Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский государственный политехнический университет»

На правах рукописи

АРАНОВ Владислав Юрьевич

МЕТОД И СРЕДСТВА ЗАЩИТЫ ИСПОЛНЯЕМОГО ПРОГРАММНОГО КОДА ОТ ДИНАМИЧЕСКОГО И СТАТИЧЕСКОГО АНАЛИЗА

Специальность 05.13.19 – «Методы и системы защиты информации,

информационная безопасность»

Диссертация на соискание ученой степени кандидата

технических наук

Научный руководитель:

д.т.н., проф. Заборовский В.С

Санкт-Петербург
2014

РЕФЕРАТ

Пояснительная записка 131 страница, 19 рисунков., 17 таблиц, 37 источников.

ЗАЩИТА КОДА, ВЗЛОМ ЗАЩИТЫ ПРОГРАММ, ПРОТЕКТОР, ОБФУСКАЦИЯ, ВИРТУАЛИЗАЦИЯ КОДА, АНТИОТЛАДЧИК, СЕТИ ПЕТРИ, МОРФИРОВАНИЕ КОДА.

Предмет исследования диссертационного исследования – методы и средства защиты программного кода от анализа, реконструкции и модификации алгоритмов, заданных в машинном коде, которые представляют государственную тайну или являются продукцией защищенной законом об авторском праве.

Целью исследования является разработка метода защиты исполняемого программного кода от компьютерных атак обратного проектирования на основе динамического и статического анализа данных.

Для достижения поставленной цели в диссертационной работе были решены следующие задачи:

1. Разработка модели угроз, связанных с использованием технологий обратного проектирования исполняемого программного кода, основанных на динамическом анализе(отладке) и статическом исследовании исполняемого кода.
2. Разработка метода защиты от компьютерных атак обратного проектирования прикладного программного обеспечения, основанного на замещении выбранного критически важного сегмента кода защищенным программным объектом.
3. Разработка масштабируемого алгоритма, позволяющего динамически модифицировать глубину защиты программного кода для его исполнения в среде виртуальных машин.
4. Разработка инструментальных средств, используемых разработчиками программного обеспечения для затруднения использования стандартных методов обратного проектирования за счет преобразований на основе полиномиального алгоритма.

Методы исследования: для решения сформулированных задач использовался аппарат теории графов, теории автоматов, теории защиты информации и методы обратного проектирования программных кодов.

В 1 главе пояснительной записки описаны актуальные проблемы защиты программного кода и постановка задачи защиты от динамического и статического анализа. Во 2 главе предлагается метод построения виртуальной машины и описывается общая структура разработанного прототипа, в том числе интерфейсные модули со стандартными средами разработки через объектные файлы, также описан разработанный метод построения

кодогенератора для создания байткода виртуальной машины и создание обфускатора на основе сетей Петри. В 3 главе описаны детали реализации защиты x86 ассемблерного кода, специфичные для данной платформы и реализованные на кроссплатформенном уровне антиотладочные приемы, а также указаны дополнительные методы защиты, реализованные при помощи стандартных методов и алгоритмов, разработанных другими разработчиками. В 4 разделе проведен анализ достигнутых результатов и указаны границы применимости разработанного метода.

Область применения результатов – наукоемкие промышленные программные продукты, а также программы с засекреченными алгоритмами.

ВВЕДЕНИЕ

В современных программных продуктах зачастую реализованы наукоёмкие алгоритмы, в которых заложены «know-how» не только из области информационных технологий, но и смежных областей народного хозяйства. При этом лицензионные соглашения между поставщиком и пользователем обычно ограничивают конечного пользователя лишь возможностью использования результатов выполнения программ, но не передает права на сами алгоритмы, используемые в программном продукте. Несанкционированный доступ к исполняемым кодам такого программного обеспечения может привести к анализу и реконструкции (реинжинирингу) этих алгоритмов третьими лицами, что влечёт за собой нарушение прав интеллектуальной собственности, кражу технологий (промышленный шпионаж), несанкционированную модификацию программного обеспечения (ПО) с целью внедрения программных злоупотреблений, а в худшем случае – раскрытие государственной тайны. Поэтому техническая защита от подобных действий является важной государственной задачей.

Нынешний уровень коммерческих систем защиты недостаточен для надежной защиты прикладных программ ответственного назначения. Большинство этих защит взламывается злоумышленниками довольно быстро после их ввода в эксплуатацию.

В диссертационной работе проводится сравнительное исследование существующих технологий защиты ПО от реинжиниринга, выбор наиболее эффективных из них, разработка оригинальной технологии, а также ее реализация в виде комплекта макетов программных инструментов, содержащего прототипы модулей системы защиты.

Задачей второго этапа исследований было создание прототипа системы защиты на основе описанных и выборанных методов защиты на первом этапе.

1 АКТУАЛЬНЫЕ ЗАДАЧИ ЗАЩИТЫ ИСПОЛНЯЕМОГО ПРОГРАММНОГО КОДА ОТ ДИНАМИЧЕСКОГО И СТАТИЧЕСКОГО АНАЛИЗА И ПОСТАНОВКА ЗАДАЧИ ИССЛЕДОВАНИЯ

1.1 Проблема защиты исполняемого программного кода от анализа в средах с неограниченным доступом к исполняемому коду

Защита программного обеспечения является важной задачей не только для производителей программного обеспечения, но и для экономики всей страны в целом. Из-за компьютерного пиратства страдают и местные дистрибуторы и поставщики услуг, они лишаются выручки, которая использовалась бы для создания новых рабочих мест и новых налоговых поступлений. Согласно данным Business Software Alliance (BSA), 63% программного обеспечения, установленного на персональные компьютеры в России в 2011, году было пиратским. Коммерческая стоимость этого программного обеспечения составила 3,3 млрд. долларов США [1]. Согласно данным этого же исследования потери от использования пиратского программного обеспечения во всем мире составили в 2011 63 млрд. долларов США. Это доказывает актуальность задачи защиты программного обеспечения от анализа и несанкционированного распространения в настоящий момент.

С неправомерным использованием программного обеспечения можно бороться различными способами. Среди них, конечно, должен быть легитимный. Взлом, анализ, незаконное копирование и распространение программного обеспечения должны быть правильно описаны в соответствующих законах, и государство должно регулировать ответственность за их несоблюдение. Но на данном этапе, государство не может решить эту проблему правовым путем и наиболее эффективными являются программные и аппаратные методы защиты.

При этом инструменты защиты программ от анализа и обратного проектирования (реинжиниринга) принципиально отличаются от обычных средств защиты от несанкционированного использования и копирования программного обеспечения, которые перечислены ниже:

привязка к дистрибутивному носителю,

предварительная или периодическая аутентификация пользователя,

парольная защита,

привязка к уникальным признакам компьютера,

электронные ключи.

Специфика программ защиты кода от реинжиниринга заключается в том, что нужно защищать не двоичный код сам по себе, а закодированный в нем алгоритм, что намного сложнее.

Злоумышленник может быть легальным пользователем, приобретшим программный продукт, и ему не нужно взламывать вышеперечисленные виды защиты.

Обычно злоумышленник дизассемблирует двоичный код исполняемой программы и по ассемблерному коду пытается восстановить алгоритм. Опытному программисту сделать это нетрудно, если поставщик не включил в программу специальных средств защиты от анализа.

Более того, многие средства защиты от несанкционированного использования и копирования программного обеспечения, существующие на рынке, взломаны именно из-за слабой защищенности от изучения. После анализа злоумышленником алгоритмов работы защиты, серийные ключи генерируются, аппаратные - успешно эмулируются. Ситуацию может исправить разработка эффективного метода защиты программного обеспечения от изучения, применяя который к алгоритмам других защит, можно было бы качественно поднять их уровень.

Целью данного диссертационного исследования является разработка и программная реализация генератора виртуальных машин, как эффективного метода защиты ПО от анализа и реинжиниринга.

Главной характеристикой этого протектора, основанного на использовании генератора виртуальных машин, является модификация кода программного продукта, представленного в виде исходных кодов, и скомпилированного исполняемого кода к виду, сохраняющему ее функциональность, но затрудняющему анализ, понимание алгоритмов работы и, соответственно, модификацию третьими лицами. Другая важная характеристика этого протектора – многоплатформенность. Наиболее распространенные на данный момент операционные системы – это MS Windows и Linux, поэтому, прежде всего именно для них проектируются средства защиты в данном диссертационном исследоввании, хотя исследуемые принципы применимы и для других ОС. Таким образом, в данной работе проводится разработка метода защиты программ от анализа и программная реализация их в виде многоплатформенной инструментальной системы.

1.2 Анализ современных подходов и технологий защиты программного кода

История защиты программ от изучения начинается в 80-х годах прошлого века, как история самозащиты вирусов [2]. Первым вирусом, который попытался решить задачу защиты своего тела от уже существовавших тогда антивирусных утилит, был DOS-вирус Cascade (Virus.DOS.Cascade). Его «самозащита» заключалась в частичном шифровании собственного кода. Эта задача оказалась не решена, поскольку каждый новый экземпляр вируса, хотя и был уникален, все же содержал в себе неизменную часть, которая «выдавала» его и позволяла антивирусам его обнаружить. Через два года появился первый полиморфный вирус Chameleon (Virus.DOS.Chameleon), а его ровесник Whale использовал для защиты своего кода сложное

шифрование и обфускацию. Еще через два года начали появляться так называемые полиморфик-генераторы, которые можно было применить в качестве готового решения для защиты кода вредоносной программы.

В настоящее время известно много методов защиты программного обеспечения от изучения. Основные из них - следующие:

а) компрессия/шифрование: изначально программа упаковывается / шифруется, и затем сама производит обратный процесс дешифрования и распаковки по мере выполнения;

б) обфускация (запутывание) - искусственное усложнение кода с целью затруднить его читабельность и отладку (перемешивание кода, внедрение ложных процедур, передача лишних параметров в процедуры и т.п.);

в) мутация: создаются таблицы соответствия операндов - синонимов и заменяются друг на друга при каждом запуске программы по определенной схеме или случайным образом;

г) виртуализация процессора: создается процессор, исполняющий обфусцированный код(ПИОК) со своей системой команд; защищаемая программа компилируется для нее и затем выполняется на целевой машине с помощью симулятора виртуальной машины;

д) морфирование, или изменение кода;

е) затруднение дизассемблирования и отладки;

ж) нестандартные методы работы с аппаратным обеспечением: модули системы защиты обращаются к аппаратуре ЭВМ, минуя процедуры операционной системы, и используют малоизвестные или недокументированные её возможности.

Следует отметить, что терминология в области защиты программ размыта, и разные авторы трактуют понятия по-разному. Например, и мутацию, и морфирование кода многие считают частным случаем обфускации.

Нетрудно видеть, что все перечисленные методы являются, по существу, внесением избыточности в программный код, которая и мешает восстановить его алгоритм. В то же время эта избыточность приводит к замедлению выполнения программы (иногда существенному) и росту объема занимаемой ею памяти. Поэтому критериями оценки методов должны быть:

- устойчивость к различным видам анализа и реинжиниринга программы,
- степень потери эффективности программы по времени и памяти,
- сложность метода построения защиты, ручного или автоматического.

Компрессия/шифрование – это традиционный способ защиты информации от несанкционированного доступа. В нашем случае уязвимым местом является наличие механизма декомпрессии/дешифрации в самой защищаемой программе, что дает возможность опытному злоумышленнику выделить и задействовать его для раскрытия программного кода. Другой вариант: взломщик после приобретения лицензионной копии программы извлекает

расшифрованные части программы в процессе ее работы из памяти. Кроме того, нужно учитывать, что чем выше криптостойкость шифра, тем длительнее дешифрация, и замедление программы может стать неприемлемым.

Обфускация кода. Существует множество способов запутать программный код. К сожалению, большинство из них применимо только к исходному коду программы на языке высокого уровня (C/C++, Java, Python и т.д.), а не к машинному (исполняемому) коду. (Обзоры этих методов можно найти в [3 - 5].) Разнообразие способов запутывания машинного кода гораздо меньше хотя бы потому, что в нем меньше разнообразие сущностей (имен, структур управления и т.д.), чем в исходном коде. Для нас же важна защита именно исполняемого кода.

Простейший способ обфускации машинного кода — вставка в него мусорных, недействующих конструкций (таких как `or ax, ax`). При этом простая вставка таких инструкций не сильно усложняет процесс анализа кода т.к. существует много программных средств дизассемблирования кода, которые анализируют код и упрощают его, в том числе и убирая недействующие конструкции. Таким образом, этот метод эффективен, если данные инструкции не просто недействующие, а недействующие в данном исходном коде, например, это могут быть реальные инструкции, но оперирующие регистрами, не используемыми в данном коде.

Виртуализация процессора. Этот метод состоит в построении псевдокода, работающего на виртуальной машине, и, на сегодняшний момент он является наиболее актуальным и эффективным. Суть технологии состоит в том, что исполняемые файлы дизассемблируются, анализируются и преобразуются в защищенный код некоторой уникальной защищенной виртуальной машины. Сама виртуальная машина генерируется тут же.

Анализировать алгоритм работы защищенного подобным образом кода существенно сложнее, чем стандартные инструкции Intel совместимых процессоров, поскольку для него не существует никакого стандартного инструментария (отладчиков, дизассемблеров). К тому же, защищенный код не содержит в явном виде методов восстановления оригинального кода [6]. Поэтому злоумышленнику приходится все делать вручную, самому, что занимает несравнимо больше времени, чем использование готовых инструментов.

Задача реинжиниринга сводится к изучению архитектуры симулятора, симулируемого им процессора, созданию дизассемблера для последнего, и, наконец, анализу дизассемблированного кода. Ведь злоумышленник не имеет доступа ни к описанию архитектуры виртуального процессора, ни к информации по организации используемого симулятора.

К сожалению, применение данного метода затруднено ввиду высокой сложности и, соответственно, стоимости его реализации. Другой недостаток – существенное замедление исполняемой программы (на один-два порядка величины). Несмотря на эти недостатки,

передовые производители защитного ПО все же реализуют его в новейших продуктах: StarForce3, NeoGuard, VMProtect и др.

Морфирование (морфинг) кода. Этот метод в простейшем варианте (полиморф) добавляет в код мусорные инструкции, как при обфускации, а в более сложном варианте (метаморф) целиком изменяет вид кода, сохраняя при этом оригинальный алгоритм его работы, для чего он заменяет инструкции их синонимами, состоящими в свою очередь из одной или нескольких других инструкций. Эта замена может делаться неоднократно. Число проходов (циклов замены) морфера называется глубиной морфинга. Чем она больше, тем более запутанным будет получаемый код. После морфинга инструкции компилируются обратно в машинный код.

Затруднение дизассемблирования и отладки необходимо для противодействия злоумышленникам, пытающимся преодолеть защиту кода. Поэтому средства, обеспечивающие такое затруднение, обычно включаются в защищаемую программу.

Нестандартные методы работы с аппаратным обеспечением – модули системы защиты обращаются к аппаратуре ЭВМ, минуя процедуры операционной системы, и используют малоизвестные или недокументированные её возможности. Этот подход чреват слишком большой зависимостью от конкретного вида аппаратуры и, значит, противоречит требованию многоплатформенности.

1.3 Недостатки существующих существующих подходов и средств защиты и постановка задачи исследования

Исследуемая область защиты ПО богата и диверсифицирована из-за большой востребованности этих средств. Ее можно классифицировать по двум признакам.

По целям защиты программные средства защиты делится на следующие виды:

– защита алгоритмов работы ПО (т.е. защита от реинжиниринга),

– защита ПО от несанкционированного использования (после покупки требуется ввод серийного номера, привязанного к оборудованию),

– защита ПО от несанкционированного копирования (например, DVD-дисков).

По операционным системам данные решения делится на:

– защита Windows-приложений

– защита мобильных приложений на всех платформах (основные: Windows Mobile, WinCE, Android, iPhone)

– защита для Apple/Mac

– защита для Linux

Существуют, по меньшей мере, два десятка успешных производителей средств защиты (будем называть такие программные продукты протекторами). Существуют компании, которые

имеют решения для всех видов защит. Например, компания Oreans предоставляет ПО для всех видов защит, что описано ниже. Есть компания, которая предоставляет средства защиты почти для всех операционных систем, в списке нет только Apple. Это компания Flexera, ее продукт FlexLM поддерживает Windows (все версии), WinCE, Linux, VxWorks.

Наиболее крупные компании в этой сфере – это следующие пять компаний: Flexera Software (США), Oreans Technologies (США), StarForce (Россия), VMProtect (Россия), Silicon Realms(США). Краткая информация об их продуктах представлена в табл. 1.1.

Таблица 1.1

Характеристики протекторов – лидеров в области защиты ПО

Компания	Продукт	Качество	Популярность в мире
Flexera Software (США) (входит в Rovi Corp)	FlexNet Publisher (FlexLM)	Среднее	Высокая
Oreans Technologies (США)	Code Virtualizer - защита кода Themida - защита приложения WinLicense - серийные номера	Отличное	Высокая
StarForce (Россия)	FrontLine ProActive Crypto	Хорошее	низкая; в РФ – высокая
VMProtect Software (Россия, Екатеринбург)	VMProtect Lite / Professional / Ultimate	Хорошее	Низкая
Silicon Realms (США, группа Digital River)	SoftwarePassport (Armadillo) - защита приложения	Среднее	Средняя

Крупнейшей системой защиты в мире является «Flex» («FlexLM», «FlexNet»). Следует отметить, что все версии этой защиты были взломаны, в частности, для таких защищенных продуктов-гигантов, как Adobe Photoshop, Adobe Creative Suite, Autodesk Autocad, Autodesk 3DS MAX. Несмотря на существующие взломанные версии, производители ПО продолжают использовать эту защиту.

Крупнейшей системой защиты в России является StarForce. Особенность продуктов StarForce – наличие специализированных версий для всех возможных вариантов использования ПО: от записи CD/DVD-дисков и защиты образовательных программ до распространения программ с серийными ключами в интернете и многопользовательских онлайн игр. Все популярные игры с защитой StarForce были взломаны. Надо отметить активную работу компании по выпуску «патчей», т.е. обновлений системы защиты для противодействия взлому.

Как видно в табл. 1.1 компания Oreon Technologies представляет наиболее качественные услуги по защите программного обеспечения. Кроме хорошего качества, она так же выделяется обширной линейкой продуктов, и, таким образом, ее интересы сосредоточены в различных направлениях защиты ПО. Продукт Themida – это протектор для приложений. WinLicense – это тот же протектор с добавлением возможности защиты приложений на основе разного рода серийных номеров [7]. SDK WinLicense управляет генерацией этих серийных номеров, их проверкой, безопасным хранением, привязкой к железу, созданием временных серийных номеров с истечением срока к указанной дате или через указанное число запусков, созданием приложений, защищенных паролем и так далее. И, наконец, CodeVirtualizer – это независимая часть Themida, которая позволяет исключительно преобразовывать указанные функции внутри приложения в код для виртуальной машины Themida. Никакой другой защиты он не обеспечивает (защиты от отладки, шифрования, проверок целостности и прочего в нем нет). Таким образом, CodeVirtualizer для защиты кода использует метод виртуализации, что в не последнюю очередь повлияло на качество этого продукта.

Так же хочется выделить компанию VMProtect, которая для своих приложений использует метод виртуализации кода, как один из основных методов защиты программного обеспечения. В настоящее время их наиболее популярный продукт - VMProtect SenseLock Edition (VMProtect SE) - это совместная разработка компании "VMProtect Software" и "Seculab", в которой реализованы все новейшие достижения в области защиты программного обеспечения от несанкционированного использования [8,9]:

- виртуализация исполняемого кода;
- упаковка и шифрование защищаемого файла;
- выполнение кода защиты с использованием электронного ключа;
- возможность создавать демонстрационные лицензии, ограничивающиеколичество запусков, устанавливать ограничения по времени работы программы, лицензировать разные участки кода с привязкой к различным лицензиям;
- протокол обмена данными с электронным ключом на основе асимметричного алгоритма RSA-1024, исключающий появление эмуляторов.

Среди коммерческих приложений, направленных на защиту исполняемого кода и данных в двоичных файлах, детально будут рассмотрены лишь те, кто имеет наибольшую долю рынка на конец 2012 года, а именно:

FlexNet(FlexLm) разрабатывается компанией Flexera Software и на конец 2012 года обладает наибольшей долей рынка согласно опубликованной информации в сочетании со средней надежностью. По состоянию на конец 2012 года все версии данной защиты взломаны.

WinLicense(Themida) разрабатывается компанией Oreans Technologies и обладает наилучшей защитой имея невзломанные версии на конец 2012 года.

FrontLine(StarForce) разрабатывается компанией StarForce Protection Technologies и обладает самой высокой популярностью на территории РФ в сочетании с высоким уровнем защиты на конец 2012 года.

VMProtect разрабатывается компанией VMP Soft и обладает средней популярностью и хорошим уровнем защиты, однако, последние версии программ всех крупных клиентов этого разработчика защит взломаны по состоянию на конец 2012 года.

Armadillo(software Passport) разрабатывается компанией Silicon Realms и обладает высокой популярностью и средним уровнем защиты. По состоянию на конец 2012 года все версии данной защиты взломаны.

Execrypter до 2007 года разрабатывался компанией StrongBit и обладала очень высокой взломоустойчивостью. К сожалению, в выпуск новых версий прекращен в связи со сменой собственника у компании разработчика.

ASProtect куплен компанией StarForce Protection Technologies и обладает средней популярностью на территории РФ и низкой за рубежом. Не имеет невзломанных версий по состоянию на конец 2012 года.

Obsidium куплен компанией Obsidium Software и обладает низкой популярностью. Предоставляет низкий уровень защиты от взлома. Не имеет невзломанных версий по состоянию на конец 2012 года.

1.3.1 Набор функциональностей, направленный на защиту кода в пользовательском пространстве

Данный набор функциональностей включает в себя следующие пункты (см. таблицу 1.2)

- Защита при помощи виртуальной машины (VM)
- Защита при помощи случайной виртуальной машины (RVM)
- Технология, препятствующая созданию дампа (Anti-dump) (AD)
- Случайное расположение данных и кода в памяти (RD)
- Наличие алгоритма ZPerm (ZP)

– Вставка мусорного кода (GC)
– Защита, основанная на преобразованиях кода (см. таблицы 1.2 и 1.3)
    1) Полиморфный код фрагмента пользовательской программы (PC)
    2) Метаморфный код фрагмента пользовательской программы (MC)
    3) Полиморфный код фрагмента пользовательской программы при каждом запуске защищенного приложения(RPC)
    4) Метаморфный код фрагмента пользовательской программы при каждом запуске защищенного приложения (RMC)

Таблица 1.2

Сравнение методов защиты пользовательского кода

Протектор	VM	RVM	AD	RD	ZP	GC
FlexNet	нет	нет	да	да	нет	нет
WinLicence	да	да	да	да	да	да
FrontLine	да	да	да	да	да	да
VMProtect	да	да	да	да	нет	да
Armadillo	да	нет	да	да	нет	нет
ExeCrypter	да	нет	да	да	нет	нет
ASProtect	нет	нет	да	да	нет	нет
Obsidium	нет	нет	да	да	да	да

Таблица 1.3

Сравнение защиты основанной на преобразованиях кода

Протектор	PC	MC	RPC	RMC
FlexNet	нет	нет	нет	нет
WinLicence	да	да	нет	да
FrontLine	да	да	нет	нет
VMProtect	да	да	нет	нет
Armadillo	нет	да	нет	нет
ExeCrypter	да	да	да	нет
ASProtect	нет	да	да	нет
Obsidium	нет	да	нет	нет

Защита, основанная на преобразованиях кода с использованием, например, сетей Петри, не используется продуктах по состоянию на конец 2012 года.

1.3.2 Интерфейс разработчика

Системы защиты зачастую характеризуются наличием SDK разработчика, позволяющего интегрировать защиту непосредственно в код защищаемого приложения и удобным инструментарием разработчика (см. таблицу 1.4).

Таблица 1.4

Сравнение интерфейсов разработчика

Протектор	SDK	Интерфейс командной строки	Графический интерфейс	Встраивание в процесс разработки
FlexNet	Обязательно	Да	нет	Да
WinLicence	Опционально	Да	Да	Да
FrontLine	Опционально	нет	Да	Нет
VMProtect	Опционально	Не	Да	Нет
Armadillo	Опционально	Да	Да	Да
ExeCrypter	Нет	Нет	Да	Нет
ASProtect	Нет	Да	Да	Да
Obsidium	Нет	Нет	Да	Нет

1.3.3 Дополнительная функциональность систем защит

Зачастую протекторы кроме непосредственно защиты исполняемого приложения предлагают еще и другую, смежную функциональность, такую как:

- Модули лицензирования, позволяющие создавать серийные номера (лицензии), как бессрочные, так и на ограниченный срок (LM)
- Привязка к оборудованию (HL)
- Предотвращение работы отладчиков (DD)
- Поддержка дополнительных платформ (EP), за исключением x86
- Возможность настройки глубины защиты кода (FT)

Сравнение характеристик дополнительной функциональности протекторов приведено в таблице 1.5.

Таблица 1.5

Сравнение списка дополнительных функциональностей

Протектор	LM	HL	DD	EP	FT
FlexNet	Да	Да	Формально, да	Да, более 20	Да
WinLicence	Да	Да	Да	X64	Да
FrontLine	Да	Да	Да	X64, ARM	Да
VMProtect	Да	Да	Да	Нет	Да
Armadillo	Да	Да	Да	X64	Да
ExeCrypter	Да	Да	Да	Нет	Да
ASProtect	Нет	Да	Да	Нет	Да
Obsidium	Да	Да	Да	X64	Да

Проведенное сравнение позволяет говорить о недостатках в существующих системах защиты машинного кода от несанкционированного изучения и обратного проектирования.

В завершении анализа хочется отметить, что метод виртуализации в защите ПО с различными ограничениями реализован в таких крупных продуктах как StarForce3, Themida, NeoGuard, VMProtect и др. и, таким образом, только комплексное использование всех вышеизложенных методов позволит создать по настоящему эффективный продукт способный противодействовать современным вызовам.

Как было показано, в настоящее время на рынке существует большое количество протекторов, аналогичных разрабатываемой системе защиты. Часть из них используется только для проектов «черного рынка» и поэтому их покупка невозможна (RDG Tejon Crypter, L33T Crypter и т.д.), и такие программы не будут участвовать в сравнении. Далее будет рассмотрен набор функциональностей. которые присутствуют только в коммерческих системах защиты программного кода.

Таким образом, необходимо предложить метод защиты иполнемого программного кода устойчивый к описанным выше недостаткам и способный эффективно осуществлять запутывающие преобразования машинного кода н основе метода виртуализации процессора, так, чтоб взлом одной виртуальной машины не приводил к взлому всех виртуальных машин, порожденных данным методом защиты. Также необходимо предложить метод защиты без применения виртуализации, для эффективной защиты с интерпретатора виртуализованного кода.

2 МЕТОД ЗАЩИТЫ ИСПОЛНЯЕМОГО КОДА ОТ ДИНАМИЧЕСКОГО И СТАТИЧЕСКОГО АНАЛИЗА НА ОСНОВЕ МНОГОУРОВНЕВЫХ ЗАПУТЫВАЮЩИХ ПРЕОБРАЗОВАНИЙ

2.1 Модель угроз динамического и статического анализа исполняемого кода

Рассмотрим модель угроз обратного проектирования исполняемого кода путем анализа и отладки, состоящую из трех компонент <S, O, A>: S – субъект атаки, O – объект атаки, A – атакующее действие

- Субъект атаки S = <s, ss, is>
    - Множество автоматизированных средств отладки кода (s): отладчики, песочницы
    - Множество полуавтоматических средств анализа кода (ss): дизассемблеры, анализаторы графа достижимости
    - Множество интеллектуальные средства анализа (is): хакер, семантическая сеть, генетические алгоритмы
- Объекты атаки O = <m, t, p>
    - Алгоритм, который реализует исполняемый код (m). Обычно он недоступен для атакующего воздействия, так как кодируется программистом в код на языке высокого уровня (ЯВУ) и потому не предается в недоверенную вычислительную среду
    - Код на ЯВУ (t). Обычно он также недоступен для атакующего воздействия, так как транслируется в машинный код инструментального процессора компилятором и потому не предается в недоверенную вычислительную среду
    - Машинный код (p). Непосредственно передается в недоверенную вычислительную среду и, потому, непосредственно доступен для атаки
- Атакующее действие A = <o, c, r>
    - Операция (o) по сбору данных о переменных, адресах областей памяти исполнения кода и их порядка выполнения
    - Преобразование (c) кода к текстовому формату
    - Оперативное восстановление (r) алгоритма

Таким образом модель объекта атаки редуцируется до <S, p, A> и далее будет рассматриваться только атака на машинный код и виртуализация этого кода является, как свидетельствуют многочисленные публикации одним из наиболее эффективных методов защиты

исполняемого кода. Поэтому перейдем к понятию обфусцирующей виртуальной машины. Также необходимо отметить что несмотря, я на то, что методы о и с напрямую не запрещаются предлагаемыми методами защиты, атакующий, тем не менее, может собрать только информацию о алгоритме защиты (например, перевести в тестовый вид сам интерпретатор виртуальной машины), но не защищенный алгоритм.

2.2 Виртуализация кода процессором с псевдослучайным набором инструкций

2.2.1 Понятие виртуальных машин

Существует несколько подходов к определению виртуальной машины.

Виртуальная машина — это:

– программная и/или аппаратная система, эмулирующая аппаратное обеспечение некоторой гостевой (целевой) платформы и исполняющая программы для целевой платформы на хост-платформе;

– система, виртуализирующая некоторую платформу и создающая на ней среды, изолирующие друг от друга программы и даже операционные системы;

– спецификация некоторой вычислительной среды.

Виртуальная машина, используемая для защиты исполняемого кода от анализа и обратного проектирования, является ПИОК со своим набором команд (инструкций), набором регистров и областью памяти. Эта виртуальная машина работает как обычное приложение в основной ОС и поддерживает единый процесс. Она создается, когда этот процесс начинает выполняться, и уничтожается, когда процесс заканчивает выполнение. Целью виртуальных процессов является обеспечение независимой от платформы среды программирования, которая абстрагируется от деталей основного оборудования или операционных систем и позволяет программам выполняться так же, как на других платформах.

Во время защиты исполняемого файла, разрабатываемый протектор преобразует код программы в байт-код своей виртуальной машины и записывает его вместо обычного, не защищенного кода. Байт-код — машинно-независимый код низкого уровня, генерируемый транслятором и исполняемый интерпретатором. Трансляция в байт-код занимает промежуточное положение между компиляцией в машинный код и интерпретацией.

Рассмотрим работу виртуального процессора подробнее. Для этого введем понятие алгоритма.

Алгоритм - набор инструкций, описывающих порядок действий исполнителя для достижения результата решения задачи за конечное время. Каждая инструкция имеет символическое имя, код (уникальное число, однозначно характеризующее действие) и набор

аргументов-операндов, которые параметризуют действие. Каждая инструкция представляет собой виртуальную команду. Виртуальная программа - это последовательность виртуальных команд. На уровне пользователя виртуальная программа - это текст, а на уровне виртуального процессора - это последовательность кодов, соответствующих тексту. Виртуальная программа может быть транслирована в последовательность фиксированных кодов, так же как обычная программа транслируется в последовательность машинных команд.

