптопрограммирование посредством использования инкрементальных алгоритмов

Одним из основных инструментов методологии криптопрограммирования являются инкрементальные 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 данный документ. В тех приложениях, когда указанные алгоритмы являются, например, алгоритмами шифрования электронных документов или их цифровой подписи, требование повышения эффективности инкрементальных алгоритмов является крайне желательным. Один из основных методов применения инкрементальных алгоритмов заключается в использовании их аутентификационных признаков для антивирусной защиты.

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 [28]) 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.

Внедрение вируса в файл не позволяет ему заново вычислять признак, и 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. Ясно, что наиболее привлекательным способом является модернизация аутентификационного признака быстрее, чем необходимость его вычисления заново. Эта проблема особенно сложна в том случае, когда локальная память не достаточно большая для того, чтобы хранить (даже временно) фрагмент файла или когда «слишком дорого» ввести в локальную память полный файл.

Идея инкрементальных алгоритмов, состоит в том, чтобы воспользоваться какими-либо имеющимися преимуществами такой организации программно-аппаратного взаимодействия и найти способы криптографических преобразований над электронными данными D не заново, а, скорее, как получение быстродействующей функции от значений криптографических преобразований над данными из которых D был получен. Когда «изменения» не велики, инкрементальные методы могут дать большие преимущества по эффективности.

Основные элементы инкрементальной криптографии

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» преобразованиях, а именно для генерации подписи и шифровании, но все рассуждения будут верны и для «сопряженных» преобразований, а именно для верификации подписи и дешифрования.

Операторы модификации. Рассмотрим проблему модификации защищаемого файла в терминах применения фиксированного набора основных операций по модификации электронного документа. Например: 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.

Инкрементальные алгоритмы. Зафиксируем базовое криптографическое преобразование $\boldsymbol{T}$ (например, цифровая подпись документа с некоторым ключом). Каждой элементарной операции модификации текста (например, вставки) будет соответствовать инкрементальный алгоритм $\boldsymbol{I}$. На вход этого алгоритма подаются: исходный файл; значения преобразования $\boldsymbol{T}$ на нем; описание модификаций; и возможно соответствующие ключи или другие параметры. Это позволит вычислить значение $\boldsymbol{T}$ для результирующего файла. Основная проблема здесь заключается в проектировании схем обработки файлов, обладающих эффективными инкрементальными алгоритмами. Предположим, что имеется подпись σ_{old} для файла D_{old} и файл D'_{old}, измененный посредством вставки в файл D_{old} некоторых данных. Необходимо получить подпись путем подписывания строки, состоящей из σ_{old} и описания модификаций над документом D_{old}. Это схема называется *схемой, зависящей от истории*. Могут иметься приложения, когда такие действия являются приемлемыми. В большинстве же случаев это не желательно, так как делается большое количество изменений и затраты на верификацию подписи пропорциональны числу изменений. 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.

Безопасность. Свойство инкрементальности вводит новые проблемы безопасности и вызывает необходимость новых определений. Рассмотрим случай схем подписи и аутентификации. Разумно предположить, что противник не только имеет доступ к предыдущим подписанным версиям фа-

лов, но также способен выдавать команды на модификацию текста в существующих файлах и получать инкрементальные подписи для измененных файлов. Такая атака на основе выбранного сообщения для инкрементальных алгоритмов подписи может вести к «взлому» используемой схемы подписи, которая не может быть взломана при проведении противником других атак, в тех случаях, когда не используются инкрементальные алгоритмы. Кроме того, в некоторых сценариях, например, при вирусных атаках можно предположить, что противник вмешивается в содержание существующих документов и аутентификационных признаков, полученных посредством схем подписи/аутентификации. Соответственно рассматриваются два определения безопасности: базовое и более сильное понятие безопасности, когда доказывается стойкость защиты от вмешательств.

Секретность инкрементальных схем. Исходя из вышесказанного, появляется новая проблема, которая проявляется в инкрементальном сценарии, а именно - проблема секретности различных версий файлов. Предположим μ - подпись кода M и μ' является подписью слегка измененного кода M'. Тогда, наилучшим вариантом было бы получить такую подпись μ', которая давала бы как можно меньше информации, об оригинальном коде M.

Метод защиты файлов программ посредством инкрементального алгоритма маркирования

Основные определения. Пусть АУТ(m) - обычный алгоритм аутентификации сообщений и АУТ$_\alpha$(m)- функция маркирования сообщения m, индуцированная схемой АУТ с ключом аутентификации α. Пусть ВЕР$_\alpha$(m,β) - 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 [2]. 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 способом 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 каждого из блоков файла F=F[1],...,F[l] (далее такое дерево будет назы-

ваться *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:

- для каждого i, пусть $T(w)=\text{АУТ}_\alpha(F[i])$, где w – i-тый концевой узел;
- для каждого не - концевого узла w, пусть $T(w)=\text{АУТ}_\alpha((L_1,L_2,L_3),рзм)$, 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);
- для корня дерева $T(\lambda)=\text{АУТ}_\alpha((L_1,L_2,L_3),Id,счт)$, 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.

Пусть $u_0,...u_h$ - путь из корня $u_0=\lambda$ к j-тому концевому узлу обозначается как u_h. Тогда:

- проверить, что ВЕР_α принимает $T(\lambda)$ как корректный АП строки $T(\lambda)=\text{АУТ}_\alpha((L_1,L_2,L_3),Id,счт)$, где *Id* - название документа и *счт* - текущее значение счетчика (связанного с этим документом);
- для $i=1,...,h$-1 проверить, что ВЕР_α принимает $T(u_i)$ как корректный АП строки $((L_1,L_2,L_3),рзм)$, 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));
- проверить, что ВЕР_α принимает $T(u_h)$ как корректный АП блока $F[j]$.

Если все эти проверки успешны, тогда совокупный АП файла F получается следующим образом:

- установить $T(u_h):=\text{АУТ}(F(\sigma))$;
- для $i=h$-1,...,1 установить $T(u_i):=\text{АУТ}(T(u_i1),T(u_i2),T(u_i3))$;
- установить $T(\lambda):=\text{АУТ}((T(u_i0),T(u_i1),T(u_i1)),Id,счт+1)$.

Следует подчеркнуть, что значения T на всех других вершинах (то есть, не стоящих на пути $u0,...,u_h$) остаются неизменяемыми.

Следует 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 или итерационно изменяемыми переменными.

2.4.5. Защита программ и забывающее моделирование на RAM-машинах

Основные положения

В этом разделе рассматриваются теоретические аспекты защиты программ от копирования. Эта задача защиты сводится к задаче эффективного моделирования *RAM*-машины (машины с произвольным доступом к памяти [2]) посредством забывающей *RAM*-машины. Следует заметить, что основные результаты по данной тематике (их можно получить на соответствующем личном интернетовском сайте) принадлежат О. Голдрайху и Р. Островски 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.

Центральный процессор, имитирующий взаимодействие. Неформально, будем говорить, что *центральный процессор имитирует взаимодействие* с соответствующими зашифрованными программами, если никакой вероятностный полиномиально-временной противник не может разли-

чить следующие два случая, когда по данной зашифрованной программе как входу:

- противник *экспериментирует с оригинальным защищенным ЦП*, который пытается выполнить зашифрованную программу, используя память;
- противник *экспериментирует с «поддельным» (фальсифицированным) ЦП*. Взаимодействия поддельного ЦП с памятью почти идентичны тем, которые оригинальный ЦП 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 ЦП (некоторым «волшебным» образом) записывает в память тот же самый выход, который подлинный ЦП написал, выполняя «реальную» программу.

При создании ЦП, который имитирует эксперименты, имеются две проблемы. Первая заключается в том, что необходимо скрывать от противника значения, сохраненные и восстановленные 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*).

Сокрытие оригинальной модели доступа к памяти – это абсолютно новая проблема и традиционные криптографические методы здесь не применимы. Цель в таком случае состоит в том, чтобы сделать невозможным для противника получить о программе что-либо полезное из модели доступа. В этом случае ЦП не будет выполнять программу обычным способом, однако он заменяет каждый оригинальный цикл «выборки/запоминания» многими циклами «выборки/запоминания». Это будет «путать» противника и предупреждать его от «изучения» оригинальной последовательности операций доступа к памяти (в отличие от фактической последовательности). Следовательно, противник не может улучшить свои способности по восстановлению 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 ЦП с k битами внутренней защищенной памяти и соответствующей зашифрованной программы такой, что ЦП имитирует взаимодействие с зашифрованной программой, реализуемое за время, ограниченное poly(k). Кроме того, t команд оригинальной программы выполняются с использованием менее чем за $tk^{0(1)}$ команд зашифрованной программы, а увеличение размера внешней памяти ограничиваются коэффициентом k.

Вышеупомянутый результат доказывается посредством сведения задачи создания ЦП, который нарушает эксперименты по вмешательству, к задаче сокрытия модели доступа. По-существу, последняя задача формулируется как задача моделирования на независимой забывающей *RAM*-машине (см. ниже).

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). 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.

Пусть *RAM*(m) означает *RAM*-машину с m ячейками памяти и доступом к случайному оракулу [13]. Тогда t шагов независимой *RAM*(m)-программы может моделироваться за менее чем $O(t(\log_2 t)^3)$ 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 (от размера памяти) 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 $\max\{m,(t\text{-}1)\log m\}$ операций доступа, чтобы смоделировать t шагов оригинальной программы.

Далее рассмотрим сценария наихудшего случая, в котором наблюдатель (или в данном случае противник) активно пытается получить информацию, вмешиваясь в процесс вычислений. Вопрос заключается в том, можно ли гарантировать, что воздействие противника является забывающим по входу. 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 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.

Модели и определения

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.

RAM-машина как пара интерактивных машин. В данном разделе *RAM*-машина представляется как две интерактивные машины: центральный процессор (ЦП) и модуль памяти (МП). Задача исследований сводится изучению взаимодействия между этими машинами. Для лучшего понимания необходимо начать с определения интерактивной машины Тьюринга.

Интерактивная машина Тьюринга – многоленточная машина Тьюринга, 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 завершен с 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 ту же самую 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 $ITM(k+2+O(k),2^kO(k))$, 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 является *машиной, управляемой сообщениями*. После получения сообщения (i,a,v), где $i\in\{0,1\}^2\equiv$ {«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 числа a (рабочей ленты).

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 по-

сылает сообщение, определенное 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. Число шагов каждого вычисления на рабочей ленте ограничено фиксированным полиномом от k.

Единственная роль входа ЦП должна заключаться в инициализации регистров ЦП, и этот вход может игнорироваться при последовательном обращении. «Внутреннее» вычисление 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 совокупность RAM_k-машин для каждого k.

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 $y=(\Pi,x)$. 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 $RAM_k(\cdot,(\Pi,x))$. 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.

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.

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-чтения» 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 $f:\{0,1\}^{O(k)} \to \{0,1\}$.

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 CPU_k заменен вероятностным CPU_k. (Другими словами, вероятностная RAM_k-машина – это оракульная RAM_k-машина с доступом к однородно выбранной функции).

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 следующее выполнение с рабочими лентами ЦП и МП, имеющих содержимое, идентичное их содержимому при окончании предыдущего выполнения программы.

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.

Эксперименты с RAM-машиной. Рассматриваются два типа противников. Оба могут неоднократно инициировать работу *RAM*-машины на вхо-

дах 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 не надо позволять противнику 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 внутренним лентам ЦП (т.е., 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.

Невмешивающийся противник, обозначаемый как *ADV* является вероятностной машиной, которая на входе k и «шифрованной программе» α, которая имеет следующий 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.

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.

Преобразования, защищающие программное обеспечение

Определим компиляторы, которые по данной программе П, производят пару (f,Π_f), где f - случайно выбранная функция и Π_f – «зашифрованная программа», которая соответствует П и f. Здесь имеется в виду оракульная *RAM*-машина, которая на входе (Π_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.

Компилятор, обозначаемый через C, является вероятностным отображением, которое по входу целочисленного параметра k и программы П для RAM_k возвращает пару (f,Π_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 $\Pi(x)$.

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 «зашифрованные 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 $(\Pi(x),t_\Pi(x),s_\Pi(x))$.

Отметим, что $t_\Pi(x)$ и $s_\Pi(x)$ обозначает время выполнения и пространственные размеры программы П на данных x. Далее даются основное определение для задачи защиты программ. В этом определении *ADV* может рассматриваться как вмешивающийся, так и невмешивающийся противник.

Для данного компилятора C и противника *ADV*, компилятор *C защищает программное обеспечение против противника ADV*, если существует вероятностная оракульная (в стандартном смысле) 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 $(k',|\alpha|)$ (и 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 следованием двух распределений вероятностей ограничено $2^{-k'}$.

Распределение выхода машины *ADV* при экспериментировании с $RAM_{k'}^{f}$ на входе Π_f, где $(f,\Pi f)\leftarrow C(\Pi)$. Отметим, что $RAM_{k'}^{f}$ обозначает интерактивную пару $(CPU_{k'},MEM_{k'})$, где $CPU_{k'}$ имеет доступ к оракулу f. Рас-

пределение 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.

Распределение выхода оракульной машины M на входе $(k',O(|\Pi|))$ 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 будет определяться затраты защиты программ. Необходимо напомнить, что для простоты, мы ограничиваем время выполнения программы П следующим условием: $t_{\Pi}(x)>|\Pi|+|x|$ для всех x.

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 (Π_f,x) и при доступе к оракулу f ограничены сверху $g(T)T$, где $T=t_{\Pi}(x)$.

Определение забывающей RAM-машины и забывающего моделирования

Необходимо начать с определения модели доступа как последовательности ячеек памяти, к которым ЦП обращается в процессе вычислений. Это определение распространяется также на оракульный ЦП.

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 CPU_k при взаимодействии с $MEM_k(y)$ 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.

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 частичного входа.

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.

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 мо-

дель 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 забывающего моделирования. Для данных вероятностных машин $RAM'_{k'}$ и RAM_k предположим, что вероятностная $RAM'_{k'}$ моделирует забывающим образом вычисления RAM_k и путь g:**N**→**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.

Моделирование с метками времени. В заключение этого подраздела приводится свойство некоторого *RAM*-моделирования. Это свойство требует, чтобы при восстановлении значения из ячеек памяти, ЦП «знает» сколько раз содержимое этих ячеек модифицировалось. То есть, для данного адреса МП a и общего числа команд (обозначенных j), выполнение всех команд ЦП «запомнить в ячейку a» может быть эффективно вычислено алгоритмом $Q(j,a)$. Далее рассматривается только моделирование детерминированных *RAM*-машин.

Для данной оракульной машины $RAM'_{k'}$, 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,·), посланных $CPU'_{k'}$, равняется точно (j,a). Далее, необходимо обратить запустить алгоритм на $Q(j,a)$ для получения номер *версии*(a) в раунде j.

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 для получения доказуемой защиты программ от вмешательства.

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 за-

щиты 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.

