НОВГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ
ЯРОСЛАВА МУДРОГО

На правах рукописи

Фурсова Наталья Игоревна

МЕТОДЫ МОНИТОРИНГА ОБЪЕКТОВ ОПЕРАЦИОННОЙ СИСТЕМЫ, ВЫПОЛНЯЮЩЕЙСЯ В ВИРТУАЛЬНОЙ МАШИНЕ

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
кандидат технических наук, доцент
Макаров Владимир Алексеевич

Великий Новгород — 2017

Оглавление

Стр.

Введение

Актуальность. Безопасность программного обеспечения связана с анализом недокументированных возможностей и вредоносного внешнего влияния. Программы как правило поставляются производителем в виде бинарного кода (исполняемых файлов), а их исходные коды при этом обычно недоступны. Анализ бинарного кода сопряжен с решением сложных задач, но взамен дает лучшее понимание за счет того, что анализируется именно тот код, который будет выполняться.

Существует много способов исследования бинарного кода, например, анализ помеченных данных (taint-анализ) [1], поведенческий анализ [2], трассировка выполнения машинных команд [3]. Для этого требуется наличие высокоуровневой информации об исследуемых приложениях и операционных системах (открываемые файлы, работающие процессы, вызываемые функции, загружаемые модули), получение которой сопряжено с проблемой, называемой семантический разрыв (semantic gap).

Семантический разрыв — это разрыв в представлении наблюдаемых в системе данных и данных, требуемых для проведения анализа. Например, анализатор требует информацию о файлах, а имеется только поток инструкций. В этой работе поведение операционных систем (ОС) исследуется в виртуальной машине, поэтому вопросы мониторинга на реальном аппаратном обеспечении не рассматриваются. Выполняемая в виртуальной машине ОС называется гостевой. Для анализа в виртуальной среде применяются различные эмуляторы: QEMU [4], Xen [5], VMWare [6], Bochs [7].

Виртуальная машина предоставляет для анализа поток инструкций виртуализированного процессора и ее оперативную память. Выделить из этого потока высокоуровневую информацию вручную практически невоз-

можно. Поэтому возникает необходимость в автоматических методах мониторинга высокоуровневых событий, например, операций с файлами и процессами. Эти события будут накладываться на низкоуровневую трассу машинных команд, улучшая как понимание происходящего в исследуемой системе, так и поддержку работы других методов анализа: анализа помеченных данных, выявления аномалий в поведении и других.

Передовыми исследователями в области мониторинга объектов ОС являются команды, реализующие такие инструменты, как PANDA [8], DECAF [9], RTKDSM [10]. Методы, заложенные в эти решения, имеют особенности: использование программ-агентов — приложений, загружаемых в систему для сбора данных о структурах, адресах функций и других данных.

Исходя из особенностей, можно определить что такие подходы имеют трудности с анализом систем с закрытым исходным кодом, а также с системами, не предусматривающими загрузку в них программ извне.

Несмотря на описанные особенности, эти инструменты можно использовать для анализа ОС настольных компьютеров, таких как Windows и Linux, но их тяжело применять для анализа встроенных систем. Как правило, исследования встроенных систем проводятся в виртуальной среде, в которую загружается ПО, извлеченное из ПЗУ исследуемого устройства. В этом случае подход с программами-агентами будет крайне трудозатратен или невозможен по причине того, что бинарный код агента может быть несовместим с системой (зачастую версия ОС и параметры сборки неизвестны), а исходный код в таких системах невозможно скомпилировать.

Чтобы обойти трудности с использованием программы-агента, необходимо использовать методы мониторинга, не предполагающие внедрение инструментального кода в окружение, в котором выполняется исследуемая программа.

В соответствии с вышесказанным можно выделить ряд требований, которые должны учитываться в новом методе мониторинга объектов ОС. Во-первых, не использовать загрузку в нее сторонних модулей, во-вторых, обеспечить работу анализатора с семействами исследуемых систем без внесения изменений в алгоритмы анализа. При выполнении этих требований также появляется возможность анализировать образы систем, извлеченных из ПЗУ. Под семействами ОС понимаются системы, разработанные на одном ядре, например, к семейству Windows относятся все пользовательские версии этой операционной системы.