Компилятор преобразует текстовую форму виртуальной программы в код, который будет интерпретироваться виртуальным процессором. Так же, как и реальный процессор, виртуальный процессор использует программный счетчик (program counter), который указывает на текущую виртуальную команду. Виртуальный процессор выбирает очередную команду (получает ее кодовый номер) и вызывает инструкцию с указанным номером. Инструкция получает свои аргументы, следующие за кодом команды, и выполняет действия, реализующие эту команду.

Заметим, что существует два разных подхода к трансляции исходного кода программ: компиляция и интерпретация.

Компилятор сначала несколько раз внимательно просматривает исходный код, потом обращается к необходимым библиотекам, и, наконец, выдает готовую программу — исполнимый код. Исполняемый код всегда машинно-зависим. Его можно запускать только на компьютерах тех систем, на которые был рассчитан компилятор. Чтобы программа могла работать на компьютерах других систем, ее надо перекомпилировать из исходного кода другими компиляторами, рассчитанными на эти системы.

Интерпретатор — это программа, которая построчно читает исходный текст, затем переводит его в машинный код процессора и тут же подает на исполнение. Исполнив одну строку исходного кода, интерпретатор переходит к другой, и так далее. Таким образом, программа, созданная программистом, работает на компьютере клиента под управлением интерпретатора. Интерпретируемые программы можно сделать машинно-независимыми и работающими на компьютерах любых систем. Для этого нужно, чтобы каждый компьютер имел свою версию интерпретатора. Хотя разные интерпретаторы будут обрабатывать одну и ту же программу по-разному, работать она сможет на любом компьютере, на котором есть соответствующий интерпретатор.

В современных условиях машино-независимость — очень ценный фактор, ведь программист заранее не знает, на каком компьютере ему придется работать. Но интерпретируемые программы обычно работают гораздо медленнее компилируемых и, к тому же, имеют значительно большие размеры. Поэтому удобно использовать промежуточный подход, сочетающий достоинства как того, так и другого метода. Этот подход называется компиляцией в промежуточный код.

При данном подходе исходная программа компилируется не в машинный код конкретного процессора, а в некий промежуточный код виртуального процессора (виртуальной машины). Тогда, на компьютере любой системы можно поставить программу-интерпретатор, которая способна понимать этот промежуточный код и переводить его в код реального процессора того компьютера, на котором этот интерпретатор работает. Скорость и емкость программы обеспечиваются за счет компиляции в код виртуальной машины, а машино-независимость достигается благодаря тому, что этот код будут понимать все компьютеры, на которых заранее установлен интерпретатор. В этом случае получается удобное сочетание, которое обеспечивает многоплатформенность протектора.

Для выполнения диссертационного исследования для создания промежуточного представления был выбран инструмент LLVM (Low Level Virtual Machine). Проект LLVM представляет собой универсальную систему анализа, трансформации и оптимизации программ. Несмотря на свое название, LLVM не является традиционной виртуальной машиной, но предоставляет полезные библиотеки, которые могут быть использованы для их создания. В основе LLVM лежит промежуточное представление кода (intermediate representation, IR), над которым можно производить трансформации во время компиляции, компоновки и выполнения. На выбор LLVM повлияли следующие его преимущества:

- библиотеки LLVM обеспечивают генерацию кода для многих популярных процессоров, среди которых x86, x86-64, ARM, PowerPC, SPARC, MIPS, IA-64, Alpha. Эти библиотеки строятся вокруг четко определенного представления кода, известного как промежуточное представление LLVM (LLVM IR). Таким образом, использование LLVM обеспечивает возможность компиляции программы в переносимый между разными платформами универсальный байт-код, что обеспечивает многоплатформенность нашего протектора;
- система имеет модульную структуру, отдельные ее модули могут быть встроены в различные программные комплексы, она может расширяться дополнительными алгоритмами трансформации и кодогенераторами для новых аппаратных платформ [1];
- библиотеки LLVM являются хорошо документированными, что позволяет использовать LLVM как оптимизатор и генератор кода. LLVM предоставляет API для создания промежуточного предоставления, а так же генерации виртуальных машин. Кроме того, LLVM предоставляет API ко всем частям компилятора.

2.2.2 Виртуальная машина Тьюринга

Простейшим примером виртуальной машины является виртуальная машина Тьюринга. Машина Тьюринга — абстрактный исполнитель (абстрактная вычислительная машина), предложенная Аланом Тьюрингом в 1936 году для формализации понятия алгоритма. В состав машины Тьюринга входит бесконечная в обе стороны лента (возможны машины Тьюринга, которые имеют несколько бесконечных лент), разделённая на ячейки, и управляющее устройство, способное находиться в одном из множества состояний. Число возможных состояний управляющего устройства конечно и точно задано. Управляющее устройство может перемещаться влево и вправо по ленте, читать и записывать в ячейки ленты символы некоторого конечного алфавита. Можно сказать, что машина Тьюринга представляет собой простейшую вычислительную машину с линейной памятью, которая согласно формальным правилам преобразует входные данные с помощью последовательности элементарных действий. Элементарность действий заключается в том, что действие меняет лишь небольшой кусочек данных в памяти (в случае машины Тьюринга — лишь одну ячейку), и число возможных действий конечно.

Управляющее устройство работает согласно правилам перехода, которые представляют алгоритм, реализуемый данной машиной Тьюринга. Каждое правило перехода предписывает машине, в зависимости от текущего состояния и наблюдаемого в текущей клетке символа, записать в эту клетку новый символ, перейти в новое состояние и переместиться на одну клетку влево или вправо. Некоторые состояния машины Тьюринга могут быть помечены как терминальные, и переход в любое из них означает конец работы, остановку алгоритма.

Рассмотрим состав конкретной машины Тьюринга. Она задаётся перечислением элементов множества букв алфавита A, множества состояний Q и набором правил, по которым работает машина [10]. Для каждой возможной конфигурации $<q_i, a_j>$, где q_i – состояние и a_j – буква алфавита, имеется ровно одно правило перехода. Правил нет только для заключительного состояния, попав в которое машина останавливается. Кроме того, необходимо указать конечное и начальное состояния, начальную конфигурацию на ленте и расположение головки машины.

Таким образом, по аналогии с машиной Тьюринга, для задания виртуальной машины достаточно задать набор инструкций, включая инструкцию завершения работы, набор регистров данной виртуальной машины, а так же область в памяти, где эта виртуальная машина работает.

Важно отметить, что согласно тезису Чёрча — Тьюринга, всякий алгоритм может быть задан в виде соответствующей машины Тьюринга, а класс вычислимых функций совпадает с классом функций, вычислимых на машинах Тьюринга. Таким образом, машина Тьюринга способна имитировать все другие исполнители с помощью задания правил перехода. Таким

образом, несмотря на простоту машины Тьюринга, на ней можно вычислить всё, что можно вычислить на любой другой машине, осуществляющей вычисления с помощью последовательности элементарных действий. Это свойство называется полнотой. Таким образом, для написания виртуальной машины, способной выполнить любой алгоритм, необходимо чтобы эта виртуальная машина была полна по Тьюрингу. Для этого нужно определить и написать столько виртуальных инструкций, сколько требуется для описания любого возможного алгоритма.

Но, имея исходный алгоритм, можно построить виртуальную машину, содержащую не полный по Тьюрингу набор команд, а набор команд достаточный для выполнения данного алгоритма. Например, если исходная программа является обычным арифметическим выражением, то для виртуальной машины достаточно задать арифметические команды, команды работы со стеком и команду прекращения работы и нет необходимости определять команды ветвления, циклов и т.д.

2.2.3 Виртуализация в защите ПО от реинжиниринга

Виртуализация кода состоит в преобразовании кода исходной программы в байт-код созданной виртуальной машины и в последующем выполнении преобразованного кода. Виртуализированные части кода исполняются интерпретатором виртуальной машины без трансляции в оригинальный машинный код. Таким образом, реинжиниринг требует изучения архитектуры виртуальной машины и создания дизассемблера, распознающего данную архитектуру, что требует больших временных затрат. Каждый раз при запуске протектора генерируется новая виртуальная машина со своим набором инструкций, регистров и т.д. Так что, если злоумышленник разберется в архитектуре конкретной виртуальной машины, ему придется начинать сначала для следующего защищенного приложения. Это значит, что определенный блок инструкций ассемблера x86 может быть трансформирован в различные блоки инструкций для каждой виртуальной машины, что способствует защите от анализа злоумышленником генерируемых виртуальных опкодов после трансформации инструкций x86 различными виртуальными машинами.

На рисунке 2.1 показано, как блок инструкций x86 преобразуется в различные виртуальные инструкции, соответствующие блоку исходного кода.

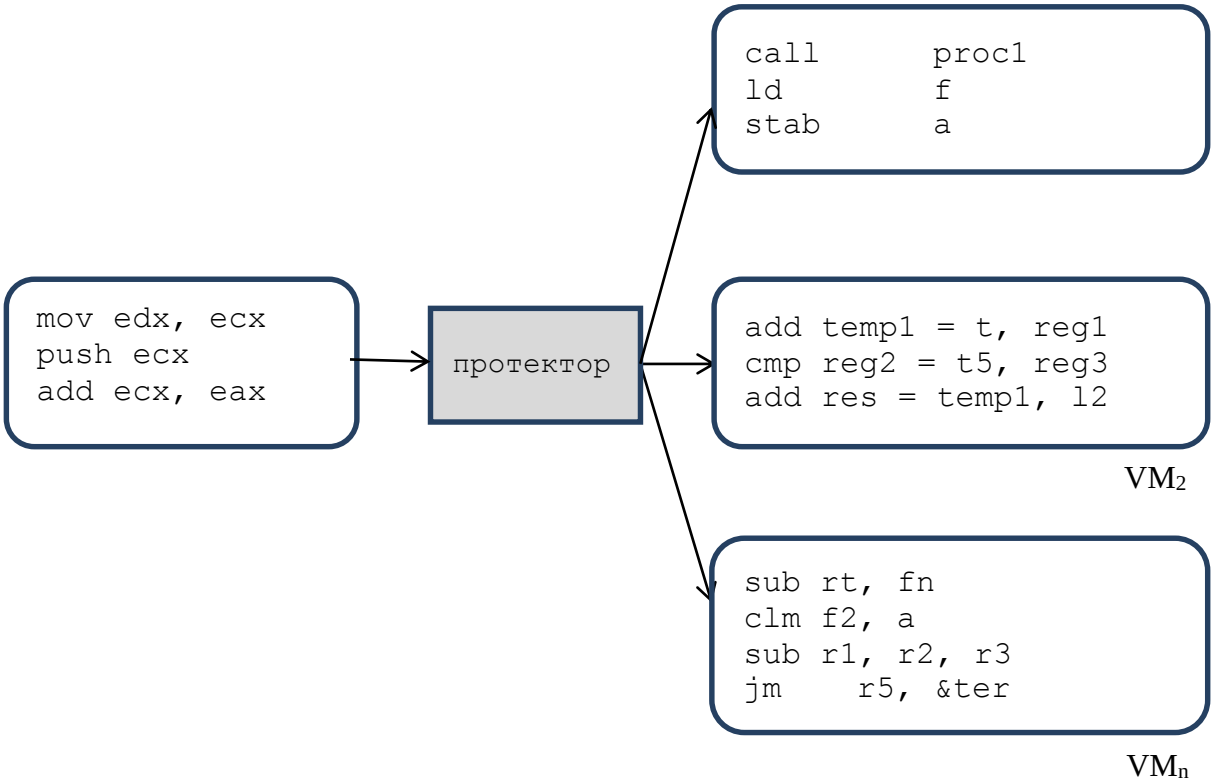

Рисунок 2.1 - Генерация различных кодов виртуальной машиной на основе одного блока исходного кода

Когда злоумышленник попытается дизассемблировать блок кода, защищенный протектором, он не увидит оригинальных инструкций x86. Вместо этого он обнаружит абсолютно новый набор инструкций, который неизвестен злоумышленнику или какому-либо декомпилятору. Таким образом, чтобы получить исходный алгоритм, злоумышленник должен проделать очень сложную работу по распознаванию семантики каждого опкода и того, как определенная виртуальная машина работает для каждого защищаемого приложения. Поэтому важно варьировать максимально большое количество параметров виртуальных машин таким образом, чтобы виртуальные машины, создаваемые генератором виртуальных машин, кардинально отличались друг от друга. Кроме того, привязка создаваемой виртуальной машины к защищаемому коду алгоритму является дополнительным осложнением, помогающим защитить генератор от анализа и обратного проектирования.

2.2.4 Выбор защищаемого участка кода

Как уже было сказано, главная цель нашего протектора – защита алгоритмов, являющихся интеллектуальной собственностью и коммерческой тайной. Очевидно, что нет необходимости защищать приложение полностью, так как это увеличивает время работы программного продукта и предоставляет большее количество данных для анализа (например, статистического). Поэтому разрабатываемый протектор позволяет защитить именно ту часть исполняемого файла, которая содержит этот алгоритм. Для этого необходимо выбрать функцию, в которой содержится тот код, который представляет определенную ценность. Список всех доступных функций строится на основе анализа секций исполняемого файла с помощью API LLVM. Затем, определенный блок инструкций ассемблера x86 может быть трансформирован в различные блоки инструкций для каждой виртуальной машины, что способствует защите от анализа злоумышленником генерируемых виртуальных опкодов после трансформации инструкций x86 различными виртуальными машинами. Рисунок 2.2 иллюстрирует преобразование части исполняемого файла в защищенный код.

Протектор выделяет указанную часть кода и преобразует её в уникальную последовательность инструкций виртуальной машины, которые будут созданы виртуальной машиной во время запуска исполняемого файла.

Как показано на рисунке 2.2, протектор вставляет сгенерированную виртуальную машину в защищенный исполняемый файл для распознавания и выполнения виртуальных инструкций в момент запуска приложения.

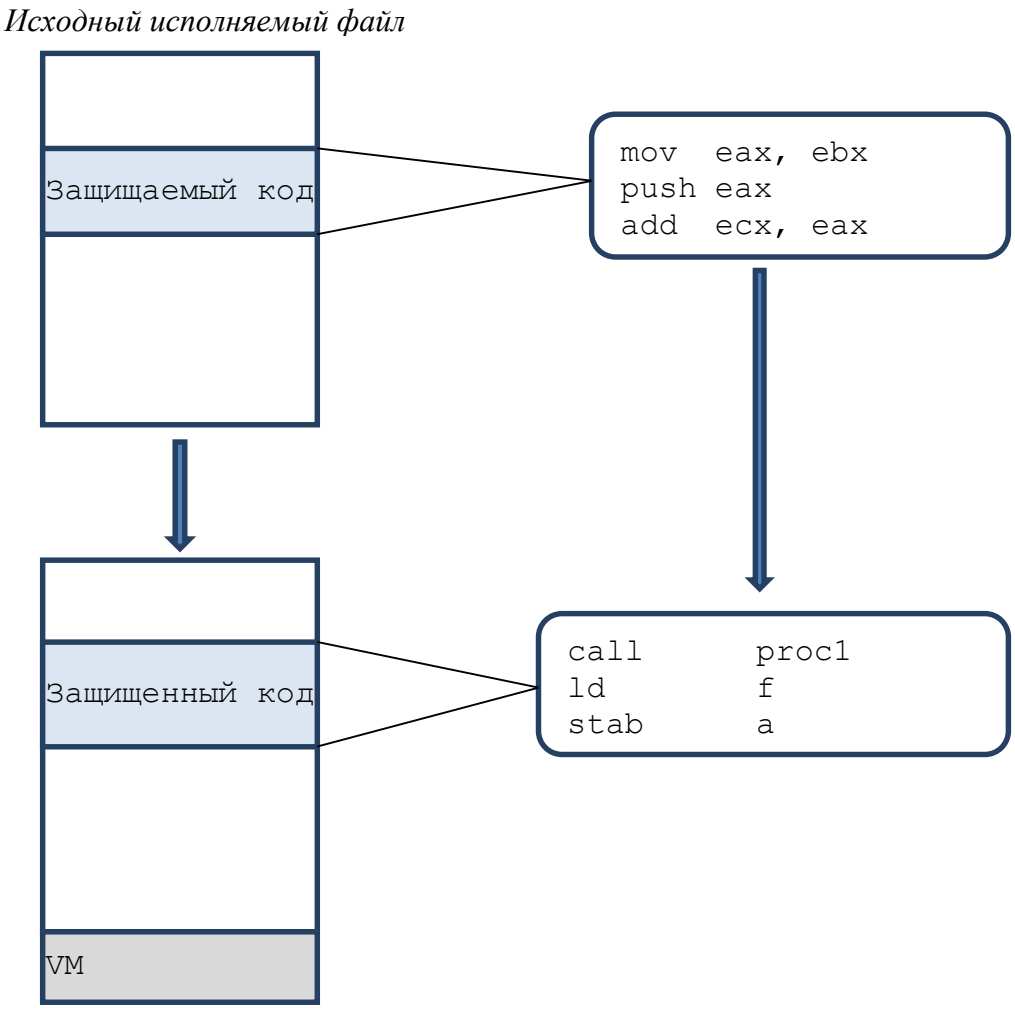

Рисунок 2.2 - Исполняемый файл и соответствующий защищенный файл

2.2.5 Создание виртуальной машины и общая схема защиты исполняемого файла

При создании конкретной виртуальной машины необходимо выполнить следующие этапы:

1. Выявить весь объём данных, которые в процессе выполнения задачи могут подвергаться преобразованию. Эти данные образуют область действия виртуальной машины. Механизм выбора этих данных описан в разд. 1.3.

2. Определить набор команд, который будет служить для конкретной виртуальной машины. Последовательность команд из этого набора, определяющая решение конкретной задачи, записывается по соответствующему адресу в некоторое программное поле виртуальной машины и не меняется в процессе её функционирования.

3. Реализовать интерпретатор виртуальной машины, представляющий собой совокупность механизмов. Посредством этих механизмов виртуальная машина осуществляет интерпретацию команд, находящихся в программном поле.

4. Реализовать транслятор из внешнего представления бинарного файла в последовательность команд виртуальной машины.

Для виртуальной машины должен быть определён способ её использования, то есть:

- способ передачи входной информации и формирования точки входа в виртуальную машину;
- способ активизации (запуска) системы;
- способ останова системы;
- способ извлечения переработанной информации [11].

Рассмотрим еще раз, что представляет собой виртуальная машина. Она состоит из интерпретатора, исполняющего последовательность известных ему команд и имеющего доступ к области памяти для хранения данных.

Одним из наиболее эффективных, с точки зрения защиты, полезных способов усложнения виртуальных машин является вызов одной виртуальной машины из другой. При его реализации вызываемые и вызывающие виртуальные машины могут как совпадать, так и существенно различаться по своей структуре. В наборе команд той виртуальной машины, из которой происходит вызов, выделяется команда или команды для порождения и вызова другой виртуальной машины. Порождённая машина получает ссылки на своё поле данных и последовательность команд, после чего запускается. Поведение вызывающей машины при этом может варьироваться: либо ожидание завершения вызванной машины, либо продолжение своего выполнения без ожидания завершения вызванной. Следует отметить, что вызовы могут быть многократно вложенными. При этом вызвавшая машина после осуществления вызова блокируется до окончания работы вызванной, после чего продолжает своё выполнение. Таким образом, можно задавать глубину виртуализации кода путем задания многократного запуска протектора.

2.2.6 Классы используемых виртуальных машин

Все многообразие виртуальных машин, которое строится генератором виртуальных машин, можно разделить на следующие классы:

- по способу адресации: регистровые и стековые;
- по составу и сложности команд: CISC или RISC;
- виртуальные машины с командами одинаковой или разной длины;
- виртуальные машины с регистрами одинаковой или разной длины;

- виртуальные машины с регистрами одного или разных типов.

Эти классы виртуальных машин полны по Тьюрингу т.е. на них можно реализовать любую вычислительную функцию. Рассмотрим их подробнее.

2.2.7 Стековые и регистровые виртуальные машины

Стековые виртуальные машины

Этот класс виртуальных машин определяется организацией регистрового файла в виде стека, и косвенной адресацией регистров через указатель стека, который определяет положение вершины стека [12]. Операции производятся над значениями на вершине стека, и результат кладётся также на вершину.

Все команды используются без операндов, поэтому арифметические операции кодируются в нуль-операндные инструкции. Например, команда sub (вычесть) берет с вершины стека два числа, вычитает первое из второго и результат вталкивает в стек. Разумеется, до этого необходимые значения вставляются в стек.

Отдельной команды, которая кладет в стек значение переменной, или константу нет.

Переменная или константа сама является командой вставки этой константы в стек.

Вот пример кода для выражения a*(b-10) (действие команды MUL аналогично SUB, только вместо вычитания она перемножает значения в стеке):

a

b

10

SUB

MUL

Таким образом, стековая система команд для выражения - это система команд в виде обратной польской записи.

Достоинства:

- простота аппаратной реализации;

- простота записи алгоритмов вычисления;

- простота мнемонического описания микроопераций (с одним или без операндов).

Недостатки:

- Стек – запоминающее устройство с последовательным доступом обладающее медленной скоростью работы. Но этот недостаток не играет существенной роли т.к. главная задача – надежная защита кода [13].

К стековым виртуальным машинам относятся:

– Java Virtual Machine;
– UCSD Pascal p-machine;
– VES.

Регистровые виртуальные машины

Регистровая архитектура характеризуется свободным доступом к регистрам для выборки всех аргументов и записи результата [14]. Это значит, что для регистровой виртуальной машины, в отличие от стековой, в качестве хранилища промежуточных результатов вычислений используется не стек, а регистры общего назначения. Эти регистры, в каком-то смысле, можно рассматривать как явно управляемый кэш для хранения недавно использовавшихся данных. Размер регистров может быть как фиксирован, так и нет. К любому регистру можно обратиться, указав его имя.

Общий вид операций наших регистровых виртуальных машин имеет вид:

```
OP получатель, источник1, источник2, … , источникN
```

и

```
OP получатель
```

OP - это код операции. В качестве источника может быть либо регистр, либо переменная, либо число. Из источника берутся данные для выполнения операции. Получателем для наших виртуальных машин может быть как регистр, так и переменная. Результат операции помещается в получателя.

Код операции вместе с кодами операндов для создаваемых виртуальных машин занимают максимум 64 бита. При этом если у инструкции есть операнды, то их коды должны идти следом за кодом инструкции и их число не должно превышать семи.

Пример команды MOV(загрузить) с различными источниками:

```
MOV R1, S (загрузить значение переменной S в регистр R1)
MOV R2, 24 (загрузить число 24 в регистр R2)
MOV R1, R5 (загрузить значение из регистра R5 в регистр R1)
```

Регистровые виртуальные машины допускают расположение операндов в одной из двух запоминающих сред: основной памяти или регистрах. Мы будем рассматривать три подвида команд обработки: регистр-регистр; регистр-память; память-память.

- В варианте «регистр-регистр» операнды могут находиться только в регистрах. В них же записывается и результат.

- Подтип «регистр-память» предполагает, что один из операндов размещается в регистре, а второй в основной памяти. Результат обычно замещает один из операндов.

- В командах типа «память-память» оба операнда хранятся в основной памяти. Результат также заносится в память.

Вариант «регистр-регистр» является основным в вычислительных машинах типа RISC. Команды типа «регистр-память» характерны для CISC-машин. В наших же виртуальных машинах оба эти варианта будут равноправны для RISC и CISC виртуальных машин.

Достоинства:

- Обеспечивается высокая скорость работы с кэшем, что позволяет эффективнее осуществлять арифметические операции.

Недостатки:

- При переходе к решению задач управления эффективность таких виртуальных машин падает, так как при переключениях подпрограмм необходимо разгружать и загружать регистры в кэш [17].

Примерами регистровых виртуальных машин являются:

– Dalvik Turbo virtual machine;
– SWEET16;
– Parrot virtual machine;
– Dis virtual machine.

2.2.8 RISC – CISC виртуальные машины

Двумя основными классами виртуальных машин по составу команд являются виртуальные машины с архитектурой CISC и RISC.

а) RISC (Reduced (Restricted) Instruction Set Computer) – уменьшенный набор команд, которыми пользуется виртуальная машина, содержащая только наиболее простые команды.

Особенности этого класса виртуальных машин.

– Эти виртуальные машины имеют набор однородных регистров универсального назначения, причем их число может быть большим. Система команд отличается относительной простотой, коды инструкций имеют четкую структуру, как правило, с фиксированной длиной.
– Для обработки данных, как правило, используются трехадресные команды, что помимо упрощения дешифрации дает возможность сохранять большее число переменных в регистрах без их последующей перезагрузки.
– Обращение к памяти допускается лишь с помощью специальных команд чтения и записи.
– Небольшое количество форматов команд и способов указания адресов операндов.

– Реализация сложных команд за счет последовательности простых, но быстрых RISC-команд [16].

Примерами RISC виртуальных машин являются:

– TRVM, the Tiny *RISC Virtual Machine*;
– DALVIC virtual machine.

б) CISC (Complete Instruction Set Computer) – полный набор команд микропроцессора. Для виртуальных машин CISC-архитектуры характерны:

– наличие сравнительно небольшого числа регистров общего назначения;
– большое количество машинных команд, некоторые из них аппаратно реализуют сложные операторы виртуальной машины;
– разнообразие способов адресации операндов;
– множество форматов команд различной разрядности;
– наличие команд, где обработка совмещается с обращением к памяти.

Особенностью виртуальных машин с RISC архитектурой является наличие достаточно большого регистрового файла (в типовых RISC-процессорах реализуются 32 или большее число регистров по сравнению с 8–16 регистрами в CISC архитектурах). Это позволяет большему объему данных храниться в регистрах более длительное время и упрощает работу компилятора по распределению регистров под переменные.

Количество регистров в архитектурах типа CISC обычно невелико (от 8 до 32), и для представления номера конкретного регистра необходимо не более пяти разрядов, благодаря чему в адресной части команд обработки допустимо одновременно указать номера двух, а зачастую и трех регистров (двух регистров операндов и регистра результата). RISC-архитектура предполагает использование существенно большего числа регистров общего назначения (до нескольких сотен). Однако, типичная для таких ВМ длина команд (обычно 32 разряда) позволяет определить в команде до трех регистров [18]. В разрабатываемом протекторе длина команды составляет 64 бита, что позволяет создавать команды с числом операндов, не превышающим семи, при этом, на данный момент реализованы только команды, имеющие до трех операндов.

Примерами CISC виртуальных машин являются:

– DIS virtual machine;
– Python VM;
– Themida.

2.2.9 Виртуальные машины с командами одинаковой и разной длины

Средняя длина команды составляет 3,5 байта. Поскольку МП 80386 использует 16-байтовую очередь команд, в среднем 5 команд оказываются предварительно выбранными из памяти[19].

Команды содержат 0, 1, 2 или 3 операнда и, теоретически можно так же реализовать команды с числом операндов равным 4, 5, 6 и 7. Операнды могут храниться в регистре, в самой команде или в памяти. Большинство безоперандных команд занимают лишь 1 байт, в то время как двухоперандные команды обычно имеют длину 2 байт. Однако к любой команде может быть добавлен префикс замены длины операнда, действующей по умолчанию.

Принадлежность виртуальной машины к одному из этих классов определяется принадлежностью к определенному классу CISC или RISC.

Для виртуальных машин класса RISC длина команд одинаковая, а для виртуальных машин класса CISC она варьируется.

2.2.10 Виртуальные машины с регистрами одинаковой или разной длины

Регистры виртуальной машины могут иметь фиксированную или варьирующуюся длину регистров. Принадлежность к одному из этих классов для каждой виртуальной машины выбирается произвольным образом.

Примеры виртуальных машин с регистрами одной длины:

– Dalvic virtual machine (все регистры 32 бита);
– Java virtual machine (все регистры 32 бита).

Примеры виртуальных машин с регистрами разной длины:

– Oracle;
– Parrot virtual machine;
– CLR.

2.2.11 Виртуальные машины с регистрами одного или разных типов

Регистры виртуальной машины могут быть различных типов – целочисленные, числа с плавающей точкой, строковые, специальных типов данных. Принадлежность к одному из этих классов для каждой виртуальной машины выбирается произвольным образом.

Пример виртуальной машины с регистрами одного типа: Java Virtual Machine – все регистры типа float.

Примеры виртуальных машин с регистрами разных типов:

– Parrot – имеет регистры четырех типов: float, integer, string, parrot magic cookie (специальный тип для этой виртуальной машины);

– Dis virtual machine.

2.2.12 Определение набора правил, по которым происходит создание всего многообразия виртуальных машин

Каждая виртуальная машина содержит интерпретатор. Интерпретатор виртуальной машины для нашего протектора написан на языке C и реализуется в виде условного оператора типа «switch» с большим количеством условий. Интерпретатор обрабатывает входные данные, написанные на языке виртуальной машины. Язык виртуальной машины отличается от языка машин x86 и динамически формируется в процессе генерации виртуальной машины. Правила формирования языков виртуальных машин описаны дальше.

2.2.13 Коды операций виртуальных машин

Байт-код виртуальной машины представляет собой скомпилированный бинарный код, предназначенный исключительно для интерпретатора этой виртуальной машины. Алгоритм работы виртуальной машины:

1. Получить опкод виртуальной машины и ее аргументы.
2. Выполнить инструкцию.
3. Повторять шаг 1) и 2) до команды выхода.

Рассмотрим этот алгоритм на псевдокоде:

```
while (1) {
  opcode = NextOpcode();
  if (HasArg(opcode))
    oparg = NextArg();

  switch (opcode) {
  ...
  }
}
```

Код операции, опкод — часть машинного языка, называемая инструкцией и определяющая операцию, которая должна быть выполнена.

Определение и формат кодов инструкций зависит от системы команд данного процессора. В отличие от самого опкода, инструкция обычно имеет одно или больше определений для операндов, над которыми должна выполняться операция. В настоящий момент, для задания каждой инструкции используются 64 бита, где первые 8 зарезервированы непосредственно для определения команды, а 7 остальных для ее операндов. Таким образом, максимальное количество операндов инструкции равно 7.

В зависимости от архитектуры, операнды могут быть значениями регистров, значениями стека, значениями в памяти, непосредственными значениями.

У нас есть множество кодов операций для каждой построенной виртуальной машины. Эти опкоды уникальны по построению. Закодируем опкоды, чтобы скрыть структуру построения. Нам необходимо сохранить уникальность опкодов, и, кроме того, полученные опкоды должны так же иметь размерность 8 бит. Таким образом, необходимо построить автоморфизм – изоморфизм группы на себя. Для каждого опкода сгенерируем 3 произвольных значения:

```
key1 = randFromTo(0, 7);
key2 = randFromTo(0, 0xFF);
key3 = randFromTo(0, 7);
```

И выполним следующие автоморфные преобразования опкодов.

```
b = ror(b, key1);   циклический сдвиг вправо
b = b ^ key2;       поразрядное исключающее или
b = rol(b, key3);   циклический сдвиг влево
```

Так как расшифровывать опкоды нет необходимости, можно использовать любые автоморфные преобразования.