Защита программ от невмешивающихся противников. 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 2.1. 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 $i\in\{$*выборка*, *сохранить*, *стоп*$\}$, $a\in\{1,2,...,2^k\}$ и $v\in\{0,1\}^{O(k)}$, в то время как сообщения от MEM_k до 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) обеспечивает любую информацию посредством модификации ЦП так, чтобы всегда имелся доступ к ячейкам памяти при первой выборке. Следовательно, все что осталось должно «зашифровывать» содержимое области значений (т.е., v) так, чтобы ЦП мог восстанавливать оригинальное значение. Идея состоит в том, чтобы выполнить шифрование, используя оракул, доступный ЦП.

Для шифрования 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 (либо старое значение, которое только читалось, либо новое 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), зашифрованное значение восстанавливается посредством вычисления $u \oplus f(j)$. Подчеркнем, что и шифрование, и дешифрование 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 одноразово выбирает функцию f и множества $\Pi_f=(\pi_1 \oplus f(2^k+1),2^k+1),...,(\pi_n \oplus f(2^k+n),2^k+n)$.

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 шифрование (которое использует шифр «одноразовый блокнот») абсолютно безопасно (в информационно-теоретическом смысле), и следовательно, противник не получает никакой информации относительно оригинального содержания области значений. ■

Отметим, что на практике можно заменять случайный оракул на псевдослучайный. Следовательно, результат будет верен только для противников, ограниченных по времени некоторым полиномом. Компилятор на параметре входа k и программе П равномерно выбирает псевдослучайную функцию f. Описание f аппаратно реализовано в CPU_k. Следовательно, ЦП способен вычислять f на входах длины k и poly(k)-временной противник может различать поведение этого ЦП от ЦП, описанного в доказательстве теоремы. Следовательно, любой противник, который может выполнять эксперименты по вмешательству за время, ограниченное poly(k), может быть вычислен за время, ограниченное poly(k), с доступом только к оракулу спецификаций. Подобные замечания верны и для следующей теоремы.

Теорема 2.2**.** 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 числа предыдущих запомненных команд

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 должен сохранить зашифрованное значение v в ячейке памяти он сначала определяет текущий номер версии 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. ■

Решение задачи «Квадратного корня»

Отметим, что тривиальное решение для забывающего моделирования *RAM*-машины заключается в полном сканировании фактической памяти RAM_k-машины для каждой виртуальной операции доступа к памяти (которая должна быть выполнена для оригинальной *RAM*-машины). Далее описывается первое нетривиальное решение (принадлежащее О. Голдрайху) задачи забывающего моделирования RAM_k-машины посредством вероятностной $RAM'_{k'}$.

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 *виртуальном* доступе к 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 содержимого единственной ячейки памяти (т.е., *выборка*(i), сопровождаемая командами *сохранить*(i,·) для некоторого i).

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.

Инициализация: Первые $m+\sqrt{m}$ слов *RAM*-моделирования расположены так, чтобы содержать m виртуальных адресов (к которым оригинальная *RAM*-машина обращается в процессе выполнения) и $\sqrt{m}$ «фиктивных» слов (см. рис.2.11). Оставшиеся $\sqrt{m}$ слов распределены так, чтобы обслужить дополнительную («краткосрочную») память (здесь и далее называемую *зщт*).

Моделирование шагов RAM-машины: 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.

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 *зщт* (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*.

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 *зщт*. Так как размер памяти *зщт* равняется числу возможных операций доступа в одном периоде, можно хранить все значения, восстановленные в течение текущего периода в памяти *зщт*. Доступ к памяти оригинальной *RAM*-машины, скажем доступ к виртуальному слову i, моделируется следующим образом:

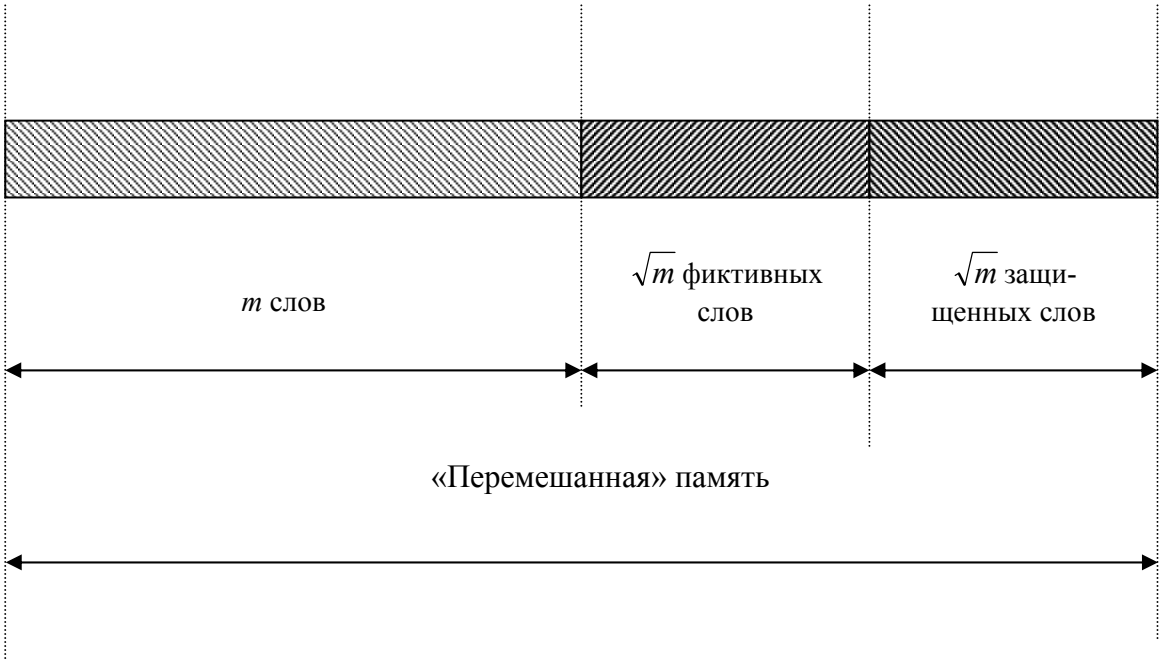

Рис. 2.11. Структура данных для решения задачи "квадратного корня"

- сканируется вся память *зщт* и проверяется, находится ли содержимое виртуального слова i в одном из слов памяти *зщт*. Подчеркнем, что здесь мы обращаемся к каждой ячейке памяти *зщт* 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 *зщт*, 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 *зщт*), 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*.

Модификация перемешанной памяти. 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 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.

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 $\{\tau(j):1\le j\le n\}$. 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, 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 наблюдателю (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 «осуществиться» доступ к некоторой виртуальной ячейке, необходимо сохранить версию этой виртуальной ячейке в памяти *зщт*. 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 *зщт*. Когда оригинальная *RAM*-машина осуществляет доступ к *i*-тому виртуальному слову, забывающая *RAM*-машина работает следующим образом:

- (2a) сканирует полностью память *зщт* 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 посредство 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 *зщт* снова и записывает модифицируемое значение *i*-того виртуального слова в памяти *зщт*. 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 shelter-е.
- Значение *счт* увеличивается на 1.

Подчеркнем, что наблюдатель 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 наблюдателя, не вскрывается.

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 $s=\Theta(\log m\sqrt{m})$.

Заключительные замечания

В данном подразделе 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 возможное приложение – 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 контроль структуры данных, которая следует из определений самотестирующихся программ, рассматриваемых выше. В этой конструкции желательно сохранить структуру данных при использовании малого количества достоверной 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.

2.5. Подходы к защите разрабатываемых программ от автоматической генерации инструментальными средствами программных закладок

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 [45]. 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 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.2.12).

Заметим, чти на рис.2.12 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 исполняемы код устанавливается код в какой-либо рабочий директорий (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.

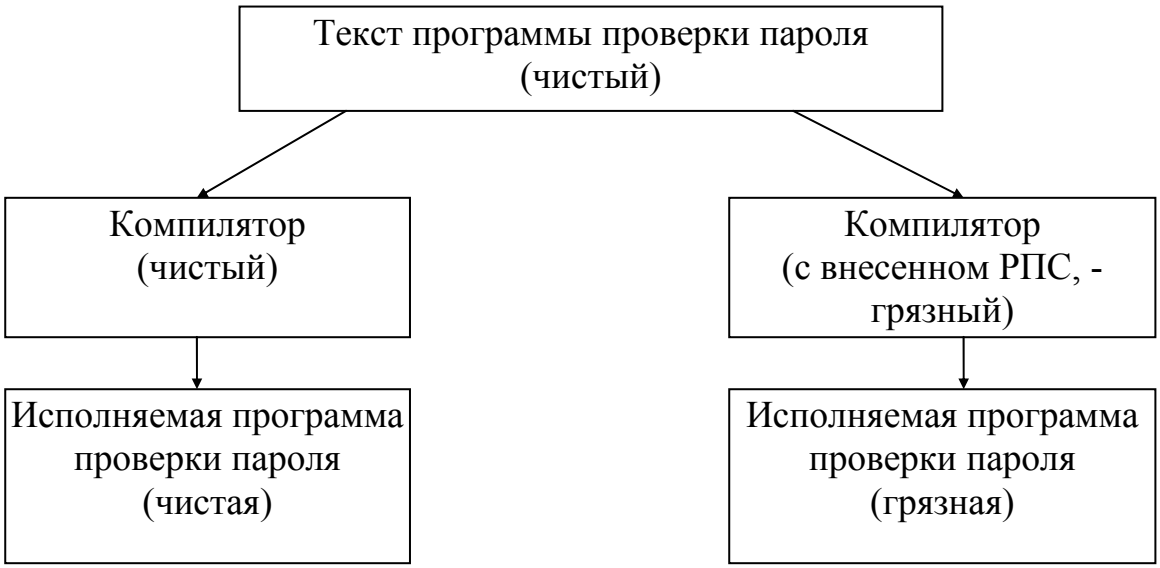

Рис.2.12. Работа компилятора с привнесенным РПС

2.6. Методы идентификации программ и их характеристик

2.6.1. 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 относится, например, распределение

операторов по тексту программы [15], 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 ой 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 выражена 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 может существенно уменьшаться.

2.6.2. 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.2.13 приведена гистограмма операторов для информационно - поисковой системы (ИПС) [15]; на абсциссе графика отмечено количество возможных операторов интерпретатора языка, используемого в этих текстах. Было выявлено [15], что некоторые операторы очень часто встречаются в программах различного назначения, некоторые редко, а некоторые вообще не встречаются. Для сравнения на рис.2.14 приведена гистограмма для программы редактирования текстов (РТ) [15], которая отличается от гистограммы на рис.2.13.

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.

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

$$A_n = \frac{S_n}{2} - D_n = \frac{f_n^{(1)} + f_n^{(2)}}{2} - \left| f_n^{(1)} - f_n^{(2)} \right|, \qquad (2.6.1)$$

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.

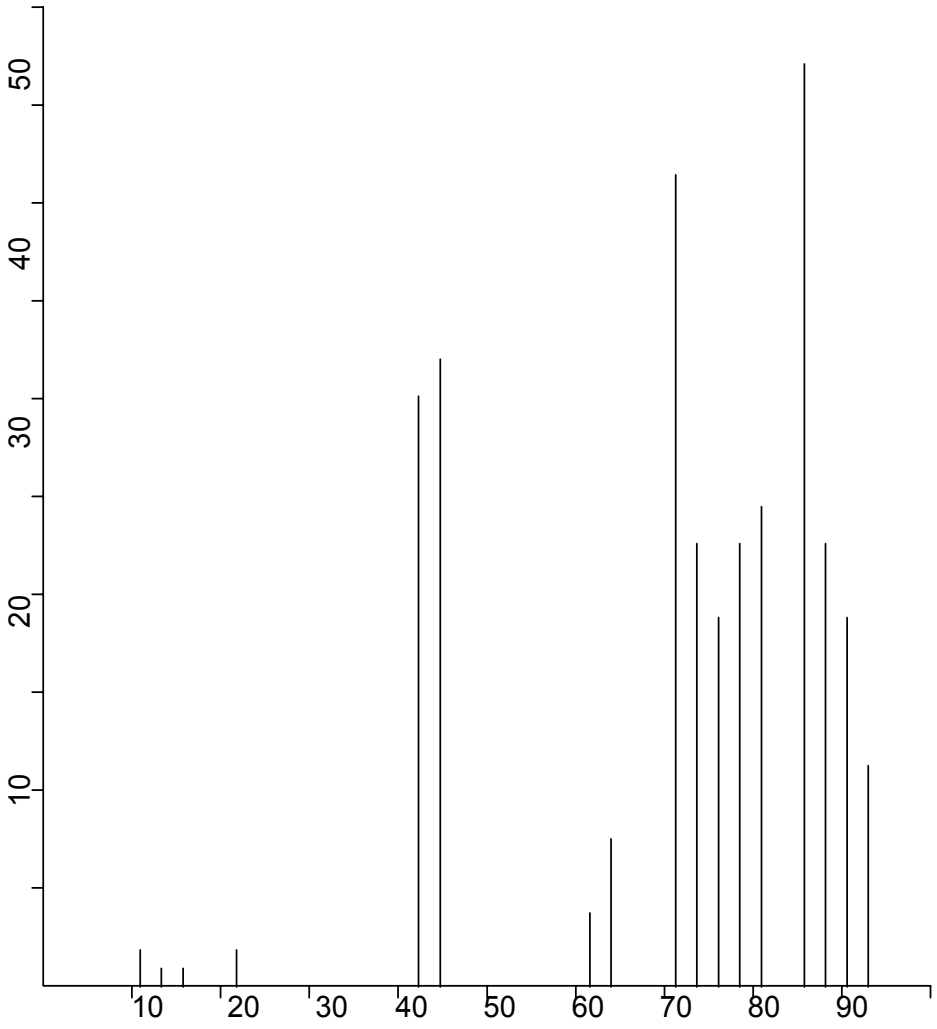

Рис.2.13. Гистограмма частоты появления операторов в программе для информационно-поисковой системы