В диссертационной работе предлагаются методы мониторинга объектов операционной системы без использования программных агентов. Объектами ОС в контексте этой работы называются файлы, процессы, модули, вызываемые функции. Как правило, объекты имеют характерные атрибуты, например, для файла это имя и дескриптор, для процесса идентификатор и адресное пространство. Для работы предлагаемых методов не требуются знания о внутреннем устройстве системы, а необходимая информация получается из открытых описаний программных интерфейсов. Кроме того, набор данных, собранный для одной ОС, применим ко всему семейству этой ОС.

Целью диссертационной работы является исследование и разработка методов и инструментальных средств, реализующих мониторинг объектов гостевой операционной системы при ее выполнении в полносистемном программном эмуляторе. Разработанные методы должны обеспечивать конфигурирование и работу анализатора без загрузки инструментального кода в исследуемую систему. Единожды сконфигурированный анализатор должен работать для операционных систем из одного семейства без перенастройки.

Для достижения поставленной цели необходимо решить следующие **задачи**:

1. Предложить модель для операционных систем общего назначения, работающих на настольных компьютерах, мобильных устройствах и коммуникационном оборудовании.
2. Разработать метод мониторинга событий виртуальной машины для получения информации об объектах ОС без внедрения инструментального кода в исследуемую систему.
3. Разработать метод вызова системных функций по запросу анализатора для получения заданных атрибутов объектов ОС.
4. Реализовать инструмент для мониторинга объектов ОС по предложенным методам в виде плагинов к эмулятору QEMU.
5. Провести функциональное тестирование разработанного инструмента, определить масштаб накладных расходов, а также проверить достоверность получаемых результатов.

Научная новизна:

1. Предложен новый метод получения информации об объектах ОС через мониторинг событий виртуальной машины, позволяющий анализировать встроенные системы, а также семейства ОС.
2. Предложен новый метод получения заданных атрибутов объектов ОС по запросу анализатора через встраивание вызовов системных функций в поток инструкций виртуальной машины.

Практическая и теоретическая ценность работы.

Разработанные методы представляют интерес для разработчиков инструментов динамического анализа, требующих для своей работы высокоуровневую информацию, например, анализ помеченных данных, поведенческий анализ, кроме того, они подходят для анализа встроенных си-

стем. Такими работами занимается Институт системного программирования РАН.

Также представленные методы могут использоваться в исследованиях по информационной безопасности, а разработанный инструмент — лабораториями, проводящими исследования поведения программных систем.

Методология и методы исследования. Результаты диссертационной работы получены на базе использования методов динамического анализа и обратной инженерии бинарного кода, системного анализа, а также математического моделирования. При проведении экспериментов и оценке их результатов применялись методы теории вероятностей.

Основные положения, выносимые на защиту:

1. Метод мониторинга событий виртуальной машины для получения информации об объектах гостевой операционной системы без внедрения инструментального кода на уровне исследуемой системы.
2. Метод вызова системных функций по запросу анализатора для получения заданных атрибутов объектов операционной системы.
3. Инструментальная среда, работающая с тремя семействами операционных систем общего назначения (Windows, Linux, FreeBSD).

Апробация работы. Основные результаты работы опубликованы в статьях [1; 11—16], из них работы [1; 11—13] опубликованы в изданиях из перечня ВАК, статьи [12; 15; 16] индексированы Scopus. В статье [14] представлена концептуальная модель разработанного автором метода мониторинга объектов ОС. В статье [15] описаны предложенные автором методы и их реализации для мониторинга объектов операционных систем. В статьях [11; 12; 16] описан разработанный автором метод мониторинга событий виртуальной машины для получения информации об объектах ОС. В статье [13] описаны разработанные автором плагины для перехвата системных вызовов, отслеживания файлов и процессов. В работе [1] упоминается

разработанное автором средство мониторинга файлов, используемое для осуществления анализа помеченных данных.