2.2.14 Регистры виртуальных машин

Для построения языков виртуальных машин определим сначала, что представляют собой регистры виртуальных машин. Рассмотрим конечное множество R регистров заданной виртуальной машины. Каждый регистр характеризуется именем и размером:

```
∀r ∈ R, r = <name, size>
```

Если переменная, помещаемая в этот регистр, имеет меньший размер, то свободное место будет заполняться нулями. Такой подход позволяет упростить правила для данного класса операций, но при этом усложнить анализ, так как каждый регистр будет содержать скрытую реализацию дополнения свободного места нулями. Это также позволит скрыть реальный размер переменных.

Множество имен регистров будет уникальным для каждой виртуальной машины. На этапе генерации виртуальной машины случайным образом сгенерируется конечное множество имен регистров. Так как все регистры имеют одинаковый размер не надо делать акцент на их имени, который позволит по именам определять размер регистра (как это сделано в языке ассемблер: eax, ebx – 32-разрядные регистры, ax, bx – 8-разрядные регистры и т.д.).

Пример:

```
c := a + b
```

Код на языке ассемблера:

```
mov eax, a
add eax, b
mov c, eax
```

Код на языке виртуальной машины:

```
mvvm17 R11, a
mvvm17 R12, b
advm17 R13, R11, R12
mvvm17 c, R13
```

Набор создаваемых кодов (имен) регистров уникален по построению. Чтобы скрыть структуру построения кодов регистров используем автоморфные преобразования, аналогичные преобразованиям для кодов инструкций.

Также для всех виртуальных машин введем следующие ограничения по числу регистров:

– количество регистров не может быть меньше двух;

– максимальное количество регистров не превышает целого числа n, которое динамически определяется для каждой виртуальной машины.

Остальные правила будут изменяться в зависимости от классификации, приведенной в п. 3.4. Создадим набор правил построения таких виртуальных машин.

2.2.15 Правил построения виртуальных машин разных классов

1) Стековые или регистровые

а) Стековые

Для стековых виртуальных машин все операции будут описываться в польской записи. Операнды для каждой операции будут браться из вершины стека. Класс операций пересылки данных для таких виртуальных машин не определен, так как сама по себе переменная или константа означает, что она лежит в вершине стека.

На данный момент, все создаваемые виртуальные машины являются регистровыми, а стек существует как дополнительный механизм. Реализация стековых виртуальных машин может стать предметом дальнейшей работы.

б) Регистровые

Для регистровых виртуальных машин все операции представляются в виде:

```
операция приемник, источник1, исторник2 …
```

Для каждой операции необходимо явно указывать регистр или переменную, куда будет записан результат операции. Приемник может одновременно являться и источником, в таком случае результат операции будет положен в приемник, а предыдущее значение не сохраняется.

В отличие от x86, количество источников для каждой операции может варьироваться, но не превышать 7, как описано в разд. 3.4. Наиболее распространенный случай для арифметических операций - один приемник и два источника, например:

add32 R1, R2 (сложить значения регистров 32 –битных регистров R2 и R1, результат положить в R1)

Но также могут встречаться операции с одним или тремя … или с семью источниками.

2) CISC или RISC архитектура

а) CISC

Для виртуальных машин типа CISC будет переопределено всё многообразие команд x86. Основными правилами построения таких виртуальных машин является наличие небольшого числа регистров общего назначения, большое количество машинных команд, разнообразие способов адресации операндов, наличие команд, где обработка совмещается с обращением к памяти.

б) RISC

Виртуальные машины типа RISC будут содержать ограниченный (базовый) набор инструкций. Это означает, что одна инструкция типа CISC будет представлена в виде нескольких инструкций виртуальной машины.

Например, инструкция со сложным обращением к памяти

```
mov eax, [ebx+ecx*4+123456h]
```

будет заменена на следующую последовательность инструкций:

```
mul ecx, 4
add ebx, ecx
add ebx, 123456h
mov eax, ebx
```

3) Виртуальные машины с командами одинаковой или разной длины

В виртуальных машинах с командами одинаковой длины команды имеют размер 8 байт.

Длина команд в виртуальных машинах с различной длиной команд зависит от количества операндов:

– 1 байт для команд без операндов,

– 2 байта для команд с одним операндом,

– 3 байта для двухоперандных команд.

Максимальный размер команды – 8 байт, и таким образом теоретически может быть до 7 операндов.

4) Виртуальные машины с регистрами одинаковой или разной длины

В виртуальных машинах, принадлежащих классу виртуальных машин с регистрами одинаковой длины, все регистры имеют размер 32 бита.

Виртуальные машины с регистрами разной длины будут поддерживать размеры регистров 8, 16, 32, 64 бита, а так же 80 бит для некоторого подкласса команд.

4) Виртуальные машины с регистрами одного или разных типов

В виртуальных машинах с регистрами одного типа все регистры имеют тип float.

В виртуальных машинах с регистрами разных типов поддерживаются регистры типов float и integer.

2.2.16 Правила построения табличного, не табличного и недетерминированного языков ассемблера виртуальной машины

Мы будем рассматривать следующие классы операций для построения языков ассемблера виртуальной машины:

а) арифметические операции;

б) логические операции;

в) операции сравнения;

г) control flow;

д) операции сдвига;

е) операции со стеком;

ж) операции пересылки данных.

Перечисленные выше классы представляют основные операции языков виртуальной машины и поэтому позволяют полностью описать всё многообразие этих языков. Большинство инструкций допускает довольно широкий спектр операндов, как с точки зрения синтаксиса, так и с точки зрения семантики.

Введем условные обозначения, используемые для описания операций виртуальных машин. Для определения этих инструкций используются следующие обозначения:

– сначала указывается мнемокод, отображающий семантику команды:

- xor – логическая операция «исключающего или»,
- or – операция логического сложения,
- and – операция логического умножения,
- not – операция логического отрицания,
- add – арифметическая операция сложения,
- sub – арифметическая операция вычитания,
- mul – арифметическая операция умножения,
- div – арифметическая операция деления,
- cmp – операция сравнения,
- je – операция перехода, если сравниваемые значения равны,
- jne – операция перехода, если сравниваемые значения не равны,
- jl – операция перехода для знаковых операндов, если значение первого операнда инструкции сравнения меньше значения второго,
- jg – операция перехода для знаковых операндов, если значение первого операнда инструкции сравнения больше значения второго
- jle– операция перехода для знаковых операндов, если значение первого операнда инструкции сравнения меньше или равно значению второго,
- jge– операция перехода для знаковых операндов, если значение первого операнда инструкции сравнения больше или равно значению второго,
- ja - операция перехода для беззнаковых операндов, если значение первого операнда инструкции сравнения выше значения второго
- jae - операция перехода для беззнаковых операндов, если значение первого операнда инструкции сравнения выше или равно значению второго
- jb - операция перехода для беззнаковых операндов, если значение первого операнда инструкции сравнения ниже значения второго,
- jbe – операция перехода для беззнаковых операндов, если значение первого операнда инструкции сравнения ниже или равно значению второго,
- load – операция извлечения из стека,
- store – операция вставки в стек,
- mov – операция пересылки данных,
- shr – операция побитового сдвига вправо,
- shl – операция побитового сдвига влево,
- rol – операция циклического побитового сдвига влево,
- ror – операция циклического побитового сдвига вправо,

    - call – вызов процедуры;
- для команд с вещественными операндами перед соответствующей командой ставится префикс f;
- затем определяются размеры операндов (8, 16, 32, 64 или 80 бит) – одно число, если у всех операндов один и тот же размер, несколько чисел, каждое из которых соответствует размеру одного из операндов, если операнды имеют разный размер. Так как после размера операнда указывается его тип, визуально эти значения разделены;
- как было сказано выше, после размера указывается тип операнда:
    - m – ячейка памяти,
    - r – регистр,
    - i – непосредственное значение,
    - b – регистр, содержащий адрес ячейки памяти;
- если команда имеет различные операции для знаковых/ беззнаковых соответствующих инструкций, то после указания типа операнда (операндов) указывается знаковый тип операнда (операндов):
    - s – знаковый (signed),
    - u – беззнаковый (unsigned);
- для команд перехода после этого определяется тип перехода: abs – абсолютный переход, переход по адресу в памяти, rel –относительный переход, переход относительно текущей позиции;
- затем через пробел указываются операнды - роль и тип (источник – приемник). Источник обозначается через i (input), приемник через o (output), операнд одновременно являющийся приемником и источником через io. Для команд условных переходов у второго операнда стоит буква j – jump, что говорит о том, что этот операнд определяет; для команд сдвига второй операнд с префиксов sh определяет сдвиг (shift). По аналогии с описанным выше типы операндов:
    - reg –регистр,
    - mem –ячейка в памяти,
    - imm - непосредственное значение,
    - brg - регистр, содержащий адрес ячейки памяти.

Например,

`div64s_mr io_mem, i_reg` – команда знакового деления 64 битных значений, где первый операнд – ячейка памяти, второй – регистр, при этом первый операнд является одновременно приемником и источником, а второй только источником.

Рассмотрим каждый из классов операций виртуальных машин подробнее.

2.2.17 Класс арифметических операций

К арифметическим операциям относятся сложение, вычитание, умножение и деление. Они могут быть разделены на две основные группы:

– операции с целочисленными данными;

– операции с вещественными данными.

Для каждой группы строится множество инструкций, основанное на изменении следующих параметров операндов:

1) Тип операндов

Виртуальные машины различают следующие типы операндов.

– Конкретное значение, известное на этапе компиляции — например, числовая константа или символ. Эти операнды называются непосредственными;

– Регистр;

– Указатель на ячейку в памяти;

– Регистр, содержащий адрес ячейки памяти.

При этом если операнд-источник может быть любого из перечисленных типов, то операнд-приемник не может быть конкретным значением.

Примеры арифметических инструкций с операндами разных типов:

```
add32_ri reg16, 25
sub64_mi [1456], 25
sub16_rr reg16, reg2
add32_ri reg13, 52
mul8_rm reg3, [0000h]
add6_mr [0145h], reg6
div64s_mr reg6, reg26
```

2) Размер операндов

Реализуется набор инструкций с операндами следующих размеров: 8, 16, 24, 32, 64 и 80 бит.

Примеры арифметических инструкций с разными размерами операндов:

```
add8_rr reg5, reg9
add16_rr reg5, reg9
add32_rr reg5, reg9
add64_rr reg5, reg9
```

```
add80_rr reg5, reg9
```

В данных примерах размер регистров составляет соответственно 8 бит для первого примера, 16 для второго, 32 для третьего.

3) Знак операндов

Все операнды инструкций виртуальных машин делятся на знаковые и беззнаковые. Инструкции сложения, вычитания и умножения для этих двух типов не отличаются, а для операции деления реализуется две отдельные инструкции – знакового и беззнакового деления.

4) Число операндов.

Все арифметические инструкции делятся на двухоперандные и трехоперандные.

В случае двухоперандных инструкций, первый операнд одновременно является и источником и приемников, а второй только источником. Это означает, что второй операнд не изменяет своего содержимого, а результат операции помещается в первый операнд.

Примеры двухоперандных инструкций:

```
add8_rr reg5 reg9
sub8_rr reg5 reg9
```

В случае трехоперанных инструкций, первый операнд является приемником, а второй и третий – источниками. Результат так же помещается в первый операнд, а второй и третий не изменяют своего содержимого.

Примеры трехоперандных арифметических инструкций:

```
add8_rr reg65, reg5, reg9
sub8_rr reg65, reg5, reg9
```

5) Значения операндов могут быть типа int или float, и, соответственно для разных типов значений используются разные команды.

Примеры операций с вещественными операндами:

```
fadd8_rr reg5 reg9
fsub8_rr reg5 reg9
```

Таким образом, для команды сложения реализовано 600 различных команд:

4*3*2*5 =120 двухоперандных и 4*4*3*2*5 = 480 трехоперандных.

Аналогично, для остальных арифметических команд. С учетом различных команд для операций знакового и беззнакового деления, общее число арифметических команд составляет $3*10^3$.

2.2.18 Класс логических операций

Логические команды выполняют над операндами логические (побитовые) операции, то есть они рассматривают операнды не как единое число, а как набор отдельных битов. Этим они отличаются от арифметических команд. Логические инструкции выполняют следующие основные операции:

- логическое И;
- логическое ИЛИ;
- сложение по модулю 2 (исключающее ИЛИ);
- логическая инверсия.

Таким образом, команды логических операций позволяют побитно вычислять основные логические функции от двух входных операндов. Кроме того, операция И (AND) используется для принудительной очистки заданных битов (в качестве одного из операндов при этом используется код маски, в котором разряды, требующие очистки, установлены в нуль). Операция ИЛИ (OR) применяется для принудительной установки заданных битов (в качестве одного из операндов при этом используется код маски, в котором разряды, требующие установки в единицу, равны единице). Операция «Исключающее ИЛИ» (XOR) используется для инверсии заданных битов (в качестве одного из операндов при этом применяется код маски, в котором биты, подлежащие инверсии, установлены в единицу). Команды требуют двух входных операндов и формируют один выходной операнд.

Для логических операций многообразие инструкций строится по тем же правилам, что и для арифметических.

Примеры логических операций виртуальных машин:

```
or8_rr reg65, reg5
xor8_rr reg65, reg5, reg9
and16_ri reg13, 25
not16_r reg5
```

Таким образом, число логических команд превышает 10^3.

2.2.19 Класс операций сравнения

Инструкции виртуальных машин для сравнения чисел являются аналогами ассемблерной инструкции cmp. После этих команд используются команды условного перехода, описанные в пункте 3.7.4. и, таким образом, используя последовательно cmp и команды условного перехода, организовываются условные операторы, циклы и так далее, как описано в пункте 3.7.4.

Операции сравнения имеют 2 операнда, первый является одновременно приемником и источником, второй только источником. Первый операнд сравнивается со вторым и результат сравнения записывается в первый операнд.

Для создания многообразия команд сравнения данных варьируются параметры операндов.

Тип операндов: в качестве первого операнда может выступать регистр, указатель на ячейку в памяти, или регистр, содержащий адрес ячейки памяти. В качестве второго - конкретное значение, регистр, указатель на ячейку в памяти, или регистр, содержащий адрес ячейки памяти.

Размер операндов может быть одним из следующих: 8, 16, 32, 64 бит. При этом для операций сравнения размеры первого и второго операнда могут различаться.

Операнды могут быть **знаковыми** и **беззнаковыми**. В случае знаковых операндов можно сделать вывод о том, что первый операнд больше/меньше/равен второму, а в случае беззнаковых выше/ниже/равен, и, в зависимости от этого, использовать различные команды условного перехода.

Число операндов: они могут быть двухоперандные и трехоперандные:

в двухоперандных инструкциях первый операнд является одновременно приемником и источником, второй является источником, в трехоперандных – первый является приемником, второй и третий источником.

Примеры операций сравнения:

```
cmp64bu16iu [reg1], 15
cmp16bs64ms [reg17], 34
cmp32bu16bu [reg19], [reg13]
cmp16rs32rs reg3, reg4
cmp16bs16bs [reg5], [reg6]
cmp16rs16bs16bs reg8 [reg5], [reg6]
```

Общее число команд сравнения превышает 10^3.

2.2.20 Класс control flow

В императивном программировании control flow (порядок исполнения, порядок вычислений) — это способ упорядочения инструкций программы в процессе ее выполнения.

Инструкции, входящие в программу, могут исполняться как последовательно, одна за другой, так и одновременно, однократно или многократно. Последовательность исполнения инструкций может совпадать с последовательностью их расположения в записи программы или не совпадать, а также зависеть от текущего состояния вычислителя, исполняющего программу и

от внешних событий, образовывая, таким образом, разнообразные порядки выполнения инструкций [21].

Организация желаемого порядка выполнения может быть осуществлена с помощью различных механизмов, таких как специализированные инструкции или управляющие конструкции высокоуровневых языков программирования или встроенные механизмы для прерывания, сохранения и восстановления состояния, модификация и генерация инструкций программы и других.

Простейшим порядком выполнения является естественный порядок, когда инструкции выполняются последовательно, в порядке их появления в записи программы.

Отклонение от естественного для применяемого способа записи порядка называется переход. В этом случае после окончания выполнения текущей инструкции вычислитель переходит не к следующей в записи, а к некоторой другой, произвольно заданной инструкции [21]. При безусловном переходе инструкция перехода выбирается без учета состояния вычислителя, а при условном переходе выбирается в зависимости от состояния вычислителя путём проверки условия.

Команды перехода виртуальных машин будут иметь тот же принцип работы, что и инструкции x86. Если в коде встречается инструкция, меняющая порядок выполнения, то осуществляется переход на указанную область памяти и с неё продолжается выполнение программы. Все команды перехода делятся на команды перехода с возвратом и без возврата.

1. Инструкции перехода без возврата записывают в регистр-счетчик команд новое значение и тем самым вызывают переход не к следующей по порядку команде, а к любой другой команде в памяти программ. И далее процессор продолжает выполнение программы с новой точки.

Команды переходов делятся на две группы:

- команды безусловных переходов;
- команды условных переходов.

1.1. Команды безусловных переходов вызывают переход в новый адрес независимо ни от чего. Они могут вызывать переход на указанную величину смещения (вперед или назад) или же на указанный адрес памяти. В многообразии инструкций виртуальных машин инструкции безусловного перехода делятся на два класса: инструкции относительного перехода (на величину смещения) и абсолютного перехода (на указанный адрес памяти). Величина смещения или новое значение адреса указываются в качестве единственного входного операнда. Этот операнд всегда 32 битный, что соответствует размеру поля перехода

Примеры:

```
jmp_rel -5     – относительный переход
jmp_abs [reg3] – переход по адресу,
находящемуся в reg3
```

1.2. Команды условных переходов вызывают переход при выполнении заданных условий. Эти команды имеют два входных операнда: значение первого операнда выступает в качестве условия, в качестве второго операнда выступает величина смещения или новое значение адреса. Таким образом, команды условного перехода так же делятся на команды относительного и абсолютного перехода. Несколько примеров команд условных переходов:

- переход, если равно нулю;
- переход, если не равно нулю;
- переход, если больше нуля;
- переход, если больше или равно нулю;
- переход, если меньше нуля;
- переход, если меньше или равно нулю.

Кроме того, многообразие инструкций условного перехода определяется изменением следующих параметров:

- размер первого операнда: могут быть использованы операнды следующих размеров 8, 16, 32, 64, 80. При этом второй операнд всегда 32 битный;
- тип операндов: операнды могут быть регистрами, адресами ячеек памяти или непосредственными значениями, регистрами, содержащими адрес ячейки памяти;
- первый операнд может быть знаковый или беззнаковый.

Совместное использование нескольких команд условных и безусловных переходов позволяет виртуальному процессору выполнять разветвленные алгоритмы любой сложности.

Примеры команд условных переходов:

```
jae32bb_abs [reg3], [reg6]
jle32rr_abs reg7, reg9
jbe32mr_rel [0146h], reg4
jne32mr_abs [0245h], reg3
```

Таким образом, команд условного перехода более $1.5*10^3$.

2. Команды переходов с дальнейшим возвратом в точку, из которой был произведен переход, применяются для выполнения подпрограмм, то есть вспомогательных программ. Эти команды называются также командами вызова подпрограмм (аналог ассемблерной команды CALL). Использование подпрограмм позволяет упростить структуру основной

программы, сделать ее более логичной, гибкой, легкой для написания и отладки.

Все команды переходов с возвратом предполагают безусловный переход. При этом они требуют одного входного операнда, который может указывать как абсолютное значение нового адреса, так и смещение, складываемое с текущим значением адреса. Текущее значение счетчика команд (текущий адрес) сохраняется перед выполнением перехода.

Для обратного возврата в точку вызова подпрограммы (точку перехода) используется специальная команда возврата (аналог ассемблерной инструкции RET). Эта команда получает сохраненное значение адреса команды перехода и записывает его в регистр-счетчик команд.

2.2.21 Класс операций сдвига

Команды сдвигов позволяют побитно сдвигать значение операнда вправо (в сторону младших разрядов) или влево (в сторону старших разрядов). Циклические сдвиги позволяют сдвигать биты значения операнда по кругу (по часовой стрелке при сдвиге вправо или против часовой стрелки при сдвиге влево) [22]. Эти команды имеют два операнда: в первом находится значение, сдвиг которого мы хотим осуществить, во втором - величина сдвига.

Множество команд сдвигов основывается на варьировании следующих параметров операндов.

Тип операндов

В качестве первого операнда может выступать регистр, указатель на ячейку в памяти, или регистр, содержащий указатель на ячейку памяти. В качестве второго - конкретное значение, регистр, указатель на ячейку в памяти, или регистр, содержащий адрес ячейки памяти.

Размер операндов

Реализуется набор инструкций с операндами следующих размеров: 8, 16, 24, 32, 64 и 80 бит.

Число операндов

В двухоперандных инструкциях – первый операнд является одновременно приемником и источником, второй является источником; в трехоперандных – первый является приемником, второй и третий источником.

Примеры операций сдвига:

```
shl32_rr reg1, reg2
shr16_ri reg6, 13
```

```
shr32_rri reg3, reg5, 34
rol64_ri reg7, 21
ror8_mr [0000h], reg5
```

Таким образом, число команд сдвигов превышает 1000.

2.2.22 Класс операций для работы со стеком

Стек - это способ хранения данных. В языке виртуальных машин можно выделить 2 группы команд работы со стеком: команды, которые кладут новую порцию данных на верхушку стека (STORE), и команды которые получают данные с вершины стека (LOAD). Использование стека позволяет сохранять данные в памяти, не заботясь об их абсолютных адресах.

Пример вызова этих функций: команда вызова функции CALL32 func записывает в стек адрес следующей за ней команды, а команда возврата RET достает его из стека.

Формально стек ограничен лишь протяженностью адресного пространства. Направление роста стека: от больших адресов – к меньшим. Еще говорят, что стек растет снизу вверх.

Если генерируемая виртуальная машина принадлежит к классу регистровых виртуальных машин, то класс операций работы со стеком аналогичен языку ассемблер: виртуальные машины могут класть в стек или брать из стека как регистры, так и переменные.

Если же генерируемая виртуальная машина принадлежит классу стековых виртуальных машин, то класс операций работы со стеком немного иной. Для стековых виртуальных машин не существует отдельных операций, которые кладут в стек значения переменных или констант, так как они сами по себе являются командами вставки в стек, как описано в п.3.4.1.

Для регистровой машины эти инструкции содержат два операнда: первый – регистр, в который загружается в стек (которое загружается из стека), второй – значение сдвига. Для создания множества инструкций работы со стеком варьируются следующие параметры операндов:

- размер регистра (первый операнд) – он может быть одним из следующих: 8, 16, 32, 64, 80 бит.
- величина переменной, хранящей смещение (второй операнд) – он может быть 8, 16 или 32 бит.

Примеры инструкций работы со стеком:

```
load80_16 reg3, 4
load16_8 reg1, 8
store64_8 reg8, 4
store64_16 reg4, 16
```

2.2.23 Класс операций пересылки данных

Инструкции виртуальных машин, осуществляющие пересылку данных, являются аналогом ассемблерной команды mov за тем исключением, что машины архитектуры x86 не поддерживают адресации типа память – память; одно из обрабатываемых чисел обязательно должно находиться в регистре или представлять собой непосредственное значение. Генерируемые виртуальные машины это жесткое правило соблюдать не будут. Арифметические операции над данными в памяти будут разрешены. Таким образом, для выполнения арифметической операции данные не надо будет предварительно класть в стек и, таким образом, мы усложним проблему анализа языка виртуальной машины.

Пример:

```
temp := a
a := b
b := temp
```

Код на языке ассемблер:

```
mov eax, a
mov ebx, b
xchg eax, ebx
```

Код на языке виртуальной машины:

```
chgvm_mm a, b
```

Команды пересылки данных не требуют выполнения никаких операций над операндами. Операнды просто пересылаются (точнее, копируются) из источника в приемник.

Для создания многообразия команд пересылки данных варьируются параметры операндов.

- В качестве первого операнда (приемника) может выступать регистр, указатель на ячейку в памяти, или регистр, содержащий адрес ячейки памяти. В качестве второго (источника)- конкретное значение, регистр, указатель на ячейку в памяти, или регистр, содержащий адрес ячейки памяти.
- Размер операндов может быть одним из следующих: 8, 16, 32, 64, 80 бит.

2.2.24 Получение полного набора инструкций

У разных процессоров системы команд существенно различаются по количеству, составу, а так же семантикой. Если рассматривать количество команд, то у процессора 8086 — 133

команды. У современных мощных процессоров количество команд достигает нескольких сотен [23].

Как было сказано в разделе 2, имея исходный алгоритм, можно построить виртуальную машину, содержащую не полный по Тьюрингу набор команд, а набор команд достаточный для выполнения данного алгоритма. Кроме того, набор инструкций генерируемых протектором явно избыточен. Из раздела 3 видно, что общее число команд, создаваемое генератором виртуальных машин, составляет около $8*10^3$ команд. Этот набор включает множество одинаковых инструкций с различной семантикой, например, он содержит множества одинаковых инструкций, у которых отличается только размер операндов. Таким образом, если использовать весь набор этих инструкций, то защищенные файлы будут весить гораздо больше исходных, так как виртуальная машина записывается в защищенный исполняемый файл. Кроме того это предоставляет большое количество данных для анализа системы защиты злоумышленником. Поэтому для каждой виртуальной машины произвольным образом из всего многообразия выбирается функционально-полный набор из 200 команд, причем состав этого набора будет специфичный для каждого запуска генератора виртуальных машин. Число 200 выбрано, как среднее число команд у современных процессоров.

Очевидный способ ограничения набора инструкций – использовать из множеств одинаковых инструкций с различными длинами операндов, только одну инструкцию с произвольным размером операндов. Это так же повысит надежность защиты генератора, так как добавится еще один параметр варьирования.

Кроме того, можно ограничить число используемых инструкций, используя инструкции только с определенным типом операндов для всех команд кроме команд пересылки данных. Благодаря многообразию команд пересылки данных, одни типы операндов преобразуются к другим, и набор команд остается функционально-полным.

Кроме того, существуют различные инструкции, которые могут быть выражены друг через друга. К ним относятся логические инструкции, и инструкции сравнения, а так же инструкции циклического сдвига. Рассмотрим их подробнее.

2.2.25 Полнота подмножеств инструкций

Рассмотрим следующие множества логических инструкций:

- множества логических инструкций,
- множества инструкций сравнения,
- множества «избыточных» инструкций.

2.2.26 Логические инструкции

Пусть $P = \bigcup_0^\infty P_n$ - это множество всех булевых функций с знаками операций из множества { ∧, ∨, ¬, ⊕ } .

Определение 4.1. Система булевых функций *F* называется *полной*, если формулами над этой системой можно задать любую булеву функцию из *P* .

Определение 4.2.

Функция $f \in P$ сохраняет 0 (сохраняет 1),если f(0,0,…0)=0 (f(1,1,…,1)=1). Класс всех *функций, сохраняющих 0*, обозначим через S_0, а класс всех *функций, сохраняющих 1,* - через S_1.

Функция $f \in P$ называется *самодвойственной*, если для любого набора аргументов $(\sigma_1, \sigma_2, \ldots, \sigma_n) \in B^n$ выполняется равенство $f(\sigma_1, \sigma_2, \ldots, \sigma_n) = \neg f(\neg\sigma_1, \neg\sigma_2, \ldots, \neg\sigma_n)$. Класс всех *самодвойственных функций* обозначим через *S*.

Функция $f \in P$ называется *линейной*, если она может быть задана линейным многочленом Жегалкина вида $\alpha_0 + \alpha_1 X_1 + \alpha_2 X_2 + \ldots + \alpha_n X_n$, где $\alpha_i \in \{0,1\}$ при $i=1..n$. Класс всех *линейных функций* обозначим через *L*.

Функция $f \in P$ называется *монотонной*, если для любых двух наборов аргументов $(\sigma_1, \sigma_2, \ldots, \sigma_n) \in B^n$ и $(\rho_1, \rho_2, \ldots, \rho_n) \in B^n$ таких, что для всех $j \in [1,n]$ $\sigma_1 \geq \rho_n$ имеет место неравенство $f(\sigma_1, \sigma_2, \ldots, \sigma_n) \geq f(\rho_1, \rho_2, \ldots, \rho_n)$. Класс всех *монотонных функций* обозначим через *M*.

Теорема 4.1 (Теорема Поста о полноте): Система булевых функций F является *полной* тогда и только тогда, когда она не содержится ни в одном из классов S_0, S_1, S, M и L. Другими словами, система булевых функций F является полной тогда и только тогда, когда в ней имеется хотя бы одна функция, не сохраняющая 0, хотя бы одна функция, не сохраняющая 1, хотя бы одна несамодвойственная функция, хотя бы одна немонотонная функция и хотя бы одна нелинейная функция [24].

Рассмотрим таблицу истинности (табл. 4.1) для логических инструкций и сделаем вывод о соответствии между булевскими функциями, входящими в набор инструкций виртуальных машин и классами Поста.

Таблица 5.1

Таблица истинности стандартных булевских функций

x	y	x∨y	x∧y	x⊕y	¬x	¬y	$\overline{\bar{x} \vee \bar{y}}$	$\overline{\bar{x} \wedge \bar{y}}$	$\overline{\bar{x} \oplus \bar{y}}$
0	0	0	0	0	1	1	0	0	0
0	1	1	0	1	1	0	0	1	1
1	0	1	0	1	0	1	0	1	1
1	1	1	1	0	0	0	1	1	1

На основании табл. 5.1 построим табл. 5.2 принадлежности булевских функций классам Поста. Знаком «#» отмечается не принадлежность функции классу, «+» - принадлежность.

Таблица 5.2

Булевские функции и классы Поста

Функции	*Классы булевых функций*				
	S_0	S_1	S	L	M
$x \vee y$	+	+	#	#	+
$x \wedge y$	+	+	#	#	+
$x \oplus y$	+	#	#	+	#
$\neg x$	#	#	+	+	#
1	#	+	#	+	+
0	+	#	#	+	+

Таким образом, функционально полными являются следующие наборы инструкций виртуальных машин:

1) OR, AND, XOR, NOT

Это функционально полный набор логических операций, включающий все возможные логические инструкции виртуальных машин протектора.

2) OR, NOT

Выразим остальные инструкции, через инструкции OR и NOT

AND: $x \wedge y = \overline{\bar{x} \vee \bar{y}}$ - правило де Моргана.

XOR: $x \oplus y = \bar{x}\, y \vee x\, \bar{y} = \overline{x \vee \bar{y}} \vee \overline{y \vee \bar{x}}$

3) AND, NOT

Выразим остальные инструкции, через инструкции AND и NOT

OR: $x \vee y = \overline{\bar{x} \wedge \bar{y}}$

XOR: $x \oplus y = \bar{x}\, y \vee x\, \bar{y} = \overline{\overline{\bar{x}\, y} \wedge \overline{x\, \bar{y}}}$

4) OR, NOT, XOR

В зависимости от функций, составляющих функционально полный набор, остальные функции могут выражаться по-разному, увеличивая многообразие виртуальных машин. Так для этого команда AND может выражаться 2 способами:

AND: $x \wedge y = (x \vee y) \oplus x \oplus y$

AND: $x \wedge y = \overline{\bar{x} \vee \bar{y}}$

5) AND, NOT, XOR

В этом случае, команда OR выражается двумя способами:

OR: $x \vee y = \overline{\bar{x} \wedge \bar{y}}$

OR: $x \vee y = (x \wedge y) \oplus (x \oplus y)$

6) XOR, AND, OR

NOT: $\neg x = x \oplus 1$

7) XOR, AND

NOT: $\neg x = x \oplus 1$

OR: $x \vee y = ((x \oplus 1) \wedge (y \oplus 1) \oplus 1$

8) XOR, OR

NOT: $\neg x = x \oplus 1$

AND: $x \wedge y = ((x \oplus 1) \vee (y \oplus 1)) \oplus 1$

2.2.27 Инструкции сравнения

Аналогично с логическими инструкциями, инструкции сравнения так же могут быть выражены друг через друга. Рассмотрим полные наборы операций сравнения.

