МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ (ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ)

РАЗРАБОТКА КРОССПЛАТФОРМЕННОГО ДИНАМИЧЕСКОГО АНАЛИЗАТОРА БИНАРНОГО КОДА НА ОСНОВЕ QEMU

Дипломная работа студента 919 группы ФРТК
Перова Максима Николаевича

Научный руководитель: кандидат военных наук,
доцент Семенихин Игорь Викторович

Рецензент:
Фонин Юрий Николаевич

Москва 2015

Оглавление

Введение.

С развитием информационных технологий значительно выросла сложность программного обеспечения, используемого в информационных системах: программы стали более динамическими, а их поведение теперь может быть оценено только в процессе выполнения. При этом степень безопасности приложений оценить сложнее, чем ранее. Из этого следует, что использование только статических подходов анализа программного обеспечения становится невозможным. Например, из-за динамически генерируемого кода возможность гарантировать полное покрытие кода при статическом анализе сводится к нулю. Такие темпы развития программного обеспечения требуют использования инструментов динамического анализа, который осуществляется в отношении ПО, выполняющегося в момент проведения анализа на реальном или виртуальном процессоре.

Кроме того, в деятельности специалистов в области защиты информации возникают ситуации, когда требуется изучить программное обеспечение без исходных кодов. Можно привести следующие примеры: во-первых, это проверка коммерческого ПО, которое не поставляется с исходным кодом, на предмет недекларированных возможностей (НДВ); во-вторых, проведение сертификации предоставленного софта. Следовательно, работа может происходить непосредственно с бинарным файлом.

Для реализации динамического анализа идеально подходит технология, называемая динамической бинарной инструментацией (Dynamic Binary Instrumentation, DBI). Суть этой технологии заключается во вставке в бинарный исполняющийся код анализирующих процедур. Вообще, инструментация – это процесс модификации исследуемой программы с целью её дальнейшего анализа.

За вставку дополнительного кода обычно отвечают инструментирующие процедуры, которые вставляются единожды, а вызываются каждый раз при достижении участка кода, в который была произведена вставка. Таким образом, можно перехватывать и обрабатывать события возникновения исключений, запуск системных вызовов, порождение потоков, вызов функций и т.п.

На сегодняшний день существует множество фреймворков, реализующих DBI. Самыми популярными из них являются PIN, Valgrind, TEMU, DECAF. Однако все они обладают общим недостатком: число поддерживаемых ими архитектур сложно расширить, и в большинстве случаев они ограничиваются поддержкой x86 и x86-64. А как известно, к настоящему моменту набирают популярность те устройства, архитектура которых отличается от x86, например, среди мобильных устройств ARM занимает лидирующие позиции на рынке.

В последнее время можно заметить растущие масштабы продвижения идеи «умный дом» в жизнь рядовых потребителей. Данная идея воплощается в различных «Интернет-вещах», находящихся под управлением микропроцессоров и подключенных к сети Интернет. Вместе с ростом количества и разновидностей самих устройств растет и количество архитектур, которые в них используют. Но суть проблемы находится в том, что в настоящее время рынок наполнен продукцией иностранных производителей, а получение от них исходного кода прошивок представляется едва возможным. Следовательно, анализ прошивки можно осуществить только посредством анализа бинарного файла. В данном случае анализ крайне необходим, так как при наличии НДВ в прошивке «умного устройства» мы можем обнаружить все, что угодно. Поэтому уже сейчас специалистам по защите информации требуются такие кроссплатформенные анализаторы кода, в которых будет обеспечена возможность добавления поддержки новых архитектур.

Подобный инструмент может быть построен на основе открытой программы-эмулятора QEMU, поддерживающей множество архитектур. В основе QEMU лежит принцип двоичной трансляции, который делает эмулятор довольно быстрым и масштабируемым. Инструкции гостевой архитектуры представляются в промежуточном коде. Сначала данный код оптимизируется, а затем преобразуется в инструкции машины, на которой работает QEMU. TCG (Tiny Code Generator) является основным модулем, выполняющим данные преобразования. Таким образом, на стадии оптимизации промежуточного кода существует возможность внедрить дополнительные внешние процедуры, которые и будут отвечать за инструментацию кода.

Цель дипломной работы – создание инструмента динамического анализа бинарного кода, поддерживающего множество архитектур, встраивание новой архитектуры в который будет происходить в полуавтоматизированном режиме. Для достижения поставленной цели требуется решить следующие задачи:

1. Исследовать инструменты, основанные на технологии DBI.
2. Разработать инструмент идентификации переполнения буфера в стеке в бинарном исполняемом файле.
3. Разработать технологию автоматизации процесса встраивания новой архитектуры в QEMU.

Актуальность работы определяется тем, что в настоящее время требуются инструменты динамического анализа кода, которые поддерживают архитектуры, отличные от x86 и x86-64 и способные осуществлять анализ как операционной системы и прошивки устройства, так и любого другого программного обеспечения.

Новизна – заключается в предложении реализации кроссплатформенного динамического анализатора кода, которая позволяет расширять число поддерживаемых архитектур в полуавтоматизированном режиме.

Глава 1. Динамический анализ кода

Часть 1. Общие термины и определения

Для того, чтобы реализовывать инструмент на базе какой-то технологии, прежде всего нужно детально ее изучить. В том числе стоит выделить основные употребляемые термины.

Статический анализ кода – анализ программного обеспечения, проводимый без выполнения исследуемых программ.

Динамический анализ кода – анализ программного обеспечения, осуществляемый в отношении ПО и выполняемый в момент проведения анализа на реальном или виртуальном процессоре.

Инструментация – процесс модификации исследуемой программы с целью её дальнейшего анализа.

Кроссплатформенное программное обеспечение – программное обеспечение, работающее более чем на одной аппаратной платформе и/или операционной системе.

Трасса приложения – список выполненных инструкций программы.

Байт-код – машинно-независимый код низкого уровня, генерируемый транслятором и исполняемый интерпретатором или виртуальной машиной.

Самомодифицируйщийся код – программный приём, при котором приложение создаёт или изменяет часть своего программного кода во время выполнения.

Часть 2. Сравнение статического и динамического анализа

Исходными данными для проведения анализа программы могут быть не только текстовые исходные коды, но и скомпилированные бинарные файлы. Однако чаще всего специалистам по информационной безопасности приходится сталкиваться именно со вторым вариантом. Исходя из этого утверждения, можем сделать вывод, что анализ бинарных исполняемых файлов более востребован.

Использование статического анализа приводит к возникновению целого ряда проблем. Например, если исследуемая программа использует динамически генерируемый код, то по исходным кодам мы не сможем увидеть полную картину ее работы. Более того, программы все чаще собираются и определяются во время их исполнения, используя виртуальные функции, динамически генерируемый код и другие возможности высокоуровневых языков программирования.

Следовательно, инструменты, основанные на статическом анализе, не способны обеспечить полного покрытия кода, хотя ранее это было их неоспоримым преимуществом. В связи с этим появляется явная необходимость в динамическом подходе для проведения анализа исследуемой программы.

Для того, чтобы подчеркнуть основные различия двух типов анализа, мы можем привести наглядную таблицу сравнения статического и динамического подходов при отсутствии исходного кода.

Таблица 1.1. Сравнение статического и динамического подхода при отсутствии исходного кода.

Критерий	**Статический анализ**	**Динамический анализ**
Покрытие кода	Большое покрытие кода	Выполненные пути
Самомодифицирующийся код	Не анализируется	анализируется
Взаимодействие с ОС	Не учитывается	Учитывается
Определенность значений параметров	Отсутствует	Вся информация известна
Проблема отличия кода от данных	Нерешаема	Отсутствует
Неиспользуемый код	Анализируется	Не анализируется

Из таблицы 1.1 мы видим, что именно динамический подход более удобен для проведения анализа программы без исходного кода.

Часть 3. Этапы динамического анализа

Динамический анализ делится на три основных этапа:

1) инструментация приложения;
2) профилирование приложения;
3) анализ полученной информации.