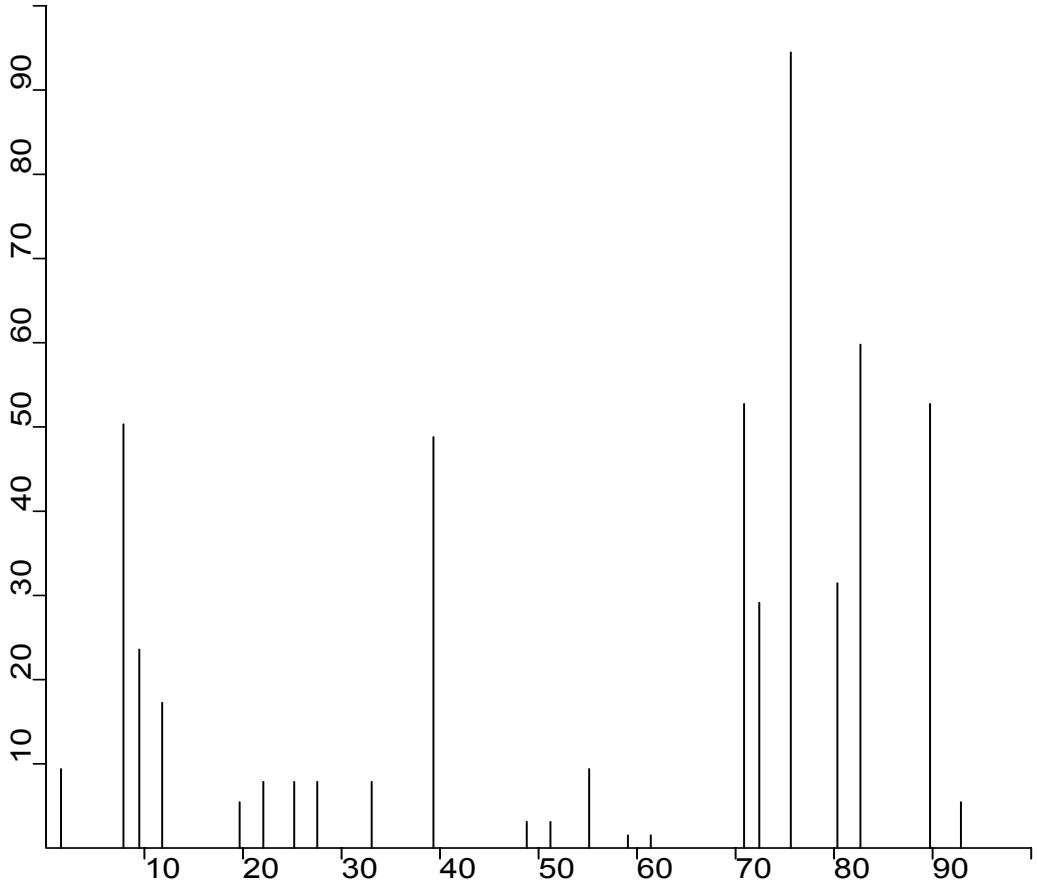

Рис.2.14. Гистограмма частоты появления операторов в программе для редакторов текста

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:

$$A(1,2) = \frac{\sum_{n=1}^{N} S_n - 2D_n}{\sum_{n=1}^{N} S_n}, \qquad (2.6.2)$$

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 (2.6.2) всегда находятся в пределах от 0 до 1. Если программы имеют почти один и тот же вид подобия 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 должен быть увеличен 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, от-

ношение 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 то 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. Автокорреляция представляет значительный интерес, поскольку дает некоторую числовую характеристику программы. По всей вероятности автокорреляционные функции различного типа можно использовать и 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 3. ОБЕСПЕЧЕНИЕ ЭКСПЛУАТАЦИОННОЙ БЕЗОПАСНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

3.1. Методы и средства защиты программ от компьютерных вирусов

3.2.1. 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 [3,5,16,59,65]. 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 которой 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 магнитные носители (дискеты), так и каналы вычислительных сетей. Ви-

русы, 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.

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.

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.

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.

MtE-вирусы делятся на

- обычные вирусы-мутанты, в разных копиях которых различаются только зашифрованные тела, а расшифровщики совпадают;
- полиморфные вирусы, в разных копиях которых различаются не только зашифрованные тела, но их дешифровщики.

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 дешифровщиков (с размером порождаемых дешифровщиков от 0 до 512 байтов) непохожих друг на друга. При этом в дешифровщиках могут встречаться практически все команды процессора Intel и даже использоваться некоторые специфические особенности его реального режимы 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 Exel for Windows, Lotus Ami Pro, Lotus 1-2-3, Lotus Notes, в операционных системах фирм Microsoft и Apple [5].

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 [59]. 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 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 системы.

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, уничтожение очагов инфекции в локальных и глобальных сетях не станет проще. Время компьютерных вирусов «общего назначения» уходит в прошлое.

3.2.2. Общая характеристика средств нейтрализации компьютерных вирусов

Наиболее распространенным средством нейтрализации ПВ 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 соот-

ветствующим 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 увеличения 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 [54] 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, разработка аппаратных средств защиты от вирусов и соблюдение технологии защиты от вирусов.

3.2.3. 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.

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 про-

грамм на соответствие значениям, указанным в документации. Большинство известных файловых и бутовых вирусов можно выявить на этапе входного контроля. Для этой цели используется батарея (несколько последовательно запускаемых программ) детекторов. 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 хранится 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 можно 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.

- *Каталог детекторов*. Детекторы, включенные в каталог, должны запускаться из операционной среды комплекса. При этом должна быть обеспечена возможность подключения к каталогу 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.* Отдельная компонента может 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.

3.2. Методы защиты программного обеспечения от внедрения на этапе его эксплуатации и сопровождения программных закладок

3.2.1. Классификация средств исследования программ

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 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 прерывания.

3.2.2. Методы защиты программ от исследования

Для защиты программ от исследования необходимо применять методы защиты от исследования файла с ее исполняемым кодом, 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 секретной части программы, а во втором - на блокировании доступа к исполняемому коду программы в оперативной памяти со стороны отладчиков [17].

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 секретную часть;
- деструктор (деициниализатор).

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 секретной части программы;
- передача управления секретной части программы.

Секретная часть программы предназначена для выполнения основных целевых функций программы и защищается шифрованием для предупреждения внесения в нее программной закладки.

Деструктор после выполнения секретной части программы должен выполнить следующие действия:

- обнуление секретного 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 секретная 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 безопасно-

сти, 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 секретных 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.

3.2.3. Анализ программ на этапе их эксплуатации

В данном разделе будут рассмотрены методы поиска и нейтрализации РПС с помощью дизассемблеров и отладчиков на этапе эксплуатации программ. То есть задача защиты в отличии задач защиты в предыдущих разделах здесь решается «с точностью до наоборот».

Основная схема анализа исполняемого кода, в данном случае, может выглядеть следующим образом [45] (см. также раздел 2.2.):

- 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 защиты осуществляется поиск сигнатур (лексем) РПС. Примеры сигнатур РПС приведены в работе [45]. Окончание этапа дизассемблирования предшествует синтаксическому анализу, то есть процессу отождествлению лексем, найденных во входной цепочке, одной из языко-

вых конструкций, задаваемых грамматикой языка, то есть синтаксический анализ исполняемого кода программ состоит в отождествлении сигнатур, найденных на этапе лексического анализа, одному из видов РПС.

При синтаксическом анализе могут встретиться следующие трудности:

- 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 об отсутствии или наличии 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 том, содержит ли входной исполняемый код РПС, и если да, то какого типа. При этом используется вся информация, полученная на всех предыдущих этапах. Кроме того, необходимо учитывать, что эта информация может считаться правильной лишь с некоторой вероятностью, причем не исключены вообще ложные факты, или умозаключения исследователей. В целом, задача семантического анализа является сложной и ресурсоемкой и скорее не может быть полностью автоматизирована.

3.3. Методы и средства обеспечения целостности и достоверности используемого программного кода

3.2.1. 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 может 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, закладка обнаружена 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 ГОСТ 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.

Можно еще более усложнить действия злоумышленника по нарушению целостности целевых программ, используя схемы подписи с верификацией по запросу [27,30]. 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.

3.2.2. Краткое описание криптографических средств контроля целостности и достоверности программ

Основные положения криптологии и базовые криптографические понятия

Термин «криптология» происходит от двух греческих слов: «крипто», что означает «тайный» и «логос», т.е. – учение. *Криптология* как наука, состоит из двух тесно теоретически и практически связанных дисциплин: криптографии и криптоанализа. *Криптография* - наука о способах преобразования (шифрования) информации с целью ее защиты от незаконных пользователей. *Криптоанализ* - наука (и практика ее применения) о методах и способах вскрытия шифров. Криптография и криптоанализ очевидным образом связаны друг с другом, так как не бывает хороших криптографов, не владеющих методами криптоанализа, и наоборот - хороший криптоаналитик должен быть знаком со всеми известными способами построения шифров. Ниже даются базовые понятия и определения криптологии, в т.ч. используемые и в настоящем разделе.

Шифр (криптосистема) - способ, метод преобразования информации с целью ее защиты от незаконных пользователей (от противника). Для противника возникает сложная задача вскрытия шифра. *Вскрытие (взламывание) шифра* - процесс получения информации из шифрованного сообщения (шифртекста) без знания примененного шифра.

Шифрование - процесс применения шифра к защищаемой информации, т.е. преобразование информации в шифрованное сообщение с помощью определенных правил, содержащихся в шифре.

Дешифрование - процесс, обратный шифрованию, т.е. преобразование шифрованного сообщения в защищаемую информацию с помощью определенных правил, содержащихся в шифре.

Исходное сообщение, имеющее, как правило, смысловое (логически значимое) содержание, которое необходимо зашифровать называется *открытым текстом*. Зашифрованное сообщение, имеющее, как правило, вид случайного набора символов (цифр) называется *шифртекстом* или *криптограммой*.

Под *ключом* в криптографии понимают сменный элемент шифра, который применен для шифрования конкретного открытого текста (сообщения).

В криптографии обычно общепринято следующее допущение. Криптоаналитик (противник) почти всегда имеет полный шифртекст. Помимо этого в криптографии принято правило Керкхоффа, которое гласит, что «стойкость шифра должна определяться только секретностью его ключа».

В этом случае задача противника сводится к попытке раскрытия шифра (попытке осуществления атаки) *на основе шифртекста*. Если же противник имеет к тому же некоторые отрывки открытого текста и соответствующие им элементы шифртекста, тогда он пытается осуществлять *атаку на основе открытого текста*. *Атака на основе выбранного открытого текста* заключается в том, что противник, используя свой открытый текст, получает правильный шифртекст (например, используя «вслепую» некоторую шифрмашину) и пытается в этом случае вскрыть шифр. Попытку раскрытия шифра можно осуществить, если противник подставляет свой ложный шифртекст и при дешифровании получает необходимый для раскрытия шифра открытый текст. Такой способ раскрытия называется *атакой на основе выбранного шифртекста*.

Теоретически существует *абсолютно стойкий шифр*, но единственным таким шифром является какая-нибудь форма так называемой ленты однократного использования (или так называемый «одноразовый блокнот»), в которой открытый текст «объединяется» с полностью случайным ключом такой же длины. Этот результат был доказан К. Шенноном с помощью разработанного им теоретико-информационного метода исследования шифров.

Для абсолютной стойкости существенным является каждое из следующих требований к ленте однократного использования:

- полная случайность (равновероятность) ключа (это, в частности, означает, что ключ нельзя вырабатывать с помощью какого-либо детерминированного устройства);
- равенство длины ключа и длины открытого текста;
- однократность использования ключа.

В то же время, именно эти условия и делают абсолютно стойкий шифр очень ресурсозатратным и непрактичным. Прежде чем пользоваться таким шифром, необходимо обеспечить всех абонентов достаточным запа-

сом случайных ключей и исключить возможность их повторного применения. А это сделать необычайно трудно и дорого. В силу данных причин абсолютно стойкие шифры применяются только в сетях связи с небольшим объемом передаваемой информации, обычно это сети для передачи особо важной государственной информации.

В 1976 г. опубликовав свою работу «Новые направления в криптографии», американские ученые У. Диффи и М. Хеллман выдвинули следующую удивительную гипотезу: «Возможно построение практически стойких криптосистем, вообще не требующих передачи секретного ключа». Такие криптосистемы, получившие название *криптосистем с открытым ключом*, основываются на введении понятий «односторонней функции» и «односторонней функции с секретом». Понятие односторонней функции было введено в подразделе 2.4

Вопрос о существовании односторонней функции с секретом является столь же гипотетическим, что и вопрос о существовании односторонней функции. Для практических целей было построено несколько функций, которые могут оказаться односторонними, а это означает, что задача инвертирования эквивалентна некоторой давно изучаемой трудной математической задаче (см., например, [70]).

Применение односторонних функций в криптографии позволяет: во-первых, организовать обмен шифрованными сообщениями с использованием только открытых каналов связи и, во-вторых, решать новые криптографические задачи, такие как электронная цифровая подпись.

В большинстве схем электронной подписи используются хэш-функции. Это объясняется тем, что практические схемы электронной подписи не способны подписывать сообщения произвольной длины, а процедура, состоящая в разбиении сообщения на блоки и в генерации подписи для каждого блока по отдельности, крайне неэффективна. Под термином «*хэш-функция*» понимается функция, отображающие сообщения произвольной длины в значение фиксированной длины, которое называется *хэш-кодом*.

Далее рассмотрим базовые криптографические методы, широко применяющиеся в современных системах обеспечения безопасности информации.

Краткое описание основных криптографических методов защиты данных

Симметричный шифр ГОСТ 28147-89

Пусть L и R - последовательности битов, LR означает их конкатенацию. Под обозначением $\oplus$ будет пониматься операция сложения по модулю 2 или логическая операция XOR (исключающая ИЛИ), символом [+] -

операция сложения по модулю 2^{32} двух 32-разрядных чисел. Числа суммируются по следующему правилу:

$$A[+]B=A\oplus B, \text{ если } A\oplus B<2^{32}$$

$$A[+]\text{B}=A\oplus B-2^{32}, \text{ если } A\oplus B\geq 2^{32}.$$

Символом {+} обозначается операция сложения по модуль 2^{32}-1 двух 32-разрядных чисел. Правила суммирования чисел следующие:

$$A\{+\}B=A\oplus B, \text{ если } A\oplus B<2^{32}-1$$

$$A\{+\}\text{B}=A\oplus B-2^{32}, \text{ если } A\oplus B\geq 2^{32}-1.$$

Во всех режимах работы алгоритма используется ключ длиной 256 битов, который представляется в виде восьми 32-разрадных чисел $X(i)$.Если обозначить ключ через W, то

$$W=X(7)X(6)X(5)X(4)X(3)X(2)X(1)X(0).$$

Дешифрование, как и в любой симметричной криптосистеме осуществляется на том же ключе, что и шифрование.

Ниже приводится описание двух наиболее используемых режимов шифра: режима простой замены и режима генерации имитовставки [6].

Описание режима простой замены. Код программы T разбивается на блоки по 64 бита в каждом, которые обозначаются $T(j)$. Очередная последовательность битов $T(j)$ разбивается на две последовательности $B(0)$ (левые или старшие биты) и A(0) (правые или младшие биты), каждая из которых содержит 32 бита. Затем выполняется итеративный процесс шифрования, который описывается следующими формулами:

при $i=1,2,...,24$; $j=i-1(\text{mod } 8)$

$$A(i)=f(A(i-1)[+]X(j)(+)B(i-1));$$

$$B(i)-A(i-1);$$

при $i=25,26,...,31$; $j=32-i$

$$A(i)=f(A(i-1)[+]X(j)(+)B(i-1));$$

$$B(i)-A(i-1);$$

при $i=32$

$$A(32)=A(31);$$

$$B(32)=f(A(31)[+]X(0)(+)B(31)),$$

где i обозначает номер итерации ($i=1,2,...,32$). Функция f называется функцией шифрования. Ее аргументом является сумма по модулю 2^{32} числа $A(i)$, полученного на предыдущем шаге итерации, в числа $X(j)$ ключа (размерность каждого из этих чисел 32 знакам).