1) (“ > ”, “ = ”)

Выразим для этого набора все остальные операции сравнения. Для остальных наборов операции сравнения, не вошедшие в набор, выражаются аналогично.

$x > y \approx \neg(x<y) \wedge \neg(x=y)$

$x >= y \approx \neg(x<y)$

$x <= y \approx (x<y) \vee (x=y)$

$x != y \approx \neg(x<y) \wedge \neg(x=y)$

2) (“ < ”, “ = ”)

3) (“ > ”, “ < ”)

4) (“ !=”, “ > ”)

5) (“ != ”, “ < ”)

6) (“ > =”, “ = ”)

7) (“ > =”, “ = ”)

8) (“ > ”, “ = ”)

Выше приведен список минимальных функционально-полных наборов инструкций сравнения. Очевидно, что при добавлении к ним дополнительных инструкций свойство функциональной полноты сохраняется, но набор инструкций изменяется.

2.2.28 «Избыточные» инструкции

К этому классу инструкций относятся инструкции циклического сдвига и пара инструкций (add, sub).

Циклический сдвиг

Действительно, инструкции циклического сдвига выражаются через инструкции правого и левого сдвига.

Так циклический сдвиг влево на s бит может быть выражен следующим образом:

```
int rotl(int a, int s) {
 return (a<<s) | (a>>sizeof(a)-s);
}
```

И циклический сдвиг вправо на s бит следующим образом:

```
int rotr(int a, int s) {
 return (a>>s) | (a<< sizeof(a)-s);
}
```

Пара инструкций add/sub

Пара сложение/ вычитание так же является избыточной, так как эти команды выражаются друг через друга.

```
x + y = x – (– y)
x – y = x + (– y)
```

2.2.29 Набор инструкций конкретной виртуальной машины

Прежде всего, все инструкции, входящие в виртуальную машину, берутся только с одним произвольным размером операндов. Затем, из классов инструкций, приведенных в разд.5.1., выбираем только по одному функционально-полному набору. Таким образом, мы получаем набор инструкций, достаточный для того, чтобы сопоставить любой инструкции LLVM, участвующей в исходной программе, одну или несколько команд вычислительной машины.

Затем, увеличиваем этот набор инструкций до двухсот произвольными инструкциями, чтобы набор инструкций виртуальной машины по их количеству был аналогичен набору инструкций ассемблера. Это поможет запутать злоумышленника, который захочет оценить состав команд виртуальной машины.

Таким образом, мы максимизируем функцию стоимости анализа (сложность расшифровки), а так же минимизируем функцию затрат (размер исходного исполняемого файла увеличивается не значительно)

2.2.30 Повышение скрытности алгоритма с использованием теории скрытности

За счет использования виртуализированного и исполняемого ПИОК кода возможно общее повышение скрытности алгоритма. Если рассматривать алгоритм обфускации в рамках теории скрытности то можно рассматривать полезные действия, совершаемые алгоритмом, как симптомы состояний [15] инструкций машинного кода, которые подвергаются обфускации.

Таким образом при замене кода на обфусцированный происходит добавление новых симптомов, скрывающих целевые.

Известно, что потенциальная энтропийная скрытность множества симптомов вычисляется по формуле $S(Y) = -\sum_{j=1}^{B} P(y_j)\log_2 P(y_j)$, где $P(y_j) = P_j = \sum_{i=1}^{A} P(x_i)P(y_j / x_i)$, определяется по формуле полной вероятности, где значения $P(y_j / x_i) = P_{j/i}$ - условная вероятность симптома y_j при состоянии объекта x_i. В нашем случае симптомами является информация доступная непосредственно отладчику, а состояние – это изменение состояний определённых целевым алгоритмом.

Очевидно, что увеличение состояний приводит к повышению неопределенности, но остается вопрос как это происходит и как это влияет на кривую снятия неопределённости при анализе в зависимости от параметров обфускации.

В теории скрытности вводится понятие арсенальной скрытности при этом под арсеналом зачастую понимается весь возможный набор параметров, предназначенный для реализации алгоритма. Таким образом, в качестве арсенала выбирается число команд процессора, исполняющего обфусцированный код. Очевидно, что это число может быть весьма велико.

Будем считать, что набор команд конкретной виртуальной машины x_i, $i = \overline{1, A}$, выбирается из имеющегося арсенала с равными вероятностями $p_i = 1/A$. При этом условии потенциальная **арсенальная скрытность** (A-скрытность) для виртуальной машины определяется выражением $S_0 = -\log_2(1/A) = \log_2 A$, где A – общее число возможных команд виртуальной машины.

Групповая же скрытность виртуальной машины тогда будет вычисляться из соотношения $S(m) = A\cdot\log_2 A - (A-m)\cdot\log_2(A-m) - -m\cdot\log_2 m + \frac{1}{2}\log_2\left[\frac{A}{2\pi(A-m)m}\right]$, A – общее

число возможных команд виртуальной машины, а m - число команд конкретной виртуальной машины. Очевидно, что максимум этой функции достигается при $m = \frac{A}{2}$.

Из этого можно сделать вывод, что желательно иметь максимально большое число разнообразных команд и оптимально использовать половину из них в конкретно выбранной виртуальной машине.

К сожалению, с практической точки зрения данные теоретические соображения не являются разумными, так вступают в противоречие с имеющимися фактами: чем больше число команд для процессора – тем медленнее идет кодогенерация. Таким образом выбор неоправданно большой группы команд m, для конкретной виртуальной машины может привести к следующим негативным последствиям:

- Медленной кодогенерации кода виртуальной машины
- Раскрытию всего арсенала команд при исследовании защищенного кода.

Зачастую отсутствуют временные рамки на исследование защищенного кода, поэтому наш арсенал (общий набор команд виртуальной машины будет раскрыт тем раньше, чем большее число команд используется в группе.

Таким образом мы можем прийти к следующим выводам:

1. Необходимо создавать генераторы виртуальных машин с максимально большим числом команд.
2. Для наилучшей защиты одной виртуальной машины, число команд должно выбираться равным половине общего числа команд.
3. В случае, если при помощи одного набора команд защищается множество алгоритмов, необходимо в каждом наборе использовать возможно меньшее число команд.

2.3 Использование сетей Петри для обфускации двоичного кода алгоритма

2.3.1 Определение сети Петри

Сети Петри — математический аппарат для моделирования динамических дискретных систем.

Определение 2.1: Сеть Петри C является четвёркой, C = (P, T, I, O). P = {p_1, p_2, … ,p_n} – конечное множество позиций, n≥0. T = {t_1, t_2, …, t_m} – конечное множество переходов, m≥0. Множество позиций и множество переходов не пересекаются, P∩T = □. I: T → P^∞ является

входной функцией – отображением из переходов в комплекты позиций. O: $T \rightarrow P^{\infty}$ есть выходная функция – отображение из переходов в комплекты позиций.

Для работы с сетями Петри гораздо удобнее использовать графическое представление сетей Петри в виде двудольного ориентированного графа, состоящего из вершин двух типов — позиций и переходов, соединённых между собой дугами. [26]

2.3.2 Маркировка сетей Петри

Фишка (точка, маркер) – это примитивное понятие сетей Петри. Фишки присваиваются и можно считать, что они принадлежат позициям. Количество и положение фишек при выполнении сети Петри могут меняться. Фишки используются для определения выполнения сети Петри. [26]

Определение 2.2: Маркировка μ сети Петри C = (P, T, I, O) есть функция, отображающая множество позиций P в множество неотрицательных целых чисел N:

$\mu: P \rightarrow N$

Маркировка μ есть присвоение фишек позициям сети Петри.

Маркировка μ может быть также определена как n-вектор $\mu = (\mu_1, \mu_2, \ldots, \mu_n)$, где $n = |P|$ и каждое $\mu_i \in N$, $I = 1, 2, \ldots, n$. Вектор μ определяет количество фишек для каждой позиции p_i сети Петри. Количество фишек в позиции p_i есть μ_i. Связь между определениями маркировки как функции и как вектора очевидным образом устанавливается соотношением $\mu(p_i) = \mu_i$.

Определение 2.3: Маркированная сеть Петри M = (C, μ) есть совокупность структуры сети Петри C = (P, T, I, O) и маркировки μ и может быть записана в виде M = (P, T, I, O, μ).

На графе сети Петри кружком O обозначается позицией, а планкой | - переход. Фишка обозначается точкой внутри кружка (см. рРисунок 2.1).

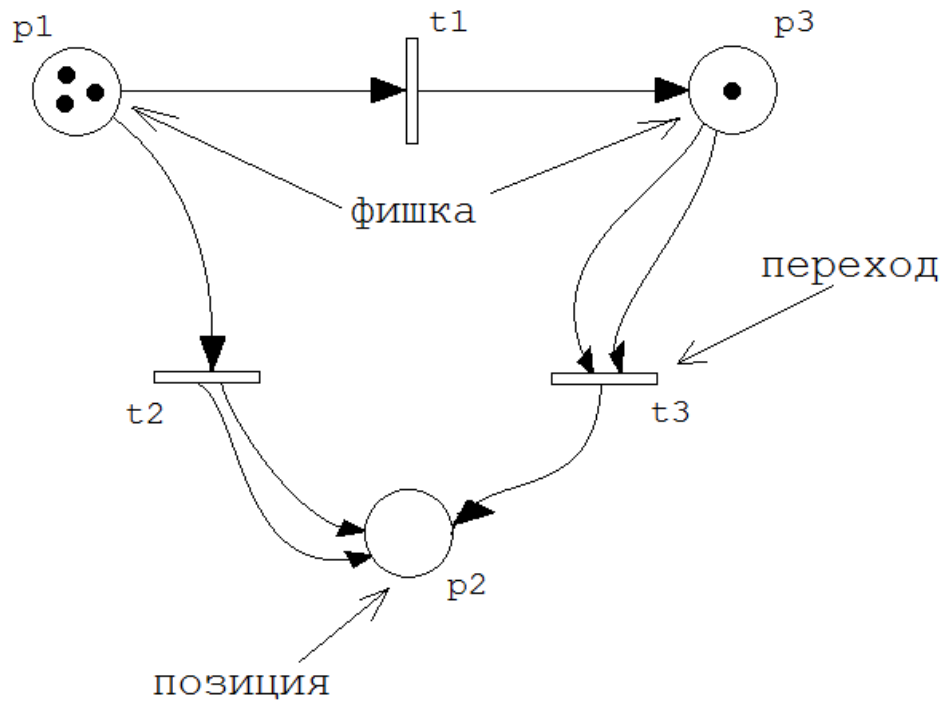

Рисунок 2.1 - Пример графа сети Петри

2.3.3 Выполнение сети Петри

Количество и распределение фишек в сети управляет выполнением сети Петри. Сеть Петри выполняется путем запуска переходов. Запуск перехода означает удаление фишек из входных позиций данного перехода и образование новых в выходных позициях.

Определение 2.4: Переход $t_j \in T$ в маркированной сети Петри $M = (P, T, I, O, \mu)$ разрешен, если для всех $p_i \in P$

$\mu(p_i) \geq \#(p_i, I(t_j))$.

Другими словами, переход разрешен, если каждая из его входных позиций содержит число фишек большее или равное числу дуг из позиций в переход. Переход может запускаться только в том случае, если он разрешен.

Определение 2.5: Разрешающие фишки – фишки во входной позиции, которые разрешают переход.

Определение 2.6: Переход t_j в маркированной сети Петри с маркировкой μ может быть запущен всякий раз, когда он разрешен. В результате запуска разрешенного перехода t_j образуется новая маркировка μ' ()

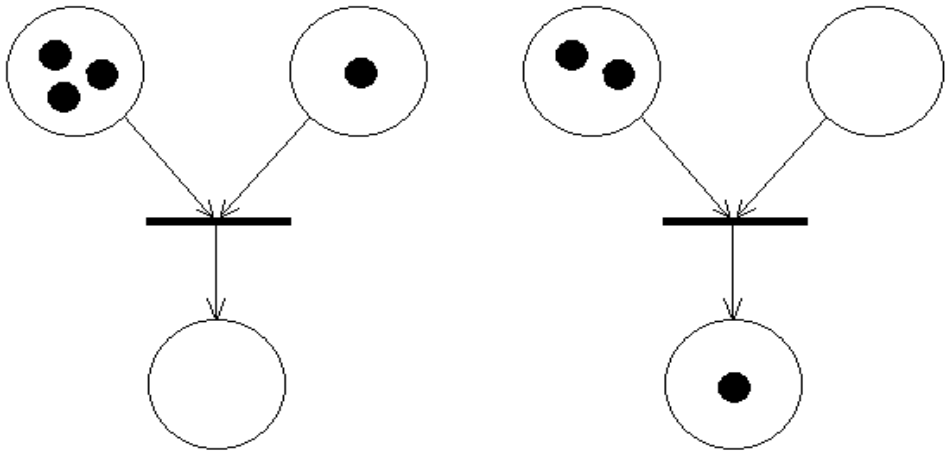

Рисунок 2.2 - Запуск разрешенного перехода

Так как можно запускать только разрешенные переходы(рис.2.2), при запуске перехода количество фишек в каждой позиции остается неотрицательным. Запуск перехода никогда не удалит фишку, отсутствующую во входной позиции. Если какая-либо входная позиция перехода не обладает достаточным количеством фишек, то такое переход не разрешен и, как следствие, не может быть запущен. Запуски могут осуществляться до тех пор, пока существует хотя бы один разрешающий переход. Когда все разрешающие переходы закончены, выполнение сети Петри прекращается [26].

2.3.4 Задача достижимости сети Петри

Задача достижимости - одна из самых важных задач анализа сетей Петри. Были поставлены следующие четыре задачи достижимости для сети Петри C = (P, T, I, O) с начальной маркировкой μ.

Определение 2.7: Задача достижимости. Выполняется ли для данной μ′: μ′∈ R(C, μ)?

Определение 2.8: Задача достижимости подмаркировки. Для подмножества P′⊆ P и маркировки μ′ существует ли μ′′∈ R(C, μ), такая, что μ′′(pi) = μ′(pi) для всех pi∈ P′?

Определение 2.9: Задача достижимости нуля. Выполняется ли μ′∈ R(C, μ), где μ′(pi) = 0 для всех pi∈ P ?

Определение 2.10: Задача достижимости нуля в одной позиции. Для данной позиции pi∈ P существует ли μ′∈ R(C, μ) с μ′(pi) = 0?

Задача достижимости подмаркировки ограничивает задачу достижения до рассмотрения только подмножества позиций, не принимая во внимания маркировки других позиций. Задача достижимости нуля выясняет, является ли достижимой частная маркировка с нулем фишек во всех позициях. Задача достижимости нуля в одной позиции выясняет, возможно ли удалить все фишки из одной позиции.

Все эти четыре задачи являются эквивалентными. Эти теоремы и доказательства описаны в работе Хэку [27]

«Поскольку задачи подмножества и равенства для множеств достижимости сетей Петри неразрешимы, то возможно, что неразрешима также и сама задача достижимости. Однако в настоящее время вопрос, разрешима ли (или неразрешима) задача достижимости, открыт. На сегодняшний день не существует ни алгоритма, решающего задачу достижимости, ни доказательства того, что такого алгоритма не может быть. [28]»

Для анализа свойств сетей Петри наиболее удобно использовать граф представления множества достижимости сетей Петри. В этом дереве представлены все достижимые состояния сетей Петри. Общий путь построения дерева достижимости сети Петри заключается в определении всех разрешенных переходов в соответствующей маркировке с последующим анализом соответствующего очередного состояния (маркировки), достигающихся при независимых автоматических последовательностях запусков всех разрешенных переходов предыдущей маркировки.

Решение задачи достижимости с использованием дерева достижимости фактически сводится к методу полного перебора. Поэтому защищенность здесь определяется только временем и ресурсами, затраченными на полный перебор. А это два наиболее важных критерия, определяющие степь защищенности.

Таким образом, сети Петри являются прекрасным инструментарием для защиты исходного кода.

2.3.5 Построение сети Петри

Сеть Петри - абстрактное понятие. Для решения поставленной задачи будем использовать модель, схожую по поведению с сетью Петри. Представим её в терминах, описанных выше.

В качестве меток сети выступают ячейки памяти. Разобьем метки на три типа:

- фиксированные метки;
- метки с вычисляемыми значениями;
- метки с автоматически сгенерированными значениями.

Первая группа - это те метки, в которых мы хотим получать искомые константы (целевые значения). Эти константы берутся из исходного кода, и именно вместо них подставляются инструкции, которые вычисляют сеть. Такие метки строго фиксированы, потому что необходимо уметь однозначно определять, в какой ячейке находится константное значение для той или иной операции исходного кода. Каждый раунд изменяет сеть: все метки могут быть пересчитаны.

Поэтому важно соблюдение порядка выполнения сети. Целевые значения в сети идут в том же порядке, что и в коде (т.е. чем позже команда в коде, тем больше номер раунда)

Ко второй группе относятся те метки, значение которых неизвестно в момент создания сети. Эти значения должны быть подобраны таким образом, чтобы после очередного раунда в заданной ячейке из первой группы получилось искомое значение.

Но если все метки сети считать неизвестными, то сложность построения такой сети возрастает. Поэтому часть неизвестных заполняется случайным образом сгенерированными значениями (третья группа).

В ходе исследования было получено, что при заполнении половины меток сети случайными значениями, всегда существует решение для данной сети, позволяющее построить её по указанным выше правилам.

Каждый узел (метка) имеет двух родителей, соответственно значение каждого узла вычисляется как сумма значений родителей. Исходя из этого, можно построить систему линейных диофантовых уравнений для расчета неизвестных узлов.

Определение 2.11: Диофантово уравнение — это уравнение вида

$$P(x_{1,\dots,} x_m) = 0$$

где P — целочисленная функция (например, полином с целыми коэффициентами), а переменные принимают целые значения.

Определение 2.12: Линейное диофантовое уравнение имеет вид:

$$a_1x_1 + a_2x_2 + \dots + a_kx_k = d$$

При разработке инструментального средства для защиты использовался метод решения системы линейных диофантовых уравнений в кольце вычетов через нормальную форму Эрмита –Смита в кольце вычетов согласно методу, преложенному Авдошиным, который также имеет вычислительную сложность $O(n^3)$[20].

Сложность решения линейного уравнения в целых числах в кольце вычетов полиномиальна и равняется $O(n^3)$ [20], значит и сложность построения такой сети Петри тоже равняется $O(n^3)$.

Получаем сеть, которую можно построить за полиномиальное время, а задача достижимости для неё является NP - полной. Такая сеть является хорошим инструментом для защиты исходного кода.

2.3.6 Оценка сложности взлома сети Петри

В том случае, если часть данных о процессе защиты известна, то задача восстановления кода становится полиномиальной. Например, если известен алгоритм защиты и структура, то при

анализе возможно построение сети Петри, а значит и удаление всех иструкиций сети за полиномиральное время.

В тоже время если ивестен генератор псвевдослучайных чисел(ГПСЧ) и его начальное значение, то можно выделить последовательность раундов, что тоже приводит к полиномиальному решению задачи.

Таким образом для эффективности защиты при помощи сети Петри, нгеобходимо обеспечить секретность:

1. Используемой сети Петри
2. Применяемого при защите ГПСЧ
3. Принципа масштабирования или генерации, в том случае, если сеть Петри получается при помощи самоподобия или каким-либо иным алгоритмом.

2.3.7 Защита исходного кода с помощью сети Петри

Для защиты программного продукта с помощью реализуемой системы защиты необходимы:

- исполняемый файл защищаемого продукта;
- виртуальный адрес защищаемой функции.

Обычно не весь исходный код несет в себе ценную информацию, а лишь какая-то его часть, например функция, реализующая алгоритм, являющийся интеллектуальной собственностью. Именно такую функцию необходимо защищать.

Имея такие начальные данные, защита строится по следующему алгоритму:

1. Из исходного исполняемого файла получается набор ассемблерных инструкций;
2. Ассемблерные инструкции преобразуются в набор инструкций LLVMIRBuilder;
3. В полученный код встраивается сеть Петри;
4. Обфусцированный код выполняется на виртуальной машине LLVM;
5. По данному виртуальному адресу в исполняемый файл встраивается новый исполняемый файл, сгенерированный виртуальной машиной LLVM, который и выполняет основную логику исходной программы.

Рассмотрим каждый из этих шагов защиты подробнее.

2.3.8 Low Level Virtual Machine и особенности ее применения

В качестве инструментария для реализации методов защиты ПО от реинжиниринга был выбран LLVM (Low Level Virtual Machine) — универсальная система анализа, трансформации и оптимизации программ, реализующая виртуальную машину с RISC-

подобными инструкциями. Может использоваться как оптимизирующий компилятор этого байткода в машинный код для различных архитектур. Система имеет модульную структуру и может расширяться дополнительными алгоритмами трансформации (compiler passes) и кодогенераторами для новых аппаратных платформ.

В основе LLVM лежит промежуточное представление кода (intermediate representation, IR), над которым можно производить трансформации во время компиляции, компоновки и выполнения. Из этого представления генерируется оптимизированный машинный код для целого ряда платформ, как статически, так и динамически (JIT-компиляция). LLVM поддерживает генерацию кода для x86, x86-64, ARM, PowerPC, SPARC, MIPS, IA-64, Alpha.

Поэтому методы защиты, реализованные с использованием LLVM, будут соблюдать одно из основных предъявленных требований - кроссплатформенность. И хотя изначально задача ставилась только для защиты исходного кода, написанного для x86, использование LLVM позволяет в теории решить более общую задачу, чем защиту только этой платформы.

LLVM IR можно охарактеризовать как типизированный трёхадресный код в SSA-форме. В конструировании компиляторов SSA-представление (Static single assignment form) - это промежуточное представление, в котором каждой переменной значение присваивается лишь единожды. Переменные исходной программы разбиваются на версии, обычно с помощью добавления суффикса, таким образом, что каждое присваивание осуществляется уникальной версии переменной.

Код в SSA-форме удобно рассматривать не как линейную последовательность инструкций, а как граф потока управления (control flow graph, CFG). Вершины этого графа — так называемые базовые блоки (basic blocks), содержащие последовательность инструкций и обозначающиеся метками, заканчивающуюся инструкцией-терминатором, явно передающей управление в другой блок. Терминаторами являются следующие инструкции:

- ret тип значение — возврат значения из функции;
- br i1 условие, label метка_1, label метка_2 — условный переход;
- br label метка - безусловный переход;
- switch — обобщение br, позволяет организовать таблицу переходов;
- invoke и unwind — используются для организации исключений;
- unreachable — специальная инструкция, показывающая компилятору, что выполнение никогда не достигнет этой точки.

Чтобы сделать переменные изменяемыми, в SSA-форме к прочим инструкциям добавляется специальная функция φ, которая возвращает одно из перечисленных значений в зависимости от того, какой блок передал управление текущему.

В LLVM этой функции соответствует инструкция phi, которая имеет следующую форму:

```
phi тип, [значение_1, label метка_1], ..., [значение_N, label метка_N]
```

В качестве примера рассмотрим функцию вычисления факториала, которую на Си можно было бы записать так:

```
int factorial(int n)
{
    int result = n;
    int i;
    for (i = n - 1; i > 1; --i)
        result *= i;
    return result;
}
```

Примечание: блок, который начинается с входа в функцию, обозначается %0.

```
define i32 @factorial(i32 %n)
{
    %i.start = sub i32 %n, 1
    br label %LoopEntry
LoopEntry:
    %result = phi i32 [%n, %0], [%result.next, %LoopBody]
    %i = phi i32 [%i.start, %0], [%i.next, %LoopBody]
    %cond = icmp sle i32 %i, 1
    br i1 %cond, label %Exit, label %LoopBody
LoopBody:
    %result.next = mul i32 %result, %i
    %i.next = sub i32 %i, 1
    br label %LoopEntry
Exit:
    ret i32 %result
}
```

LLVM также требует, чтобы все phi-инструкции шли в начале блока и до них не было никаких других инструкций. Хотя SSA-форма позволяет производить много полезных трансформаций, непосредственно генерировать её из кода на императивном языке затруднительно, хотя есть хорошо известные алгоритмы преобразования в SSA. При написании компилятора на основе LLVM нет никакой необходимости заниматься этим, потому что система умеет генерировать SSA самостоятельно.

SSA представление, в основе которого лежит разбиение кода на BasicBlock, очень удобно для реализации обфускации на базе сетей Петри. Так как внутри BasicBlock нет переходов, то выполнение сети Петри можно контролировать. То есть все инструкции, выполняющие суммирование сети, гарантированно будет выполнены, и кроме того всегда будет соблюден порядок их выполнения. При наличии терминаторов это немаловажное условие могло быть нарушено, что привело бы к некорректной работе сети: целевые значения не были бы получены в фиксированных метках. [29]

2.3.9 LLVM Pass Manager

LLVM Pass фреймворк является важной частью всей системы LLVM. Pass Manager выполняет преобразования и оптимизации, которые и составляют компилятор. Он также выполняет анализ результата, который используется для преобразований.

Все LLVM оптимизации являются наследниками класса Pass, содержащий набор виртуальных методов, которые реализуют наследники. В зависимости от того, какая задача решается с помощью LLVM оптимизаций, необходимо наследоваться от одного из следующих классов: ModulePass, CallGraphSCCPass, FunctionPass, LoopPass, RegionPass, или BasicBlockPass класс. Они дают системе больше информации о том, что конкретно делает реализуемая LLVM оптимизация и как она может сочетаться с другими оптимизациями.

При разработке нового класса оптимизации необходимо четко определить, от какого из возможных LLVM Pass классов надо наследоваться. При выборе надо руководствоваться правилом, что лучше выбирать наиболее специфичный класс, это дает LLVM больше необходимой информации, что позволяет компилятору работать быстрее. Ниже приведены классы, которые использовались в данной работе. Подробную информацию о всевозможных классах оптимизации можно получить в [31].

2.3.10 Генерация сети Петри

Класс Petri, который генерирует сеть Петри для защиты кода, проектируется как наследник класса FunctionPass. То есть в данном случае оптимизация применяется к каждой функции. В классе реализован метод runOnFunction, который в качестве параметра принимает обрабатываемую функцию.

Из этой функции выбирается самый большой BasicBlock. Это необходимо для того, чтобы присутствие в коде сети не было заметно. То есть чем больше блок, в который будет встраиваться сеть, тем сложнее злоумышленнику выявить сам факт существования такой сети.

В максимальном BasicBlock выбираются инструкции, чьи операнды являются константами - они и будут вычисляться с помощью сети. По описанным выше правилам строиться сеть Петри.

Так как целевые значения в сети фигурируют в том же порядке, что и в коде(т.е. чем позже команда в коде, тем больше номер раунда), то когда мы подходим к очередной такой команде, все нужные вычисления уже проведены (до раунда, который соответствует текущей команде). Инструкции, которые осуществляют вычисления (суммирование) сети, вставляются между этой

командой и предыдущей командой, в которой операнд был заменён на вычисляющую сеть инструкцию.

Вставка инструкций, которые производят вычисления, осуществляется равномерно, при равном количестве инструкций, они вставляются через одну.

В результате описанных действий исходный код уже не содержит определенные константы. Вместо них находятся инструкции, вычисляющие сеть. Численные значения, которые используются при построении сети, случайно генерируются. Это не дает злоумышленнику возможности воспроизвести работу сети и определить, какие константные данные были сокрыты таким способом.

2.3.11 Встраивание защищенного кода в объектный файл

На данном этапе имеется код, состоящий из инструкций LLVMIRBuilder, содержащий вместо констант инструкции, выполняющие сеть Петри.

Этот код исполняется на виртуальной машине LLVM. В результате получается новый исполняемый файл, который содержит ту же логику, что и исходная функция. Но в отличии он неё, этот файл является защищенным.

Последней итерацией необходимо встроить защищенный код в исходный исполняемый файл. Исходный код удаляется, делается это путем его обнуления. Далее создаются две дополнительные секции: данных и кода. В них и записывается новый код.

Эти секции автоматически создаются LLVM, необходимо только перенести их из модуля, который был сгенерирован LLVM в защищаемый объектный файл. Помимо этого создается ещё одна секция, которая исполняет роль переходника. В неё записывается имя защищаемой функции и именно на эту секцию происходит переход при вызове функции.

Полученный таким образом объектный файл выполняет те же действия, что и исходный, но при этом является защищенным.

Очевидно, что для результирующего файла можно применить защиту еще раз, так как полученный объектный файл, соотвествует требованиям исходной задачи.

2.3.12 Метод построения кодогенератора виртуальной машины

Кодогенератор преобразует код в формате промежуточного представления LLVM в байткод виртуальной машины. Для каждой виртуальной машины, сгенерированной по методике

из раздела 3, кодогенератор создаётся автоматически. Автоматическое создание кодогенератора проводится в три этапа (рРисунок 2.3).

На первом этапе происходит преобразование описания виртуальной машины из внутреннего формата приложения в формат LLVM. Результатом этого этапа являются файлы типа LLVM Target Description (Td файлы) и файлы с исходным кодом на языке C++ (C++ файлы).

На втором этапе осуществляется генерация файлов с исходным кодом на языке C++ (Inc файлов) из Td файлов с помощью утилиты llvm-tblgen, входящей в состав библиотек LLVM.

На третьем этапе происходит сборка кодогенератора с помощью компилятора GCC 4.6.2. При сборке используются C++ файлы, полученные на первом этапе, Inc файлы, полученные на втором этапе, и каркас кодогенератора, написанный на языке C++ и одинаковый для всех виртуальных машин.

Далее приведены подробности автоматического создания кодогенератора со структурой, представленной на рисунке 2.3.