Первый этап отвечает за подготовку приложения к профилированию. Сразу же после профилирования совершается выполнение, во время которого собирается информация о работе приложения. Накопленная информация представляет собой трассу, которая затем обрабатывается.

Часть 4. Виды инструментации

Инструментацию можно проводить с исходным кодом или без исходного кода. Наиболее интересным для рассмотрения является второй вариант, поэтому разбор инструментации с исходным кодом мы можем пропустить. Рассматривая инструментацию без исходного кода, отмечаем, что исходные данные здесь могут быть представлены в двух видах: байт-коде или исполняемом бинарном файле.

Байт-код обычно встречается при анализе программ, написанных на Java/C#, а бинарный исполняемый файл – при анализе программ, написанных на компилируемых языках программирования, то есть когда компилятор переводит исходный код в машинные команды.

Таким образом, инструментацию классифицируем следующий образом:

Исполнение байт-кода происходит следующим образом: байт-код загружается в виртуальную машину, в которой переводится в машинные инструкции. Последние выполняются в операционной системе.

Из описанной схемы легко понять, что инструментация байт-кода может совершаться на следующих стадиях:

1) до запуска;
2) во время загрузки байт-кода в виртуальную машину;
3) во время выполнения.

В первом случае возможна статическая инструментация, при которой файл с байт-кодом инструментируется до загрузки в виртуальную машину. Например, создается копия файла, инструментируется, а затем уже отправляется на исполнение. Во втором – в процессе загрузки после реализации собственного загрузчика байт-кода в виртуальную машину, им же и будет проводиться инструментация кода. В последнем случае возможна динамическая инструментация, при которой байт-код уже загружен, возможно, запущен и находится в процессе модификации.

Для того, чтобы проводить анализ бинарного исполняемого файла, можно либо влиять непосредственно на него, либо на его программное окружение в процессе работы. Если мы выбираем первый способ, то единственный возможный вариант развития событий – оперирование с копией файла. Таким образом, нам придется вставить инструментирующий код и отправить полученную копию на исполнение.

Итак, отсюда мы видим, что для анализа бинарного исполняемого файла может быть использована не только статическая бинарная инструментация, но и динамическая.

Часть 5. Статическая и динамическая бинарная инструментация

Статическая бинарная инструментация состоит из следующих этапов: вставки, удаления или изменения определенной функциональности в бинарном исполняемом файле и сохранении исходного результата в виде отдельной копии, модифицированный файл которой в дальнейшем можно отправлять на исполнение.

При данном подходе инструментация проводится только один раз, а используется многократно. В данном случае накладные расходы во время выполнения программы связаны только с выполнением вставленного кода, что является положительным моментом в использовании статической бинарной инструментации.

Однако у статического подхода есть и некоторые недостатки. Во-первых, это проблема отличия данных от кода, так как инструментация производится до выполнения программы. Во-вторых, разделяемые библиотеки должны быть инструментированы по-отдельности. Стоит подчеркнуть, что у динамического подхода приведенные выше минусы отсутствуют.

Теперь перейдем к рассмотрению второго способа инструментации. Динамическая бинарная инструментация выполняется во время работы программы и состоит из нескольких базовых этапов:

1) получение управления от приложения;
2) сохранение состояния программы;
3) выполнение задач;
4) восстановление состояния программы;
5) возвращение управления приложению.

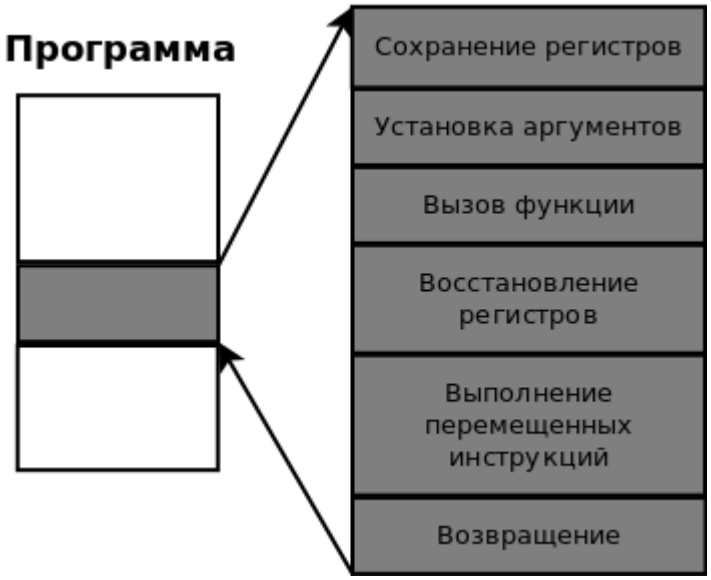

Рисунок 1.1. Вставка кода в запущенную программу.

Таким образом, следующие недостатки и преимущества динамической бинарной инструментации могут быть выделены. И те, и другие перечисляются в порядке убывания их важности.

Преимущества:

1) обработка динамически генерируемого кода;
2) обнаружение кода во время выполнения;
3) нет необходимости в перекомпиляции и перелинковке;
4) анализ всего приложения как единого целого.

Недостатки:

1) сложность реализации;
2) увеличение накладных расходов.

Во время динамической бинарной инструментации увеличиваются накладные расходы. Это связано с тем, что параллельно вставке анализирующих

функций, выполняются задачи разбора, дизассемблирования, генерации кода и т.п. Данные недостатки со временем становятся все менее заметными, так как снижение производительности устраняется использованием кэш-памяти и оптимизирующими алгоритмами.

Вывод по главе 1

В данной главе рассмотрены статические и динамические методы анализа кода. Мы не можем с точностью утверждать о преимуществах одного способа анализа перед другим, так как каждый имеет свои достоинства и недостатки. Также нам удалось выяснить, что существует ряд определенных случаев, таких как самомодифицирующийся код, использование внешних библиотек и т.д., для которых единственно возможным вариантом является применение только динамического анализа кода. Его основной этап, как мы выяснили, – инструментация. Последняя, в свою очередь бывает двух типов: статической и динамической. Несмотря на то, что оба типа имеют свои достоинства и недостатки, второй мощнее, из чего следует, что динамическая бинарная инструментация способна решить больший круг задач, нежели статическая.

Глава 2. Обзор существующих средств динамической бинарной инструментации

Часть 1. PIN

К настоящему моменту существует множество различных инструментов, умеющих работать не только с исходным кодом, но и без оного. Еще одно отличие этих инструментов состоит в том, что они проводят разные типы анализа. Но так как нас не интересуют случаи, когда имеется исходный код и применяются статические приемы анализа, мы рассмотрим только средства для динамической бинарной инструментации.

Для нашего исследования наиболее интересным вариантом являются DBI фреймворки, с помощью которых создаются динамические бинарные анализаторы. Самым популярным фреймворком является PIN, который поддерживает архитектуры x86 и x86-84. Этот фреймворк также позволяет создавать инструменты под Linux и Windows. Библиотека даёт исследователю возможность вставить произвольную процедуру, написанную на языках высокого уровня C или C++, в произвольный участок бинарного кода. Перед выполнением вставленного кода полностью сохраняются все данные выполняемого процесса, например, состояние регистров. И в данном случае работа программы не нарушается.

Одна из отличительных особенностей PIN – вполне обширный API, примеры использования которого предоставляются вместе с библиотекой. На сайте Intel мы можем найти подробное описание всех примеров. Все это делает PIN достаточно простым в использовании фреймворком. Следуя инструкциям с сайта, достаточно легко научиться им пользоваться. Из недостатков же следует отметить, что PIN является проприетарным приложением с закрытыми исходными

кода, а это значит, что встраивать новую архитектуру могут лишь разработчики компании Intel.