Функция шифрования включает две операции над полученной 32-разрядной суммой. Первая операция называется подстановкой K. Блок подстановки K состоит из восьми узлов замены $K(1)...K(8)$ с памятью 64 бита каждый. Поступающий на блок подстановки 32-разрядный вектор разбивается на 8 последовательно идущих 4-разрядных векторов, каждый из которых преобразуется в 4-разрядный вектор соответствующим узлом

замены, представляющим собой таблицу из 16 целых чисел в диапазоне 0,...,15.

Входной вектор определяет адрес строки в таблице, число из которой является выходным вектором. Затем 4-разрядные выходные векторы последовательно объединяются в 32-разрядный вектор. Таблицы блока подстановки блока подстановки *K* содержат редко изменяемые ключевые элементы, общие для некоторой компьютерной системы.

Вторая операция - циклический сдвиг влево 32-разрядного вектора, полученного в результате подстановки *K*. 64-разрядный блок зашифрованных данных $T_ш$ представляется в виде:

$$T_ш = A(32)B(32).$$

Остальные блоки кода программы в режиме простой замены шифруются аналогично.

Режим генерации имитовставки. Для получения имитовставки код программы представляется в виде 64-разрядных блоков $T(i)$, =1,2,..,m. Где m определяет объем кода программы. Первый блок кода программы $T(i)$ подвергается преобразованию, соответствующему первым 16 циклам алгоритма шифрования в режиме простой замены, причем в качестве ключа для выработки имитовставки используется ключ, по которому шифруются данные.

Полученное после 16 циклов работы 64-разрядное число суммируется по модулю 2 со вторым блоком открытых данных $T(2)$. Результат суммирования снова подвергается преобразованию, соответствующему 16 циклам алгоритма шифрования в режиме простой замены. Полученное 64-разрядной число суммируется по модулю 2 с третьим блоком данных $T(3)$ и т.д. Последний блок $T(m)$, при необходимости дополненный до полного 64-разрядного блока нулями, суммируется по модулю 2 с результатом работы на шаге m-1, после чего шифруется в режиме простой замены по первым 16 циклам алгоритма. Из полученного 64-разрядного числа выбирается отрезок $И_p$ длиной p битов. Данный отрезок и является имитовставкой $И_p$, полученной для кода программы T.

Открытый ключевой обмен Диффи-Хеллмана и криптосистемы с открытым ключом

Основная задача ключевого обмена Диффи-Хеллмана заключается в следующем: «Каким образом можно установить секретный ключ между абонентами **A** и **B** по открытому каналу связи а затем использовать его для шифрованной передачи сообщений?» Для этих целей, пусть абонент **A** выбирает какую-нибудь функцию f_k с секретом k. Он публикует в открытом сертифицированном справочнике описание функции f_k в качестве своего алгоритма шифрования. Однако, значение секрета k он никому не сообщает, т.е. держит его в тайне от других.

Если абонент **B** хочет послать **A** защищаемую информацию $x \in X$, то он вычисляет $y=f_k(x)$ и посылает y по открытому каналу к **A**. Поскольку **A** для своего секрета k умеет инвертировать f_k, то он вычисляет x по полученному y. Так как никто другой не знает k и поэтому в силу свойств функции с секретом не сможет за полиномиальное время по известному шифрованному сообщению y вычислить защищаемую информацию x.

Описанная выше схема является криптосистемой с открытым ключом, поскольку алгоритм шифрования f_k является общедоступным или открытым. Такие криптосистемы называют еще *асимметричными*, поскольку в них есть асимметрия в алгоритмах: алгоритмы шифрования и дешифрования различны. В отличие от таких систем традиционные шифры называют *симметричными*, так как в них ключ для шифрования и дешифрования один и тот же, и именно поэтому его нужно хранить в секрете. Для асимметричных систем алгоритм шифрования общеизвестен, но восстановить по нему алгоритм дешифрования за полиномиальное время невозможно.

Электронная цифровая подпись

Основные определения, обозначения и алгоритмы. Для реализации схем *электронной цифровой подписи* (или просто цифровой подписи) требуются три следующих эффективно функционирующих алгоритма:

- A_k - алгоритм генерации секретного и открытого ключей для подписи кода программы, а также проверки подписи, - s и p соответственно;
- A_s - алгоритм генерации (проставления) подписи с использованием секретного ключа s;
- A_p - алгоритм проверки (верификации) подписи с использованием открытого ключа p.

Алгоритмы должны быть разработаны так, чтобы выполнялось основное принципиальное свойство, - свойство невозможности получения нарушителем (противником) алгоритма A_s из алгоритма A_p.

Таким образом, если A_k - алгоритм генерации ключей, тогда определим значения $(s,p)=A_k(\alpha,\beta)$ как указанные выше сгенерированные ключи, где α - некоторый параметр безопасности (как правило, длина ключей), а β - параметр, характеризующий случайный характер работы алгоритма A_k при каждом его вызове.

Ключ s хранится в секрете, а открытый ключ p делается общедоступным. Это делается, как правило, путем помещения открытых ключей пользователей в открытый сертифицированный справочник. Сертификация открытых ключей справочника выполняется некоторым дополнительным надежным элементом, которому все пользователи системы доверяют обработку этих ключей. Обычно этот элемент называют Центром обеспечения безопасности или Центром доверия.

Непосредственно процесс подписи осуществляется посредством алгоритма A_s. В этом случае значение $c=A_s(m)$ - есть подпись кода программы m, полученная при помощи алгоритма A_s и ключа s.

Процесс верификации выполняется следующим образом. Пусть m^* и c^* - код программы и подпись для этого кода соответственно, A_p - алгоритм верификации. Тогда, выбрав из справочника общедоступный открытый ключ p, можно выполнить алгоритм верификации: $b=A_p(m^*,c^*)$, где предикат $b\in\{\text{true,false}\}$. Если b=true, то код m^* действительно соответствует подписи c^*, полученной при помощи секретного ключа s, который, в свою очередь, соответствует открытому ключу p, то есть $m=m^*$, $c=c^*$ и наоборот если b=false, то код программы ложен и (или) код подписан ложным ключом.

Примеры схем электронной цифровой подписи. К наиболее известным схемам цифровой подписи с прикладной точки зрения относятся схемы RSA, схемы Рабина-Уильямса, Эль-Гамаля, Шнорра и Фиата-Шамира [15], а также схема электронной цифровой подписи отечественного стандарта ГОСТ Р 34.10-94 [7].

Рассмотрим несколько подробнее четыре наиболее известные схемы цифровой подписи:

Подпись RSA: *Вход:* Числа n, p, q, где p и q - большие l - разрядные простые числа, $n=pq$. Открытый ключ $p=(e,n)$, секретный ключ $s=d$, такой, что $ed\equiv1(\text{mod }\varphi(n))$ и наибольший общий делитель НОД$(e,\varphi(n))=1$, где $\varphi(n)=(p-1)(q-1)$.

Генерация ключей: $(d,(e,n))=A_k(l,b)$.

Подпись: A_s: 1.$m^d(\text{mod }n)\equiv c$, где c - подпись кода программы m.

Верификация: A_p: 1.$[c^*]^e(\text{mod }n)\equiv m^{**}$,
2.если $m^{**}=m^*$, подпись верна.

Подпись Эль-Гамаля: *Вход:* Числа p и g, где p - простое l - разрядное число, а g - первообразный корень по модулю p. Секретный ключ $s=d$, открытый ключ $p=e$, такой, что $e\equiv g^d(\text{mod }p)$, m - подписываемый код программы.

Генерация ключей: $(d,(e,g,p))=A_k(l,b)$.

Подпись: A_s: 1.$r\equiv g^k(\text{mod }p)$, где $k\in_R Z_{p-1}$;
2.находится такое c, что $m\equiv[kc+dr](\text{mod }p-1)$, где (c,r) - подпись кода m.

Верификация: A_p: 1.если $g^{m^*}=e^r r^{c^*}(\text{mod }p)$, то подпись верна.

Подпись Фиата - Шамира: *Вход:* Числа n, p, и q, где p и q большие l - разрядные простые числа, открытый ключ p есть вектор $(v_1,v_2,...,v_k)$, где v_j -

квадратичные вычеты по модулю n, $j=\overline{1,k}$, секретный ключ p есть вектор $(s_1,s_2,...,s_k)$, где каждый s_j - наименьший квадратный корень из v^{-1}_j, m - подписываемый код, f - псевдослучайная функция.

Генерация ключей: $((s_1,s_2,...,s_k),(v_1,v_2,...,v_k))=A_k(l,b)$.

Подпись: A_s: 1.$x_i \equiv r^2_i (\mathrm{mod}\ n)$, где $\{r_1,r_2,...,r_k\} \in_R Z_n$;
2.вычисляется значение $a=f(m,x_1,x_2,...,x_t)$;
3.выбирается первые kt битов числа a как матрица e_{ij} ($i=\overline{1,t}$, $j=\overline{1,k}$);
4.вычисляется: $y_i \equiv r_i \prod_{e_{ij}=1} s_j (\mathrm{mod}\ n)$,
где $i=\overline{1,t}$.
Тогда (e_{ij},y_i) - подпись кода m.

Верификация: A_p: 1. $z_i \equiv [y^*_i]^2 \prod_{e^*_{ij}=1} v_j (\mathrm{mod}\ n)$, где $i=\overline{1,t}$.
2.если первые kt битов значения функции $f(m^*,z_1,z_2,...,z_t)$ равны e^*_{ij}, подпись верна.

Подпись стандарта ГОСТ Р 34.10-94. *Вход:* Числа p, g и q где p - простое l -разрядное число, g - первообразный корень по модулю p, а q - большой простой делитель p-1. Пусть также $g^q \equiv 1 (\mathrm{mod}\ p)$, $g \neq 1$. Секретный ключ подписи x ($1<x<q$) и открытый ключ $y \equiv g^x (\mathrm{mod}\ p)$.

Генерация ключей: $(x,(g,p,q,y))=A_k(l,b)$.

Подпись: A_x: 1.$r \equiv [g^k (\mathrm{mod}\ p)] (\mathrm{mod}\ q)$, где $k \in_R Z_q$.
2.$s \equiv [xr+km)] (\mathrm{mod}\ q)$, где $m=h(M)$ - рассматривается как значение хэш-функции h, соответствующей отечественному стандарту на функцию хэширования сообщений ГОСТ Р 34.11-94. Таким образом, пара (r,s) - есть подпись кода программы M.

Верификация: A_y: 1.$v \equiv m^{-1} (\mathrm{mod}\ q)$.
2.$u \equiv [g^{sv} y^{-rv} (\mathrm{mod}\ p)] (\mathrm{mod}\ q)$;
3.Если $u=r$, то (r,s) - есть подпись кода программы M, где $m=h(M)$.

Под *стойкостью схемы цифровой подписи* понимается стойкость в теоретико-сложностном смысле, то есть, как отсутствие эффективных алгоритмов ее подделки и (или) раскрытия [6]. По-видимому, до сих пор основной является следующая классификация:

•*атака с открытым ключом*, когда противник знает только открытый ключ схемы;

•*атака на основе известного открытого текста* (известного кода программы), когда противник получает подписи для некоторого (ограниченного) количества известных ему кодов (при этом противник никак не может повлиять на выбор этих кодов);

•*атака на основе выбранного открытого текста*, когда противник может получить подписи для некоторого ограниченного количества выбранных им кодов программы (предполагается, что коды выбираются независимо от открытого ключа, например, до того как открытый ключ станет известен);

•*направленная атака на основе выбранного открытого текста* (то же, что предыдущая, но противник, выбирая код программы, уже знает открытый ключ);

•*адаптивная атака на основе выбранного открытого текста*, когда противник выбирает коды программы последовательно, зная открытый ключ и зная на каждом шаге подписи для всех ранее выбранных кодов.

Атаки перечислены таким образом, что каждая последующая сильнее предыдущей.

Угрозами для схем цифровой подписи являются раскрытие схемы или подделка подписи. Определяются следующие типы угроз:

•*полного раскрытия*, когда противник в состоянии вычислить секретный ключ подписи;

•*универсальной подделки*, когда противник находит алгоритм, функционально эквивалентный алгоритму вычисления подписи;

•*селективной подделки*, когда осуществляется подделка подписи для кода программы, выбранного противником априори;

•*экзистенциональной подделки*, когда осуществляется подделка подписи хотя бы для одного кода программы (при этом противник не контролирует выбор этого кода, которое может быть вообще случайным или бессмысленным).

Угрозы перечислены в порядке ослабления. Стойкость схемы электронной подписи определяется относительно пары (тип атаки, угроза).

Если принять вышеописанную классификацию атак и угроз, то наиболее привлекательной является схема цифровой подписи, стойкая против самой слабой из угроз, в предположении, что противник может провести самую сильную из всех атак. В этом случае, такая схема должна быть стойкой против экзистенциональной подделки с адаптивной атакой на основе выбранного открытого текста.

Разновидности схем подписи электронных сообщений. Для обеспечения целостности и достоверности информации при ее передаче и хранении,

а также для проведения аутентификации абонентов системы и решения других задач защиты существует достаточно много всевозможных разновидностей схем подписи [15]. Без подробного описания ниже приводится одна из разновидностей схем подписи, которая, как правило, использует рассмотренный выше математический аппарат и может использоваться для защиты программ.

Схема подписи с верификацией по запросу. В работах Д. Шаума (см., например, [27,30]) 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 [30]. 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).

I. Генерация подписи. 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 $w \equiv g^u \pmod p$. Открытый ключ P публикуется в открытом сертифицированном справочнике.

Подпись кода m вычисляется следующим образом. Выбирается $k \in_R Z_q$ и вычисляется $r \equiv g^k \pmod p$. Затем вычисляется $s \equiv [xr+mku] \pmod q$. Пара (r,s) является подписью для кода m. Подпись считается корректной тогда и только тогда, когда $r^u \equiv g^{sw} y^{-rw} \pmod p$, где $w \equiv m^{-1} \pmod q$.

Проверка подписи (с участием подписывающего) осуществляется посредством следующего интерактивного протокола.

II. Протокол верификации. Абонент вычисляет $\gamma \equiv g^{sw} y^{-rw} \pmod p$ 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 $\delta \equiv r^a g^b \pmod p$ и посылает δ абоненту **S**.

2. Абонент **S** выбирает $t \in_R Z_q$, вычисляет $h_1 \equiv \delta g^t \pmod p$, $h_2 \equiv h_1^u \pmod p$ и посылает h_1 и h_2 абоненту **V**.

3. Абонент **V** высылает параметры a и b.

4. Если $\delta \equiv r^a g^b \pmod p$, то абонент **S** посылает **V** параметр t; в противном случае - останавливается.

5. Абонент **V** проверяет выполнение равенств $h_1 \equiv r^a g^{b+t} \pmod p$ и $h_2 \equiv \gamma^a w^{b+t} \pmod p$.

Если проверка завершена успешно, то подпись принимается как корректная.