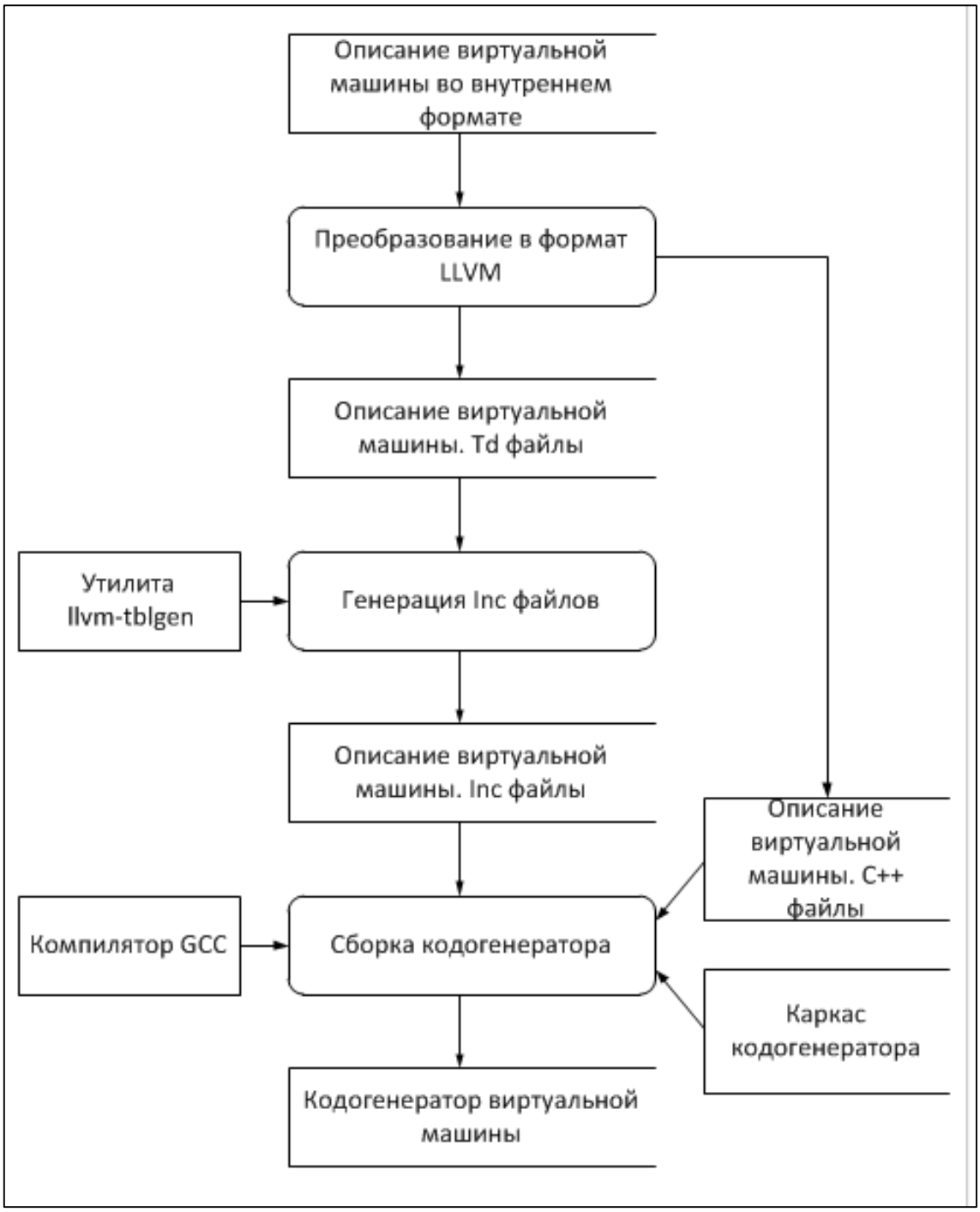

Рисунок 2.3 - Автоматическое создание кодогенератора

2.3.13 Быстрое создание кодогенератора

В текущей версии прототипа автоматическое создание кодогенератора занимает около одной минуты, причём 95% этого времени приходится на сборку кодогенератора с помощью компилятора GCC. Далее в разделе показано, каким образом уменьшить время автоматического создания генератора до нескольких секунд.

Во-первых, необходимо собирать кодогенератор как динамически подключаемую библиотеку (.dll) под ОС Windows или как разделяемый объект (shared object .so) под ОС Linux. Сейчас кодогенератор представляет собой отдельную программу и подавляющую часть времени сборки исполняемого файла этой программы с помощью компилятора GCC составляет компоновка с библиотеками LLVM. В случае использования динамически подключаемой библиотеки (разделяемого объекта) код библиотек LLVM не требуется включать в собираемый .dll (.so) файл, поскольку они уже присутствуют в адресном пространстве приложения (библиотеки LLVM нужны на более ранних этапах защиты, например, для преобразования машинного кода в промежуточное представление LLVM).

Во-вторых, следует уменьшить количество файлов, содержащих C++-код кодогенератора. Каждый C++-файл является единицей трансляции, а уменьшение единиц трансляции значительно ускоряет сборку. Число файлов должно быть равно один плюс число файлов, сгенерированных утилитой llvm-tbgen.

В-третьих, временные файлы нужно держать в памяти, а не сохранять на диск. В текущей версии прототипа все временные файлы сохраняются на диск. Поскольку доступ к информации на диске на три порядка медленнее, чем доступ к данным в памяти, избежание такого сохранения позволит значительно ускорить автоматическое создание кодогенератора.

В-четвёртых, утилита llvm-tbgen должна быть интегрирована в приложение. В текущей версии прототипа утилита llvm-tbgen представляет собой отдельную программу. Интеграция утилиты (возможная, благодаря открытому исходному коду LLVM) llvm-tbgen позволит избежать лишней траты времени на создание нового системного процесса.

В-пятых, следует заменить компилятор GCC на компилятор LLVM Clang. Для сборки кодогенератора предпочтительнее применять компилятор LLVM Clang, поскольку он уже используется приложением для сборки интерпретатора. В текущей версии прототипа применяется компилятор GCC, т. к. в C++-коде кодогенератора используется стандартная библиотека C++, которая присутствует в пакете программ GCC и отсутствует в Clang. Таким образом, для использования Clang необходимо исключить вызовы функций стандартной библиотеки C++ из исходного кода кодогенератора.

После осуществления приведённых в этом разделе мер по ускорению автоматического создания кодогенератора из процесса создания будут исключены копирование кода библиотек LLVM, открытие/закрытие/чтение/запись большого количества файлов и вызов новых системных процессов. Время создания кодогенератора будет определяться временем компиляции нескольких файлов компилятором Clang, временем генерации C++-кода утилитой llvm-tbgen и временем генерации описания виртуальной машины в формате LLVM. Практика показывает, что все эти этапы в сумме занимают менее 3 секунд.

3 СРЕДСТВА ЗАЩИТЫ ИСПОЛНЯЕМОГО КОДА ОТ ДИНАМИЧЕСКОГО АНАЛИЗА С ИСПОЛЬЗОВАНИЕМ ПЛАТФОРМЕННО ЗАВИСИМЫХ ПОДХОДОВ

3.1 Средства противодействия статическому анализу исполняемого кода

Защита x86 ассемблерного кода осуществляется при генерации кода целевой платформы (x86) из кода в формате промежуточного представления LLVM.

Для генерации x86 кода используется кодогенератор, входящий в состав LLVM. Процесс кодогенерации представляет собой несколько проходов по графу потока управления (control flow graph, CFG) программы с заменой команд промежуточного представления на команды x86. Под проходом понимается выполнение основной процедуры экземпляра класса, являющегося наследником класса LLVM Pass. Запуском проходов управляет LLVM Pass Manager.

Для осуществления защиты x86 ассемблерного кода кодогенератор LLVM был дополнен тремя проходами. Первый проход вставляет мусорные инструкции, второй проход вставляет инструкции для метаморфирования кода и третий проход вносит в код полиморфизм. Далее в этой главе представлено подробное описание всех трёх проходов.

3.1.1 Вставка мусорного x86 кода

При вставке мусорного x86 кода происходит запутывание кода, а именно, преобразование исходного текста или исполняемого кода программы в вид, сохраняющий ее функциональность, но затрудняющий анализ, понимание алгоритмов работы и модификацию при декомпиляции. Таким образом, данные инструкции «мусорного» кода позволяют решить следующие задачи:

- Вывод из строя автоматов-детекторов.
- Борьба со статистическими методами обнаружения: мусор состоит из последовательностей команд, которые встречаются в “легальных” программах.
- Увеличивается сложность изучения кода при трассировке файла.
- Увеличение элемента случайности в расшифровщике. С «мусором» появляются новые варианты компоновки кода. При этом вставка таких команд имеет так же огромное влияние на размер кода.

Рассмотрим группы «мусорных» инструкций x86, используемые в нашем протекторе.

1) *Зеркальные команды*. Последовательное использование данных команд аннулирует изменения, вносимые ими.

- *Сложение/вычитание*

```
add reg_n, α
sub reg_n, α
```

Первая команда увеличивает значение регистра reg_n на α, а затем вторая команда уменьшает его на α.

Таким образом, значение регистра не изменяется. Данные команды используются так же и в обратном порядке – сначала вычитание, затем сложение.

Примеры:

а)
```
add eax, 3
sub eax, 3
```

б)
```
sub cx, 8
add cx, 8
```

Аналогичные зеркальные команды создаются для ячейки памяти:

```
add mem_n, α
sub mem_n, α
```

Первая команда увеличивает значение в ячейке памяти mem_n на α, а затем вторая команда уменьшает его на α.

Примеры:

а)
```
add BYTE PTR [eax], 10
sub BYTE PTR [eax], 10
```

б)
```
sub DWORD PTR [ebx], 15
add DWORD PTR [ebx], 15
```

При этом для передачи значения, на которое временно изменяется значение регистра или ячейки памяти, используются не только непосредственные значения, но и регистры и ячейки памяти.

Примеры:

а)
```
add BYTE PTR [eax], ecx
sub BYTE PTR [eax], ecx
```

б)
```
sub eax, DWORD PTR [ebx]
add eax, DWORD PTR [ebx]
```

– *Инкрементирование/декрементирование*

```
inc reg_n
dec reg_n
```

Первая команда увеличивает значение регистра reg_n на 1, а затем вторая команда уменьшает его на 1. Таким образом, значение регистра не изменяется. Данные команды так же используются в обратном порядке – сначала декрементирование, затем инкрементирование.

Примеры:

а)
```
inc eax
```

```
dec eax
```

б)
```
dec cx
inc cx
```

Аналогичные зеркальные команды создаются для ячейки памяти:

```
inc mem_n
dec mem_n
```

Первая команда увеличивает значение в ячейке памяти mem_n на 1, а затем вторая команда уменьшает его на 1.

Примеры:

а)
```
inc BYTE PTR [eax]
dec BYTE PTR [eax]
```

б)
```
dec DWORD PTR [ebx]
inc DWORD PTR [ebx]
```

– *Вставка/извлечение значений из стека*

```
push reg_n
pop reg_n
```

Первая команда вставляет в стек значение из регистра reg_n, а затем вторая команда извлекает его из стека. Таким образом, содержимое стека не изменяется. В отличие от команд сложения/вычитания и инкрементирования/декрементирования эту пару команд можно использовать только в указанной последовательности.

Пример:

```
push eax
pop eax
```

Аналогичные зеркальные команды создаются для ячеек памяти:

```
push mem_n
pop mem_n
```

Первая команда вставляет в стек значение из ячейки памяти mem_n, а затем вторая команда извлекает его из стека.

Пример:

```
push BYTE PTR [eax]
pop BYTE PTR [eax]
```

– *Циклический сдвиг влево/ циклический сдвиг вправо*

```
rol reg_n, α
reg reg_n, α
```

Первая команда осуществляет циклический сдвиг влево для значения в регистре reg_n на α бит, а затем вторая команда осуществляет обратный циклический сдвиг вправо на α бит. Таким образом, значение регистра не изменяется. Данные команды используются и в обратном порядке – сначала циклический сдвиг вправо, затем циклический сдвиг влево.

Примеры:

а)
```
rol ecx, 5
ror ecx, 5
```

б)
```
ror ax, 4
rol ax, 4
```

Аналогичные зеркальные команды создаются для ячейки памяти:

```
rol mem_n, α
ror mem_n, α
```

Первая команда осуществляет циклический сдвиг влево для значения в ячейке памяти mem_n на α бит, а затем вторая команда осуществляет для него обратный циклический сдвиг вправо на α бит.

Примеры:

а)
```
rol BYTE PTR [ax], 2
ror BYTE PTR [ax], 2
```

б)
```
ror DWORD PTR [ebx], 7
rol DWORD PTR [ebx], 7
```

При этом для передачи числа бит, на которое осуществляется циклический сдвиг, используются не только непосредственные значения, но и регистры и ячейки памяти.

Примеры:

а)
```
rol BYTE PTR [eax], ecx
ror BYTE PTR [eax], ecx
```

б)
```
ror eax, DWORD PTR [ebx]
rol eax, DWORD PTR [ebx]
```

Кроме пары команд `rol/ror` циклический сдвиг так же может осуществляться с помощью команд `rcl/rcr`. Отличие между командами `rol` и `rcl` в том, что `rol` при каждом сдвиге выдвигаемый слева бит операнда помещает в него же справа, в то время как команда `rcl` выдвигаемый бит прежде, чем поместить с другой стороны, помещает в CF и помещает обратно в операнд лишь на следующем шаге цикла. Для команд rcr и ror аналогично.

Для пары зеркальных команд `rcl/rcr` осуществляются вставки инструкций, аналогичные вставке инструкций для команд `rol/ror`.

Примеры:

а)
```
rcl edx, 1
rcr edx, 1
```
б)
```
rcr bx, 4
rcl bx, 4
```
в)
```
rcl BYTE PTR [bx], 3
rcr BYTE PTR [bx], 3
```
г)
```
rcr DWORD PTR [eax], 5
rcl DWORD PTR [eax], 5
```
д)
```
rcl BYTE PTR [ebx], ecx
rcr BYTE PTR [ebx], ecx
```
е)
```
rcr eax, DWORD PTR [ebx]
rcl eax, DWORD PTR [ebx]
```

2) *Группа инструкций, которые не вносят изменений*. Данные инструкции просто изменяют сигнатуру кода, но реально не вносят даже временных изменений, как это делают зеркальные команды.

– *Операция логического умножения, где в обоих операндах одно и то же значение.*

```
and reg_n, reg_n
```

Из теории булевой алгебры, очевидно, что в результате выполнения данной инструкции значение регистра reg_n не изменится.

Примеры:

а) `and ax, ax`

б) `and ebx, ebx`

В качестве операнда так же выступают ячейки памяти:

```
and mem_n, mem_n
```

Примеры:

а) `and DWORD PTR [ebx], DWORD PTR [ebx]`

б) `and BYTE PTR [ebx], BYTE PTR [ebx]`

– *Операция логического сложения, где в обоих операндах одно и то же значение.*

```
or reg_n, reg_n
```

В результате выполнения данной инструкции значение регистра reg_n не изменится.

Примеры:

а) `or bx, bx`

б) `or ecx, ecx`

В качестве операнда может так же выступать ячейка памяти:

```
or mem_n, mem_n
```

Примеры:

а) `or DWORD PTR [ecx], DWORD PTR [ecx]`

б) `or BYTE PTR [bx], BYTE PTR [bx]`

- *Операция пересылки данных из операнда в этот же операнд.*

```
mov reg_n, reg_n
```

При выполнении данной операции в регистр reg_n помещается значение из этого же регистра reg_n, и, таким образом, его значение не изменяется.

Примеры:

а) `mov ax, ax`

б) `mov ebx, ebx`

В качестве операнда так же выступают ячейки памяти:

```
and mem_n, mem_n
```

Примеры:

а) `mov DWORD PTR [eax], DWORD PTR [eax]`

б) `mov BYTE PTR [cx], BYTE PTR [cx]`

Кроме инструкций mov в данную группу «мусорных» команд относятся инструкции обмена значений между операндами xchg, где в обоих операндах одно и то же значение. Формирование данных «мусорных» команд аналогично формированию «мусорных» команд с инструкцией mov.

Примеры:

а) `xchg cx, cx`

б) `xchg eax, eax`

в) `xchg DWORD PTR [ebx], DWORD PTR [ebx]`

г) `xchg BYTE PTR [ax], BYTE PTR [ax]`

– *nop («no operation» - «нет операции»)*

Данная команда изменяет сигнатуру кода, но сама не вносит никаких изменений, что и следует из ее перевода.

– *вставка команды daa*

Данная команда используется для двоично-десятичной арифметики. В настоящее время данная арифметика используется крайне редко для специфических задач, поэтому вставляя команды daa, меняем сигнатуру кода, но не влияем на функциональность.

3) *Группа инструкций, выполнение которых дважды, аннулирует изменения*

Однократное выполнение инструкций из данной группы может внести изменение в функционирование программы. При этом выполнение данных инструкций четное число раз аннулирует вносимые изменения.

– Двойное выполнение команды сложения по модулю 2 с одними и теми же операндами не изменяет значения этих операндов.

```
xor reg1, reg2
xor reg1, reg2
```

Из теории булевой алгебры знаем, что первое выполнение данной инструкции изменяет значение регистра reg1, при этом повторное ее выполнение восстанавливает значение этого регистра reg1.

Примеры:

а)
```
xor eax, ebx
xor eax, ebx
```

б)
```
xor ax, dx
xor ax, dx
```

В качестве операндов так же выступают ячейки памяти и комбинации регистров и ячеек памяти.

Примеры:

а)
```
xor DWORD PTR [eax], DWORD PTR [edx]
xor DWORD PTR [eax], DWORD PTR [edx]
```

б)
```
xor BYTE PTR [cx], ax
xor BYTE PTR [cx], ax
```

в)
```
xor eax, DWORD PTR [ebx]
xor eax, DWORD PTR [ebx]
```

– Двойное логическое отрицание, примененное к одному и тому же операнду, не изменяет его.

```
not reg_n
not reg_n
```

Первая команда заменяет биты значения в reg_n на противоположные, вторая команда возвращает их значения к исходным.

Примеры:

а)
```
not eax
not eax
```

б)
```
not bx
not bx
```

В качестве операндов так же выступают ячейки памяти:

```
not mem_n
not mem_n
```

Примеры:

а)
```
not [ax]
not [ax]
```

б)
```
not [ebx]
not [ebx]
```

– Двойное отрицание, примененное к одному и тому же операнду, сохраняет его знак.

```
not reg_n
not reg_n
```

Однократное выполнение данной команды изменяет знак операнда на противоположный; двойное выполнение сохраняет исходный знак значения.

Примеры:

а)
```
neg cx
neg cx
```

б)
```
neg ebx
neg ebx
```

В качестве операндов так же выступают ячейки памяти:

```
neg mem_n
neg mem_n
```

Примеры:

а)
```
neg [ax]
neg [ax]
```

б)
```
neg [edx]
neg [edx]
```

4) *Группа инструкций, выполняющие ложные переходы*

– *Переход на метку, находящуюся сразу после перехода*

Данный переход происходит вне зависимости от результата сравнения.

Приведем примеры шаблонов для данного типа мусорных команд, используемых в протекторе:

а)
```
cmp reg_n, reg_m
jz label
label:
```

б)
```
cmp reg_n, mem_m
```

```
    jnz label
    label:
```

в)
```
    cmp mem_n, mem_m
    jle label
    label:
```

г)
```
    cmp mem_n, reg_m
    jbe label
    label:
```

Так же используются всевозможные сочетания типов операндов в команде cmp и различные операции условных переходов.

– *Вставка блоков осмысленных команд, на которые никогда не выполняется переход*

Данные «мусорные» команды сложнее детектировать с помощью сигнатур, так как в командах, относящихся к невыполняемому переходу, содержится действующий, влияющий на функциональность код.

Приведем примеры шаблонов для данного типа мусорных команд, используемых в протекторе:

а)
```
    cmp reg_n, reg_n
    jne label
```

б)
```
    cmp mem_n, mem_n
    jne label
```

в)
```
    jmp label2
    push reg1
    label2:
```

Значащая команда push reg1 всегда будет пропущена после перехода на метку label2.

г)
```
    jmp label2
    mov reg1, reg2
    inc reg2
    label2:
```

Значащие команды mov reg1, reg2 и inc reg2 всегда будут пропущены после перехода на метку label2.

– *Вызов ложных функций*

В данном типе ложных команд вызывается функция, содержащая «мусорные» инструкции, и, таким образом, выполнение данной функции не влияет на функциональность, но изменяет сигнатуру кода.

Примеры:

а)
```
label:
push reg_n
pop reg_n
ret
call label
```

б)
```
label:
inc mem_m
dec mem_m
ret
call label
```

в)
```
label:
add reg1, α
sub reg1, α
ret
call label
```

3.1.2 Вставка инструкций для метаморфирования кода

Метаморфирование кода представляет собой изменение вида кода при сохранении оригинального алгоритма его работы. Метаморфизм - технология позволяющая изменять полный код протектора, каждый раз при создании копии.

Рассмотрим общую схему метаморфа, представленную в статье [34] (см. рисунок 3.1), где части кода метаморфа выделены серым, а белым выделен исходный код.

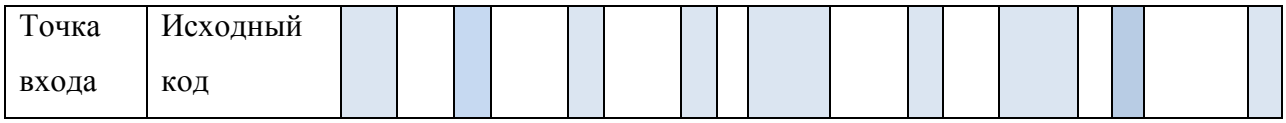

Рисунок 3.1 - Общая схема метаморфа

В этом случае точка входа может остаться связанной с исходным защищаемым кодом. Метаморф перехватывает управление во время выполнения исходного кода. Эта техника является примером метода обфускации точки входа. Это означает, что злоумышленник не может

концентрироваться только на поиске точки входа исполняемого файла, чтобы получить сигнатуру генератора, а уже с ее помощью получить исходный код.

В общем случае метаморф имеет метаморфный генератор, встроенный в него. В течение выполнения защищаемого кола, метаморф создает измененную копию самого себя, используя этот генератор. Обычный метаморфный генератор состоит из следующих функциональных частей, показанных на рисунке 3.2 [34].

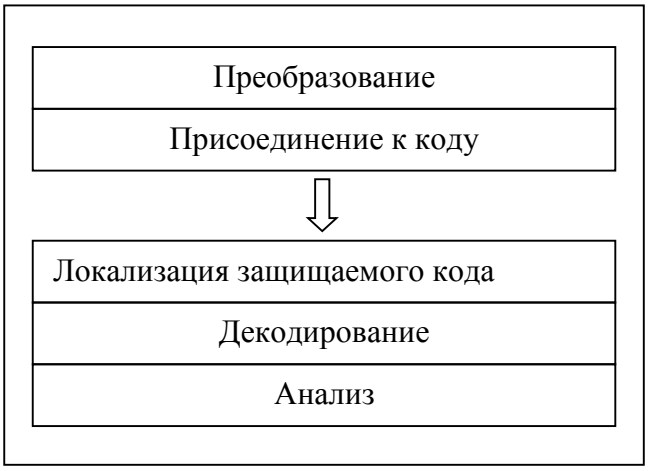

Рисунок 3.2 - Функциональные части метаморфного генератора

Метаморфные преобразования включают следующие методы запутывания кода:

- переименование регистров,
- изменение порядка инструкций,
- подмена инструкций,
- встраивание подпрограмм,
- перенос подпрограмм,
- перестановка кода,
- вставка мусора.

Рассмотрим примеры морфирования, используемые в нашем протекторе.

1) *Переименование регистров* – это изменение имен переменных или регистров, используемых в метаморфе. Когда регистры изменяются, они имеют различные коды операций, что усложняет поиск сигнатуры программы. Рассмотрим примеры переименования регистров, приведенные в табл. 3.1.

Таблица 3.1

Примеры переименования регистров для метаморфирования кода

Исходный код	Преобразованный код
mov ebp, esp sub esp, 10h	mov ebp, esi sub esi, 10h

mov eax, edx
mov ecx, [ebp+0002h]
shl eax, 001Ch
mov edx, [ebp+0003h]
shl edx, cl
add eax, edx
	mov ecx, ebx
	mov eax, [ebp+0002h]
	shl ecx, 001Ch
	mov ebx, [ebp+0003h]
	shl ebx, cl
	add ecx, ebx

pop eax
mov esi,ebp
sub eax,000Ch
add edx,0088h
mov edi,0004h
mul ebx,[edx]
mov [esi+eax*4],ebx
	pop edx
	mov eax,ebp
	sub edx,000Ch
	add esi,0088h
	mov ebx,0004h
	mul eax,[esi]
	mov [eax+edx*4],ecx

2) *Изменение порядка инструкций кода с использованием Jump*

Изменение порядка инструкций заключается в перестановке команд, но при этом оригинальный защищаемый код выполняется в логическом порядке с помощью команды языка ассемблера Jump.

Рассмотрим пример применения этого метода, приведенный в табл. 3.2. Через junk обозначаются «мусорные» инструкции, используемые для изменения сигнатуры преобразования. Множество вставляемых «мусорных» инструкций описано в предыдущем пункте.

Таблица 3.2

Пример изменения порядка инструкций с помощью команды Jump

Исходный код	Преобразованный код_1	Преобразованный код_2
jmp Start Start: Instruction1 Instruction2 Instruction3 Instruction4 Instruction5 jmp End End:	jmp Start Instr4: Instruction4 Jmp Instr5 junk Instr2: Instruction2 Instruction3 Instr5: Instruction5	jmp Start Instr3: Instruction3 jmp Instr4 junk Start: Instruction1 Instruction2 jmp Instr3 junk

	Jmp End junk Start: Instruction1 jmp instr2 End:	Instr4: Instruction4 Instruction5 jmp End End:

3) *Перестановка подпрограмм*

Это простой метод запутывания кода, когда подпрограммы переопределяются. Это не повлияет на работу программы, так как порядок описания подпрограмм не важен. Таким образом, метаморф, содержащий n подпрограмм может организовать n! перестановок (см. таблицу 3.2)

4) *Подмена инструкций*

Многие инструкции в ассемблере являются взаимозаменяемыми. Так команды сложения/ вычитания можно выразить через команды инкрементирования/декрементирования и наоборот; команды пересылки данных через команды вставки/ извлечения из стека и т.д.

Используя различные операции для выполнения одних и тех же действий, можно производить морфирование кода.

Рассмотрим несколько примеров замены инструкций, используемых в протекторе:

– Варианты записи инструкции `mov [mem addr], 7`

а)
```
push 7
pop [mem addr]
```

б)
```
mov eax, mem addr
mov [eax], 7
```

в)
```
mov edi, mem addr
mov eax, 7
stosd
```

г)
```
push mem addr
pop edi
push 7
pop eax
stosd
```

– Варианты записи инструкции `sub edx, 3`

а)
```
mov eax, 3
sub edx, eax
```

б)
```
dec edx
dec edx
```

```
        dec edx
в)      move eax, 3
        neg eax
        add eda, eax
```

- Варианты записи инструкции `mov ax, bx`

```
а)      push bx
        pop ax
```

- ```
  б)    xchg ax,bx
  ```
- ```
  в)    mov cx,bx
        mov ax,cx
  ```

Рассмотрим примеры метаморфизма, включающие несколько техник.

а)

```
G = ( N, T, S, R )
    N = { A, B, C}
    T = { a, b, c, d , e , f}
    a = "mov reg, 0 "
    b = "pop reg"
    c = "push 0 "
    d = "mov reg2 , 0"+ "push reg 2"
    e = "xor reg, reg "
    f = "sub reg, reg "
        R =  {
        S :: = A | Bb | C | D
        A :: = a
        B :: = c | d
        C :: = e
        D :: = f
        }
```

Вывод:

```
S => Bb => db
```

В итоге мы получили следующую последовательность команд:

```
mov reg 2, 0
push reg 2
```

```
pop reg
```

Этот пример иллюстрирует два метода: переименование регистров и использование мусорных команд.

б) В данном примере демонстрируется генерация различных вариантов следующего расшифровщика:

```
1. mov R1, val1
2. mov R2, val2
3. xor [R2], key1
4. add R2, 4
5. sub R1, 4
6. jnz 3
```

Алгоритм для создания соответствующего полиморфного генератора разбит на две части:

1) Инициализация переменных алгоритма;

2) Шифрование и организация цикла.

Им соответствуют два начальных правила грамматики:

```
S ::= XB
B ::= Y
```

Для команды xor существует следующая эквивалентная запись

x_1 xor x_2 = not(not x_1 xor x_2) =

= (x_1 and not x_2) or (not x_1 and x_2)

Используя данное утверждение, определим X и Y.

– Инициализация переменных:

X ::= X_1X_2 | X_2X_1

X_1 ::= JX_1 | mov R_1,val1 | push val1 + pop R_1 | xor R_1,R_1 + lea R_1,[R_1 + val1] | sub R_1,R_1 + add R_1,val1

X_2 ::= JX_2 | mov R_2,val2 | push val2 + pop R_2 | xor R_2,R_2 + lea R_2,[R_2 + val2] | sub R_2,R_2 + add R_2,val2

JX – это «мусор»,

В этой части продемонстрировано, что блоки можно переставлять, а так же возможность запутывания кода с помощью мусора.

– Шифрование:

Y -> JY | W

W -> JW | xor [R_2], key H_1

W -> not [R_2] + xor [R_2],key + not [R_2] H_1

W -> mov R_3,[R_2] + not R_3 + and R_3,key + and [R_2],not key + or [R_2],R_3 H_1

H_1 -> JH_1 | add R_2,4 H_2 | sub R_2,-4 H_2 S_4 -> GS_4 | sub R_2,4 | add R_2,-4

H_2 -> JH_2 | sub R_1,4 + jnz xxx | sub R_1,4 + jz yyy + jmp xxx

H_2 -> add R_1,-4 + jnz xxx | add R_1,-4 + jz yyy + jmp xxx

H_2 -> sub ecx,3 + loop xxx <=> R_1 == ecx

Правила расшифровывания W1 реализуют три эквивалентных формулы для операции XOR. Правила H1 и H2 описывают изменение счетчиков и условный переход.

в) В данном примере демонстрируется генерация различных вариантов следующего расшифровщика:

```
1. mov R1, val1
2. mov [R2], val2
3. add [R1], key1
4. sub [R2], key2
5. add R2, 3
6. sub R1, 7
7. jnz 3
```

Алгоритм для создания соответствующего полиморфного генератора так же разбит на две части: инициализация переменных и шифрование.

Им соответствуют два начальных правила грамматики:

```
S ::= XB
B ::= Y
```

– Инициализация переменных:

X ::= X_1X_2 | X_2X_1

X_1 ::= JX_1 | mov R_1,val1 | push val1 + pop R_1 | mov R_1,0 + add R_1,val1

X_2 ::= JX_2 | mov R_2,val2 | push val2 + pop R_2 | mov R_2,0 + add R_2,val2

– Шифрование:

Y -> JY | $W_1W_2H_1$ |$W_2W_1H_1$

W_1 -> JW_1 | add [R_1], key1 | neg key1 + sub [R_1], key1

W_2 -> JW_2 | sub [R_2], key2 | neg key2 + add [R_2], key2

H_1 -> JH_1 | add R_2,3 H_2 | sub R_2,-3 H_2

H_2 -> JH_2 | sub R_1,7 + jnz xxx | sub R_1,7 + jz yyy + jmp xxx

H_2 -> add R_1,-7 + jnz xxx | add R_1,-7 + jz yyy + jmp xxx

г) Рассмотрим грамматику G

```
G = ( N, T, S, R )
    N = { A, B}
    T = { a, b, c, d, x , y }
    a = "nop"
    b = "add eax, 0"
    c = "push ecx "+ "pop ecx"
    x = "AND [EDI], AL"
    y = "DEC AL"
    e = ""
    R = {
        S::= aS | bS | eS | xA
        A::= aA | bA | cA | yB
        B::= aB | bB | cB | e
    }
```

Примеры вывода:

S => aS => abS => abxA => abxcA => abxcyB => abxcybB => abxcybe

S => bS => bxA => bxaA => bxacA => bxacyB => bxacycB => bxacyce

S => xA => xbA => xbaA => xbacA => xbacyB => xbacye

}

Так при различных запусках мы можем получить различные последовательности команд. Приведем последовательности команд, соответствующие примерам выводов выше.

```
Nop                 add eax, 0          AND [EDI] , AL
add eax , 0         AND [EDI] , AL      add eax, 0
```

```
AND [EDI], AL      nop                nop
push ebx           push ebx           push ebx
pop ebx            pop ebx            pop ebx
DEC AL             DEC AL             DEC AL
add eax , 0        push ebx
                   pop ebx
```

В данном примере команды a="nop ", b = "add eax, 0 ", c = "push ebx "+ "pop ebx" являются «мусором», запутывающим код.