Существует некоторое количество проектов, относящихся к безопасности и использующих PIN. Из них стоит отметить следующие: VERA служит для визуализации работы программы; Kerckhoffs – для детектирования криптографических примитивов; Tripoux – анализатор упаковщиков; а Privacy Scope ищет утечки критичной информации. И это далеко не полный список. Все упомянутые выше проекты имеют открытый исходный код, а это дает возможность изучить их, а также написать собственные инструменты.

Часть 2. Valgrind

Кроме уже известного нам фреймворка PIN, существует еще один не менее мощный – Valgrind. Valgrind – это система динамического профилирования с последующим анализом, которая может выполнять на следующих операционных системах:

1) Linux на платформах x86, x86-64, ARM, PowerPC, MIPS;
2) Android на платформах x86, ARM, MIPS;
3) Mac OS X.

Valgrind немного уступает PIN в плане платформ, так как не может быть запущена из-под операционных систем семейства Windows. Однако по сравнению с PIN, данная система поддерживает гораздо больший список архитектур.

Еще одно неоспоримое преимущество Valgrind состоит в том, что это свободно распространяемое программное обеспечение, т.е. любой желающий имеет возможность не только изучить работу данной среды, но и дополнить ее.

Более того, открытые исходные коды позволяют детально разобраться в продукте. Таким образом, пользователь приобретает прекрасный опыт перед тем, как написать собственный анализатор.

Теперь перейдем непосредственно к рассмотрению «внутренностей» Valgrind. В среде Valgrind анализируемая программа не выполняется напрямую. Вместо этого она динамически транслируется в промежуточное представление Valgrind – VEX. Промежуточное представление является унифицированной и независящей от целевой платформы. Анализаторы, за основу которых взят Valgrind, выполняют необходимые преобразования над полученным IR, после чего IR транслируется обратно в машинный код. Таким образом, выполнение программы в рамках Valgrind можно представить на рисунке 2.1.

Рисунок 2.1. Сравнительная схема выполнения программы с Valgrind и без Valgrind.

Итак, каждый исполняемый базовый блок транслируется в IR. Данная процедура занимает пять этапов:

1) представление исполняемого кода в виде двухадресных инструкций Valgrind, используя виртуальные регистры;
2) оптимизация IR;

3) внесение инструментального кода, определяемого анализатором;
4) отображение виртуальных регистров, используемых в IR, на 6 аппаратных регистров: eax, ebx, ecx, edx, esi, edi.
5) конвертация каждой инструкции IR в одну или несколько инструкций целевой архитектуры.

Но Valgrind не может транслировать код ядра операционной системы, поэтому системные функции выполняются строго по алгоритму:

1) сохранение стека анализируемой программы;
2) копирование значений из памяти на аппаратные регистры;
3) исполнение системной функции;
4) копирование значений аппаратных регистров после системного вызова в память;
5) восстановление стека анализируемой программы.

Среди самых известных модулей Valgrind стоит отметить Memcheck. Он выявляет попытки использования неинициализированной памяти, утечку памяти, чтение/запись в память после её освобождения и другие ошибки.

Стоит также упомянуть модуль TaintCheck, так как наибольший интерес для нас представляют проекты, полезные для специалистов по информационной безопасности. Этот модуль анализирует «помеченные» данные, в частности, отслеживание потоков данных, полученных из внешних источников. Такой анализ называется Dynamic Taint Analysis (DTA), который детально рассмотрен в третьей части второй главы.

Часть 3. Существующие инструменты динамического анализа бинарного кода основе QEMU

Так как существующие распространенные решения динамической бинарной инструментации не являются полноценными эмуляторами, осуществить запуск полноценной операционной системы из-под них невозможно. Анализ прошивок «Интернет-вещей» также становится невозможным. Следовательно, стоит рассмотреть средства, построенные на полноценных эмуляторах с исходными кодами, которые поддерживают множество архитектур. Самый мощный из таких эмуляторов – QEMU.

Группа исследователей в университете Беркли в рамках проекта BitBlaze создала инструмент динамического анализа, который был назван TEMU. TEMU анализирует состояние программ и операционной системы непосредственно во время их работы на виртуальной машине. Во время анализа дополнительно извлекается семантика уровня ОС – данные о процессах, нитях, загруженных модулях и именах функций. Внутреннее устройство TEMU изложено на рисунке 2.2.

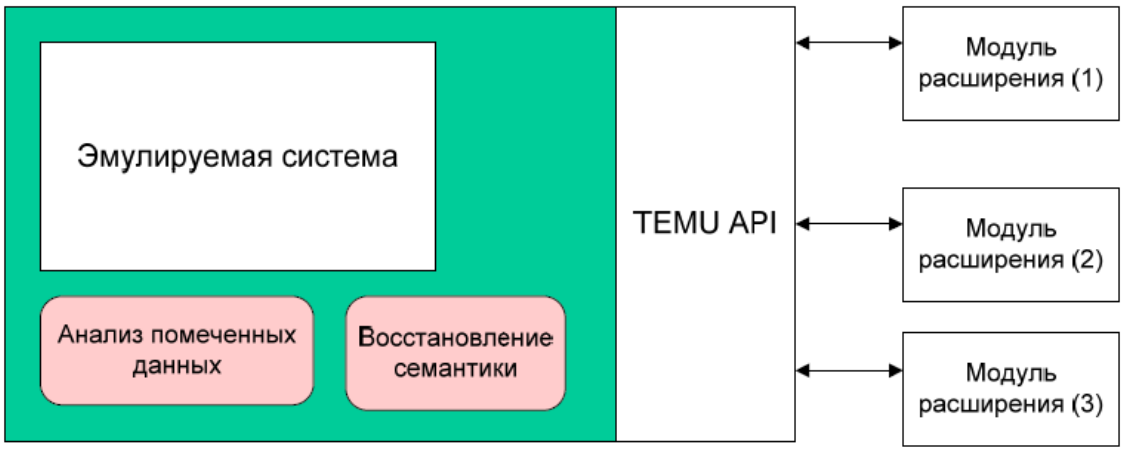

Рисунок 2.2. Компоненты TEMU.

С помощью TEMU можно осуществлять снятие бинарной трассы уровня машинных инструкций, а также создавать собственные модули на основе предоставляемого API.

Одна из ключевых возможностей TEMU – анализ помеченных данных, который выполняется во время работы системы. Для каждого помеченного байта создается структура данных, где хранится информация о месте использования данного байта. Пользовательский ввод, сетевые пакеты, данные жесткого диска, а также высокоуровневые абстрактные объекты, например, результаты выполнения функции могут выступать в качестве источников помеченных данных.

Трасса выполнения программы или операционной системы является предварительным результатом анализа. Для получения итогового результата необходима разность трассы с уязвимостью и без нее. DTA, упомянутый в предыдущей части, является таким анализом.

В отличие от PIN и Valgrind, TEMU обладает возможностью проводить анализ операционной системы. Однако данное средство несовершенно, так как не является кроссплатформенным, то есть поддерживает ограниченное количество архитектур. С добавлением архитектур также возникают проблемы, так как усилия для произведения данной процедуры сопоставимы с реализацией отдельного средства динамического анализа.

Другой недостаток TEMU (даже без плагинов) – ощутимое замедление работы гостевой ОС по сравнению с оригинальной версией QEMU. Кроме этого, TEMU использует закрытую библиотеку, выполняющую роль «драйвера» при взаимодействии с операционной системой.

Существует ещё один проект на основе QEMU – DECAF (Dynamic Executable Code Analysis Framework). Вместе с тем, этот проект является преемником TEMU. Отличие TEMU и DECAF состоит в том, что у последнего нет зависимости от закрытой библиотеки-драйвера. Также он обладает большим

количеством поддерживаемых архитектур. В DECAF добавлены модули анализа операционных систем, а также увеличена производительность от проведения DTA. Кроме того, преемник TEMU, в отличие от него самого, продолжает свое развитие в рамках упомянутого проекта BitBlaze.

Однако не стоит забывать о том, что оба инструмента имеют некоторые трудности при запуске на различных операционных системах. Например, на старой версии ядра операционной системы Linux анализаторы на основе DECAF запускаются, а на более новой – нет.