III. Отвергающий протокол. 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 $a \equiv g^e \pmod p$, $b \equiv w^e \pmod p$, если $\beta=0$ и $a \equiv r^e \pmod p$, $b \equiv \gamma^e \pmod p$, если $\beta=1$. Посылает **S** значения a, b, d.

2. Абонент **S** проверяет соотношение $a^u \pmod p \equiv b$. Если оно выполняется, то $\alpha=0$, в противном случае $\alpha=1$. Выбирает $R \in_R Z_q$, вычисляет $c \equiv d^\alpha g^R \pmod p$ 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: $a \equiv g^e \pmod p$, $b \equiv w^e \pmod p$ и $a \equiv r^e \pmod p$, $b \equiv \gamma^e \pmod p$. Если да, то посылает **V** значение R. Иначе останавливается.

5. Абонент **V** проверяет, что $d^\beta g^R \pmod p \equiv c$.

Если во всех l циклах проверка в п.5 выполнена успешно, то абонент **V** принимает доказательства.

Таблица 3.1. Протокол верификации является интерактивным протоколом доказательств с абсолютно нулевым разглашением.

Доказательство. Требуется доказать, что вышеприведенный протокол удовлетворяет трем свойствам: полноты, корректности и нулевого разглашения [27,29].

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

$$h_1 \equiv r^{a_1} g^{b_1+t_1} \equiv r^{a_2} g^{b_2+t_2} \pmod{p}$$

и

$$h_2 \equiv \gamma^{a_1} w^{b_1+t_1} \equiv \gamma^{a_2} w^{b_2+t_1} \pmod{p},$$

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

$$\gamma \equiv w^{\lambda} \equiv g^{\lambda \log_g^{(p)} w} \equiv r^{\log_g^{(p)} w} \pmod{p},$$

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 доказательст-

вом 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 $\eta \in_R Z_q$ и вычисляет $h_1' \equiv g^{\eta} (\mathrm{mod} p)$ и $h_2' \equiv r^{\eta} (\mathrm{mod}\, p)$.

3. Получает от $\mathbf{V}^*$ значения a и b. Если $\delta \neq r^a g^b (\mathrm{mod}\, p)$, то M_{V^*} останавливается.

4. Машина M_{V^*} "отматывает" $\mathbf{V}^*$ на состояние, которое было запомнено в конце шага 1. Выбирает $t \in_R Z_q$ и вычисляет $h_1 \equiv r^a g^{b+t} (\mathrm{mod}\, p)$ и $h_2 \equiv \gamma^a w^{b+t} (\mathrm{mod}\, p)$.

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 $\delta \equiv r^a g^b \equiv r^{a'} g^{b'} (\mathrm{mod}\, p)$. Предположим, что $b \neq b'$. Из этого следует, что $a \neq a'$. Следовательно, M_{V^*} может вычислить $r \equiv g^{(b'-b)/(a-a')} (\mathrm{mod}\, p)$. 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 $h_1 \equiv g^{\eta} (\mathrm{mod}\, p)$ и $h_2 \equiv w^{\eta} (\mathrm{mod}\, p)$ и передает их V^*.

8. Получает от $\mathbf{V}^*$ значения a и b. Если $\delta \neq r^a g^b (\mathrm{mod}\, p)$, то M_{V^*} останавливается. В противном случае вычисляет $t=[\eta - al - b] (\mathrm{mod}\, q)$, 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

$$h_2 \equiv w^{\eta} \equiv w^{b+t} w^{al} \equiv w^{b+t} \gamma^{a} (\mathrm{mod}\, p).$$

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 3.2. 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 $c \equiv dg^{R_1} (\mathrm{mod}\, p)$ и $c \equiv g^{R_2} (\mathrm{mod}\, p)$, 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 $c \equiv d^{\alpha} g^{R} (\mathrm{mod}\, p)$. 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 $a \equiv g^{e} (\mathrm{mod}\, p)$, $b \equiv w^{e} (\mathrm{mod}\, p)$ 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 [27] 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.

Процедуры арбитража. Часто разработчиками схем цифровой подписи при их создании, наряду с процедурами генерации и верификации цифровой подписи, не приводятся процедуры арбитража, которые позволяют установить корректность/ некорректность подписи в случае возникновения

споров между участниками схемы. Проблеме арбитража в литературе уделяется мало внимание еще по-видимому и потому, что разработка арбитражных процедур считается прерогативой конкретного пользователя системы.

В случае возникновения проблемы по поводу корректности подписи основные действия участников процесса должны сводиться к следующему:

1). Абонент *V* предъявляет арбитру сообщение и его подпись.

2). Арбитр требует предъявить секретный ключ подписи абоненту *S*. Если абонент *S* отказывается, тогда арбитр дает заключение, что подпись подлинная. (То есть, в данном случае возможное нарушение сводится к тому, что абонент отказывается признать представленную подпись своей).

3). Арбитр выбирает из открытого сертифицированного справочника открытый ключ абонента *S* и проверяет его соответствие секретному ключу подписи. При обнаружения несоответствия (например, по вине службы ведения такого справочника) арбитр признает подпись, предъявленную абонентом *V*, подлинной.

4). Арбитр проверяет корректность подписи при помощи проверочных соотношений, используемых как при генерации, так и при верификации данной подписи.

Одна из основных проблем при арбитраже связана с тем, что арбитр получает секретный ключ абонента *S*. В том случае, если арбитр является нечестным, то тогда никаких способов защиты пользователей системы от него не может быть в принципе. Помимо этого, следует также сказать о том, что желательно, чтобы пользователь генерировал свои секретные ключи самостоятельно, так как даже создание специального центра генерации ключей не может полностью обезопасить пользователей от слабых ключей и полностью исключает в данном случае возможность решения споров в суде.

Хэш-функции

Основные определения, обозначения и алгоритмы. Под термином «*хэш-функция*» понимается функция, отображающая электронные данные произвольной длины (иногда длина ограничена, но достаточно большим числом), в значения фиксированной длины. Последние часто называют *хэш-кодами*. Таким образом, у всякой хэш-функции h имеется большое количество *коллизий*, то есть пар значений x и y таких, что $h(x)=h(y)$. Основное требование, предъявляемое криптографическими приложениями к хэш-функциям, состоит в отсутствии эффективных алгоритмов поиска коллизий. Хэш-функция, обладающая таким свойством, называется *хэш-функцией, свободной от коллизий*. Кроме того, хэш-функция должна быть *односторонней*, то есть функцией, по значению которой вычислительно

трудно найти ее аргумент и, в то же время, функцией, для аргумента которой, вычислительно трудно найти другой аргумент, который давал бы то же самое значение функции.

Таким образом, хэш-функции вместе со схемами электронной цифровой подписи предназначены для решения задач обеспечения целостности и достоверности электронных данных. В прикладных компьютерных системах требуется применение так называемых криптографически стойких хэш-функций. Под термином «*криптографически стойкая хэш-функция*» понимается функция h, которая является односторонней и свободной от коллизий.

Введем следующие обозначения. Хэш-функция h обозначается как $h(\alpha)$ и $h(\alpha,\beta)$ для одного и двух аргументов соответственно. Хэш-код функции h обозначается как H. При этом $H_0=I$ обозначает начальное значение (вектор инициализации) хэш-функции. Под обозначением $\oplus$ будет пониматься операция сложения по модулю 2 или логическая операция XOR (исключающая ИЛИ). Результат шифрования блока B блочным шифром на ключе k обозначается $E_k(B)$.

Для лучшего понимания дальнейшего материала приведем небольшой пример построения хэш-функции. Предположим нам необходимо получить хэш-код для некоторой программы M. В качестве шифрующего преобразования в хэш-функции будут использоваться процедуры шифра DES с ключом k. Тогда, чтобы получить хэш-код H программы M при помощи хэш-функции h необходимо выполнить следующую итеративную операцию:

$$H_i = E_{H_{i-1}}(M_i) \oplus M_i \text{, где } i = \overline{1,n}\text{, } H_0=I\text{, } M=M_1,M_2,...,M_n,$$

где код программы M разбит на n 64-битных блока. Хэш-кодом данной хэш-функции является значение $H=h(M,I)=H_n$.

Стойкость (безопасность) хэш-функций. Один из основных методов криптоанализа хэш-функций заключается в проведении криптоаналитиком (противником) атаки, основанной на «парадоксе дня рождения», основные идеи которой, излагаются ниже. Для понимания сущности предлагаемого метода криптоанализа достаточно знать элементарную теорию вероятностей.

Атака, основанная на «парадоксе дня рождения», заключается в следующем. Пусть $\alpha\sqrt{n}$ предметов выбираются с возвращением из некоторого множества с мощностью n. Тогда вероятность того, что два из них окажутся одинаковыми, составляют $1-e^{-\alpha^2/2}$.

Практически это означает, что в случайно подобранной группе из 24 человек вероятность наличия двух лиц с одним и тем же днем рождения

составляет значение примерно 1/2. Этот старый и хорошо изученный парадокс и положен в основу криптоанализа хэш-функций.

Например, для криптоанализа хэш-функций, основанных на использовании криптоалгоритма DES, указанная атака может быть проведена следующим образом. Пусть n - мощность области хэш-кодов (для криптоалгоритма DES она равна 2^{64}). Предположим, что есть две программы m_1 и m_2. Первая программа достоверна, а для второй криптоаналитик пытается получить то же самое значение хэш-кода, выдав таким образом программу m_1 за программу m_2 (то есть криптоаналитик пытается получить коллизию). Для этого криптоаналитик подготавливает порядка $\sqrt{n}$ различных, незначительно отличающихся версий m_1 и m_2 и для каждой из них вычисляет хэш-код. С высокой вероятностью криптоаналитику удается обнаружить пару версий m_1' и m_2', имеющих один и тот же хэш-код. В дальнейшем при использовании данного хэш-кода можно выдать программу m_2' вместо программы m_1', содержание которой близко к содержанию программы m_1.

К основным методам предотвращения данной атаки можно отнести: увеличение длины получаемых хэш-кодов (увеличение мощности области хэш-кодов) и выполнение требования итерированности шифрующего преобразования.

Методы построения криптографически стойких хэш-функций. Практические методы построения хэш-функций можно условно разделить на три группы: методы построения хэш-функций на основе какого-либо алгоритма шифрования (пример, приведенный выше), методы построения хэш-функций на основе какой-либо известной вычислительно трудной математической задачи и методы построения хэш-функций «с нуля» [28].

Рассмотрим примеры построения *хэш-функций на основе алгоритмов шифрования*. Наряду с примером, приведенным выше, покажем, как строить хэш-функции на основе наиболее известных блочных шифров ГОСТ 28147 - 89, DES и FEAL.

В качестве шифрующего преобразования будут использоваться некоторые режимы шифров ГОСТ 28147-89, DES и FEAL с ключом k. Тогда, чтобы получить хэш-код H программы M при помощи хэш-функции h, необходимо выполнить следующую итеративную операцию (например, с использованием алгоритма ГОСТ 28147-89):

$$H_i = E_k(M_i \oplus H_{i-1}), \text{ где } i = \overline{1,n}, H_0=I, M=M_1,M_2,...,M_n.$$

Таким образом, хэш-кодом данной хэш-функции является значение $H=h(M,I)=H_n$. При этом используется режим выработки имитовставки ГОСТ 28147-89, а шифрующее преобразование 64-битных блоков заклю-

чается в выполнении 16 циклов алгоритма шифрования в режиме простой замены.

Алгоритм ГОСТ 28147-89 в качестве базового используется в хэш-функции отечественного стандарта на функцию хэширования сообщений ГОСТ Р 34.11-94, являющегося основным практическим инструментом в компьютерных системах, требующих обеспечения достоверности и целостности электронных данных.

Алгоритм DES (в режиме CFB) можно использовать в качестве базового, например, в следующей хэш-функции (с получением хэш-кода $H=h(M,I)=H_n$):

$$H_i = E_{M_i}(H_{i-1}), \text{ где } i = \overline{1,n},\ H_0=I,\ M=M_1,M_2,...,M_n.$$

Другие примеры хэш-функций на основе алгоритмов шифрования. N-хэш алгоритм разработан фирмой Nippon Telephone Telegraph в 1990 году. *N*-хэш алгоритм использует блоки длиной 128 битов, шифрующую функцию, аналогичную функции алгоритма шифрования FEAL, и вырабатывает блок хэш-кода длиной 128 битов. На вход пошаговой хэш-функции в качестве аргументов поступают очередной блок кода M_i длиной 128 битов и хэш-код H_{i-1} предыдущего шага также размером 128 битов. При этом $H_0=I$, а $H_i=E_k(M_i,H_{i-1})\oplus M_i\oplus H_{i-1}$. Хэш-кодом всего кода программы объявляется хэш-код, получаемый в результате преобразования последнего блока текста.

Идея использовать алгоритм блочного шифрования, стойкость которого общеизвестна, для построения надежных схем хэширования выглядит естественной. Однако для некоторых таких хэш-функций возникает следующая проблема. Предлагаемые методы могут требовать задания некоторого ключа, на котором происходит шифрование. В дальнейшем этот ключ необходимо держать в секрете, ибо противник, зная этот ключ и значение хэш-кода, может выполнить процедуру в обратном направлении.

Еще одной слабостью указанных выше схем хэширования является то, что размер хэш-кода совпадает с размером блока алгоритма шифрования. Чаще всего размер блока недостаточен для того, чтобы схема была стойкой против атаки на основе «парадокса дня рождения». Поэтому были предприняты попытки построения хэш-функций на базе блочного шифра с размером хэш-кода в k раз (как правило, k=2) большим, чем размер блока алгоритма шифрования.

В качестве примера можно привести хэш-функции MDC2 и MDC4 фирмы IBM. Данные хэш-функции используют блочный шифр (в оригинале DES) для получения хэш-кода, длина которого в два раза больше длины блока шифра. Алгоритм MDC2 работает несколько быстрее, чем MDC4, но представляется несколько менее стойким.

Следует также отметить, что существует два основных режима применения хэш-функций: поточный и блочный. Выбор режима зависит от используемого в хэш-функции шифрующего преобразования.

В качестве примера *хэш-функций, построенных на основе вычислительно трудной математической задачи* можно привести функцию из рекомендаций MKKTT X.509.

Криптографическая стойкость данной функции основана на сложности решения следующей труднорешаемой теоретико-числовой задачи. Задача умножения двух больших (длиной в несколько сотен битов) простых чисел является простой с вычислительной точки зрения, в то время как факторизация (разложение на простые множители) полученного произведения является труднорешаемой задачей для указанных размерностей.

Следует отметить, что задача разложения числа на простые множители эквивалентна следующей труднорешаемой математической задаче. Пусть $n=pq$ произведение двух простых чисел p и q. В этом случае можно легко вычислить квадрат числа по модулю n: $x^2(\mathrm{mod}\, n)$, однако вычислительно трудно извлечь квадратный корень по этому модулю.