Результаты работы обсуждались на следующих конференциях:

1. Международная конференция РусКрипто, Солнечногорск, Россия, 25-28 марта, 2014.
2. Открытая конференция по компиляторным технологиям, Москва, Россия, 2 декабря, 2015.
3. SAC '16 Proceedings of the 31st Annual ACM Symposium on Applied Computing, Пиза, Италия, 4-8 апреля, 2016.
4. Международная конференция РусКрипто, Солнечногорск, Россия, 21-24 марта, 2017.
5. Международная Ершовская конференция по информатике PSI–2017, Москва, Россия, 27–29 июня 2017.
6. Научно-исследовательский семинар Института системного программирования РАН.
7. 11th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2017).

Личный вклад. Все представленные в диссертации результаты получены лично автором.

Объем и структура работы. Диссертация состоит из введения, пяти глав, заключения и приложения. Полный объём диссертации составляет 120 страниц с 10 рисунками и 9 таблицами. Список литературы содержит 54 наименования.

Глава 1. Изучение предметной области и обзор существующих подходов к мониторингу объектов ОС

Мониторинг объектов операционных систем широко известная тема исследований. Все работы, рассмотренные автором, являются англоязычными, в них принято использовать термин интроспекция (Introspection), поэтому в этой главе понятия мониторинг объектов ОС и интроспекция считаются эквивалентными.

Чем больше имеется информации об окружении, в котором осуществляется работа, тем более эффективно можно ее анализировать и защищать. Глубина информации — фундаментальное преимущество концепции, называемой интроспекцией виртуальных машин. Интроспекция ВМ это исследование текущего состояния ВМ на системном уровне. Для интроспекции виртуальных машин текущее состояние может быть определено довольно широко, включая информацию о регистрах процессора, памяти, диске, сети и любых других событиях.

Фундаментальной проблемой интроспекции виртуальных машин является разрыв в уровнях представлений, так называемый семантический разрыв [17; 18].

Цель интроспекции — получение как можно более полного представления о гостевой ОС. Следовательно, эволюция интроспекций подчиняется вопросу эффективного преодоления семантического разрыва [19; 20].

Другими словами, интроспекция — это извлечение данных из операционной системы, которые используются системой для ее работы, от пользователя эти данные скрыты.

1.1 Свойства методов мониторинга объектов ОС

Интроспекция широко распространена в различных областях. В зависимости от области применения разработчики наделяют методы различными свойствами. Однако, можно выделить несколько свойств, которые будут уместны для методов мониторинга объектов ОС независимо от их области применения.

Такими свойствами являются:

- Минимальное влияние на производительность. Реализация методов интроспекции должна по возможности оказывать на систему как можно меньшее влияние. Методы интроспекции должны работать независимо и вносить в гипервизор минимум изменений. Это важно для обеспечения переносимости разработанных методов на новые версии виртуальной машины.
- Нет модификациям гостевой системы. Гипервизоры поддерживают почти все возможные ОС в качестве гостевой системы. Если код интроспекции должен быть изменен для каждого гостя, его широкая применимость становится очень сомнительной. Даже незначительные изменения и периодические патчи для конкретной ОС могут создавать проблемы.
- Прозрачность в работе. Работа методов интроспекции должна быть прозрачна для гипервизора, гостевой ОС и любой программы в гостевой ОС.
- Независимость гипервизора. Техника интроспекции не должна зависеть от любой исключительной особенности в архитектуре гипервизора. Интроспекция должна быть применима к любому виду гипервизоров независимо от реализации метода интроспекции.

– Никаких побочных эффектов. Реализация инструментов интроспекции не должна генерировать любые нежелательные результаты, которые могут привести к вредоносному поведению компонентов системы. Интроспектирующие инструменты не должны так же давать каких-то результатов, которые откроют их присутствие в системе.
– Безопасность компонентов мониторинга. Модули интроспекции могут быть расположены в гипевизоре, гостевой ОС или защищенной ВМ (хостовой). Эти модули должны быть защищены от внешних атак. Если модуль находится в гостевой ОС, то должна быть реализована дополнительная защита для сохранения целостности [21].

1.2 Разновидность методов мониторинга объектов ОС в виртуальной машине

Подходы к интроспекции делятся на две большие группы: статический анализ и динамический анализ.