Вывод по главе 2

Во второй главе мы детально изучили и рассмотрели существующие на настоящий момент средства для создания динамических анализаторов динамического кода такие, как PIN и Valgrind. Наибольший интерес из них, благодаря открытым исходным кодам и большему количеству поддерживаемых архитектур, представляет Valgrind. Однако ни один из двух инструментов не способен анализировать целые ОС или прошивки устройств. Это является большим недостатком для проведения анализа набирающих популярность «Интернет-вещей». Кроме этого, мы также рассмотрели инструменты динамического анализа бинарного кода, за основу которых взяты преемники QEMU – TEMU и DECAF. Они, напротив, применимы для анализа «Интернет-вещей», но не являются масштабируемыми в плане архитектур.

Глава 3. Эмулятор QEMU

Часть 1. Краткий обзор QEMU

QEMU – это приложение с открытым исходным кодом, способное полностью эмулировать персональный компьютер. QEMU имитирует не только процессор, но и все необходимые подсистемы, в числе которых сетевые, звуковые карты и другое периферийное оборудование. Именно поэтому QEMU используется для виртуализации гостевой операционной системы или в качестве полнофункционального машинного эмулятора, который запускает операционные системы, предназначенные для процессора хост-системы или других процессорных архитектур.

Список поддерживаемых QEMU платформ довольно широк: x86, x86-64, SPARC32, ARM, MIPS, PowerPC, m68k. То есть в режиме запуска полноценной операционной системы данные архитектуры доступны в QEMU. Запуск самого эмулятора возможен и на следующих платформах: x86, x86-64 и PowerPC. В ближайшем будущем этот список будет расширяться, так как в сейчас проводятся исследования и тестирования работы эмулятора на ARM, SPARC32, Itanium, SPARC64 и др.

Одним из ключевых критериев качества эмулятора является быстродействие. Для обеспечения максимальной скорости работы применяется бинарная трансляция, которая как раз используется в QEMU. Более того, QEMU поддерживает аппаратную виртуализацию для увеличения производительности, и тем самым дает возможность большинству гостевых операционных систем выполняться непосредственно на x86 и x86-64 процессорах с поддержкой аппаратной виртуализации Intel VT или AMD-V. Такая поддержка основана на

модуле ядра KVM, который может работать в операционной системе Linux на платформах x86 и x86-64.

Часть 2. Принципы работы QEMU

Как уже говорилось выше, QEMU – это программа-эмулятор, имитирующая достаточно большое количество архитектур. Воспроизведенные архитектуры QEMU называются target (целью). Архитектура ЭВМ, на которой работает QEMU, называется host (хостом). Перевод виртуального машинного кода в код хоста выполняется модулем в QEMU, имеющим название «tiny code generator» или сокращенно TCG. В рамках TCG, термин «target» получает другое значение. TCG создает код, чтобы имитировать target архитектуру, который и является target кодом в терминологии TCG. Таким образом, при вхождении в TCG, target обозначает сформированный код хоста. А код, который выполняется на эмулируемом процессоре, называется гостевым кодом. Для полного понимания меняющейся терминологии приведена схема на рисунке 3.1.

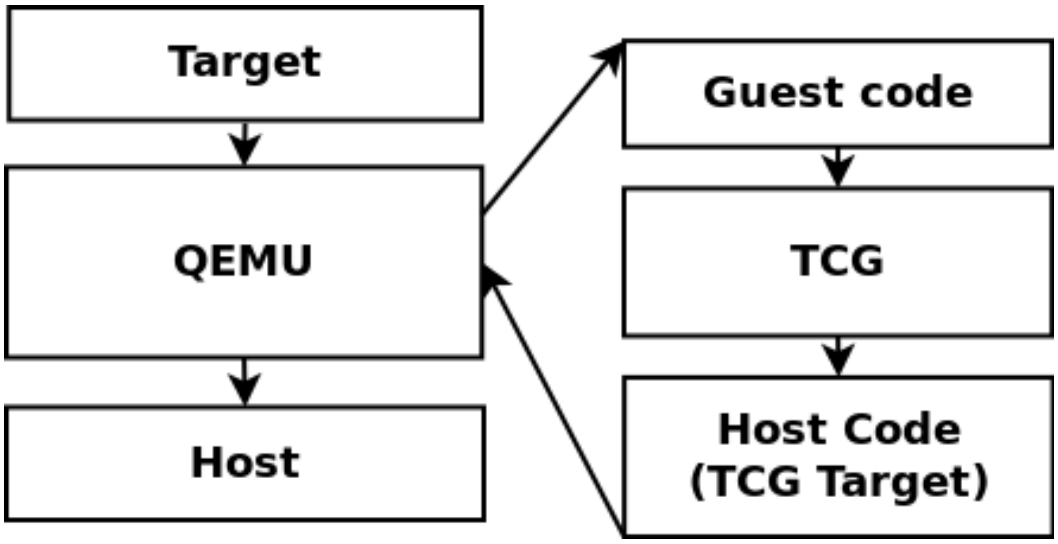

Рисунок 3.1 QEMU и TCG.

Таким образом, QEMU функционирует при помощи извлечения гостевого кода и превращения его в код хоста. В свою очередь, сам перевод состоит из двух частей: первая, блок целевого кода (Target Block, TB) конвертируется в TCG операцию – разновидность машинно-независимого промежуточного представления; вторая – TCG операция для определенного TB преобразовывается в код хоста. Между этими двумя этапами выполняются дополнительные оптимизационные проходы.

Стоит отдельно подчеркнуть, что Target Block в QEMU является неким аналогом базового блока в системе Valgrind, описанного в части 1 главы 2.

Часть 3. Внутреннее устройство QEMU

Основная часть исходных кодов QEMU состоит приблизительно из 1300 файлов, систематизированных в определенных каталогах. Однако, даже если исходный код систематизирован, он остается довольно сложным для изучения и модификации. Следовательно, для полного понимания работы QEMU стоит обратить особое внимание на исходные коды. Далее, под символом «/» будет подразумеваться корневой каталог проекта QEMU.

Главными файлами являются: /vl.c, /cpus.c, /exec-all.c, /exec.c, /cpu-exec.c. Функция main, где начинается исполнение, реализована в /vl.c. Остальные функции в этом файле создают виртуальную машинную среду согласно заданным виртуальным машинным техническим характеристикам, таким как: размер оперативной памяти, тип CPU, доступные устройства и т.д. Начиная от функции main, после того, как виртуальный процессор создан, исполнение передается через такие файлы, как: /cpus.c, /exec-all.c, /exec.c, /cpu-exec.c.

Описание преобразований TB в TCG операции находится в /target-X/, где X – название поддерживаемой для эмуляции архитектуры. Например, для i386 исходные коды расположены в /target-i386/. Эта часть называется TCG-frontend.

Описание преобразований TCG операций в хостовый код размещено в /tcg/. В /tcg/Y/, где Y – название поддерживаемой хостовой архитектуры, содержится код, преобразовывающий TCG операции в код, понятный host архитектуре. Данная часть называется TCG-backend.

В итоге, стоит перечислить базовые исходные файлы, осуществляющие всю основную работу:

1) /vl.c – главный цикл программы-эмулятора;

2) /target-X/translate.c – извлечение гостевого кода и преобразование в промежуточный;
3) /tcg/tcg.c – основные функции TCG модуля;
4) /tcg/Y/tcg-target.c – код, который преобразовывает TCG операции в код, на котором исполняется QEMU;
5) /cpu-exec.c – фукнция cpu-exec находит следующий TB. В случае, если TB не обнаружен, делаются запросы, чтобы создать TB и в конечном итоге исполнить сгенерированный код.