Таким образом, хэш-функцию MKKTT X.509 можно записать как:

$$H_i = [(H_{i-1} \oplus M_i)^2](\mathrm{mod}\, n),$$

где

$$i = \overline{1,n},\ H_0=I=0,\ M=M_1,M_2,...,M_n;$$

длина блока M_i представляется в октетах, каждый октет разбит пополам и к каждой половине спереди приписывается полуоктет, состоящий из двоичных единиц; n - произведение двух больших (512-битных) простых чисел p и q.

По-видимому, наиболее эффективными на сегодняшний день с точки зрения программной реализации и условий применения оказываются *хэш-функции построенные «с нуля»*.

Алгоритм MD4 (Message Digest) был разработан Р. Ривестом [68,69]. Размер вырабатываемого хэш-кода - 128 битов. По заявлениям самого разработчика при создании алгоритма он стремился достичь следующих целей:

После того, как алгоритм был впервые опубликован, несколько криптоаналитиков построили коллизии для последних двух из трех раундов, используемых в MD4. Несмотря на то, что ни один из предложенных методов построения коллизий не приводит к успеху для полного MD4, автор усилил алгоритм и предложил новую схему хэширования MD5.

Алгоритм MD5 является доработанной версией алгоритма MD4. Аналогично MD4, в алгоритме MD5 размер хэш-кода равен 128 битам. После ряда начальных действий MD5 разбивает текст на блоки длиной 512 битов, которые, в свою очередь, делятся на 16 подблоков по 32 бита. Выходом ал-

горитма являются 4 блока по 32 бита, конкатенация которых образует 128-битовый хэш-код.

В 1993 году Национальный институт стандартов и технологий (NIST) США совместно с Агентством национальной безопасности США выпустил «Стандарт стойкой хэш-функции» (Secure Hash Standard), частью которого является алгоритм SHA. Предложенная процедура вырабатывает хэш-код длиной 160 битов для произвольного текста длиной менее 2^{64} битов. Разработчики считают, что для SHA невозможно предложить алгоритм, имеющий разумную трудоемкость, который строил бы два различных набора данных, дающих один и тот же хэш-код (то есть алгоритм, находящий коллизии). Алгоритм SHA основан на тех же самых принципах, которые использовал Р. Ривест при разработке MD4, более того, алгоритмическая структура SHA очень похожа на структуру MD4. Процедура дополнения хэшируемого текста до кратного 512 битам полностью совпадает с процедурой дополнения алгоритма MD5.

Обобщая вышесказанное, следует отметить, что при выборе практически стойких и высокоэффективных криптографических хэш-функций можно руководствоваться следующими эвристическими принципами:

- любой из известных алгоритмов построения коллизий не должен быть эффективнее метода, основанного на «парадоксе дня рождения»;
- алгоритм должен допускать эффективную программную реализацию на 32-разрядном процессоре;
- алгоритм не должен использовать сложных структур данных и подпрограмм;
- алгоритм должен быть оптимизирован с точки зрения его реализации на микропроцессорах типа Intel.

Криптографические протоколы

Базовым объектом исследований в криптологии являются криптографические протоколы. Криптографические протоколы относится к сравнительно новому направлению в криптографии, которое в последнее время переживает бурное развитие. В то же время, они являются весьма нетривиальным объектом исследований. Даже их формализация на сегодняшний день выглядит весьма затруднительной. Тем не менее, неформально под *протоколом* будем понимать распределенный алгоритм, т.е. совокупность алгоритмов для каждого из участников вычислений, плюс спецификации форматов сообщений, пересылаемых между участниками, плюс спецификации синхронизации действий участников, плюс описание действий при возникновении сбоев [6].

Объектом изучения теории криптографических протоколов являются удаленные абоненты, взаимодействующие по открытым каналам связи. Целью взаимодействия абонентов является решение какой-то задачи. Имеется также противник, который преследует собственные цели. При этом противниками могут быть один или даже несколько абонентов, вступивших в сговор.

Приведенная выше схема подписи с верификацией по запросу, а также схемы, представленные в разделе 2.4. есть не что иное, как сложные двухсторонние или многосторонние протоколы, которые используют, в том числе и криптографические примитивы.

3.4. Основные подходы к защите программ от несанкционированного копирования

3.2.3. 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 [47].

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.

3.2.4. Основные методы защиты от копирования

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 компьютер 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.

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.

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 возведения в степень, описанная в разделах 2.1 и 3.3 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.

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.

Методы, основанные на работа 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.

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.

3.2.5. Методы противодействия динамическим способам снятия защиты программ от копирования

Набор методов противодействия динамическим способам снятия защиты программ от копирования включает следующие методы [47].

- 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.

Автокорреляция представляет значительный интерес, поскольку дает некоторую числовую характеристику программы. По всей вероятности ав-

токорреляционные функции различного типа можно использовать и 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 4. ПРАВОВАЯ И ОРГАНИЗАЦИОННАЯ ПОДДЕРЖКА ПРОЦЕССОВ РАЗРАБОТКИ И ПРИМЕНЕНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ЧЕЛОВЕЧЕСКИЙ ФАКТОР

4.1. Стандарты и другие нормативные документы, регламентирующие защищенность программного обеспечения и обрабатываемой информации

4.1.1. 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.

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.

Основные нормативно-технические документы в области информационной безопасности приведены в таблице 4.1 (название некоторых документов приводятся в сокращенном виде, - их полное название можно найти в тексте данного раздела или списке литературы). При этом существующие нормативно-методические и нормативно-технические документы привязаны к этапам жизненного цикла автоматизированных систем.

Таблица 4.1

Этапы жизненного цикла АС		Нормативные документы		
		Международные	**ГОСТы, ОСТы. Требования и рекомендации**	**Положения, РД**
Уяснение замысла на проведение работ по обеспечению информационной безопасности в АС		• DOD 5200.28-STD	• ГОСТ 24.104-85	• Положение о государственной системе защиты информации в РФ • Концепция защиты СВТ и АС от НСД к информации. РД Гостехкомиссии России
Разработка и утверждение технического задания		• DOD 5200.28-STD	• ГОСТ 34.602-89 • ОСТ 4 ГО 208.014/с • ОСТ 4 ГО 203.001/с • ОСТ 4 ГО 070ОА • СТР	• Концепция защиты СВТ и АС от НСД к информации. РД Гостехкомиссии России
Эскизное, рабочее и техническое проектирование		• ISO/IEC DTR 7498-2-89 • ISO/IEC DTR 10181-1 • ISO/IEC DTR 11586-1 • ISO/IEC DTR 10745	• ОСТ В 4ГО.132. 201-84 • ОСТВ4127.006/с-81 • ОСТВ4127.007 • СТР	• Временное положение по разработке, изготовлению и эксплуатации СЗИ. РД Гостехкомиссии России
Разработка		• Рек-ции X.800-X.849 • ISO/IEC 9798-91 • ISO/IEC 11574-94 • ISO/IEC DTR 10736 • ISO/IEC CD 13888 • M30Sec (проект)	• ОСТВ4ГО.132200-83 ЕС САСУ • ГОСТ28147-89 • ГОСТ 34.10-94 • ГОСТ 34.11-94 • СТР	• Временное положение по разработке, изготовлению и эксплуатации СЗИ. РД Гостехкомиссии России
Сертификация средств и систем защиты и аттестация объекта информатизации				• РД.50-580-88; • Положение по аттестации объектов информатизации по требованиям безопасности информации • Положение о сертификации СЗИ по требованиям безопасности информации
Прием в эксплуатацию	Испытания		• ГОСТ 20.57.406-81 • ГОСТ 21552-84 • ГОСТ 29339-92 • ГОСТ Р 50752-95 • СТР	
	Категорирование технических средств		• СТР	
	Специсследования		• ОСТВ4ГО.132201-84 • СТР	
Эксплуатация		• ISO/IEC DTR 13335-1,2,3	• ГОСТ28147-89 • ГОСТ 34.10-94 • ГОСТ 34.11-94 • ОСТВ4ГО.132. 201-85 ЕССАСУ • СТР	• Временное положение по разработке, изготовлению и эксплуатации СЗИ. РД Гостехкомиссии России
Сопровождение	Поддержание СОБИ в рабочем состоянии			• РД.50-492-84 • РД.50.680-88

Этапы жизненного цикла АС		Нормативные документы		
		Международные	ГОСТы, ОСТы. Требования и рекомендации	Положения, РД
ро-вож-де-ние	Контроль эффективности применения		• ГОСТ 24.702-85 • ГОСТ 16504-81	
	Модернизация			• РД.50-492-84 • РД.50.680-88
	Ведение документации		• ГОСТВ2.904-74 • ЕСКД • ГОСТ РВ 50600-93 • ГОСТ 34.003-90	• РД.50-34.698-90 • Термины и определения. РД Гостехкомиссии России

Архитектура безопасности Взаимосвязи открытых систем

Большинство современных сложных сетевых структур, лежащих в качестве телекоммуникационной 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 ВС. Требование защиты ресурсов сети от НСД является обязательным при проектировании и реализации большинства современных ИВС, 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 ИВС, 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 ИВС, 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 уров-

нях 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.4.2.).

- 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);
- 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.

Таблица 4.2

Назначение службы	Номер службы	Процедура защиты	Номер уровня
Аутентификация:			
Одноуровневых объектов	1	Шифрование, цифровая подпись Обеспечение аутентификации	3,4 3,4,7
источника данных	2	Шифрование Цифровая подпись	3,4 3,4,7
Контроль доступа	3	Управление доступом	3,4,7
Засекречивание:			
соединения	4	Шифрование Управление трафиком	1-4,6,7 3
в режиме без соединения	5	Шифрование Управление трафиком	2-4,6,7 3
выборочных полей	6	Шифрование	6,7
потока данных	7	Шифрование Заполнение потока Управление трафиком	1,6 3,7 3
Обеспечение целостности:			
соединения с восстановлением	8	Шифрование, обеспечение целостности данных	4,7
соединения без восстановления	9	Шифрование, обеспечение целостности данных	3,4,7
выборочных полей	10	Шифрование, обеспечение целостности данных	7
без установления соединения	11	Шифрование Цифровая подпись Обеспечение целостности данных	3,4,7 4 3,4,7
выборочных полей без соединения	12	Шифрование Цифровая подпись Обеспечение целостности данных	7 4,7 7
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

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; - определение 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»;
- 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

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*-бит 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 политика крупных государств направлена, как правило, на использование собственных криптоалгоритмов.

4.1.2. Отечественная нормативно-правовая база, под действие которой подпадают АС различного назначения

Стандартизация в области защиты информации

К основным стандартам и нормативным техническим документам по безопасности информации, в первую очередь, относятся:

• в области защиты информации от несанкционированного доступа 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»;

• в области защиты информации от утечки за счет побочных электромагнитных излучений и наводок (ПЭМИН) «Специальные требования и рекомендации по защите информации, составляющей государственную

тайну, от утечки за счет ПЭМИН» (СТР), ГОСТ 29339-92 «Информационная технология. Защита информации от утечки за счет ПЭМИН при ее обработке средствами вычислительной техники. Общие технические требования», ГОСТ Р 50752-95 «Информационная технология. Защита информации от утечки за счет побочных электромагнитных излучений при ее обработке средствами вычислительной техники. Методы испытаний», методики контроля защищенности объектов ЭВТ и другие.

Особенности защиты программ нашли свое отражение в следующих документах Гостехкомиссии России: «Программное обеспечение автоматизированных систем и средств вычислительной техники. Классификация по уровню гарантированности отсутствия недекларированных возможностей» и «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 гарантированности 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 ГОСТ 28147-89, ГОСТ Р 34.10-94 и ГОСТ Р 34.11-94 [9-11] 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».

Последние два алгоритма связаны друг с другом и описывают алгоритмы выработки и проверки электронной цифровой подписи, служащей для удостоверения авторства и подлинности информации.

4.2. 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 требова-

ниям 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.

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...» 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.

4.3. Безопасность программного обеспечения и человеческий фактор. Психология программирования.

4.3.1. Человеческий фактор

Преднамеренные и непреднамеренные нарушения безопасности программного обеспечения безопасности 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 или 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, а также 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 карту вызовов (calling card) и номера

PBX (private branch exchange - частная телефонная станция с выходом в общую сеть);

- группы хакеров, которые специализируются на вычислениях. Они используют компьютеры для кражи денег, вычисления номеров кредитных карточек и другой ценной информации, а затем продают свои услуги и методы другим, включая членов организованной преступности. 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 уголовного архива ФБР 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 [54] и характеризует скорее зарубежных хакеров-любителей, в то время как второй - это обобщенный портрет отечественного злонамеренного хакера, составленный Экспертно-криминалистическим центром МВД России [55].

В первом случае отмечается, что многие хакеры обладают следующими особенностями [54]:

- *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.4.1 [54];
- *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;

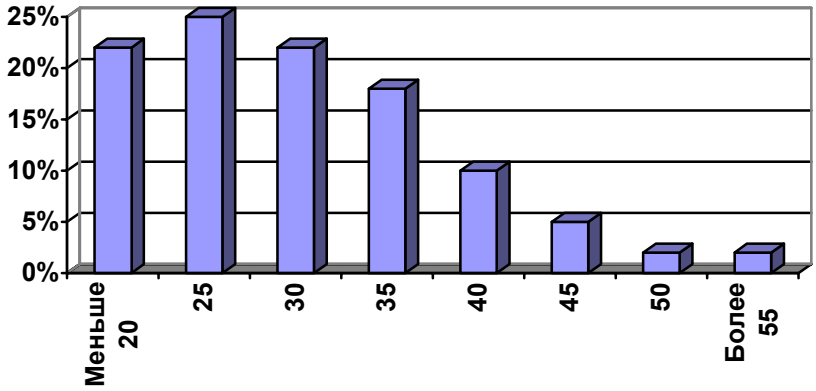

Рис. 4.1. Возрастное распределение обнаруженных компьютерных преступников

- *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 [54] 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.

Обобщенный портрет отечественного хакера выглядит следующим образом: 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 (см. табл.4.3). Это показали исследования, проведенные в 90-х гг. в США [54]. Анализ показывает, что такая тенденция постоянно сохраняется.

За последнее время в нашей стране не отмечено ни одного компьютерного преступления, которое было бы совершено одиночкой [55]. 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 (см. табл.4.4).

Специалисты правоохранительных органов России неоднократно отмечали тот факт, что большинство компьютерных преступлений в банковской сфере совершается при непосредственном участии самих служащих коммерческих банков [55]. 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.

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.