Методы статического анализа программ предполагают проведение анализа без запуска программы на исполнение. Вместо этого анализ осуществляется с помощью автоматического построения модели программы и последующей обработки построенной модели. Модель программы может быть построена таким образом, что это позволит её проводить частичный или параллельный анализ. Такой подход позволяет существенно сократить время анализа программы. Как правило, модель программы имеет некоторую степень приближения к реальному поведению программы в процессе запуска. В связи с этим поиск ошибок с помощью статического анализа

может иметь как ложные срабатывания, так и не обнаруживать некоторые ошибки, присутствующие в программе.

Статический анализ может проводиться, в том числе, в процессе написания исходного кода программы в интегрированной среде разработки, что позволяет разработчикам обнаруживать и исправлять найденные ошибки в процессе написания программы и до её фактической компиляции в исполняемый код [22; 23]. В свою очередь это позволяет уменьшить стоимость исправления ошибки. Динамический анализ основан на запуске исследуемого продукта на исполнение. Несомненным преимуществом динамического анализа является отсутствие каких-либо предположений о ходе исполнения программы и проверка её в процессе или сразу после исполнения. При этом одно из основных требований, предъявляемых к динамическому анализу — само проведение анализа должно насколько это возможно меньшим образом влиять на ход исполнения. При определённых условиях на детерминированность программы, динамический анализ позволяет избежать проблемы ложных срабатываний.

Главный минус динамического анализа состоит в том, что для получения качественного покрытия анализируемой программы, как правило, требуется неоднократный запуск программы на выполнение, что связано с большими временными затратами. Однако, при обнаружении ошибки в процессе динамического анализа, как правило, возможно сгенерировать входные данные для программы, на которых ошибка воспроизводится. Таким образом появляется возможность исключения ложных срабатываний анализатора [24].

Выше приведены классические варианты использования статического и динамического анализа. В этой работе речь идет о статическом и динамическом извлечении информации об объектах операционной системы, что является другой задачей, но суть различий остается той же.

Классификация подходов может выглядеть следующим образом:

- Статические методы
    - Основанные на анализе снимков системы
- Динамические методы
    - Основанные на событиях
    - Исследование памяти
    - Смешанный тип

Интроспекция памяти работает с анализом живой памяти. Когда ОС работает, все важные структуры данных находятся в оперативной памяти. ОЗУ содержит блоки управления процессами, значения регистров, загруженные модули ядра, таблицы страниц и т.д. ОЗУ так же содержит страницы, связанные с сегментами данных и кода выполняющегося процесса. Информация, связанная с ОС, может быть извлечена путем изучения содержимого ОЗУ. Большинство инструментов анализа вредоносного кода проверяют поведение программы, исследуя содержимое ОЗУ для данной программы. Это может быть использовано для таких задач, как обнаружение вторжения или анализа гостевых процессов [21].

Подходы, использующие только исследование памяти, заранее проигрывают, потому что память не всегда может быть актуальна, следовательно, результаты анализа могут быть неточными. В противном случае память необходимо пересканировать при каждом обращении, что существенно скажется на производительности системы. Проблемой также является то, что не всегда может быть очевидно когда именно проводить процедуру пересканирования.

Для определения текущего положения дел была определена классификация качеств систем, которые актуальны в рамках этой работы.

- Вид интроспекции

  - Динамический анализ «на лету». Анализ системы производится непосредственно во время ее работы без дополнительных средств.
  - Динамический анализ по снимкам системы. В этом случае копируется полное состояние системы через заданные промежутки времени, затем они исследуются в динамике.
  - Динамический анализ по записанным трассам. Трассы записываются в процессе работы системы, в трассу попадают определенные данные, например, DMA транзакции, значения регистров, интернет пакеты и так далее.
  - Статический анализ по снимкам системы. Как правило это один снимок, отражающий текущее состояние системы.
- Особенности метода
  - Внедрение внешних модулей в систему. Некоторые инструменты для своей настройки загружают в исследуемую систему программу-агент для первоначальной настройки.
  - Наличие исходных кодов. Для выполнения анализа системы или программы необходимо иметь доступ к их исходным кодам. Например, для первоначальной настройки интроспекции.
  - ABI. Использование двоичного интерфейса для работы метода.
- Добываемая информация
  - Объекты ядра (список дескрипторов, файлов, процессы). Характеристики этих объектов (идентификатор, имя файла, название процесса, адресное пространство)
  - Структуры данных приложений