Часть 4. Модуль TCG

До версии 0.9.1 динамический перевод кода в QEMU осуществлялся посредством DynGen. В то время блоки перевода преобразовывались в язык программирования C при помощи DynGen, а компилятор GCC уже преобразовывал код из языка C в хостовый код. Проблемным местом этой технологии являлась тесная связь DynGen с компилятором GCC. Это и было причиной возникновения трудностей при учете изменений в GCC. С целью устранить тесную привязку к GCC, разработчики QEMU перешли на TCG.

При разработке TCG было сделано так, чтобы динамический перевод кода осуществлялся только по мере необходимости, что явно отображено на рисунке 3.2.

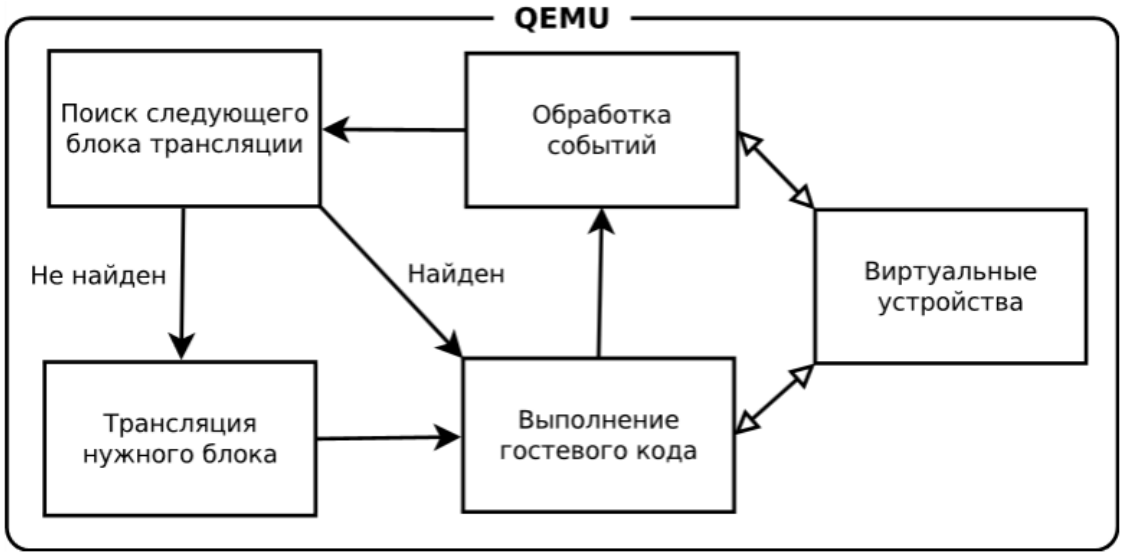

Рисунок 3.2. Ядро QEMU.

Подобная оптимизация способствует максимальной продолжительности использования сгенерированного кода. Каждый раз, когда код генерируется из ТВ, он сохраняется в кэше кодов перед тем, как быть исполненным. В основном используются одни и те же ТВ. Таким образом, вместо восстановления одного и того же кода, будет удобнее просто сохранить его. А как только кэш будет заполнен, он сбросится без использования специальных алгоритмов замещения.

Все компиляторы создают объектные файлы из исходного кода перед использованием. Для того чтобы получить выходную программу для вызова функции, такой компилятор, как GCC, создает специальный компонующий автокод. Последний выполняет все необходимое до запроса и возвращения функции. Этот компонующий автокод называется прологом и эпилогом функции. И если архитектура обладает указателями базы и стека, вводная часть функции выполняет следующие действия:

1) проталкивает текущий указатель базы в стек так, чтобы он мог быть восстановлен впоследствии;

2) замещает старый указатель базы текущим указателем стека так, что новый стек может быть создан поверх старого;
3) перемещает указатель вдоль стека, чтобы создать место в текущем стековом фрейме для локальных переменных функции.

Эпилог функции совершает действия, противоположные вводной части функции (прологу), и возвращает управление вызывающей функции:

1) замещает указатель стека текущим указателем базы. Таким образом, указатель стека восстанавливается в своем значении до пролога;
2) выталкивает указатель базы из стека так, что он восстанавливается в своем значении до пролога;
3) возвращается к вызывающей функции посредством выталкивания счетчика команд предыдущего фрейма из стека и скачком переходит к ней.

Таким образом, TCG по функциональности схож с компилятором, создающим объектный код «на лету». Этот код впоследствии хранится в кэше, откуда осуществляется и выполнение программы.

Весь процесс динамического перевода кода наглядно показан на рисунках 3.3, 3.4 и 3.5. На входе TCG модуля имеется некий гостевой код, ISA которого относится к некой поддерживаемой в QEMU архитектуре.

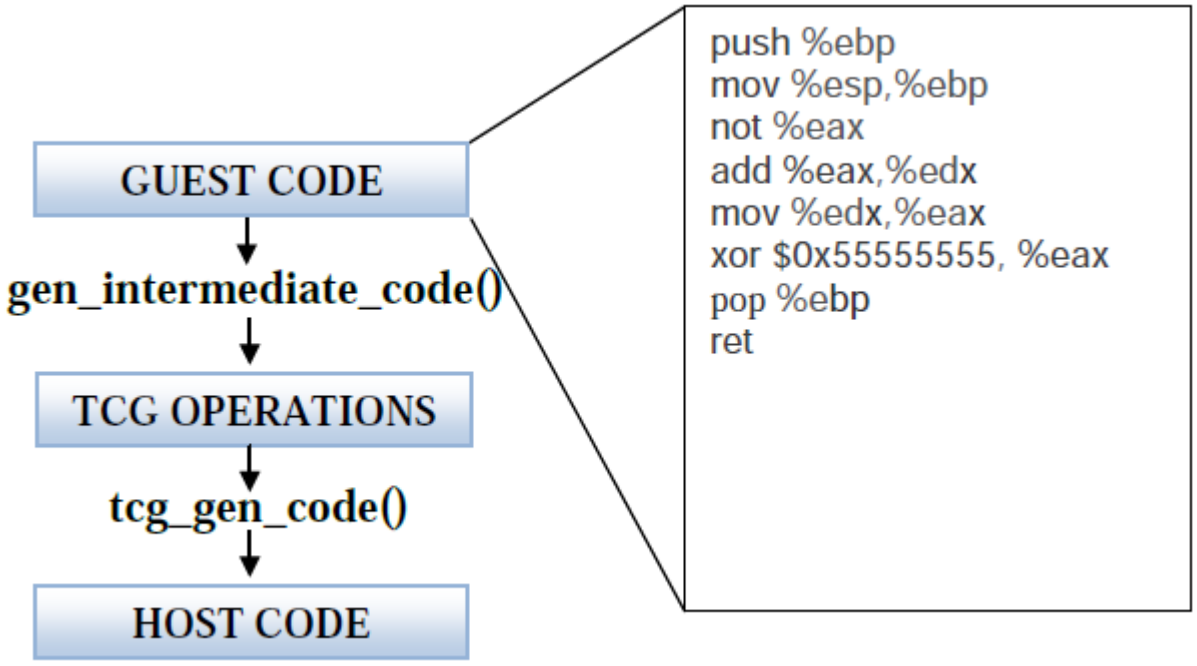

Рисунок 3.3. Гостевой код.

Функция gen_intermediate_code осуществляет активный перевод из гостевого кода в промежуточное представление QEMU, которое состоит, в то же время, из TCG операций.