Таблица 4.3

Год	События, цифры, факты
21.11. 1988	Вирус Морриса на 24 часа вывел из строя сеть ARPANET. Ущерб составил 98 млн. долларов.
1989	К. Митник подключился к одному из компьютеров ИВС объединенной системы ПВО Североамериканского континента (North American Air Defense Command)
1989	Группа хакеров MOD уничтожила почти всю информацию в компьютере, используемом корпорацией Educational Broadcasting Corp., общественной телевизионной станцией WNET, канал 13 в Нью-Йорке
1990	К. Нейдорф осуществил доступ к телефонную сеть системы 911 в девяти штатах США и получил конфиденциальную информацию в виде кодов доступа
1994	Национальная аудиторская служба Великобритании (National Audit Office - NAO) зарегистрировала 655 случаев НСД и 562 случая заражения вирусами компьютерных систем британских правительственных организаций, что в 1.4 и 3.5 раза соответственно превышает уровень 1993 г.
1994	В США ущерб от НСД к информационно-вычислительным ресурсам превысил $10 млрд. (в 1991 г. по оценкам USA Research $1.75 млрд.)
1995	В США из 150 проверенных исследовательских и производственных вычислительных комплексов 48% подверглись успешной реализации НСД.
1995	В Великобритании ущерб от НСД к информационно-вычислительным ресурсам превысил 5 млрд. Фунтов стерлингов (в 4 раза больше по сравнению с 1989 г.)
1995	В сети Bitnet (международная академическая сеть) за 2 часа вирус, замаскированный под рождественское поздравление, заразил более 500 тысяч компьютеров по всему миру, при этом сеть IBM прекратила вообще работу на несколько часов.
Декабрь 1996	Компьютерная атака на WebCom (крупнейшего провайдера услуг WWW в США) вывела из строя на 40 часов больше 3000 абонентских пунктов WWW. Атака представляла собой «синхронный поток», которая блокирует функционирование сервера и приводит к «отказу в обслуживании». Поиск маршрута атаки длился 10 часов.

Таблица 4.4

Год	События, цифры, факты
1993	Была совершена попытка хищения 68 миллионов долларов путем манипуляции с данными в компьютерных сетях Центрального Банка России
1994	В. Левин проник в компьютерную систему Ситибанка и сумел перевести 2.8 миллиона долларов на счета своих сообщников в США, Швейцарии, Нидерландах и Израиле
1995	Ущерб, нанесенный банкам США за счет несанкционированного использования компьютерных сете путем введения и «навязывания» ложной информации из Москвы и Санкт-Петербурга российскими хакерами составил за I квартал 1995 г. составил $300 млн.
1997	Правоохранительными органами России было выявлено 15 компьютерных преступлений, связанных НСД к банковским базам данных. В ходе расследования установлены факты незаконного перевода 6,3 млрд. рублей. Доля компьютерных преступлений от общего числа преступлений в кредитно-финансовой сфере в 1997 г. составила 0,02% при их раскрываемости не более 1-5%.
1998	Предотвращено хищения на сумму 2 млрд. рублей из филиала одного из самых крупных коммерческих банков России [55]

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 [54] 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».

4.3.2. 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. 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 деятельности крупных коммерческих организаций (см. таблицы данного раздела), 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 (см. табл.4.5) 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;
- использование средств информационно-психологического воздействия на личный состав;
- воздействие радиоэлектронными излучениями;
- физические разрушения систем обработки информации.

Таблица 4.5

Год	События, цифры, факты
1985	Разведслужбой ФРГ с засекреченного объекта в пригороде Франкфурта успешно проведена операция “Project RAHAB” по проникновению в ИВС и базы данных государственных учреждений и промышленных компаний Великобритании, Италии, СССР, США, Франции и Японии.
1985	Спецподразделение LAKAM разведслужбы Израиля МОССАД осуществило НСД к ИВС Центра по обеспечению разведывательных операций ВМС США (US Naval Intelligence Support Center - NISC).
1988	Матиас Шпеер из Ганновера осуществил НСД к информации о программе СОИ и разработках ядерного, химического и биологического оружия из секретных электронных досье Пентагона.
Декабрь 1988	В Ливерморской лаборатории США (Lawrence Livermore National Laboratories - LLNL), занимающейся разработкой ядерного оружия, было зафиксировано 10 попыток НСД через каналы связи с INTERNET.
1989	Во Франции зарегистрированы попытки НСД в информационные банки данных военного арсенала в Шербуре.
Конец 80-х	Группа компьютерных взломщиков из ГДР (Д. Бржезинский, П. Карл, М. Хесс и К. Кох) овладели паролями и кодами доступа к военным и исследовательским компьютерам в США, Франции, Италии, Швейцарии, Великобритании, ФРГ, Японии.
1995	В космическом центре NASA им. Джонсона (Johnson Space Center) регистрировалось 3-4 попытки НСД в сутки (на 50% больше чем в 1994 г.), 349.2 часов затрачено на восстановление функционирования сети.
1995	Центр информационной борьбы ВВС (Air Force Information Warfare Center) за первых 3 недели после создания зарегистрировал более 150 попыток НСД только к одному сетевому узлу.
Январь 1997	Через Internet выведена из строя главная машина для хранения информационного архива по проекту FreeBSD (freefall.freebsd.org).

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.

4.3.3. Психология программирования

При создании высокоэффективных и надежных 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» [54].

- 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 применение 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 научно-практические основы обеспечения безопасности программного обеспечения компьютерных систем, рассматриваются методы и средства защиты программ от воздействия разрушающих программных средств на различных этапах их жизненного цикла.

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 (ознакомиться с большей их частью можно в главе 4), а также международные стандарты и рекомендации по управлению качеством программного обеспечения, которые позволяет предъявить к создаваемым и эксплуатируемым программных комплексам требуемый уровень реализации защитных функций.

Изложенный материал, по мнению автора, позволит при изучении технологии проектирования систем обеспечения безопасности 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.

ЛИТЕРАТУРА

1. Абрамов С.А. Элементы анализа программ. Частичные функции на множестве состояний. - М.: Наука, 1986.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. - М.: Мир, 1979.
3. Безруков Н.Н. Компьютерная вирусология// Справ. - Киев: Издательство УРЕ, 1991.
4. Беневольский С.В., Бетанов В.В. Контроль правильности расчета параметров траектории сложных динамических объектов на основе алгоритмической избыточности// Вопросы защиты информации. - 1996.- №2.- С.66-69.
5. Белкин П.Ю. Новое поколение вирусов принципы работы и методы защиты// Защита информации. - 1997.- №2.-С.35-40.
6. Варновский Н.П. Криптографические протоколы// В кн. Введение в криптографию/ Под. Общ. Ред. В.В. Ященко М.: МЦНМО, “ЧеРо”, 1998.
7. Вербицкий О.В. О возможности проведения любого интерактивного доказательства за ограниченное число раундов// Известия академии наук. Серия математическая. - 1993. - Том 57. - №3. - С.152-178.
8. Герасименко В.А. Защита информации в АСОД.- М.: Энергоиздат, 1994.
9. ГОСТ 28147–89. Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования.
10. ГОСТ Р 34.10–94. Информационная технология. Криптографическая защита информации. Процедуры выработки и проверки электронной цифровой подписи на базе асимметричного криптографического алгоритма.
11. ГОСТ Р 34.11–94. Информационная технология. Криптографическая защита информации. Функция хэширования.
12. Дал У., Дейкстра Э., Хоор К. Структурное программирование/ М.: Мир, 1975.
13. Гэри М., Джонсон Д. Вычислительные машины и труднорешаемые задачи. М.: Мир, 1982.
14. Дейкстра Э.В. Смиренный программист// В кн. Лекции лауреатов премии Тьюринга за первые 20 лет 1966-1985.- М.: Мир, 1993.
15. Защита программного обеспечения: Пер. с англ./ Д. Гроувер, Р. Сатер, Дж. Фипс и др./ Под редакцией Д. Гроувера.- М.: Мир, 1992.
16. Зегжда Д., Мешков А., Семьянов П., Шведов Д. Как противостоять вирусной атаке. – СПб.: BHV-Санкт-Петербург, 1995.
17. Зегжда Д.П., Шмаков Э.М. Проблема анализа безопасности программного обеспечения// Безопасность информационных технологий. - 1995.- №2.- С.28-33.

18. Зима В.М., Молдовян А.А., Молдовян Н.А. Защита компьютерных ресурсов от несанкционированных действий пользователей. - Учеб пособие. - СПб: Издательство ВИКА им. А.Ф. Можайского, 1997.
19. Ефимов А.И., Пальчун Б.П. О технологической безопасности компьютерной инфосферы// Вопросы защиты информации. - 1995.- №3(30).- С.86-88.
20. Ефимов А.И. Проблема технологической безопасности программного обеспечения систем вооружения// Безопасность информационных технологий. - 1994.- №3-4.- С.22-33.
21. Ефимов А.И., Ухлинов Л.М. Методика расчета вероятности наличия дефектов диверсионного типа на этапе испытаний программного обеспечения вычислительных задач// Вопросы защиты информации. - 1995.- №3(30).-С.86-88.
22. Ефимов А.И., Пальчун Б.П., Ухлинов Л.М. Методика построения тестов проверки технологической безопасности инструментальных средств автоматизации программирования на основе их функциональных диаграмм// Вопросы защиты информации. - 1995.- №3(30).-С.52-54.
23. Казарин О.В. Самотестирующиеся и самокорректирующиеся программы для систем защиты информации// В кн.: Тезисы докл. конференции «Методы и средства обеспечения безопасности информации», С.- Петербург. - 1996.- С.93-94.
24. Казарин О.В. О создании информационных технологий, исходно ориентированных на разработку безопасного программного обеспечения// Вопросы защиты информации. - 1997. - №№1-2 (36-37). - С.9-10.
25. Казарин О.В. Два подхода к созданию программного обеспечения на базе алгоритмически безопасных процедур// Безопасность информационных технологий.-1997.-№3.- С.39-40.
26. Казарин О.В. Основные принципы построения самотестирующихся и самокорректирующихся программ// Вопросы защиты информации. - 1997. - №№1-2 (36-37). - С.12-14.
27. Казарин О.В. Конвертируемые и селективно конвертируемые схемы подписи с верификацией по запросу// Автоматика и телемеханика. - 1998. - №6. - С.178-188.
28. Казарин О.В., Лагутин В.С., Петраков А.В. Защита достоверных цифровых электрорадиосообщений. - М.: РИО МТУСИ, 1997.
29. Казарин О.В., Ухлинов Л.М. Интерактивная система доказательств для интеллектуальных средств контроля доступа к информационно - вычислительным ресурсам// Автоматика и телемеханика. - 1993.- №11.- С.167-175.
30. Казарин О.В., Ухлинов Л.М. Новые схемы цифровой подписи на основе отечественного стандарта// Защита информации. - 1995.- №5.- С.52-56.

31. Казарин О.В., Ухлинов Л.М. Принципы создания самотестирующейся / самокорректирующей программной пары для некоторых схем защиты информации// Тезисы докладов Международной конференции IT+SE'97, Украина, Ялта, 15-24 мая 1997. - С.109-111.
32. Казарин О.В., Ухлинов Л.М. К вопросу о создании безопасного программного обеспечения на базе методов конфиденциального вычисления функции// Тезисы докладов Международной конференции - IP+NN'97, Украина, Ялта, 10-17 октября 1997. - С.3-6.
33. Казарин О.В., Скиба В.Ю. Об одном методе верификации расчетных программ// Безопасность информационных технологий.-1997.-№3.- С.40-43.
34. Кнут Д. Искусство программирования для ЭВМ. Том 2.: Мир, 1976.
35. Лагутин В.С., Петраков А.В. Утечка и защита информации в телефонных каналах. - М.: Энергоатомиздат, 1998.
36. Липаев В.В. Управление разработкой программных средств. – М.: Финансы и статистика, 1993.
37. Липаев В.В., Позин Б.А., Штрик А.А. Технология сборочного программирования. - М.: Радио и связь, 1992.
38. Липаев В.В. Сертификация информационных технологий программных средств и баз данных. Методы и стандарты. – Казань: Издательство ЦНИТ РТ, 1995.
39. Надежность и эффективность в технике (в 10-ти томах). Справочник. - М.: Машиностроение, 1989.
40. Организация и современные методы защиты информации (под общей редакцией Диева С.А., Шаваева А.Г.). - М.: Концерн «Банковский деловой центр», 1998.
41. Пальчун Б.П. Проблема взаимосвязи надежности и безопасности информации// В кн.: Тезисы докл. конференции "Методы и средства обеспечения безопасности информации", С.-Петербург. - 1996.- С.184-185.
42. Петраков А.В. Защита и охрана личности, собственности, информации. М.: Радио и связь, 1997.
43. Петраков А.В., Лагутин В.С. Телеохрана. М.: Энергоатомиздат, 1998.
44. Пальчун Б.П., Юсупов Р.М. Оценка надежности программного обеспечения. - СПб.: Наука, 1994.
45. Проблемы безопасности программного обеспечения. Под ред. П.Д. Зегжда. - СПб.: Издательство СПбГТУ, 1995.
46. Правиков Д.И., Чибисов В.Н. Об одном подходе к поиску программных закладок// Безопасность информационных технологий. - 1995.- №1.- С.76-79.
47. Расторгуев С.П., Дмитриевский Н.Н. Искусство защиты и «раздевания» программ. М.: Издательство «Совмаркет», 1991.

48. Руководящий документ. Концепция защиты средств вычислительной техники и автоматизированных систем от НСД к информации. М.: Гостехкомиссия России, 1992.
49. Руководящий документ. Защита от несанкционированного доступа к информации. Термины и определения. М.: Гостехкомиссия России, 1992.
50. Руководящий документ. Автоматизированные системы. Защита от НСД к информации. Классификация АС и требования по защите информации. М.: Гостехкомиссия России, 1992.
51. Руководящий документ. Средства вычислительной техники. Защита от несанкционированного доступа к информации. Показатели защищенности от НСД к информации. М.: Гостехкомиссия России, 1992.
52. Руководящий документ. Временное положение по организации разработки, изготовления и эксплуатации программных и технических средств защиты информации от НСД в автоматизированных системах и средствах вычислительной техники. М.: Гостехкомиссия России, 1992.
53. Руководящий документ. Средства вычислительной техники. Межсетевые экраны. Защита от несанкционированного доступа к информации. Показатели защищенности от НСД к информации. М.: Гостехкомиссия России, 1997.
54. Стенг Д., Мун С. Секреты безопасности сетей. - К.: «Диалектика», 1995.
55. Сырков Б. Компьютерная преступность в России. Современное состояние// Системы безопасности связи и телекоммуникаций. - 1998. - №21. - С.70-72.
56. Тейер Т., Липов М., Нельсон Э. Надежность программного обеспечения. - М.: Мир, 1981.
57. Томпсон К. Размышления о том, можно ли полагаться на доверие// В кн. Лекции лауреатов премии Тьюринга за первые 20 лет 1966-1985.- М.: Мир, 1993.
58. Ухлинов Л.М., Казарин О.В. Методология защиты информации в условиях конверсии военного производства// Вестник РОИВТ.- 1994.- №1-2.- С.54-60.
59. Файтс Ф., Джонсон П., Кратц М. Компьютерный вирус: проблемы и прогноз. - М.: Мир, 1994.
60. Щербаков А. Разрушающие программные воздействия. - М.: ЭДЕЛЬ, 1993.
61. Ben-Or M., Canetti R., Goldreich O. Asynchronous secure computation// Proc 25th ACM Symposium on Theory of Computing. – 1993. – P.52-61.
62. Blum M., Luby M., Rubinfeld R. Self-testing/ correcting with applications to numerical problems// Proc 22th ACM Symposium on Theory of Computing. - 1990. - P.73-83.