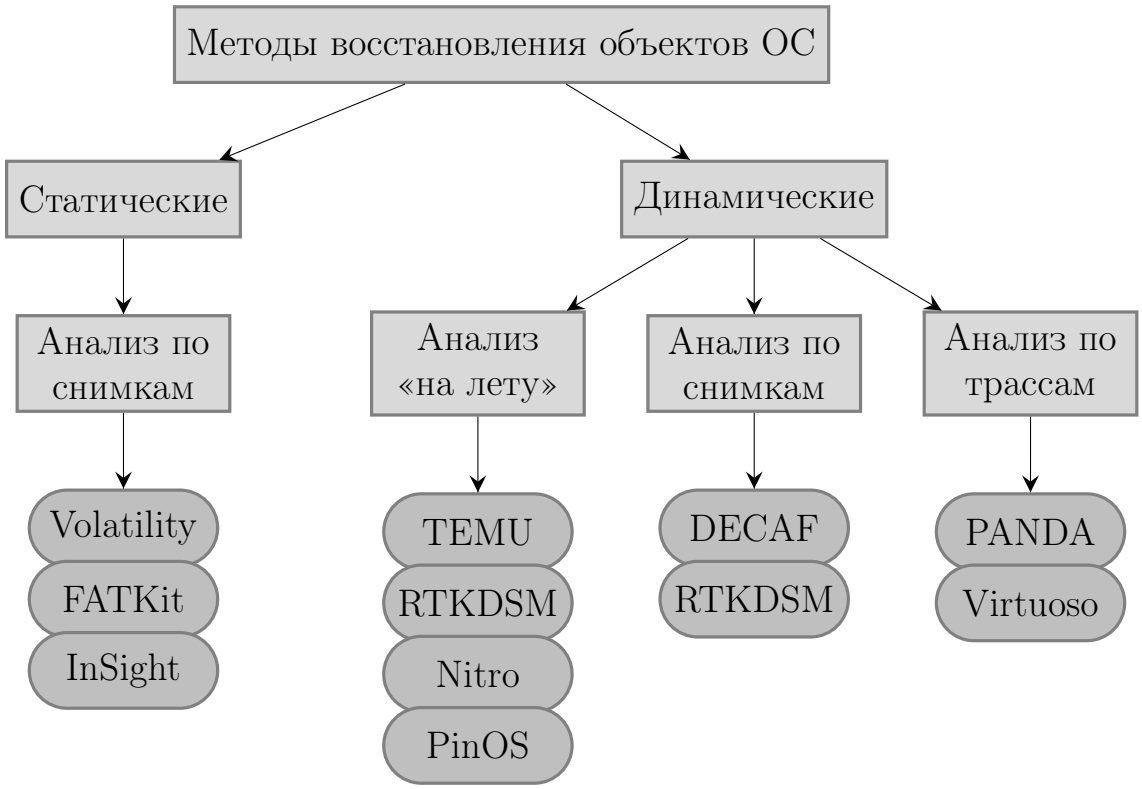

Рисунок 1.1 — Классификация методов мониторинга объектов ОС

– Содержимое памяти. Дампы физической памяти в процессе работы системы.
– Вызываемые функции API
– Вызываемые системные функции

На рисунке 1.1 представлена классификация методов мониторинга объектов ОС с обозначением представителей каждого из них.

Далее в главе будут описаны инструменты, реализующие эти методы, а также проведен анализ этих инструментов в соответствии с вышеописанной классификацией.

1.2.1 Методы, основанные на статическом анализе

В этом разделе рассмотрены инструменты, использующие статический анализ.

Википедия определяет статический анализ как анализ программного обеспечения, производимый (в отличие от динамического анализа) без реального выполнения исследуемых программ. В большинстве случаев анализ производится над какой-либо версией исходного кода, хотя иногда анализу подвергается какой-нибудь вид объектного кода. Термин обычно применяют к анализу, производимому специальным программным обеспечением.

В зависимости от используемого инструмента глубина анализа может варьироваться от определения поведения отдельных операторов до анализа, включающего весь имеющийся исходный код. Способы использования полученной в ходе анализа информации также различны — от выявления мест, возможно содержащих ошибки, до формальных методов, позволяющих математически доказать какие-либо свойства программы (например, соответствие поведения программы определенной спецификации).