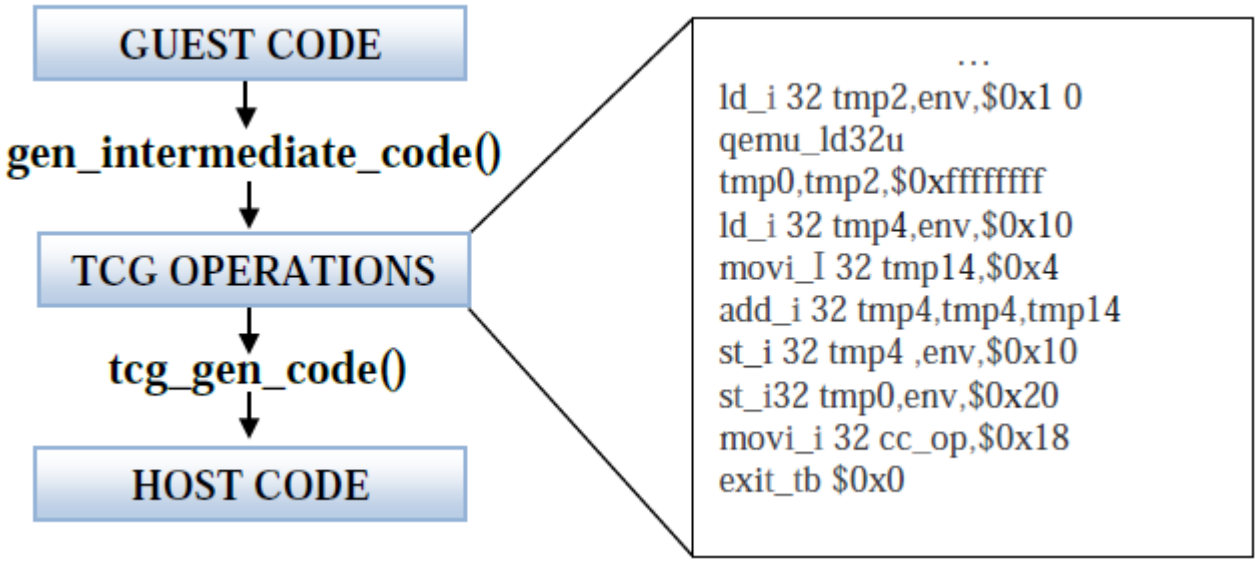

Рисунок 3.4. Промежуточный код.

Из получившегося промежуточного кода функция tcg_gen_code способна осуществить перевод TCG операций в код платформы, на которой запущен эмулятор QEMU.

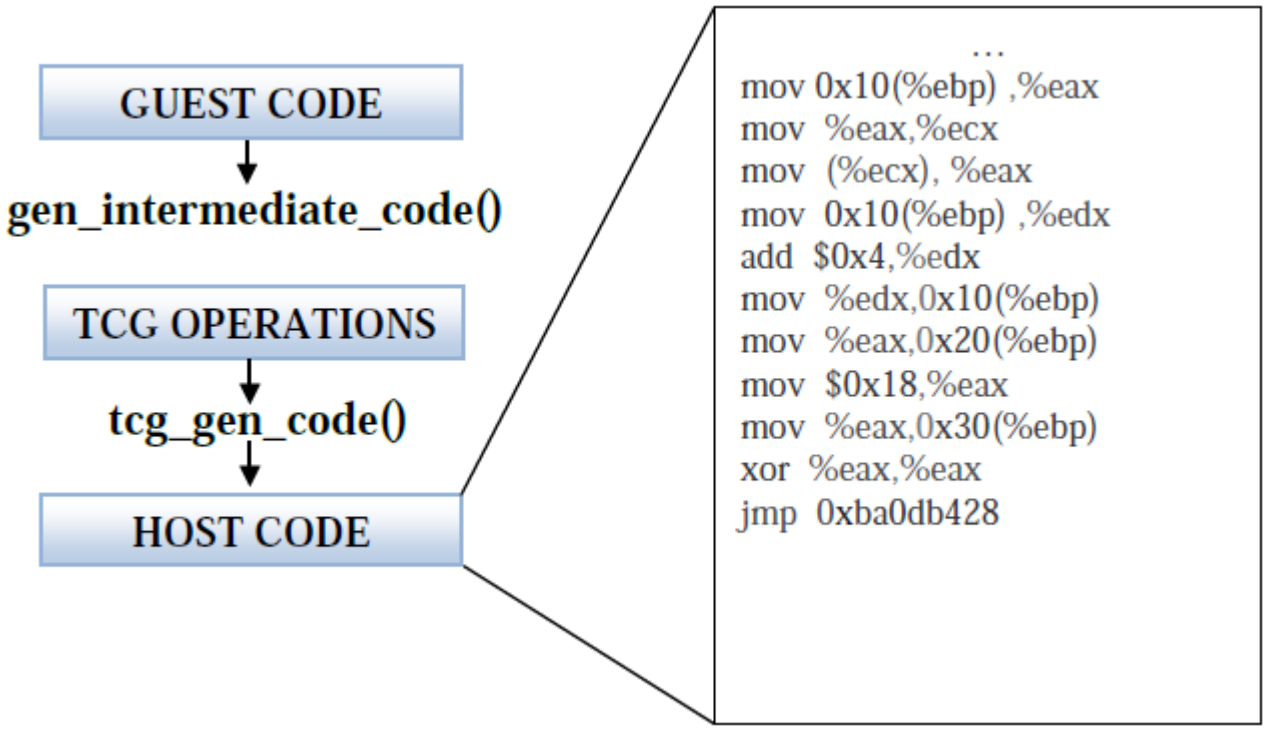

Рисунок 3.5. Хостовый код.

Вывод по главе 3

В данной главе мы разобрали внутреннее устройство и принцип работы программы QEMU. Также мы детально изучили модуль TCG, который выполняет основную роль в QEMU. Проведенное нами исследование необходимо не только для разработки QEMU, но и для написания других программ, основанных на эмуляторе QEMU.

Глава 4. Встраивание новой архитектуры в QEMU
Часть 1. PPDL

PPDL – это язык описания ядер микропроцессоров высокого уровня. Ключевой принцип PPDL – «одно описание – несколько компонентов». Описания на языке PPDL достаточно для автоматической генерации следующего набора компонентов ядра процессора, в который входят:

1) симулятор: C++, SystemC или симулятор, основанный на QEMU;
2) синтезируемая Verilog-модель с автоматически генерируемой логикой для отладки процессора. Она необходима для запуска и остановки процессора, определения точек останова (breakpoint), а также чтения и записи регистров процессора и памяти, подключенной к процессору;
3) ассемблер и дизассемблер – платформозависимые части GNU assembler и GNU disassembler;
4) набор тестов на языке ассемблера;
5) профилировщик;
6) платформозависимая (back-end) часть компилятора.

В PPDL функциональность и другие параметры (инструкции, обработчики прерываний и т.д) описываются на C-подобном языке. В отличии от C и C++, в PPDL мы имеем возможность обращаться к отдельным битам регистра, порта или локальной переменной (по аналогии с Verilog). Как и в языках C и C++, в PPDL имеется C-подобный препроцессор.

Использование PPDL позволит не только сократить размер команды разработчиков микропроцессора, уменьшив себестоимость разработки, но и сократить время последней:

1) для архитектур, создаваемых «с нуля» – в 3-5 раз;
2) при модификации ранее созданных архитектур под требования приложений – в 7-10 раз.

Часть 2. Генерация кода для модуля TCG

Компилятор PPDL позволяет создавать формальное описание процессора в широко известном формате XML. Такое описание можно преобразовать в TCG-fronted функции, которые участвуют в динамическом переводе гостевого кода QEMU в хостовый.

Для того, чтобы увеличить скорость встраивания архитектуры в QEMU, в рамках данной дипломной работы был разработан кодогенератор на языке высокого уровня Python. Данный кодогенератор осуществляет описанное выше преобразование операций из XML в функции QEMU. Пример перевода приведён на рисунке 4.1.