Каждая из этих цепочек является измененным вариантом исходного защищаемого кода.

3.1.3 Вставка инструкций для создания полиморфного кода

Полиморфизм заключается в формировании программного кода «на лету» — уже во время исполнения, при этом сама процедура, формирующая код, также не должна быть постоянной и видоизменяется при каждом новом запуске. Особенностью полиморфизма является самомодификация исполняемого кода.

Все полиморфные генераторы обязательно снабжаются расшифровщиком кода, который по определенному принципу преобразует переданный ему код, вызывая при этом стандартные функции и процедуры операционной системы.

Рассмотрим общую схему полиморфа, представленную в статье [35] (рисунок 3.3).

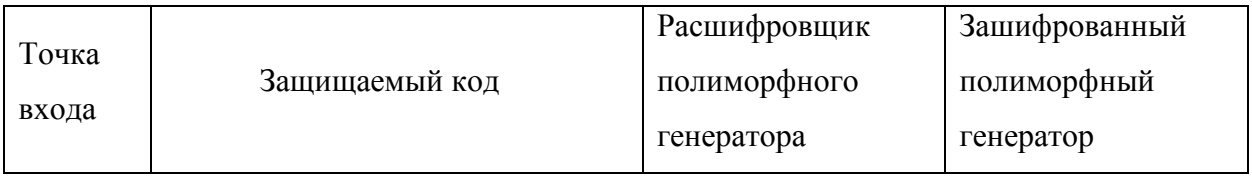

Рисунок 3.3 - Общая схема полиморфа

Методы полиморфирования включают преобразования, позволяющие производить самомодификацию кода во время выполнения. В данном протекторе используются следующие методы полиморфирования:

- Модификация имен регистров
- Модификация команд

Рассмотрим примеры использования полиморфизма в нашем протекторе, в которых используются данные методы.

1) *Модификация имен регистров*

Модификация имен регистров происходит уже во время выполнения кода. Каждый регистр имеет свой код, который можно изменить во время выполнения программы, используя различные арифметические и логические операции. Таким образом, при дизассемблировании видно, что операции выполняются с определенными регистрами, при этом коды этих регистров преобразовываются в другие и операция выполняется с другими регистрами. Изменение кодов регистров усложняет поиск сигнатуры программы. Рассмотрим примеры модификации регистров, используемые в данном протекторе.

а) Рассмотрим 2 команды и соответствующие им машинные коды

Инструкция ассемблера	Машинный код
`add edx, ebx`	`03D3`
`add edx, ecx`	`03D1`

При анализе следующего кода видно, что последовательность команд

```
     mov eax, offset label1
     inc eax
     or  byte ptr[eax],  0x2
label1:
     add edx, ecx
```

на самом деле выполняет сложение со вторым операндом – регистром ebx, а не ecx:

```
     add edx, ebx
```

б) Рассмотрим другую пару команд:

Инструкция ассемблера	Машинный код
`mov ebx, ebp`	`8BDD`
`mov ebx, ecx`	`8BD9`

При анализе этих кодов видно, что последовательность команд

```
     mov eax, offset label1
     inc eax
     and byte ptr[eax],  0xFB
label1:
```

```
mov ebx, ebp
```

на самом деле выполняет сложение со вторым операндом – регистром ecx, а не ebp:

```
mov ebx, ecx
```

В этих предыдущих примерах имеют место перемещения в памяти (address relocation). Рассмотрим так же используемые методы модификации кодов регистров без перемещений в памяти.

а) Преобразование ebx в ecx

Инструкция ассемблера	Машинный код
inc ebx	43
inc ecx	41

В протекторе используются следующие способы автоматической модификации данных регистров в команде инкрементирования:

1 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+8	EB 01
$+7 RETN	C3
$+8 MOV EAX, DWORD PTR SS:[ESP-4]	8B4424 FC
$+C ADD EAX, 0D	83C0 0D
$+F AND BYTE PTR DS:[EAX], 0xE9	8020 E9
$+12 INC ebx	43

2 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+B	EB 01
$+7 MOV EAX, DWORD PTR SS:[ESP]	8B0424
$+A RETN	C3
$+B ADD EAX, 0C	83C0 0C
$+E AND BYTE PTR DS:[EAX], 0xE9	8020 E9
$+11 INC EBX	43

Таким образом, в обоих случаях вместо инструкции с кодом 43, выполнится инструкция 41.

б) Преобразование edx в ebx

Инструкция ассемблера	Машинный код
`add edx, 0x6`	`83C2 06`
`add ebx, 0x6`	`83C3 06`

В протекторе используются следующие способы автоматической модификации команды edx в ecx в команде сложения:

1 способ:

`$ => CALL $+7`	`E8 02000000`
`$+5  JMP SHORT $+8`	`EB 01`
`$+7  RETN`	`C3`
`$+8  MOV EAX, DWORD PTR SS:[ESP-4]`	`8B4424 FC`
`$+C  ADD EAX, 0E`	`83C0 0C`
`$+F  INC EAX`	`40`
`$+10 ADD BYTE PTR DS:[EAX], 0x1`	`8000 01`
`$+13 ADD EDX, 0x6`	`83C2 06`

2 способ:

`$ =>  CALL $+7`	`E8 02000000`
`$+5   JMP SHORT $+B`	`EB 01`
`$+7   MOV EAX, DWORD PTR SS:[ESP]`	`8B0424`
`$+A   RETN`	`C3`
`$+B   ADD EAX, 0D`	`83C0 0B`
`$+E   INC EAX`	`40`
`$+F   AND BYTE PTR DS:[EAX], 0x1`	`8000 01`
`$+12  ADD EDX, 0x6`	`83C2 06`

Таким образом, в обоих случаях вместо инструкции с кодами `83C2 06`, выполнится команда `83C3 06`.

Аналогичные преобразования можно делать для любых пар регистров.

2) *Модификация инструкций*

Аналогично с модификацией кодов регистров, возможна модификация кодов команд ассемблера при помощи логических и арифметических операций.

Рассмотрим несколько примеров модификации инструкций, используемых в протекторе:

а) Модификация команды add в sub

Инструкция ассемблера	Машинный код
`ADD DWORD PTR DS:[5530C],10`	`8305 0C530500 10`
`SUB DWORD PTR DS:[5530C],10`	`832D 0C530500 10`

Рассмотрим полиморфизм, позволяющий заменить команду add на sub:

```
    mov eax, offset label1
    inc eax
    or byte ptr[eax],  0x2D
label1:
    add dword ptr[0x5530C], 10
```

Таким образом, на самом деле, выполняется команда

```
    sub dword ptr[0x5530C], 10
```

И, наоборот, преобразование инструкции sub в add

```
    mov eax, offset label1
    inc eax
    and byte ptr[eax],  0x07
label1:
    sub dword ptr[0x5530C], 10
```

Таким образом, на самом деле, выполняется команда

```
    add dword ptr[0x5530C], 10
```

б) Модификация команды inc в dec

Инструкция ассемблера	Машинный код
`inc  ebx`	`43`
`dec  ebx`	`4B`

Рассмотрим полиморфизм, позволяющий заменить команду inc ebx на dec ebx:

```
    mov eax, offset label1
    xor byte ptr[eax],  0x08
label1:
    inc ebx
```

Таким образом, на самом деле, выполняется команда

```
    dec ebx
```

И, наоборот, преобразование инструкции dec ebx в inc ebx

```
    mov eax, offset label1
    sub byte ptr[eax],  0x08
label1:
    dec ebx
```

Таким образом, на самом деле, выполняется команда

```
inc ebx
```

в) Модификация команды jle в jge

Инструкция ассемблера	Машинный код
JLE +5	0F8E 27D107FF
JGE +5	0F8D 21D107FF

Рассмотрим полиморфизм, позволяющий заменить команду jge на jle:

```
    mov  eax, offset label1
    inc eax
    xor  byte ptr[eax],  0x03
label1:
    jge +5
```

Таким образом, на самом деле, выполняется команда

```
jle +5
```

И, наоборот, преобразование инструкции jle в jge:

```
    mov  eax, offset label1
    inc eax
    dec  byte ptr[eax]
label1:
    jle +5
```

Таким образом, на самом деле, выполняется команда

```
jge +5
```

Аналогичные преобразования используются для пар команд `jae/jbe`, `ja/jb`, `jl/jg`.

г) Модификация команды je в jne

Инструкция ассемблера	Машинный код
JE short 7C91AE21	74 14
JNE short 7C91AE21	75 14

Рассмотрим полиморфизм, позволяющий заменить команду je на jne:

```
     mov  eax, offset label1
     inc  byte ptr[eax],  0x01
label1:
     je short 7C91AE21
```

Таким образом, на самом деле, выполняется команда

```
     jne short 7C91AE21
```

И, наоборот, преобразование инструкции jne в je

```
     mov  eax, offset label1
     dec  byte ptr[eax]
label1:
     jne short 7C91AE21
```

Таким образом, на самом деле, выполняется команда

```
     je short 7C91AE21
```

В этих предыдущих примерах имеют место перемещения в памяти (address relocation). Рассмотрим так же используемые методы модификации кодов команд без перемещений в памяти.

а) Модификация команд add, sub

Инструкция ассемблера	Машинный код
`add ebx, 0x8`	`83C3 08`
`sub ebx, 0x8`	`83EB 08`

В протекторе используются следующие способы автоматической модификации команды add в sub:

1 способ:

`$ => CALL $+7`	`E8 02000000`
`$+5  JMP SHORT $+8`	`EB 01`
`$+7  RETN`	`C3`
`$+8  MOV EAX, DWORD PTR SS:[ESP-4]`	`8B4424 FC`
`$+C  ADD EAX, 0E`	`83C0 0C`
`$+F  INC EAX`	`40`
`$+10 XOR BYTE PTR DS:[EAX], 0x28`	`8030 28`
`$+13 add ebx, 0x8`	`83C3 08`

2 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+B	EB 01
$+7 MOV EAX, DWORD PTR SS:[ESP]	8B0424
$+A RETN	C3
$+B ADD EAX, 0D	83C0 0B
$+E INC EAX	40
$+F XOR BYTE PTR DS:[EAX], 28	8030 28
$+12 ADD EBX, 0x8	83C3 08

Таким образом, в обоих случаях вместо инструкции с кодами 83C3 08, выполнится команда 83EB 08.

Верно и обратное, то есть инструкцию sub протектор может модифицировать в инструкцию add, например:

1 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+8	EB 01
$+7 RETN	C3
$+8 MOV EAX, DWORD PTR SS:[ESP-4]	8B4424 FC
$+C ADD EAX, 0E	83C0 0E
$+F INC EAX	40
$+10 AND BYTE PTR DS:[EAX], 0xD7	8020 D7
$+13 SUB ebx, 0x8	83EB 08

2 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+B	EB 01
$+7 MOV EAX, DWORD PTR SS:[ESP]	8B0424
$+A RETN	C3
$+B ADD EAX, 0D	83C0 0D
$+E INC EAX	40
$+F AND BYTE PTR DS:[EAX], 0xD7	8020 D7
$+12 SUB EBX, 0x8	83EB 08

Таким образом, в обоих случаях вместо команды с кодами 83EB 08, выполнится команда 83C3 08.

б) Модификация команд inc, dec

Инструкция ассемблера	Машинный код
inc ebx	43
dec ebx	4B

В протекторе используются следующие способы автоматической модификации команды inc в dec:

1 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+8	EB 01
$+7 RETN	C3
$+8 MOV EAX, DWORD PTR SS:[ESP-4]	8B4424 FC
$+C ADD EAX, 0D	83C0 0D
$+F ADD BYTE PTR DS:[EAX], 0x8	8000 08
$+12 INC ebx	43

2 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+B	EB 01
$+7 MOV EAX, DWORD PTR SS:[ESP]	8B0424
$+A RETN	C3
$+B ADD EAX, 0C	83C0 0C
$+E ADD BYTE PTR DS:[EAX], 0x8	8000 08
$+11 INC EBX	43

Таким образом, в обоих случаях вместо инструкции с кодом 43, выполнится инструкция 4B.

Верно и обратное, то есть инструкцию dec протектор может модифицировать в инструкцию inc, например:

1 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+8	EB 01

$+7 RETN	C3
$+8 MOV EAX, DWORD PTR SS:[ESP-4]	8B4424 FC
$+C ADD EAX, 0D	83C0 0C
$+F SUB BYTE PTR DS:[EAX], 0x8	8000 08
$+12 DEC EBX	4B

2 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+B	EB 01
$+7 MOV EAX, DWORD PTR SS:[ESP]	8B0424
$+A RETN	C3
$+B ADD EAX, 0C	83C0 0B
$+E SUB BYTE PTR DS:[EAX], 28	8000 08
$+11 DEC EBX	4B

Таким образом, в обоих случаях вместо команды с кодом 4B, выполнится команда 43.

в) Модификация команд je, jne

Инструкция ассемблера	Машинный код
JE short $+5	74 14
JNZ sort $+5	75 14

В протекторе используются следующие способы автоматической модификации команды je в jne:

1 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+8	EB 01
$+7 RETN	C3
$+8 MOV EAX, DWORD PTR SS:[ESP-4]	8B4424 FC
$+C ADD EAX, 0C	83C0 0C
$+F INC BYTE PTR DS:[EAX]	FE00
$+11 JE SHORT $+34	74 14

2 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+B	EB 04

$+7 MOV EAX, DWORD PTR SS:[ESP]	8B0424
$+A RETN	C3
$+B ADD EAX, 0B	83C0 0B
$+E INC BYTE PTR DS:[EAX]	FE00
$+10 JE SHORT $+34	74 14

Таким образом, в обоих случаях вместо команды с кодами 74 14, выполнится команда 75 14.

Верно и обратное, то есть инструкцию jne протектор может модифицировать в инструкцию je, например:

1 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+8	EB 01
$+7 RETN	C3
$+8 MOV EAX, DWORD PTR SS:[ESP-4]	8B4424 FC
$+C ADD EAX, 0C	83C0 0C
$+F DEC BYTE PTR DS:[EAX]	FE08
$+11 JNE SHORT $+34	75 14

2 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+B	EB 04
$+7 MOV EAX, DWORD PTR SS:[ESP]	8B0424
$+A RETN	C3
$+B ADD EAX, 0B	83C0 0B
$+E DEC BYTE PTR DS:[EAX]	FE08
$+10 JNE SHORT $+34	75 14

Таким образом, в обоих случаях вместо команды с кодами 75 14, выполнится команда 74 14.

г) Модификация команд jl, jg

Инструкция ассемблера	Машинный код
JL +5	0F8C EEC007FF
JG +5	0F8F EEC007FF

В протекторе используются следующие способы автоматической модификации команды jl в jg:

1 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+8	EB 01
$+7 RETN	C3
$+8 MOV EAX, DWORD PTR SS:[ESP-4]	8B4424 FC
$+C ADD EAX, 0E	83C0 0C
$+F INC EAX	40
$+10 ADD BYTE PTR DS:[EAX], 0x03	8000 03
$+13 JL +5	0F8C EEC007FF

2 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+B	EB 01
$+7 MOV EAX, DWORD PTR SS:[ESP]	8B0424
$+A RETN	C3
$+B ADD EAX, 0D	83C0 0B
$+E INC EAX	40
$+F ADD BYTE PTR DS:[EAX], 28	8000 03
$+12 JL +5	0F8C EEC007FF

Таким образом, в обоих случаях вместо инструкции с кодами0F8C EEC007FF, выполнится команда 0F8F EEC007FF.

Верно и обратное, то есть инструкцию jg протектор может модифицировать в инструкцию jl, например:

1 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+8	EB 01
$+7 RETN	C3
$+8 MOV EAX, DWORD PTR SS:[ESP-4]	8B4424 FC
$+C ADD EAX, 0E	83C0 0E
$+F INC EAX	40
$+10 AND BYTE PTR DS:[EAX], 0xFC	8020 FC
$+13 JG +5	0F8F EEC007FF

2 способ:

$ => CALL $+7	E8 02000000
$+5 JMP SHORT $+B	EB 01
$+7 MOV EAX, DWORD PTR SS:[ESP]	8B0424
$+A RETN	C3
$+B ADD EAX, 0D	83C0 0D
$+E INC EAX	40
$+F AND BYTE PTR DS:[EAX], 0xFC	8020 FC
$+12 JG +5	0F8F EEC007FF

Таким образом, в обоих случаях вместо команды с кодами `0F8F EEC007FF` выполнится команда `0F8C EEC007FF`.

Аналогичные преобразования выполняются и для других пар команд условных переходов `ja/jb, jae/jbe, jge/jle`.

3.2 Средства противодействия динамическому анализу исполняемого кода

Process Environment Block (PEB) это структура, которая содержит данные окружающей среды процесса:

- переменные среды,
- список загруженных модулей,
- адреса в памяти «Куча»,
-другие параметры, жизненно необходимые для функционирования процесса в OS MsWindows

```
typedef struct _PEB
{
        BOOLEAN InheritedAddressSpace;
        BOOLEAN ReadImageFileExecOptions;
        BOOLEAN BeingDebugged;
        BOOLEAN Spare;
        HANDLE Mutant;
        PVOID ImageBaseAddress;
        PPEB_LDR_DATA LoaderData;
        PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
        PVOID SubSystemData;
        PVOID ProcessHeap;
        PVOID FastPebLock;
        PPEBLOCKROUTINE FastPebLockRoutine;
        PPEBLOCKROUTINE FastPebUnlockRoutine;
}
```

Следует отметить, что структура PEB является недокументированной и зависит от версии Windows, поэтому обращаться с ней надо с аккуратностью.

В структуре PEB есть поле, которое может использоваться для выявления отладчика. Это поле называется NtGlobalFlag и находится по смещению 0x68 от начала PEB. Значение в этом поле по умолчанию - ноль. Начиная с Windows 2000, введены специальные значения, которые, как правило, заносятся в это поле, когда запущен отладчик. Проверка этого значения не очень надежный показатель наличия отладчика, однако, такую проверку часто используют с целью противодействия отладке защищаемого кода. Поле представлено рядом флагов. Значение,

которое предполагает присутствие отладчика, складывается из значения следующих полей и поэтому обычно равно 0x70:

```
FLG_HEAP_ENABLE_TAIL_CHECK (0x10)
FLG_HEAP_ENABLE_FREE_CHECK (0x20)
FLG_HEAP_VALIDATE_PARAMETERS (0x40)
```

Таким образом, используя описанные флаги, мы можем установить, отлаживают ли программу или нет.

Рассмотрим пример реализации этого приема, продемонстрированный в [23]:

```
mov eax, fs:[30h]
mov eax, [eax+68h]
;NtGlobalFlag
and eax, 0x70
test eax, eax
jne @DebuggerDetected
...
```

Этот метод эффективен, если отладчик не создает пустую строку "GlobalFlag" или каким-либо другим образом не задает пустое значение этой строки, например, "GlobalFlag" – это строка ключа "HKLM\System\CurrentControlSet\ Control\Session Manage", в которой содержатся все дополнительные флаги.

3.2.1 Противодействие отладке защищаемого кода и кода виртуальной машины для Windows: функция IsDebuggerPresent

Функция IsDebuggerPresent() показывает, запущен ли вызывающий ее процесс в контексте отладчика [33]. Она возвращает TRUE, если отладчик присутствует. На уровне архитектуры операционной системы это означает установка флага в поле PEB > BeingDebugged, который находится по смещению 0x02 от начала PEB.

Пример кода:

```
call IsDebuggerPresent
test al, al
jne being_debugged
```

Некоторые упаковщики избегают использовать IsDebuggerPresent() и считывают данные непосредственно из PEB.

Пример кода:

```
mov eax, fs:[30h] ;указатель на PEB
```

```
cmp b [eax+2], 0 ;проверка флага BeingDebugged
jne being_debugged
;если флаг установлен, то присутствует отладчик
```

Или в другой вариации:

```
Mov eax, large fs:18h ;получаем указатель на TEB
Mov eax, [eax+30h] ;затем указатель на PEB
movzx eax, byte ptr [ eax+2] ;в регистр EAX
;заносится значение флага BeingDebugged
retn
```

Эти проверки известны, поэтому их стоит запутывать путем использования мусорного кода мусора или других способов. Однако, эта проверка не работает, если отладчик устанавливает флаг BeingDebugged структуры PEB равным FALSE. Это можно реализовать, поместив контрольную точку на первую команду IsDebuggerPresent() и исправить возвращаемое значение на FALSE.

3.2.2 Противодействия отладке защищаемого кода и кода виртуальной машины для Linux: Функция ptrace()

Функция ptrace() используется для отладки программ. Этот антиотладочный прием заключается в следующем: программа проверяет, отлаживают ли ее, пытаясь совершить самоотладку. Тогда, если программа отлаживается, то функция ptrace() выдаст сообщение об ошибке т.к. в один момент времени может работать только один отладчик.

Пример кода:

```
int main()  {
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0) {
      printf("DEBUGGING... Bye\n");
      return 1;
    }
    printf("Hello\n");
    return 0;
}
// -- EOF –
```

3.3 Способы достижения стойкой обфускации

Для достижения стойкости обфускации в формулировке Захарова [37] необходимо использование стойких методов шифрования, что показано в работе [37] приводит нас к необходимости выбора стойкого криптографически стойкого метода шифрования.

3.3.1 Сжатие двоичного кода виртуальной машины с целью уменьшения его размера на диске

Для экономии памяти интерпретатором виртуальной машины использовался алгоритма сжатия байткодов виртуальной машины. В качестве алгоритма для сжатия байткода был выбран Lempel–Ziv–Storer–Szymanski как имеющий возможность потоковой декомпрессии и не требующий выделения динамических блоков памяти, что не всегда возможно в контексте защищенного приложения, так как добавляет зависимость от CRT библиотек. В качестве основы использовалась реализация, разработанная Haruhiko Okumura, и предоставленная автором в свободный доступ.

Изначально планировалось использовать ZLib, однако алгоритмическая сложность процедуры распаковки и необходимость выделения буферов под распаковку привела, в процессе выполнения диссертационой работы, к необходимости отказа от данного алгоритма, в пользу другого, хоть и обеспечивающего худший коэффициент сжатия.

3.3.2 Шифрование двоичного кода виртуальной машины

Иногда защищаемый алгоритм слишком ценен для того чтобы быть запущенным на неавторизованной аппаратуре. Использование надежных криптографических методов с привязкой к аппаратуре, на которой исполняется защищаемая программа, позволяет реализовать этот требование.

В случае использования шифрования байткода реализуется следующая логика:

1. При защите на основе анализа установленного оборудования и последующего хэширования полученного «отпечатка» формируется 128-битный ключ.
2. При защите байткод шифруется этим ключом с применением метода XXTEA.
3. При выполнении происходит также происходит формирование 128-битного ключа. В том случае если аппаратура является авторизованной, то ключи совпадают и расшифровка является успешной. В противном случае расшифровки и выполнения защищенного алгоритма не происходит.

В отличие от стандартных RSA-derived алгоритмов XXTEA требует значительно меньших, как вычислительных ресурсов, так и временной памяти. Вместе с тем согласно выводам корейских исследователей сложность криптоанализа превышает 2^{84} MIPS лет при наличии $2^{52.5}$ образцов и более $2^{123.37}$ MIPS лет при наличии 1920 образцов [36], что гарантирует

надежность этого метода шифрования и соответствие требованиям предъявляемыми к подобным системам в современных условиях.

В качестве дополнительных методов защиты исходного программного кода были включены антиотладочные методы для ОС Windows и Linux.

Описанные ниже методы также реализовывались с использованием функций LLVMIRBuilder. Таким образом, антиотладочные методы встраиваются в код, который генерирует LLVM для интерпретатора виртуальной машины что позволяет получать стойкую обфускацию в парадигме черного ящика.

4 АНАЛИЗ ЭФФЕКТИВНОСТИ РАЗРАБОТАННЫХ СРЕДСТВ ЗАЩИТЫ ПРОГРАММНОГО КОДА ИСПОЛНЯЕМОГО НА ПЛАТФОРМЕ X86

4.1 Результаты защиты при помощи виртуального процессора

Как было сказано выше, главной характеристикой этого метода защиты, основанного на использовании генератора виртуальных машин, является модификация кода программного продукта, представленного в виде исходных кодов, и скомпилированного исполняемого кода к виду, сохраняющему ее функциональность, но затрудняющему анализ, понимание алгоритмов работы и, соответственно, модификацию третьими лицами.

Проведем сравнительный анализ исходного исполняемого файла и файла, защищенного регистровой виртуальной машиной с 8 битными кодами инструкций и операндов.

Сначала рассмотрим защиту функции, которая просто возвращает значение, т.е. функции с псевдокодом `"return 1".` Для процессора x86 – это набор простейших операций со стеком, где ассемблерный код имеет вид

```
push    ebp
mov     ebp, esp
mov     eax, [ebp+arg_0]
add     eax, 1
pop     ebp
retn
```

И соответствующий байт-код исходной защищаемой функции имеет вид:

```
{55, 8B, EC, 8B, 45, 08, 83, C0,  01, 5D, C3}
```

Байт-код защищенного с помощью указанной виртуальной машины исполняемого файла имеет вид:

```
VMByteCode =
{c9, f0, fb, 1, 0, 0, c, c9, f0, b3, 3, 0, 0, e, 49, f0, c, d,
4, e8, fc, ff, ff, ff, d, 49, f4, e, c, 4b, f0, d, fb, 1, 0, 0,
4, e8, 4, 0, 0, 0, c, 4b, f0, d, 72, 1, 0, 0, c8, f0, c, d, 49,
f0, d, c, 4, e8, 1, 0, 0, 0, c, 4b, f0, d, 53, 2, 0, 0, 4b, f0,
c, 53, 2, 0, 0, c9, f0, 65, 3, 0, 0, c, 4, e8, 4, 0, 0, 0, c,
4b, f0, c, 65, 3, 0, 0, c8, f0, c, c, 4b, f0, c, b3, 3, 0, 0,
4d};
```

Рисунок 4.1 - Байт-код защищенной функции

При анализе важно отметить, что, на самом деле, существует отличие между представлениями байт-кода исходного файла и байт-кода защищенного, которое связано с тем, что злоумышленник не знает ни длину кодов команд, ни размеры кодов операндов. Для получения этих данных ему нужно провести дополнительный анализ защищенного файла, чему можно противодействовать с помощью различных способов борьбы с отладкой, шифрованием и другими методами. Кроме того, во время защиты исполняемого файла генерируется специфическая виртуальная машина, поэтому нет программных инструментов для анализа этого байт-кода, предоставляющих аналог ассемблерного текста, и, соответственно, анализ строится только на рассмотрении данного байт-кода.

Предположим, что злоумышленник на основе определенных рассуждений или в процессе исследования предположил, что байт-код виртуальной машины – это множество кодов с размерностью 1 байт каждый и, соответственно байт-код выглядит как на рисунке 4.1.

Прежде всего, необходимо отметить увеличение размера исполняемого файла относительно исходного, причем данный байт-код относится именно к преобразованному байт-коду защищаемой функции, а увеличение размера исполняемого файла за счет присоединения виртуальной машины здесь не рассматривается. Это увеличение обусловлено тем, что генератор создал регистровую виртуальную машину, у которой нет операций работы со стеком, и все команды работы со стеком исполняются при помощи использования набора команд пересылки данных.

Главным предметом исследования для злоумышленника является семантика команд, соответствующих этим опкодам. Покажем, что частотный анализ – не помогает соотнести опкоды в байт-коде с некоторым осмысленным набором команд.

Для этого применим частотный анализ защищенному файлу, предоставленному на рисунке 4.1. Частотный анализ — один из методов криптоанализа, основывающийся на предположении о существовании нетривиального статистического распределения отдельных

кодов и их последовательностей как в исходном тексте, так и в защищенном, которое, с точностью до замены символов, будет сохраняться в процессе шифрования и дешифрования.

Частотный анализ байт-кода предполагает, что частота появления опкода, соответствующего определенной команде, в достаточно длинных текстах одна и та же для разных программ. Аналогичные рассуждения применяются к паре кодов, и т.д. Утверждается, что вероятность появления отдельных кодов, а также их порядок в байт-коде подчиняются статистическим закономерностям.

Байт-код, представленный на рисунке 4.1, слишком мал для проведения частотного анализа, но используем его для формулировки одного вывода. Рассмотрим относительную частоту для некоторых опкодов:

```
L(0) = 27      -> частота опкода 0  = 27/112
L(c) = 14      -> частота опкода c = 14/112
L[f0] = 13     -> частота опкода f0  = 13/112
L[9] =7        -> частота опкода 9  = 9/112
```

Логично предположить, что среди этих опкодов находятся опкоды, соответствующие командам mov, push, add и pop, а так же командам косвенной адресации, так как по статистике эти команды имеют максимальную относительную частоту для машинного кода процессора x86. При этом мы знаем, что рассматриваемая виртуальная машина является регистровой, то есть совсем не содержит команд для работы со стеком. И, соответственно, опкодов, соответствующих командам push и pop, в байт-коде нет совсем. Аналогичный вывод можно было бы сделать и для большей выборки, полученной после защиты исполняемого файла со значительным количеством команд для работы со стеком. Этот пример иллюстрирует тот факт, что мы не знаем типа виртуальной машины, используемой для защиты исполняемого файла, и, таким образом, не имеем статистических данных относительно которых можно проводить частотный анализ. Более того, для тех наборов команд, в которые попадут только sub (и не попадет add), самые различные варианты команд mov (благодаря многообразию которых частота каждого из них будет значительно меньше), одни команды будут выражены через другие по правилам приведенным выше, частотный анализ даст абсолютно неверный результат т.к. нет статистики по тому набору команд, который будет использоваться в итоге.

Так же необходимо отметить, что на два опкода 0 и f0 в данном защищенном файле являются опкодами команды mov с разными типами операндов. Таким образом, в защищенном файле, при большем количестве различных команд mov, относительная частота каждой из них в

отдельности меньше. И при добавлении достаточного набора команд mov. выделить их на основе частотного анализа станет невозможно.

Рассмотрим другой пример. Проанализируем защищаемую функцию, вычисляющую некоторое арифметическое выражение. На рисунке 6.2 представлен ее шестнадцатеричный машинный код, а также ASCII- символы, соответствующие этому коду.

Из асемблерного кода видно, что в этой функции множество команд mov, add, но отсутствуют команды push, pop из-за которых в прошлом примере значительно увеличилась относительная частота команды mov.