63. Blum M., Kannan S. Designing programs that check their work// Proc 21th ACM Symposium on Theory of Computing. - 1989. - P.86-97.
64. Buell D.A., Ward R.L. A multiprecise arithmetic package// J.Supercomput. - 1989. - V.3. - №2. - P.89-107.
65. Cohen F. Computer viruses: Theory and experiments// Proceedings of the 7th National Computer Security Conference. - 1984.
66. Hunter D.G.N. RSA key calculations in ADA// The Comp.J. - 1985. - V.28. - №3. - P.343-348.
67. Micali S., Rogaway Ph. Secure computation// Advances in Cryptology – CRYPTO'91, Proceedings, Springer-Verlag LNCS, V.576. – 1992. – P.392-404.
68. Rivest R. The MD4 message digest algorithm// RFC 1186, October, 1990.
69. Rivest R. The MD5 message digest algorithm// RFC 1321, April, 1992.
70. Rivest R.L., Shamir A., Adleman L. A method for obtaining digital signatures and public-key cryptosystems// Communication of the ACM.- 1978.- V.21.- №2.- P.120-126.

ПРИЛОЖЕНИЯ

Приложение 1

Перечень типовых дефектов разработки, влияющих на безопасность ПО, и программных закладок, замаскированных под дефекты разработки.

1. Неполная проверка параметров и разброса переменных; нестрогий контроль границ их изменений.
2. Скрытое использование приоритетных данных.
3. Асинхронное изменение интервала между временем проверки и временем использования.
4. Неправильное преобразование в последовательную форму.
5. Неправильные идентификация, верификация, аутентификация и санкционирование задач.
6. Отказ предотвращения перехода за установленные в программе пределы доступа и полномочий.
7. Логические ошибки (например, больше логических выражений или результатов, чем операций перехода).
8. Незавершенные разработка и описание.
9. Недокументированные передачи управления.
10. Обход контроля или неправильные точки контроля.
11. Неправильное присвоение имен, использование псевдонимов.
12. Неполная инкапсуляция или неполное скрытие описания реализации объекта.
13. Подменяемые контрольные журналы.
14. Передача управления в середине процесса.
15. Скрытые и недокументированные вызовы из прикладных программ, команд ОС и аппаратных команд.
16. Не устранение остаточных данных или отсутствие их адекватной защиты.
17. Неправильное освобождение ресурсов.
18. Игнорирование отключения внешних приборов.
19. Неполное прерывание выполнения программ.
20. Использование параметров ОС в прикладном пространстве памяти.
21. Не удаление средств отладки до начала эксплуатации.

Формы проявления программных дефектов

(Вариант 1)

1.Выполнение арифметических операций и стандартных функций:

- деление на 0;
- переполнение разрядной сетки;
- отличие результатов арифметических операций от ожидаемых;
- обращение к стандартным функциям с недопустимыми значениями параметров.

2.Ошибки, связанные с затиранием команд и переменных.

3.Ошибки управления:

- зацикливание - бесконечное повторение одной и той же части программы;
- последовательность прохождения участков программы не соответствует ожидаемому;
- потеря управления, приводящая к ошибкам разного рода (обращение к запрещенной области памяти, попытка выполнить запрещенную программу или «не команду».
- *4.Ошибки ввода - вывода:*
- «странный» вывод (на печать, на монитор и т.д.);
- сообщения об ошибках от системных программ ввода - вывода.

Вариант 2

1.Ошибки, приводящие к прекращению выполнения основных или части функций управляющей системы на длительное и или неопределенное время:

- зацикливание, то есть последовательная повторяющаяся реализация определенной группы команд, не прекращающаяся без внешнего вмешательства;
- останов и прекращение решения функциональных задач;
- значительное искажение или потеря накопленной информации о текущем состоянии управляемого процесса;
- прекращение или значительное снижение темпа решения некоторых задач вследствие перегрузки ЭВМ по пропускной способности;
- искажение процессов взаимного прерывания подпрограмм, приводящее к блокировке возможности некоторых типов прерываний.

2.Ошибки, кратковременно, но значительно искажающие отдельные результаты, выдаваемые управляющим алгоритмом:

- пропуск подпрограмм или их существенных частей;

- выход на подпрограммы или их части, резко искажающиеся результаты;
- обработка ложных или сильно искаженных сообщений.

3.Ошибки, мало и кратковременно влияющие на результаты, выдаваемые управляющим алгоритмом.

Этот тип ошибок характерен, в основном, для квазинепрерывных величин, в которых возможны небольшие отклонения результатов за счет ошибок. Эти ошибки в среднем мало искажают общие результаты, однако отдельные выбросы могут сильно влиять на процесс управления и требуется достаточно эффективная защита от таких редких значительных выбросов.

Приложение 2

Характеристики программ с точки зрения влияния на их защищенность и результаты работы

1. Состав и количество операторов исходного текста.
2. Время работы программы.
3. Число строк комментариев.
4. Число операторов и операндов.
5. Число исполненных операторов.
6. Количество ветвей и маршрутов в программе.
7. Число точек входа/выхода.
8. Время реакции.
9. Объем ввода/вывода.
10. Количество модулей.
11. Количество переходов по условию.
12. Количество циклов.
13. Количество инструкций эксплуатационной документации.
14. Количество специфицированных функций.
15. Количество внутренних/внешних переменных.
16. Время рестарта.
17. Объем внутренней/внешней памяти.
18. Число сбоев, отказов при работе программы.

Приложение 3

Перечень международных нормативных документов, связанных с проблематикой обеспечения безопасности программного обеспечения

1. ISO 1155-78 Обработка информации. Использование четности для обнаружения ошибок в информационных сообщениях.
2. ISO 2382-8-86 Системы обработки информации. Словарь. Часть 8. Контроль, целостность и защита.
3. ISO 7498-2-89 Системы обработки информации. Взаимосвязь открытых систем. Базовая эталонная модель. Часть 2. Архитектура безопасности.
4. ISO 8730-90 Банковское дело. Требования к подлинности сообщения (стандартная форма).
5. ISO 8731-87 Банковское дело. Утвержденные алгоритмы для аутентификации сообщений.
   Часть 1. Алгоритм шифрования данных (DEA).
   Часть 2. Алгоритмы для аутентификации сообщений.
6. ISO 9579-1-93 Информационные технологии. Взаимосвязь открытых систем. Удаленный доступ к базам данных. Часть 1. Общая модель, услуги и протокол.
7. ISO 9594-8-95 Информационные технологии. Взаимосвязь открытых систем. Директории. Часть 8. Система понятий аутентификации.
8. ISO/IEC 9646-1-94 Информационные технологии. Взаимосвязь открытых систем. Методология и основы аттестационного тестирования.
   Часть 1. Основные концепции.
   Часть 2. Спецификация абстрактного набора текстов.
   Часть 3. Дерево и таблица комбинируемой нотации.
   Часть 4. Реализация теста.
   Часть 5. Требования к испытательным лабораториям и клиентам для проведения аттестации.
   Часть 6. Спецификация протокола испытаний.
9. ISO/IEC 9796-91 Информационные технологии. Методы защиты. Схема цифровой подписи для восстановления сообщений.
10. ISO/IEC 9797-89 Передача данных криптографическим методом. Механизм целостности данных с использованием функции криптографического контроля на основе алгоритма блочного шифрования.
11. ISO/IEC 9798-1-91 Информационные технологии. Методы защиты. Механизмы аутентификации объектов. Часть 1. Общая модель.

12. ISO/IEC 9798-2-94 Информационные технологии. Методы защиты. Механизмы аутентификации объектов. Часть 2. Механизмы, использующие алгоритмы симметричного шифрования.
13. ISO/IEC 9798-3-93 Информационные технологии. Методы защиты. Механизмы аутентификации объектов. Часть 3. Сущность аутентификации, использующей алгоритм с открытым ключом.
14. ISO/IEC 9798-4-95 Информационные технологии. Методы защиты. Механизмы аутентификации объектов. Часть 4. Механизм, использующий функцию криптографического контроля.
15. ISO 9834-1-93 Информационные технологии. Взаимосвязь открытых систем. Процедуры регистрации полномочий. Часть 1. Общие процедуры.
16. ISO 9834-2-93 Информационные технологии. Взаимосвязь открытых систем. Процедуры регистрации полномочий. Часть 2. Регистрация процедур для типов документов ВОС.
17. ISO 9834-3-90 Информационные технологии. Взаимосвязь открытых систем. Процедуры регистрации полномочий. Часть 3. Регистрация значения составной части идентификатора объекта для совместного использования ISO-CCITT.
18. ISO 9834-4-91 Информационные технологии. Взаимосвязь открытых систем. Процедуры регистрации полномочий. Часть 4. Регистр профилей VTE.
19. ISO 9834-5-91 Информационные технологии. Взаимосвязь открытых систем. Процедуры регистрации полномочий. Часть 5. Регистр определений объекта контроля VT.
20. ISO 9834-6-93 Информационные технологии. Взаимосвязь открытых систем. Процедуры регистрации полномочий. Часть 6. Использование процессов и объектов.
21. ISO/IEC 9979-91 Передача данных криптографическим способом. Процедуры регистрации криптографических алгоритмов.
22. ISO/IEC 10038-93 Информационные технологии. Телекоммуникации и информационный обмен между системами. Локальные сети. Методы контроля доступа посредников.
23. ISO/IEC 10116-91 Информационные технологии. Режимы работы при использовании алгоритмов кодирования и *n*-битовыми блоками.
24. ISO/IEC 10118-1-94 Информационные технологии. Методы защиты. Хэш-функции. Часть 1. Общие положения.
25. ISO/IEC 10118-2-94 Информационные технологии. Методы защиты. Хэш-функции. Часть 2. Хэш-функции, использующие алгоритм шифрования *n*-битовым блоком.
26. ISO/IEC 10164-4-92 Информационные технологии. Взаимодействие

открытых систем. Управление системами. Часть 4. Функция, сообщающая о сигнале нарушения безопасности.

27. ISO/IEC 10164-6-93 Информационные технологии. Взаимодействие открытых систем. Управление системами. Часть 6. Функция контроля журнала регистрации.
28. ISO/IEC 10164-7-92 Информационные технологии. Взаимодействие открытых систем. Управление системами. Часть 7. Функция, сообщающая о вскрытии защиты.
29. ISO/IEC 10164-8-92 Информационные технологии. Взаимодействие открытых систем. Управление системами. Часть 8. Функция, отслеживающая контроль защиты.
30. ISO/IEC 10164-9-95 Информационные технологии. Взаимодействие открытых систем. Управление системами. Часть 9. Объекты и атрибуты для контроля доступа.
31. ISO/IEC 10164-10-95 Информационные технологии. Взаимодействие открытых систем. Управление системами. Часть 10. Функция измерения частоты использования ресурса в целях учета.
32. ISO/IEC 10181-96 Информационные технологии. Взаимодействие открытых систем. Основы защиты для открытых систем.
    Часть 1. Обзор.
    Часть 2. Основы аутентификации.
    Часть 3. Основы контроля доступа.
    Часть 4. Основы обеспечения невозможности отказа партнеров по связи от факта передачи или приема сообщений.
    Часть 5. Основы конфиденциальности.
    Часть 6. Основы обеспечения целостности.
    Часть 7. Основы контроля защиты и сигналов о нарушении безопасности.
33. ISO/IEC 11131-92 Банковское дело и связанные с ним финансовые услуги. Установление подлинности при входе в систему.
34. ISO 11166-94 Банковское дело. Организация шифрования посредством асимметричных алгоритмов.
    Часть 1. Принципы, процедуры и форматы.
    Часть 2. Одобренные алгоритмы, использующие RSA-криптосистему.
35. ISO/IEC 11181-96 Информационные технологии. Взаимосвязь открытых систем.
    Часть 1. Схемы безопасности для открытых систем: обзор.
    Часть 2. Схемы безопасности для открытых систем: структура аутентификации.

Часть 3. Схемы безопасности для открытых систем: структура обеспечения контроля за доступом.
Часть 5. Схемы безопасности для открытых систем: структура обеспечения конфиденциальности.
Часть 6. Схемы безопасности для открытых систем: структура обеспечения целостности.
Часть 7. Схемы безопасности для открытых систем: схема проверки безопасности и сигналы тревоги.

36. ISO/IEC 11442-4-93 Техническая документация. Обращение машиноориентированной технической информации. Требования защиты.
37. ISO/IEC 11558-92 Информационные технологии. Уплотнение данных для обмена информацией. Адаптивное кодирование с вложенным словарем. Алгоритм DCLZ (уплотнение данных по Lempel и Ziv).
38. ISO/IEC 11576-94 Информационные технологии. Процедуры регистрации алгоритмов для сжатия данных без потерь.
39. ISO/IEC 12119-94 Информационные технологии. Пакеты программного обеспечения. Требования к качеству и тестированию.
40. ISO/IEC TR 13594-95 Информационные технологии. Модель безопасности для низких уровней.
41. ISO/IEC 14136-95 Информационные технологии. Телекоммуникации и информационный обмен между системами. Локальные сети с комплексными услугами. Спецификация, функциональная модель и информационные потоки. Дополнительные услуги по идентификации.
42. ISO/IEC DIS 11586-5 Информационные технологии. Взаимодействие открытых систем. Общая защита высших уровней.
43. ISO/IEC DIS 11586-6 Информационные технологии. Взаимодействие открытых систем. Общая защита высших уровней. Часть 6. Проформа сообщения соответствия реализации (PICS) защиты синтаксиса передачи.
44. ISO/IEC DIS 13522-5 Информационные технологии. Кодирование мультимедиа и гипермедиа информации. Часть 5. Поддержка для интерактивных приложений базового уровня.
45. ISO/IEC DIS 15200 Информационные технологии. Адаптивный алгоритм сжатия данных без потерь.
46. CD 11768-2 Техника защиты информационной технологии. Критерии оценки по защите информационной технологии. Часть 2. Функциональность систем ИТ, результатов и составных частей.
47. ISO/IEC HDTR 13335. Информационные технологии. Защита информации. Руководство по управлению и административным мерам.
Часть 1. Концепция и модели защиты информации.

Часть 2. Управление и планирование защиты информации.
Часть 3. Средства управления защитой.

48. CD 13796 Аутентификация и связанные с ней услуги защиты для распределенных приложений.
49. CD 14488 Обработка информации. Методы защиты. Цифровая подпись с дополнением. Часть 1. Общие положения.
50. CD 14488 Обработка информации. Методы защиты. Цифровая подпись с дополнением. Часть 2. Механизм, основанный на идентифицирующей информации.
51. CD 14488 Обработка информации. Методы защиты. Цифровая подпись с дополнением. Часть 3. Механизмы, основанные на удостоверяющей информации.

ОГЛАВЛЕНИЕ

Научное издание

Казарин Олег Викторович

Безопасность программного обеспечения
компьютерных систем

Печатается в авторской редакции с готового оригинал-макета.