В контексте этой работы рассматриваются инструменты статического анализа, которые направлены на интроспекцию виртуальных машин, поэтому исследуют не исходные коды напрямую, а дампы памяти, полученные в процессе работы системы. Дамп памяти — содержимое рабочей памяти одного процесса, ядра или всей операционной системы. Кроме того, дамп может включать дополнительную информацию о состоянии программы или системы, например значения регистров процессора и содержимое стека.

Статический метод интроспекции нацелен на получение максимально возможного количества данных о системе, которые можно извлечь из дампа. Как правило, нижеописанные инструменты имеют исходный код анализируемой системы и способны найти важные структуры ядра и их расположение в памяти. С помощью механизмов, заложенных в этих инструментах, данные накладываются на дампы, в результате получая выход-

ную информацию о системе. Это могут быть значения полей структур ядра (например, структура ОС Linux `task_struct` содержит значения таких полей как идентификатор процесса и потока), переменные, прочитанные или записанные данные.

Инструменты интроспекции основаны на применении различных алгоритмов обработки дампов. Полученная информация может быть использована для исследования работы системы, обнаружения вредоносного программного обеспечения, отладки драйверов или ядра. Как правило, инструменты предоставляют удобные средства для получения и исследования данных.

Статический анализ позволяет изучить работу системы в определенный момент времени, получить только ту информацию, которая содержится в снимке памяти в момент его сохранения. Это свойство накладывает на анализ серьезные ограничения:

- Невозможность получения всей картины происходящего в системе. Снимок, сделанный в определенный момент времени захватывает только те данные, которые в него попали, что происходило за секунду до или после увидеть невозможно.
- Отсутствие некоторого фрагмента данных в исследуемом снимке системы. Возможна ситуация, при которой на снимок не попадет, например, вся структура данных, потому что часть ее окажется на другой странице памяти.
- Большой объем лишней информации. Как правило, интерес представляют определенные элементы памяти, но дамп включает в себя и «мусор».

Наиболее развитым статическим анализатором является Volatility [25]. Так же к этой группе относятся такие инструменты, как FATKit [26], InSight [27].

Инструмент Volatility

Volatility — это инструмент статического анализа, позволяющий получать данные из дампов оперативной памяти исследуемой системы. Поддерживает такие операционные системы, как Windows x86 x64, Linux, Mac, Android и архитектуры Intel и ARM.

Инструмент представляет собой набор модулей, за счет чего поддается расширению не только со стороны разработчиков, но и пользователей, поскольку исходный код открыт. Написан на языке Python, что позволяет подгружать различные поддерживаемые библиотеки.

Методы извлечения данных в этом инструменте являются полностью независимыми от исследуемой системы.

Основные качества Volatility:

- Модульная структура фреймворка позволяет легко поддерживать новые ОС и архитектуры.
- Открытый исходный код. Пользователи могут скачивать, пользоваться, дописывать модули, исправлять ошибки, если они есть.
- Есть скриптовое API.
- Использует в своем анализе техники обратного инжиниринга для получения недокументированных данных. Например, то что не показывает родной отладчик Microsoft (буфер консольного ввода/-вывода, история команд, сетевые структуры данных и т.д.), может быть обнаружено с помощью Volatility.
- Быстрые и эффективные алгоритмы позволяют обрабатывать даже большие дампы без лишних накладных расходов и потребления памяти.

Инструмент FATKit

FATKit это кроссплатформенный, модульный, расширяемый фреймворк для анализа энергозависимой памяти системы. FATKit был разработан для отслеживания вредоносных вторжений в системы (проникновение/эксплойты, черви, руткиты).

Задачей FATKit является автоматизация извлечения и визуализации объектов, найденных в физической памяти, он заменяет рутинную работу аналитиков. Этот фреймворк был разработан для облегчения извлечения, анализа, обобщения и визуализации интересующих данных на различных уровнях абстракции и сложности данных. Так же он включает в себя инструменты для автоматизации разработки профилей для приложений из веб браузера в ядро операционной системы.

На данный момент FATKit включает модули для реконструкции виртуального адресного пространства, трансляции виртуальных адресов в физические и модули для визуализации. Фреймворк использует ряд методов визуализации и анализа данных (data mining) для улучшения анализа и облегчения поиска в большом объеме данных.