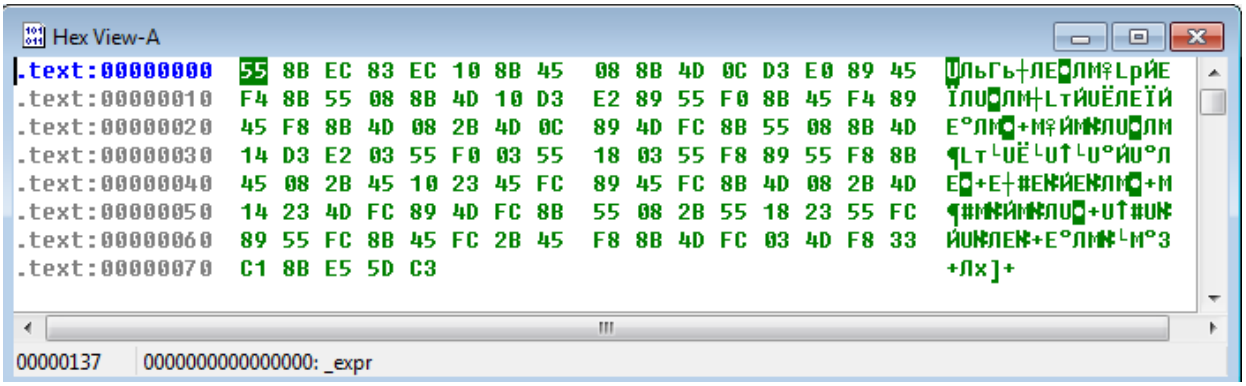

Рисунок 4.2 - Окно с шестнадцатеричным кодом защищаемой функции

Проанализировав рисунок 4.2, можно сделать вывод о том, что относительная частота кода команды mov составляет 23/133, при этом, так как мы знаем структуру ассемблерных кодов команд, а так же кодов операндов, то видим, что инструкций пересылки данных в данном файле 23/43.

Рассмотрим байт-код защищенного файла, представленный на рис. 4.3, и составим для него соответствующую частотную характеристику в таблице 4.1. Основываясь на предыдущем примере можно было бы предположить, что коды «0», «CF», «44», «45» соответствуют наиболее часто встречающимся командам ассемблера.

VMByteCode = { a3, cf, 16, 0, 0, 0, 45, a3, cf, 15, 1, 0, 0, 46, a1, cf, 45, 44, 84, c9, fc, ff, ff, ff, 44, a1, ce, 46, 45, a9, cf, 44, 16, 0, 0, 0, a9, cf, 44, 5c, 2, 0, 0, a3, cf, 16, 0, 0, 0, 44, a3, cf, 98, 3, 0, 0, 45, 84, c9, f0, ff, ff, ff, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 16, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 44, 84, c9, 8, 0, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 3f, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 44, 84, c9, c, 0, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 45, 1, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 3f, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 45, a1, ce, 44, 45, a3, cf, 5c, 2, 0, 0, 44, 84, c9, 8, 0, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 78, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 44, 84, c9, 10, 0, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 45, 1, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 78, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 45, a1, ce, 44, 45, a3, cf, 5c, 2, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 3f, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 45, a1, ce, 44, 45, a3, cf, 5c, 2, 0, 0, 44, 84, c9, 8, 0, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 45, 1, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 45, 1, 0, 0, a3, cf, 5c, 2, 0, 0, 45, a1, ce, 44, 45, a3, cf, 5c, 2, 0, 0, 44, 84, c9, 8, 0, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 78, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 44, a7, cf, 44, 44, a3, cf, 98, 3, 0, 0, 46, 80, c9, 45, 44, a9, cf, 46, 98, 3, 0, 0, a9, cf, 44, 78, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 45, 84, c9, 18, 0, 0, 0, 45, a7, cf, 45, 45, a3, cf, 98, 3, 0, 0, 46, 80, c9, 44, 45, a9, cf, 46, 98, 3, 0, 0, a9, cf, 45, 78, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 44, a7, cf, 44, 44, a3, cf, 98, 3, 0, 0, 46, 80, c9, 45, 44, a9, cf, 46, 98, 3, 0, 0, a9, cf, 44, 78, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 45, a1, ce, 44, 45, a3, cf, 5c, 2, 0, 0, 44, 84, c9, 8, 0, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 3f, 0, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 3f, 0, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a1, cf, 45, 45, a9, cf, 44, 98, 3, 0, 0, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 3f, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 45, a1, ce, 44, 45, a3, cf, 5c, 2, 0, 0, 44, 84, c9, 8, 0, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 45, 1, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 45, 1, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a1, cf, 45, 45, a9, cf, 44, 98, 3, 0, 0, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 45, 1, 0, 0, a3, cf, 5c, 2, 0, 0, 45, a1, ce, 44, 45, a3, cf, 5c, 2, 0, 0, 44, 84, c9, 8, 0, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 78, 0, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 78, 0, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a1, cf, 45, 45, a9, cf, 44, 98, 3, 0, 0, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 78, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 45, a1, ce, 44, 45, a3, cf, 5c, 2, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 3f, 0, 0, 0, a3, cf, 98, 3, 0, 0, 45, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a9, cf, 44, 3f, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 44, a7, cf, 44, 44, a9, cf, 44, 45, 1, 0, 0, a3, cf, 5c, 2, 0, 0, 45, a7, cf, 45, 45, a3, cf, 98, 3, 0, 0, 46, 80, c9, 44, 45, a9, cf, 46, 98, 3, 0, 0, a9, cf, 45, 45, 1, 0, 0, a3, cf, 98, 3, 0, 0, 44, a1, cf, 45, 45, a9, cf, 44, 98, 3, 0, 0, a1, cf, 44, 44, a9, cf, 45, 98, 3, 0, 0, a1, cf, 45, 45, a9, cf, 44, 98, 3, 0, 0, a9, cf, 45, 3f, 0, 0, 0, a3, cf, 5c, 2, 0, 0, 44, a9, cf, 44, 16, 0, 0, 0, a3, cf, 68, 3, 0, 0, 44, 84, c9, 4, 0, 0, 0, 44, a9, cf, 44, 68, 3, 0, 0, a7, cf, 44, 44, a9, cf, 44, 15, 1, 0, 0, b1}

Рисунок 4.3 - Байт-код функции, защищенной виртуальной машиной (незашифрованный и несжатый)

Но, как уже было сказано, данная функция вычисляет арифметическое выражение и наиболее часто встречающимися кодами являются коды операндов, среди которых «0», «CF», «44», «45». Таким образом, так как коды операндов и операций не разделены, то злоумышленник не может просто использовать статистические данные по командам, так как любой код – может быть как кодом операции, так и кодом операнда. Так в данном примере, коды «A9», «3», «5C» соответствуют командам mov с разными операндами. И злоумышленник имеет только соответствующие частоты 62/1056, 44/1056 и 27/1056, но не может сделать вывод о том, что это частоты опкодов или кодов операндов.

Таблица 4.1

Таблица частот опкодов защищенного файла с наибольшей относительной частотой

Код операции	Относительная частота
0	274/1056
CF	146/1056
44	143/1056
45	82/1056

A9	62/1056
A3	47/1056
3	44/1056
98	42/1056
A1	30/1056
2	27/1056
5C	27/1056
C9	18/1056
87	14/1056

При этом, как и в первом примере можно предоставить большее число различных команд mov, для уменьшения относительной частоты каждой из них в отдельности. Кроме того, для противодействия частотному анализу, защищенный код можно обфусцировать, добавив инструкции, не влияющие на функционирование программы, но противодействующие частотному анализу. Более того, так как злоумышленник анализирует только лишь набор машинного байт-кода, можно, в том числе, добавить просто «пустые» опкоды, которые могут не выполняться, но изменять частотную характеристику защищенного файла.

Результаты защиты можно также оценить при помощи теории скрытности.

Базовый набор x86 составляет 454 разных инструкции. Из них 251 непривилегированная инструкция из которых 218 реально используется компилятором. Исключаются BCD и им подобные инструкции, не используемые в реальной жизни. Таким образом, общая арсенальная скрытность стандартного набора инструкций не превышает 7.7. В случае же виртализированного кода общая арсенальная скрытность составляет 13.1, что дает общий прирост арсенальной скрытности почти в два раза.

В случае, если рассматривать групповую скрытность, то в случае незащищенного кода она примерно равна 448. Для случая защищенного кода с паромерами из примера значение групповой скрытности составит около 1370, а при выборе 3900 команд теоретическая величина групповой скрытности составит 8723. Однако, следует отметить, что в случае выбора большого числа команд необходимо предпринимать значительные усилия к тому чтобы соблюдать баланс различных команд в коде, что представляется весьма затруднительным, т.к. уже при использовании 900 команд время для генерации кода вырастает более чем в 80 раз.

К сожалению, внутренний алгоритм балансировки частоты команд приводит к тому, что хотя алгоритм кодогенератора и имеет полиномиальную сложность, но это сложность пропорциональна $O(m^3)$, где m – число команд конкретной виртуальной машины. Вследствие этого создание виртуальных машин с максимальной скрытностью является нецелесообразным с практической точки зрения.

Построим кривую снятия неопределенности (КСН), как графически выраженную зависимость средней остаточной неопределенности (энтропии) состояния объекта от числа выполненных уже раскрытых команд виртуальной машины, а таким образом, и от времени потраченного на исследование защищенного кода, т.к. эти величины связаны линейно, нормализовав ее для защищенного и незащищенного кода в предположении, что арсеналы равны (мы имеем одинаковое число возможных иструкций) и число инструкций в конкретной виртуальной машине тоже одинаково.

В начале обратим внимание на то что распределние инструкций для незащищенного кода известно и приведено на рисунке 4.4.

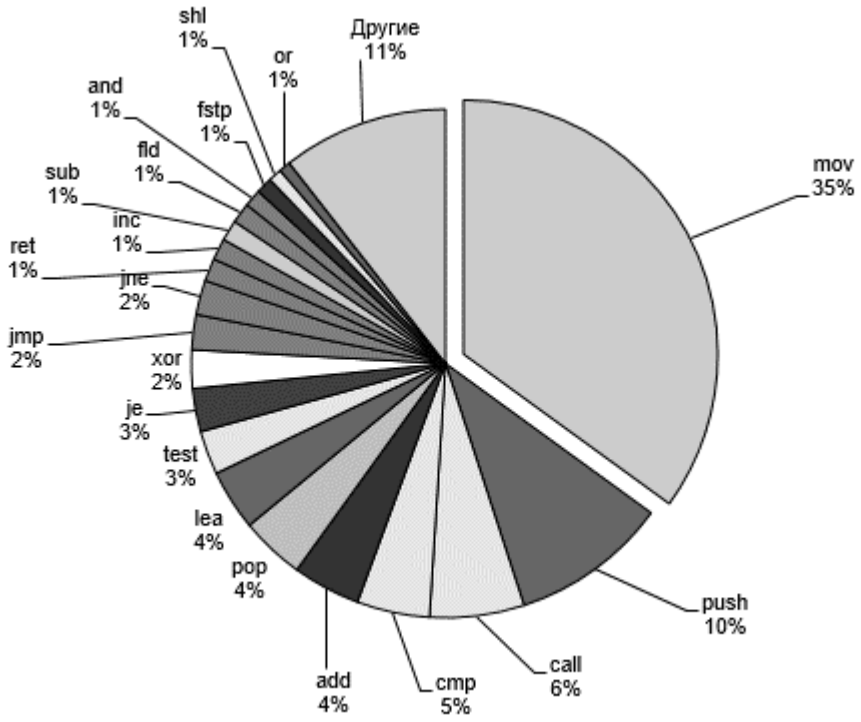

Рисунок 4.4 – Распределение команд в x86 на примере системных библиотек Windows 7.

Очевидно, что при анализе наиболее эффективная стратегия – это стратегия, угадывания, так именно она приводит к результату наиболее быстрым образом для x86 кода.

Тогда кривая КСН будет выглядеть так как показано на рисунке 4.4. При этом точка перегиба зависит непосредственно от исследователя и означет этап перехода от стратегии угадывания к непосредственному снятию неопределенности через исследование кода интрепретатотра виртуальной машины исполняющей обфусцированный код. Следует отметить, что это весьма трудоемкий процесс, и для нагляности на рисунке 4.5 он показан, как более быстрый чем на самом деле. Следует отметить, что полностью в наших силах сделать так, чтоб стратегия угадывания гарантированно не приводила к положительным результатам. Для этого необходимо всего лишь выбирать целевым распределение честот команд отличное отприведенного на рисунке 4.4.

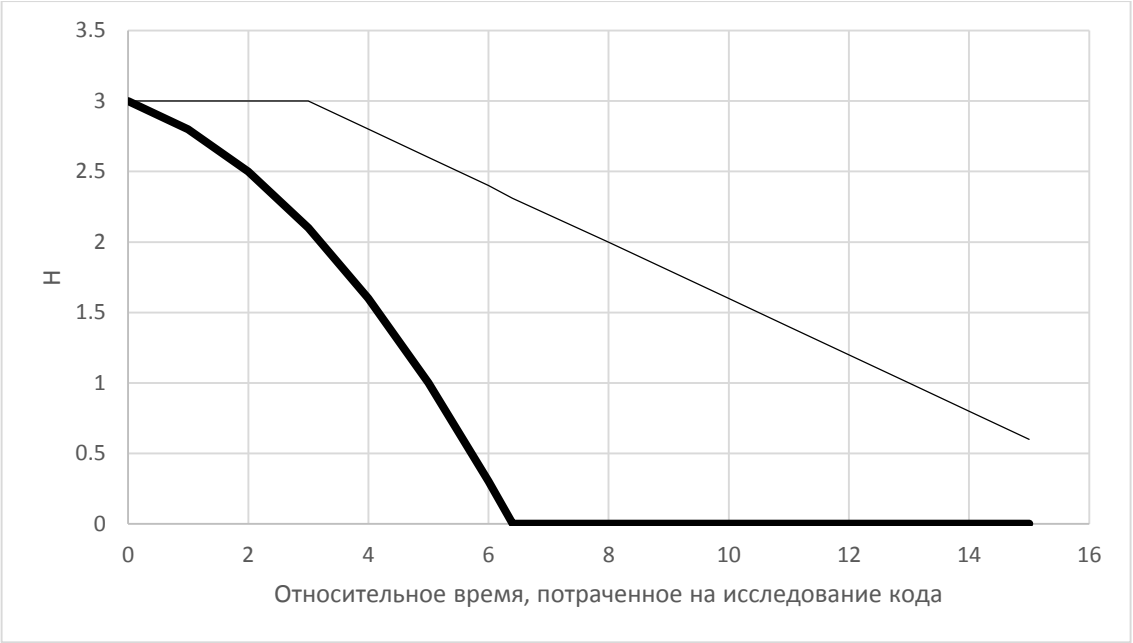

Рисунок 4.5 – Кривая снятия неопределённости в случае незащищенного машинного кода(жирная линия) и защищенного кода (тонкая линия) .

4.2 Результаты защиты при помощи сети Петри и анализ производительности защищенного кода

Работа разработанного алгоритма проверялась набором контрольных примеров. Для каждого примера изменялись различные параметры сети Петри из пункта 2.3. Оценка качества полученного защищенного исходного файла производилась по следующим параметрам:

- размер исполняемого файла;
- время выполнения программы;
- количество кода, получаемого при обфускации исходного объектного файла.

В качестве защищаемого объекта была выбрана программа, реализующая итеративное вычисление факториала. Ставилась задача защитить функцию, содержащую основной алгоритм. Характеристики соответствующего исполняемого файла приведены в Таблице 4.2.

Таблица 4.2

Характеристики исходного файла

Характеристики	Исходный файл
Размер исполняемого файла (Кб)	27
Время выполнения (мс)	0.00437

Количество кода (строки)	5

При построении сети Петри следующие параметры сети могут варьироваться:

- количество узлов сети;
- количество раундов;
- метки сети, которые заполняются случайными значениями.

Постоянной величиной является количество целевых значений сети: оно равняется количеству констант, которые находится в максимальном BasicBlock. Обозначим эту величину через N. Остальные параметры сети будем выражать через N. При изменении одного из параметров два другие будем фиксировать.

Зафиксируем количество раундов сети и метки со случайными значениями. Пусть эти величины равны соответственно N+2 и N. Таблица 4.3 иллюстрирует, как меняются основные характеристики защищенного исполняемого файла при изменении количества узлов сети.

Таблица 4.3

Характеристики защищенного исходного файла при изменении количества узлов сети

	N/2	N	N + 2	N * 2
Размер исполняемого файла (Кб)	34	35	32	42
Время выполнения (мс)	0.00981	0.0104	0.00948	0.0263
Количество кода (строки)	86	98	64	347

Из этих данных можно сделать вывод, что лучшие показатели достигаются при количестве узлов в два раза большем, чем количество целевых значений.

Теперь зафиксируем количество узлов сети и метки со случайными значениями. Пусть эти величины равны соответственно N * 2 и N. Таблица 4.4 иллюстрирует, как меняются основные характеристики защищенного исполняемого файла при изменении количества раундов сети.

Таблица 4.4

Характеристики защищенного исходного файла при изменении количества раундов сети

	N/2	N	N + 2	N * 2
Размер исполняемого файла (Кб)	28	29	47	39
Время выполнения (мс)	0.00487	0.00593	0.0318	0.0221
Количество кода (строки)	32	58	402	328

Отсюда получаем, что защищаемый файл имеет оптимальные показатели, если количество раундов сети незначительно превосходит количество целевых значений.

Проследим, как меняются основные характеристики защищенного исполняемого файла при изменении количества меток сети, содержащих случайные значения. Количество раундов сети положим N + 2, а количество узлов сети N * 2 (таблица 4.5).

Таблица 4.5

Характеристики защищенного исходного файла при изменении количества меток сети, содержащих случайные значения

	N/2	N	N + 2	N * 2
Размер исполняемого файла (Кб)	31	45	38	-
Время выполнения (мс)	0.00891	0.0301	0.0205	-
Количество кода (строки)	58	386	311	-

Если случайными значениями заполнить N * 2 и более узлов сети, то сеть не будет выполнена. То есть целевые значения не будут получены. Наилучшие результаты получаются, если количество меток, заполненных случайными значениями, равно количеству целевых значений сети.

Отсюда получаем оптимальные настройки сети Петри для защиты исходного кода, который содержит N целевых значений. При таких настройках максимально увеличивается количество кода в исполняемом файле, что затрудняет анализ при реинжиниринге. В тоже время не происходит существенного увеличения размера исходного файла и времени его выполнения (таблица 4.6).

Таблица 4.6

Оптимальные значения параметров сети Петри

Характеристика	Значение
Количество узлов сети	N * 2
Количество раундов	N + 2
Метки сети со случайными значениями	N

Так как изначально ставилась цель защиты исходного файла от дисассемблирования, то важно проследить, как изменяется блок-схема программы, которую можно получить с помощью распространенных отладочных инструментов.

В качестве такого инструмента был выбран IDA Pro Disassembler (англ. Interactive DisAssembler) — интерактивный дизассемблер, который широко используется для обратного проектирования. На рисунке 4.4 приведена блок-схема исходного исполняемого файла, полученная с помощью IDA Pro. На ней легко можно различить цикл и проверки, соответствующие вычислению факториала.

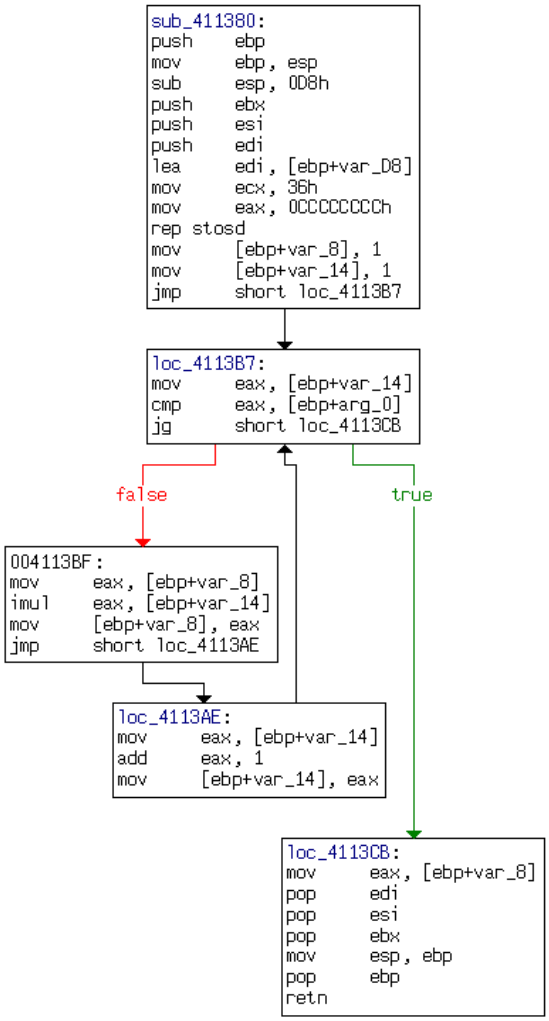

Рисунок 4.6 - Блок-схема исходной процедуры

На рРисунок 4.6 приведена блок-схема защищенного исполняемого файла, который всё также вычисляет факториал. На ней циклы и условия, которые, несомненно, присутствуют в коде, совсем неочевидны.

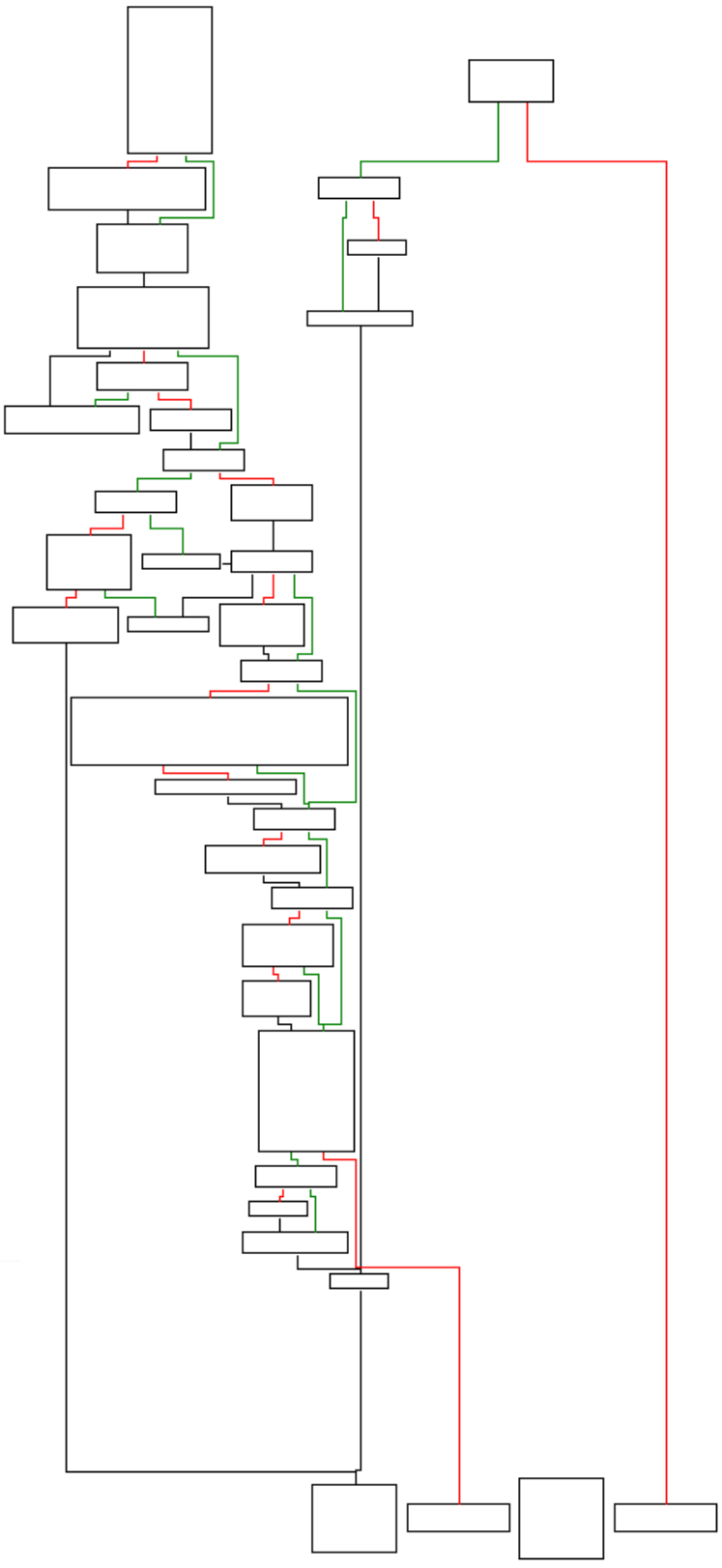

Рисунок 4.7 - Блок-схема защищенной процедуры

Рисунок 4.6 и р**Рисунок 4.**7 иллюстрируют, как меняется граф (блок-схема) исходного файла в процессе обфускации, и как сильно эти изменения затрудняют анализ кода.

По результатам анализа характеристик исполняемого файла и блок-схем, получаемых с помощью IDA Pro, можно сделать вывод, что реализуемый протектор является надежной защитой двоичного кода от реинжиниринга.

4.3 Структура инструментального средства защиты программного кода

Прототип модуля инструментального средства защиты исполняемого кода от обратного проектирования имеет модульную структуру, представленную на рРисунок 4.4.6.
В данную структуру сходят следующие модули:

1. Модуль захвата машинного кода представленного в форматах COFF и ELF
2. Модуль анализа и преобразования машинного кода в промежуточное представление для последующей защиты
3. Модуль создания виртуальной машины
4. Модуль сборки выходных данных состоящий из:
   - модуля компиляции байткода защищаемого алгоритма
   - создания интерпретатора виртуальной машины
   - защиты кода интерпретатора виртуальной машины
5. Модуль генерации и защиты кода
6. Модуль создания выходного объектного модуля для связи с остальными частями защищаемой программы при помощи стандартных утилит
7. Интерфейс программного прототипа

На рРисунок 4. 4.8 показана общая схема защиты исполняемых файлов прототипом протектора и проиллюстрирована многоплатформенность данного решения.

Сначала исполняемый файл дизассемблируется и транслируется в специальное промежуточное представление, зависимое от архитектуры процессора. Полученный код транслируется в машинно-независимое промежуточное представление (Intermediate Representation), затем, на основе входных пользовательских параметров и входных данных строится виртуальная машина, включающая интерпретатор для ее исполнения. И, наконец, новый код вместе с виртуальной машиной преобразуется в машинный код и записывается в новый машинно-зависимый исполняемый файл.

В том случае, если требуется защита иполняемого или динамически компонуемого объекта, то полученный защищенный файл необходимо обработать компановщиком с соотвествующими настройками длоя получения исполняенмого , динамически подключаемого (dll) или разделяемого (so) модуля.

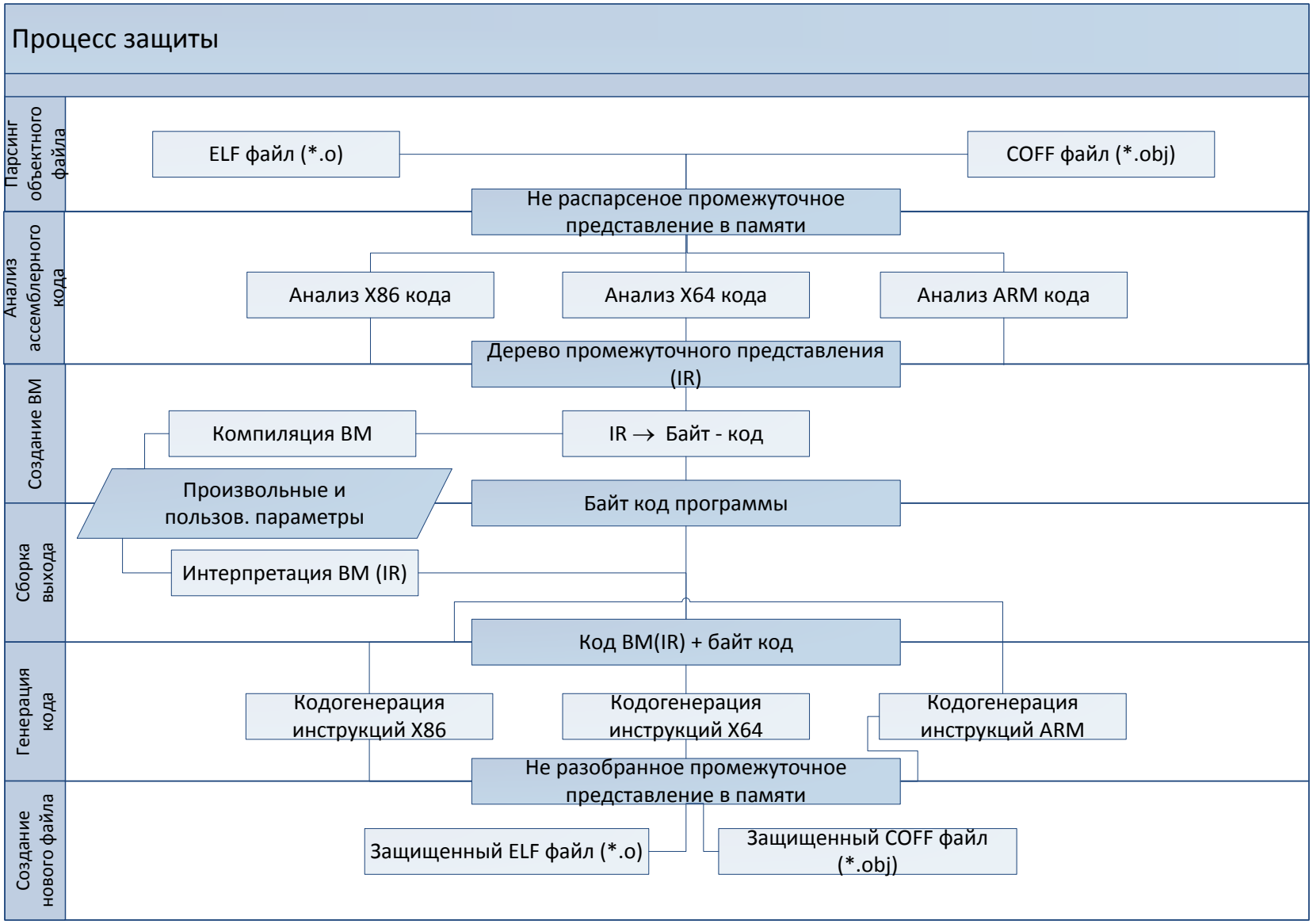

Рисунок 4.8 - Общая структура прототипа защиты

4.3.1 Модуль захвата машинного кода

Для захвата машинного кода из объектных файлов представленных в форматах COFF и ELF использовалась свободно доступная библиотека LLVMObject входящая в состав пакета LLVM версии 3.0. Захват кода, связанных с ним данных и перемещаемых символов осуществляется в модуле objectloader.cpp при помощи класса ObjectLoader. При этом широко используются интерфейсы, методы и функции из пространства имен llvm::object. К сожалению, в существующем состоянии «из коробки» модуль LLVMObject был не способен корректно определять размеры символов в объектных файлов, поэтому в процессе работы над прототипом, в данную библиотеку были внесены исправления и расширения функциональности, позволившие использовать модифицированную версию в рамках данного проекта.

4.3.2 Модуль создания выходного объектного модуля

К сожалению, не существует свободных библиотек позволяющих выполнять модификации объектных файлов, так, как это требовалось для данного проекта. Поэтому для реализации сохранения защищенного машинного кода в объектные файлы использовалась модифицированная библиотека objconv под авторством Agner Fog. К сожалению, лицензия под которой распространяется данная библиотека не позволяет использовать ее в коммерческих продуктах без публикации исходных кодов, поэтому для целей ОКР модули, ответственные за сохранение объектного кода, в частности, метод Save класса ObjectLoader, были полностью переработаны.