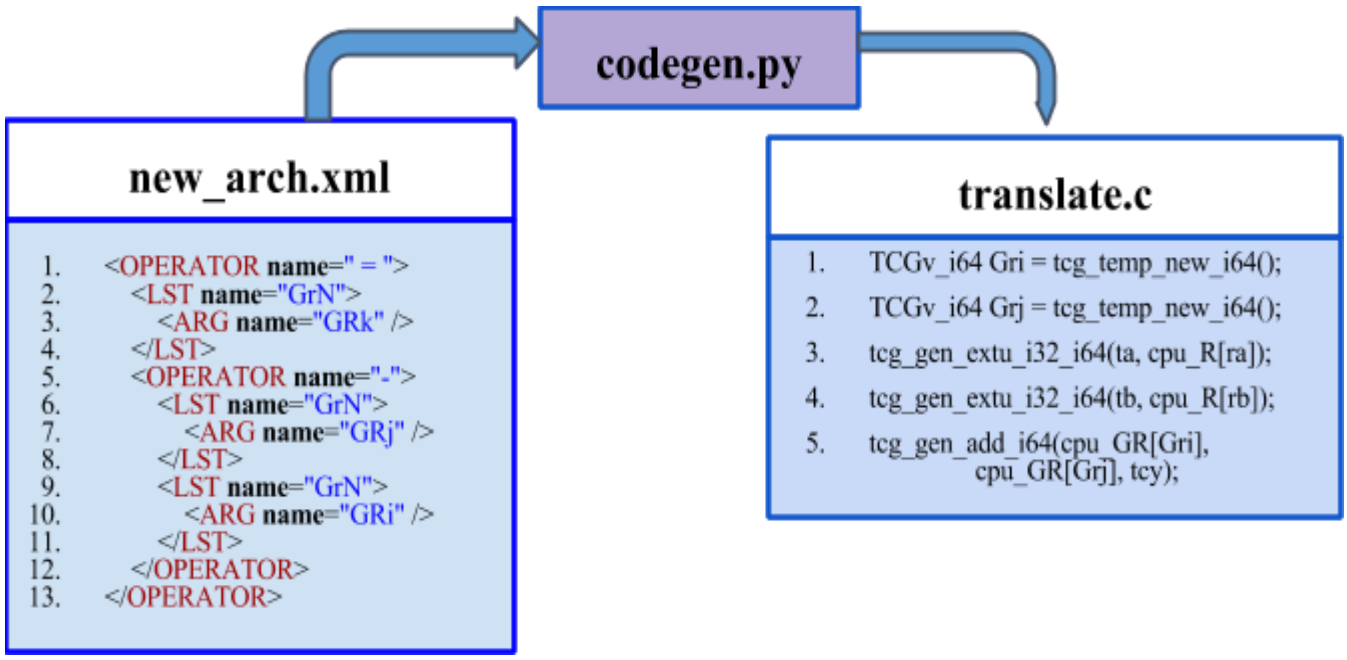

Рисунок 4.1. Часть перевода XML описания архитектуры в код QEMU.

Отсюда следует, что если мы имеем формальное описание архитектуры в формате XML, которое генерируется PPDL, процесс добавления данной

архитектуры во многом облегчается и становится возможным в полуавтоматизированном режиме.

Вывод по главе 4

Итого можем сказать, что встраивание новой архитектуры в QEMU – автоматизируемая задача. Проблемным местом при этом является унификация входных данных для генерации TCG функций. Во время эмуляции они отвечают за генерацию хостового кода. Однако если мы имеем некое понятное генератору кода описание архитектуры, то встраивание данной архитектуры в QEMU возможно осуществить в полуавтоматизированном режиме.

Глава 5. Кроссплатформенный динамический анализатор бинарного кода, перехватывающий переполнения буфера в стеке

Часть 1. Общие сведения

Так как в последующих частях данной главы будут употребляться узкоспециализированные термины, каждый из них требует точного определения:

Уязвимость – недостаток программы, посредством которого можно намеренно нарушить её целостность и вызвать неправильную работу.

Буфер – это область памяти, используемая для временного хранения данных при вводе или выводе.

Переполнение буфера – явление, возникающее, когда компьютерная программа записывает данные за пределами выделенного в памяти буфера.

Стек – структура данных, представляющая собой список элементов, организованных по принципу LIFO («последним пришёл – первым вышел»).

Адрес возврата – адрес в памяти следующей инструкции приостановленной программы; управление передается подпрограмме или подпрограмме-обработчику.

Сюръекция – свойство отображения, при котором каждому элементу множества прибытия может быть сопоставлен хотя бы один элемент области определения.

Часть 2. Уязвимая программа

Для проверки работоспособности реализованного анализатора бинарного кода существует программа, в процессе выполнения которой вызывается общеизвестная небезопасная функция strcpy(). Входные параметры этой функции не фильтруются, что может привести к переполнению буфера в стеке.

Исходный код данной программы приведён в приложении 1. Здесь будут указаны общие черты выполнения уязвимой программы. Рисунок 5.1 иллюстрирует все вышесказанное.

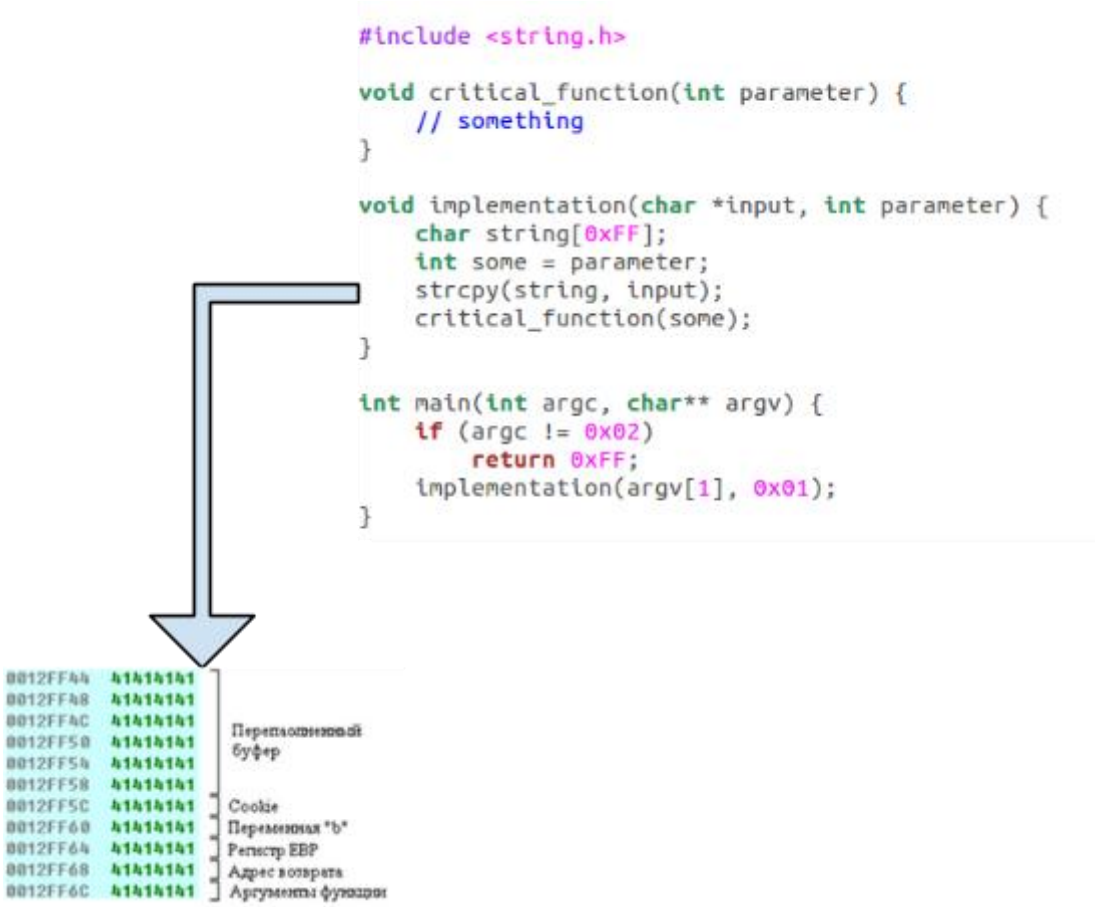

Рисунок 5.1. Выполнение уязвимого приложения.