4.3.3 Описание интерфейса прототипа командной строки

Вне зависимости от используемой среды разработки (Microsoft Visual studio IDE, make files, CMake, scons и т.д.) разработчик имеет возможность напрямую влиять на процесс сборки приложения на всех его этапах: до компиляции модулей трансляции, перед осуществлением операции связывания отдельных модулей трансляции в исполняемый модуль и в конце, после получения исполняемого модуля. Единым модулем, к которому применяется защита, является объектный файл, порождаемый как результат трансляции исходного кода на языке высокого уровня при помощи компилятора.

Данный уровень работы механизма защиты позволяет удобным образом:

а) встраивать модуль защиты в существующий процесс сборки подлежащего защите пользовательского приложения при минимальной модификации стандартных процедур сборки,

б) динамически определять качество применения запутывающих преобразований к защищаемому коду при помощи опции «adebugPoison»

в) полностью контролировать все возможности защиты и иметь возможность их отключения/включения, как для ускорения работы защищаемого кода, так и для осуществления отладки самой системы защиты на этапе изготовления прототипа и разработки конечного продукта.

Картинка-подсказка, определяющая интерфейс командной строки:

```
VMGuardian CLI started
VMGuardian CLI parameters:

Basic options:
  --help                produce help message

General:
  --iFName arg          Object file name function for protection will be taken
                        from
  --oFName arg          Object to put result to
  --funcName arg        Function name to protect
  --targetOs arg        OS that target code will be executed on

Debug options:
  --irdump              Dump IR of translated function
  --vmirdump            Dump IR of VM interpreter

Protection options:
  --sPetri                          Use Petri network obfuscation
  --petriMaxNumTargets arg (=10)    Maximum number of constants to replace with
                                    calculated values
  --petriMinNumRounds arg (=5)      Minimum number of calculation rounds
  --sInsSwap                        Use instruction switch obfuscation
  --sADebug                         Anti-debugger obfuscation
  --adebugPoisonMin arg (=1)        Left border of random range for poison value
  --adebugPoisonMax arg (=10)       Right border of random range for poison value
  --adebugCallsAllowed              Allow external calls in time-controlled blocks
  --adebugTimer arg                 Method used to measure code execution time
  --adebugStraightChecks            Add random straight debugger checks
  --adebugCheckDist arg (=100)      Average number of instructions between
                                    debugger checks
```

4.3.4 Описание прототипа с графическим интерфейсом

Модуль защиты с графическим интерфейсом не пригоден для удобного и эффективного использования в процессе разработки, однако позволяет настроить оптимальные опции для защищаемого приложения максимально удобным образом, и эффективен в тех случаях, когда надо защитить малое количество исполняемого кода, или оператор не обладает эффективными навыками работ с командной строкой.

Прототип с графическим интерфейсом повторяет прототип инструмента защиты с интерфейсом командной строки с единственным отличием: функция для защиты, равно как и

объектный модуль для защиты должны быть выбраны при помощи графического манипулятора ввода типа «мышь» или клавиатуры интерактивным образом.

На рРисунок 4.9 представлен основной интерфейс в котором можно выбрать целевую ОС, защищаемую функцию, а также исходный и целевой файлы для защиты. На рРисунок 4.10 можно наблюдать различные возможные настройки защиты, которые можно применить к выбранному файлу.

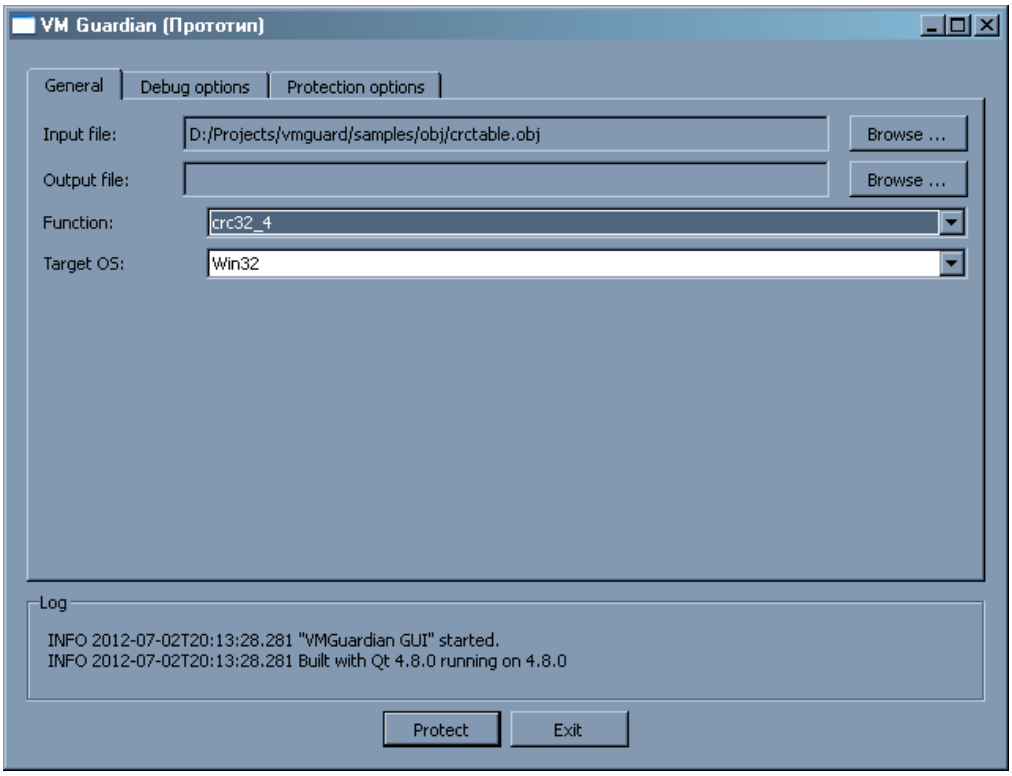

Рисунок 4.9 - Общие настройки прототипа с графическим интерфейсом

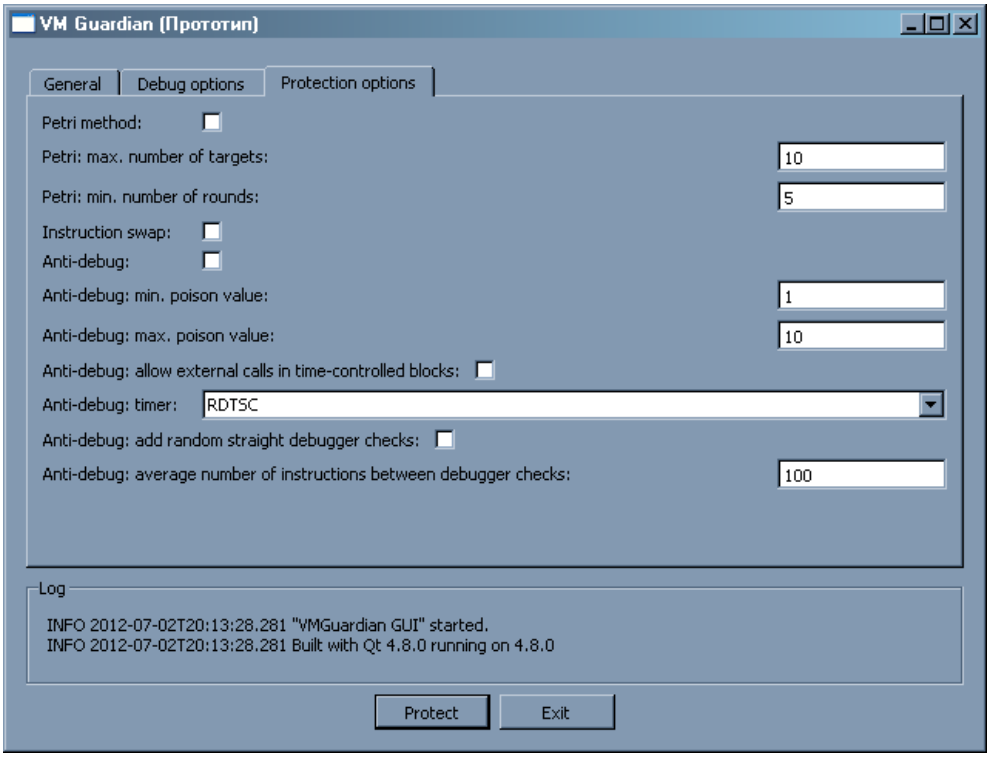

Рисунок 4.10 - Настройки защиты

Интерфейс на ОС Linux аналогичен интерфейсу под ОС Windows с незначительными модификациями косметического толка. Образец интерфейса под ОС Linux можно наблюдать на рРисунок 4.11. Данного результата удалось достигнуть благодаря использованию кроссплатформенной библиотеки (языка) QT.

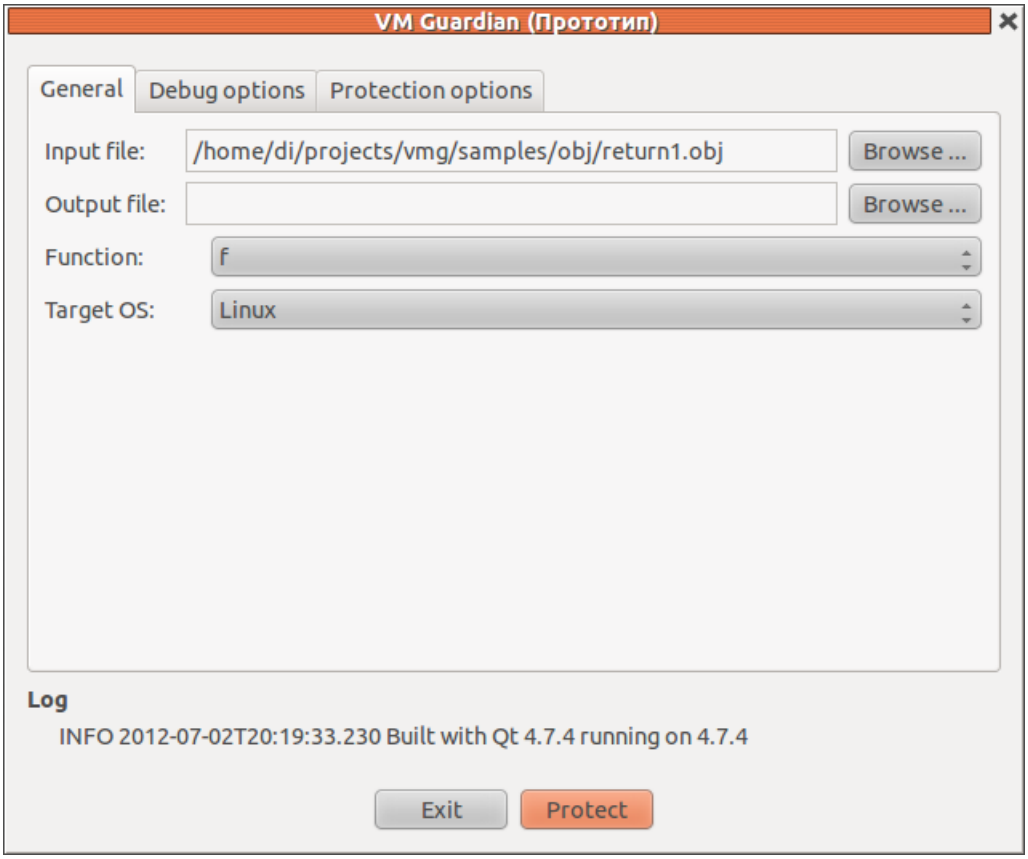

Рисунок 4.11 - Вид интерфейса под ОС Linux

4.3.5 Набор инструкций конкретной виртуальной машины

Рассмотрим набор инструкций, построенный для конкретной виртуальной машины генератором виртуальных машин. Он состоит из функционально – полного набора инструкций и тех инструкций, которыми мы его дополнили, чтобы их общее количество стало равным 200.

Расшифровка обозначений приводится в п. 2.2.

Функционально - полный набор инструкций

1) xor8ir i_imm, o_reg
2) not16r io_reg
3) and80mm i_mem, o_mem
4) add16im i_imm, o_mem
5) mul64im i_imm, o_mem
6) div32mrs i_mem, o_reg
7) div64rru i_reg, o_reg
8) mov32rr i_reg, o_reg
9) mov32mr i_mem, o_reg
10) mov16br i_brg, o_reg
11) mov16ir i_imm, o_reg
12) mov32rm i_reg, o_mem
13) mov8bm i_brg, o_mem
14) mov64mm i_mem, o_mem
15) mov32im i_imm, o_mem
16) mov16rb i_reg, o_brg
17) mov16ib i_imm, o_brg
18) mov32mb i_mem, o_brg
19) mov64bb i_brg, o_brg
20) cmp16rs32rs o_reg, i_reg
21) jle32rr_abs i_reg, j_reg
22) ja32bb_rel i_brg, j_brg
23) jge32mr_abs i_mem, j_reg
24) jb32rr_rel i_reg, j_reg
25) jne32mr_abs i_mem, j_reg
26) je32br_rel i_brg, j_reg

27) shr32rm i_reg, sh_mem
28) shl16mr i_mem, sh_reg
29) load16_8 reg, imm
30) store80_32 reg, imm

Используя первые три команды можно выразить все логические операций с произвольными типами, размерами и количеством операндов.

Инструкции 4 – 7 представляют функционально полный набор арифметических операций.

Инструкция 8 – 19 предоставляют функционально полный набор команд пересылки данных, благодаря которому можно не заботиться о типах операндов для других инструкций. Действительно, используя эти инструкции можно преобразовать типы операндов к тем, для которых есть соответствующая инструкция в построенном наборе.

Набор инструкций 20 - 26 является функционально полным набором для выражения всех команд перехода.

Инструкции 27 – 28 - набор позволяющий осуществлять побитовые сдвиги, в том числе и циклические.

И, наконец, 29 – 30 – инструкции для работы со стеком.

Таким образом, сформирован полный набор инструкций, достаточный для выполнения любой функции, не вызывающей другие функции (нет инструкции вызова процедур call).

Затем, этот набор пополняется до 200 инструкций.

31) jle32im i_imm, j_mem rel
32) add32rrr i_reg, i_reg, o_reg
33) xor64irr i_imm, i_reg, o_reg
34) or80rm i_reg, o_mem
35) sub80ir i_imm, o_reg
36) xor8br i_brg, o_reg
37) div8ir i_imm, o_reg
38) mul8ir i_imm, o_reg
39) and8im i_imm, o_mem
40) and16ib i_imm, o_brg
41) mov64ir i_imm, o_reg
42) jl32mi i_mem, j_imm abs
43) ja32rr i_reg, j_reg abs
44) sub32mr i_mem, o_reg

45) jl32ii i_imm, j_imm abs
46) je32mm i_mem, j_mem rel
47) je32im i_imm, j_mem rel
48) or80bb i_brg, o_brg
49) jg32br i_brg, j_reg rel
50) subsigned8ir i_imm, o_reg
51) jae32rm i_reg, j_mem abs
52) je32bi i_brg, j_imm abs
53) shl8ir i_imm, sh_reg
54) xor8rr i_reg, o_reg
55) subsigned32mr i_mem, o_reg
56) or32ib i_imm, o_brg
57) sub8mb i_mem, o_brg
58) jge32ir i_imm, j_reg abs
59) jl32mm i_mem, j_mem abs
60) jne32rr i_reg, j_reg reg
61) and32bb i_brg, o_brg
62) sub80bb i_brg, o_brg
63) mul80im i_imm, o_mem
64) jbe32mr i_mem, j_reg rel
65) add8ib i_imm, o_brg
66) jae32ii i_imm, j_imm abs
67) ja32mr i_mem, j_reg abs
68) addSigned8rr i_reg, o_reg
69) mov8ir i_imm, o_reg
70) add8im i_imm, o_mem
71) subunsigned32im i_imm, o_mem
72) mulSigned8mr i_mem, o_reg
73) and32bm i_brg, o_mem
74) or16br i_brg, o_reg
75) and16br i_brg, o_reg
76) jae32bm i_brg, j_mem rel
77) jge32mm i_mem, j_mem abs
78) jg32bm i_brg, j_mem abs

79) shr16ir i_imm, sh_reg
80) sub64br i_brg, o_reg
81) subsigned80mm i_mem, o_mem
82) mul8rm i_reg, o_mem
83) subsigned32ir i_imm, o_reg
84) jg32mm i_mem, j_mem reg
85) rol8mr i_mem, sh_reg
86) xor80mm i_mem, o_mem
87) jne32bi i_brg, j_imm abs
88) xor32br i_brg, o_reg
89) subsigned8rm i_reg, o_mem
90) jle32br i_brg, j_reg abs
91) jle32ir i_imm, j_reg abs
92) div8im i_imm, o_mem
93) jle32mm i_mem, j_mem abs
94) jae32rr i_reg, j_reg rel
95) rol64rm i_reg, sh_mem
96) jl32im i_imm, j_mem abs
97) jb32bi i_brg, j_imm abs
98) jne32mm i_mem, j_mem abs
99) jne32im i_imm, j_mem abs
100) subunsigned32rr i_reg, o_reg
101) jb32bm i_brg, j_mem abs
102) jge32ri i_reg, j_imm abs
103) jge32im i_imm, j_mem abs
104) shr16mm i_mem, sh_mem
105) ror8mm i_mem, sh_mem
106) mulSigned8rm i_reg, o_mem
107) jb32rr i_reg, j_reg abs
108) shr32rr i_reg, sh_reg
109) subsigned32im i_imm, o_mem
110) and8mm i_mem, o_mem
111) and16rr i_reg, o_reg
112) divSigned16mr i_mem, o_reg

113) xor8mb i_mem, o_brg
114) rol64mr i_mem, sh_reg
115) add8bb i_brg, o_brg
116) divSigned16ir i_imm, o_reg
117) or8rr i_reg, o_reg
118) jae32bi i_brg, j_imm abs
119) jg32ir i_imm, j_reg abs
120) ror8rm i_reg, sh_mem
121) subsigned80im i_imm, o_mem
122) div8mr i_mem, o_reg
123) rol32rr i_reg, sh_reg
124) subsigned8mr i_mem, o_reg
125) sub80br i_brg, o_reg
126) subunsigned8rr i_reg, o_reg
127) shr64rm i_reg, sh_mem
128) subsigned32rm i_reg, o_mem
129) jl32bi i_brg, j_imm abs
130) jne32ii i_imm, j_imm abs
131) subsigned64mr i_mem, o_reg
132) mov32ib i_imm, o_brg
133) subSigned8mr i_mem, o_reg
134) mov32ir i_imm, o_reg
135) divSigned8im i_imm, o_mem
136) jbe32br i_brg, j_reg absadd32mb i_mem, o_brg
137) mov80rb i_reg, o_brg
138) jle32mi i_mem, j_imm abs
139) div8rr i_reg, o_reg
140) mulSigned8mm i_mem, o_mem
141) subsigned16im i_imm, o_mem
142) add8mb i_mem, o_brg
143) jle32bi i_brg, j_imm abs
144) or80im i_imm, o_mem
145) rol16rr i_reg, sh_reg
146) rol32rm i_reg, sh_mem

147) subsigned8im i_imm, o_mem
148) and8mr i_mem, o_reg
149) subunsigned32ir i_imm, o_reg
150) jne32bm i_brg, j_mem rel
151) jle32ii i_imm, j_imm ab
152) addSigned8ir i_imm, o_re
153) xor32ib i_imm, o_brg
154) and8ib i_imm, o_brg
155) ror64rr i_reg, sh_reg
156) sub80rr i_reg, o_reg
157) mul8mm i_mem, o_mem
158) shr64mr i_mem, sh_reg
159) mul80ir i_imm, o_reg
160) subunsigned8rm i_reg, o_mem
161) jae32mm i_mem, j_mem rel
162) jb32ii i_imm, j_imm abs
163) shl64mr i_mem, sh_reg
164) ja32bi i_brg, j_imm abs
165) shr64mm i_mem, sh_mem
166) xor80rm i_reg, o_mem
167) xor8bm i_brg, o_mem
168) mulSigned16im i_imm, o_mem
169) or32br i_brg, o_reg
170) shr8rm i_reg, sh_mem
171) shr64rr i_reg, sh_reg
172) subsigned80mr i_mem, o_reg
173) sub8im i_imm, o_mem
174) mul16rm i_reg, o_mem
175) jbe32rm i_reg, j_mem abs
176) sub32ib i_imm, o_brg
177) subunsigned8ir i_imm, o_reg
178) jge32rm i_reg, j_mem abs
179) and80ir i_imm, o_reg
180) div32rm i_reg, o_mem

181) mul8rr i_reg, o_reg
182) jle32rm i_reg, j_mem abs
183) jl32rm i_reg, j_mem abs
184) and80im i_imm, o_mem
185) jle32im i_imm, j_mem abs
186) jae32bi i_brg, j_imm rel
187) jbe32ri i_reg, j_imm rel
188) subunsigned64mr i_mem, o_reg
189) jb32rm i_reg, j_mem rel
190) ja32im i_imm, j_mem rel
191) jb32mi i_mem, j_imm abs
192) jg32rm i_reg, j_mem abs
193) add80br i_brg, o_reg
194) subsigned64im i_imm, o_mem
195) ja32bm i_brg, j_mem abs
196) subunsigned16im i_imm, o_mem
197) jbe32ir i_imm, j_reg rel
198) jne32ii i_imm, j_imm rel
199) jbe32rr i_reg, j_reg abs
200) jb32ri i_reg, j_imm abs

ЗАКЛЮЧЕНИЕ

В ходе данной диссертационной работы решены следующие задачи:

- разработан метод защиты объектных и исполняемых файлов;
- разработан метод защиты кода, находящегося в динамически подключаемых библиотеках (DLL) и разделяемых объектах (.so);
- разработан прототип ПО, осуществляющего многоуровневую защиту кода при помощи сетей Петри, который будет обладать возможностью встраивания в стандартный процесс разработки программного обеспечения;
- решена задача противодействия отладке, как защищаемого кода, так и кода виртуальной машины с применением как общих методов противодействия отладке, так и с использованием уязвимостей конкретных отладчиков;
- решена задача предоставления возможности защиты уже защищённого кода, с рядом ограничений;
- предоставлена возможность анализа производительности защищенного кода;
- решена задача сжатия защищаемого кода, с целью уменьшения его размера на диске;
- предоставлен прототип «дружелюбного» графического интерфейса для осуществления защиты программного двоичного кода из-под графических оболочек Gnome и Windows GUI;
- изготовлены прототип программного инструмента, обеспечивающих многоуровневую защиту программного кода, с помощью которого была проведена оценка эффективности метода;

СПИСОК ЛИТЕРАТУРЫ

1. Shadow Market, 2011 [Электронный ресурс]// BSA global software piracy study, Ninth edition — 2012 — Режим доступа: http://portal.bsa.org/globalpiracy2011/ (дата обращения 30.05.2012)

2. Шевченко, А. Эволюция технологий самозащиты вредоносных программ. [Электронный ресурс]/А. Шевченко//SecureList — 2007 — Режим доступа: http://www.securelist.com/ru/analysis /204007553/Evolyutsiya_tekhnologiy_samozashchity_vredonosnykh_programm (дата обращения 30.05.2012).

3. Чернов, А. В. Анализ запутывающих преобразований программ[Электронный ресурс]/А.В. Чернов. //Библиотека аналитической информации. — Труды Института Системного программирования РАН 2003 — Режим доступа: http://www.citforum.ru/security/articles/analysis/ (дата обращения 30.05.2012).

4. Лифшиц Ю. М. Запутывание (обфускация) программ. Обзор [Электронный ресурс]/Ю.М. Лившиц//— СПб. отд. Мат. институт им. В.А. Стеклова РАН, 2004. — Режим доступа:http://logic.pdmi.ras.ru/~yura/of/survey1.pdf (дата обращения 30.05.2012).

5. Collberg, C., Thomborson, C., Low, D.. Taxonomy of Obfuscating Transformations. [Электронный ресурс]/ C. Collberg//University of Arizona — 1997 — Режим доступа:http://www.cs.arizona.edu/collberg/Research/Publications/CollbergThomborsonLow97a/index.html (дата обращения 30.05.2012).

6. Бойко В. Метод виртуального процессора в защите программного обеспечения/В. Бойко— Труды Института системного программирования РАН, 2005.

7. Раструсный, В., Защита Win32 и .NET приложений: обзор протектора Themida (X-Protector) [Электронный ресурс] / Раструсный В.// — 2012 — Режим доступа:http://habrahabr.ru/post/106920/ (дата обращения 30.05.2012).

8. Утилита автоматической защиты исполняемых файлов[Электронный ресурс]// ЗАО "Секьюлэб" — 2012 — Режим доступа:http://senselock.ru/projects/senselock-vmprotect.php (дата обращения 30.05.2012).

9. Low Level Virtual Machine // University of Illinois at Urbana-Champaign — 2012 — Режим доступа:http://ru.wikipedia.org/wiki/Low_Level_Virtual_Machine (дата обращения 30.05.2012).

10. Машина Тьюринга [Электронный ресурс] — 2012 — Режим доступа:http://ru.wikipedia.org/wiki/Машина Тьюринга (дата обращения 30.05.2012).

11. Буздин, К.В. Исполнение моделей при помощи виртуальной машины. [Электронный ресурс]/ К.В. Буздин // Труды Института Системного Программирования РАН — 2004 —. Режим доступа:http://citforum.ru/SE/project/models/ (дата обращения 30.05.2012).

12. Микропроцессор. [Электронный ресурс] // Викизнание — 2012 — Режим доступа:http://www.wikiznanie.ru/ru-wz/index.php/Микропроцессор (дата обращения 30.05.2012).

13. Ершова Н.Ю., Ивашенков О.Н., Курсков С.Ю. Архитектура микропроцессора. Режимы адресации. [Электронный ресурс]/ Ершова Н.Ю., Ивашенков О.Н., Курсков С.Ю// Петрозаводский государственный университет — 2004 — http://dfe3300.karelia.ru/koi/posob/microcpu/vved.htm Режим доступа: (дата обращения 28.05.2012).

14. Парфенов П.С., История и методология информатики и вычислительной техники:

Учебное пособие/ П.С. Парфенов // Сибирский федеральный университет — 2010.

15. З.М. Каневский, В.П. Литвиненко, Г.В. Макаров, Д.А. Максимов. Теория скрытности: Учеб. пособие. / Под ред. З.М. Каневского. Воронеж: Воронеж. гос. техн. ун-т, 2006. 211 с.

16. Sequin, C.H. , Patterson, D.A., Design and Implementation of RISC/ C.H. Sequin, D.A. Patterson// University of California, — 1982.

17. Stallings W., Computer Organization and Archtecture/W Stallings// Universidade do Minho — 2000.

18. Severance, C., Dowd K., High Performance Computing", 2nd Edition/ C. Severance, K. Dowd// O'Reilly Media — 1998.

19. Аппаратные средства микропроцессора. Длина команд. [Электронный ресурс]// Электронный ресурс «История компьютерной техники» — 2012 — Режим доступа: http://www.exicomputers.ru/ustroistvo/microprocessori6.html (дата обращения 30.05.2012).

20. Авдошин С.М.,СавельеваА.А.Алгоритм решения систем линейных уравнений в кольцах вычетов/ С.М. Авдошин // Информационныетехнологии. 2006. No2. с.50-54

21. Control flow. [Электронный ресурс]// Фонд «Викимедиа» — 2012 — Режим доступа:http://en.wikipedia.org/wiki/Control_flow (дата обращения 30.05.2012).

22. Мишунин, В. В. Микропроцессоры и цифровая обработка сигналов : Учебно-методическое пособие : Электронный ресурс / В.В. Мишунин, П.Г. Лихолоб //БелГУ. - Белгород : Изд-во БелГУ — 2010. — 210 с.

23. Новиков, Ю.В., Скоробогатов П.К., Основы микропроцессорной техники. Лекция 5: "Система команд процессора".[Электронный ресурс]/ Ю.В. Новиков, П.К. Скоробогатов // НОУ «ИНТУИТ» — 2012 — Режим доступа:http://www.intuit.ru/department/hardware/mpbasics/5/ (дата обращения 30.05.2012).

24. Стасюк В.В. Функции алгебры логики: Краткое учебное пособие./ В.В. Стасюк // Владивосток: Изд-во ДВГТУ— 2004. — 44 с.

25. Fiñones, R.G., Fernandez, R., Solving the Metamorphic Puzzle./ R.G. Fiñones R. Fernandez. // Virus Bulletin, — 2006. — стр. 14-19

26. Котов В. Е. Сети Петри/ В. Е. Котов//М.: Наука. Главная редакция физико-математической литературы — 1984 — 160 с.

27. Four F., Защита программ с помощью сетей Петри./ F. Four //Портал для программистов «ВАСМ» — 2011 — Режим доступа: http://www.wasm.ru/article.php?article=petri (дата обращения 22.10.2011).

28. Peterson, J. L., Petri net theory and the modeling of systems./ J. L. Peterson //Prentice-Hall, Englewood cliffs — 1981 — 290 стр.

29. The LLVM Compiler Infrastructure documentation [Электронный ресурс] — 2012 — Режим доступа: http://llvm.org/ (дата обращения 04.03.2012).

30. BeaEngine official documentation [Электронный ресурс] —2011 — Режим доступа: http://www.beaengine.org/ (дата обращения 14.09.2011).

31. LLVM Pass classes documentation [Электронный ресурс] — 2011 — Режим доступа: http://llvm.org/docs/WritingAnLLVMPass.html (дата обращения 11.07.2011).

32. Jackson J., An Anti-Reverse Engineering Guide. [Электронный ресурс]/J.Jackson — 2008— Режим доступа: http://www.codeproject.com/KB/security/AntiReverseEngineering.aspx (дата обращения 05.07.2011).

33. Спецификация функции IsDebuggerPresent().[Электронный ресурс] — 2012— Режим доступа: http://msdn.microsoft.com/en-us/library/windows/desktop/ms680345(v=vs.85).aspx (дата обращения 29.09.2011).

34. Raz I., Anti Debugging Tricks. [Электронный ресурс]/ I. Raz// — 2011— Режим доступа: http://www.intel-assembler.it/portale/5/anti-debug-tricks/anti-debug-tricks-release2.asp (дата обращения 2.08.2011).

35. Ferrie P., Anti-Unpacker Tricks [Электронный ресурс]/ P. Ferrie// — 2012— Режим доступа: http://pferrie.tripod.com/papers/unpackers.pdf (дата обращения 04.02.2012).

36. Hong, S.; Hong, D.; Ko, Y.; Chang, D.; Lee, W.; Lee, S., Differential cryptanalysis of TEA and XTEA/ S. Hong; D. Hong; Y. Ko; D. Chang; W. Lee; S. Lee, // 6th International Conference, Seoul, Korea — 2003 — стр. 402-417.

37. Варновский Н.П. О стойкой обфускации компьютерных программ /Н.П. Варновский, В.А. Захаров, Н.Н. Кузюрин, А.В. Шокуров // Научные ведомости БелГУ. Сер. История. Политология. Экономика. Информатика. — 2009—№ 12-1. стр. 97–104