Часть 3. Shadow Stack

Суть метода Shadow Stack заключается в создании собственного стека адресов возврата функций программы. Благодаря этому возможна идентификация переполнения буфера в стеке в момент перезаписи адреса возврата. Алгоритм такого инструмента очень прост: перед вызовом каждой функции запоминается адрес возврата в Shadow Stack, а перед выходом из функции он сравнивается со значением, сохраненным в Shadow Stack. Наглядное описание работы алгоритма изображено на рисунке 5.2.

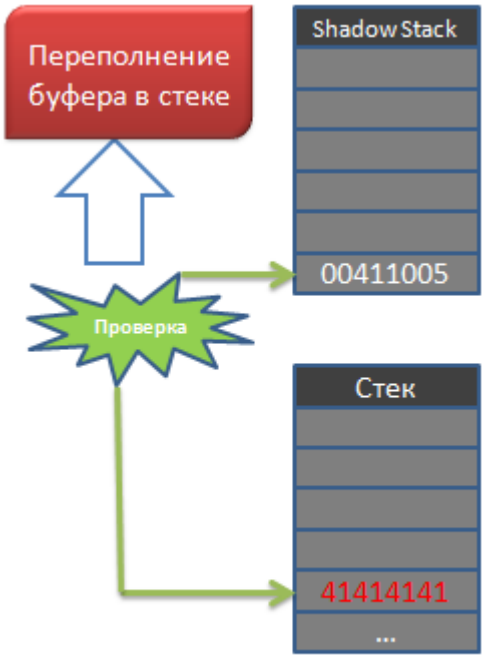

Рисунок 5.2. Алгоритм Shadow Stack.

Часть 4. Реализация динамического анализатора на основе QEMU

Для реализации кроссплатформенного динамического анализатора кода на основе QEMU существуют два способа. Во-первых, это проведение анализа гостевого кода, во-вторых, – промежуточного кода. Для проведения анализа требуются масштабируемые анализаторы. Таким образом, первый способ не подходит, т.к. задача добавления новой архитектуры схожа по сложности с задачей создания отдельного анализатора для данной архитектуры. Второй способ не обладает таким свойством, а добавление новой архитектуры в анализатор является добавлением новой архитектуры в эмулятор QEMU. На основе наработок, описанных в главе 4, мы можем сделать вывод, что второй способ идеально подходит для создания анализатора с возможностью расширения числа архитектур в полуавтоматизированном режиме.

Единственной трудностью является знание того, как интересующие нас участки гостевого кода преобразуются в промежуточный код. В связи с этим, перед написанием анализатора необходимо найти для каждого прообраза образ преобразования кода из гостевого кода в хостовый. Таким образом, промежуточный код – образ, а гостевой – прообраз. Стоит указать, что данное преобразование кода является сюръективным отображением.

В рамках данной работы прообраз отображения – это код гостевой архитектуры, отвечающий за вызов и завершение функции, а образ – соответствующий им промежуточный код.

Часть 5. DBI framework на основе QEMU

В предыдущей части мы выяснили, что для написания динамияческого анализатора нам необходимо выявить образ преобразования для гостевого кода. Данная задача является выполнимой. Для реализации DBI фреймворка необходимо определить больше соответствующих образов, чем для реализации отдельного анализатора. Отсюда следует, что возможность создания DBI фреймворка реально существует.

Вывод по главе 5

В главе 5 рассмотрена реализация динамического анализатора бинарного кода на основе QEMU, который позволяет идентифицировать переполнение буфера в стеке. Данный анализатор является кроссплатформенным, т.е. с его помощью могут быть исследованы бинарные приложения, собранные для целого ряда архитектур. Среди них: x86, x86-64, ARM, MIPS, openRISC, PowerPC, SPARC, т.е. все известные платформы эмулятору QEMU. Кроме уже поддерживаемых архитектур, анализатор может быть расширен иными платформами в полуавтоматизированном режиме на основе наработок, описанных в главе 4. Данный инструмент также демонстрирует реальную возможность создания DBI фреймворка из QEMU.

Заключение

В дипломной работе были изучены различные способы анализа приложений. В ходе исследования был выявлен наиболее эффективный для современных сложных динамичных программ анализ бинарного кода. Помимо этого были исследованы существующие инструменты, построенные на динамической бинарной инструментации.

Подобные мощные средства обладают общим недостатком – отсутствием быстрого расширения поддерживаемых архитектур. Как было установлено в самом начале работы, этот минус является существенным в современном информационном мире. Так как развитие «умных» домов и «Интернет-вещей» продвигается все дальше, появление новых архитектур неизбежно, а значит кроссплатформенные динамические бинарные анализаторы с легко расширяемой поддержкой архитектур будут востребованы.

Кроме этого, в работе была предложена базовая основа, позволяющая сократить время реализации кроссплатформенного динамического анализатора кода с расширяемым количеством поддерживаемых архитектур в полуавтоматизированном режиме. В качестве данной основы был выбран и в дальнейшем изучен эмулятор QEMU.

Полученные в результате исследования знания позволили создать кроссплатформенный динамический анализатор бинарного кода. С его помощью возможна идентификация переполнения буфера в стеке. Данный инструмент показывает, что на основе эмулятора QEMU может быть создан полноценный кроссплатформенный DBI фреймворк. Таким образом, специалисты по информационной безопасности получат возможность разрабатывать и создавать

различные динамические кроссплатформенные анализаторы бинарного кода на основе данного фреймворка.

В итоге, поставленные задачи, упомянутые нами во введении, были успешно решены и реализованы. Во-первых, были исследованы инструменты, основанные на технологии DBI. Во-вторых, на основе полученных знаний был разработан инструмент идентификации переполнения буфера в стеке в бинарном исполняемом файле. С помощью данного инструмента могут быть исследованы бинарные приложения, собранные для целого ряда архитектур, известных эмулятору QEMU. И наконец, в-третьих, также была разработана технология автоматизации процесса встраивания новой архитектуры в QEMU. Для решения этой задачи был создан кодогенератор, который позволил встраивать новые архитектуры в QEMU в полуавтоматизированном режиме.

Список литературы

1. Перов М.Н. Применение Dynamic Binary Instrumentation в области защиты информации. Комплексная защита информации. Электроника инфо. Материалы XVIII Международной конференции 21–24 мая 2013 года, Брест (Республика Беларусь). 2013. С. 140.
2. Перов М.Н., Фонин Ю.Н., Костин А.Е. Встраивание новой архитектуры в эмулятор QEMU. Труды 57-й научной конференции МФТИ с международным участием, посвященной 120-летию со дня рождения П.Л. Капицы. МФТИ. 2014.
3. Bellard F. QEMU, a Fast and Portable Dynamic Translator. USENIX Annual Technical Conference. 2005.
4. C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In Proceedings of the SIGPLAN 2005 Conference on Programming Language Design and Implementation. 2005.
5. N. Nethercote and J. Seward. Valgrind: A program supervision framework. In Proceedings of the 3rd Workshop on Runtime Verification.
6. Nicholas Nethercote. Dynamic Binary Analysis and Instrumentation or Building Tools is Easy. A dissertation submitted for the degree of Doctor of Philosophy at the University of Cambridge. 2004.
7. Dawn Song, David Brumley, HengYin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, Prateek Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. International Conference on Information Systems Security. 2008.

8. Дроздов Ю.А., Владиславлев В.Е., Новиков С.В., Фонин Ю.Н. Современные программные средства разработки новых вычислительных архитектур. Труды 57-й научной конференции МФТИ с международным участием, посвященной 120-летию со дня рождения П.Л. Капицы. МФТИ. 2014.

Приложение 1. Исходный код программы, обладающей уязвимостью

```
#include <string.h>

void critical_function(int parameter) {
   // something
}

void implementation(char *input, int parameter) {
   char string[0xFF];
   int some = parameter;
   strcpy(string, input);
   critical_function(some);
}

int main(int argc, char** argv) {
   if (argc != 0x02)
      return 0xFF;
   implementation(argv[1], 0x01);
}
```