FATKit поддерживает архитектуру x86 и операционные системы Windows и Linux.

Основные возможности фреймворка FATKit:

- Система, основанная на профилях, позволяет низкоуровневым типам отображаться на высокоуровневые конструкции и распространяться на различное ПО;
- Инструмент для автоматической генерации профилей позволяет извлекать низкоуровневые форматы объектов, когда доступен исходный код;

– Модули анализа сценариев позволяют аналитикам легко реализовывать специализированные или проприетарные методы извлечения информации с использованием высокоуровневого языка, а не ручного кодирования процедур;
– Модульное строение позволяет легко расширять фреймворк для новых архитектур и операционных систем.

В FATKit реализовано два модуля визуализации:

– Браузер объектов (обозреватель): данный модуль позволяет аналитикам интерпретировать бинарные объекты памяти на уровне абстракции языка высокого уровня исходного кода. На данный момент поддерживаются приложения, написанные на языке C, браузер позволяет аналитикам сворачивать и разворачивать объекты и их вложенные поля, указатели и приводить объекты к другим форматам данных.
– Просмотрщик адресного пространства: данный модуль позволяет аналитикам визуализировать данные в соответствующем виртуальном или физическом адресном пространстве. На данный момент набор функций включает выделение цветом объектов, представление данных в шестнадцатеричном и ASCII форматах и поддержка наложения символьных имен. Просмотрщик адресного пространства интегрирован с браузером объектов.

Инструмент InSight

InSight это инструмент для анализа памяти, разработанный для платформ IA-32 и AMD64, который делает все объекты ядра операционной системы доступными для дальнейшего анализа. Он работает с дампом физи-

ческой памяти, сохраненным на диск или с физической памятью гостевой ОС в работающей виртуальной машине. Таким образом InSight позволяет делать интроспекцию, анализ на вредоносность, а так же отладку ядра.

InSight поддерживает анализ памяти для ядра Linux и будет расширен для анализа ядра Windows в будущем.

Состояние операционной системы организовано с помощью специальных структур данных в памяти ядра (kernel memory). Экземпляры с такими структурами представляют объекты ядра. Объекты ядра содержат некоторое количество членов, которые имеют один из таких типов:

- примитивный, такой как integer или float;
- сложный тип, такой как другая структура;
- указатель на примитивный тип;
- указатель на функцию.

Значения членов примитивных типов и указателей на функции представляет актуальное состояние системы.

Целью проекта InSight является сделать эту информацию доступной для приложений, которые анализируют состояние операционной системы. Проблемой здесь является то, что организация памяти ядра в физической памяти варьируется в зависимости от различных операционных систем и их отдельных версий. Без детального знания об операционной системе, которая в настоящее время анализируется, физическая память не представляет интереса, потому что содержит только нули и единицы. Эта проблема известна как семантический разрыв в области интроспекции виртуальных машин.

InSight добавляет семантику в содержимое физической памяти, применяя к ней знания об операционной системе. При этом, InSight может пересоздать объекты ядра из памяти, их так же будет видеть ядро и все связи и значения сохранятся.

InSight работает под Linux и предоставляет следующую функциональность:

- анализ дампа физической памяти, сохраненной на диск;
- анализ физической памяти работающей виртуальной машины (если гипервизор предоставляет доступ к гостевой памяти);
- поддержка 32 и 64-битных адресных схем;
- пересоздание объектов ядра для ядра Linux;
- автоматическое удаление указателей на дальнейшие объекты;
- интерактивная оболочка для ручного анализа объектов ядра;
- JavaScript engine для автоматического анализа повторяющихся или сложных задач.

1.2.2 Методы, основанные на динамическом анализе

В контексте этой работы динамический анализ является приоритетным методом исследования систем, поскольку отслеживание поведения системы в динамике позволяет обеспечивать большее покрытие по сравнению с единоразово созданным дампом памяти. Также решение многих задач, поставленных в этой работе, требует выполнения непрерывного анализа в процессе работы системы. Динамический анализ гарантирует исследование кода, который точно будет выполнен. В этом разделе будут рассмотрены инструменты, поддерживающие данный вид анализа и использующие для этого разные подходы.

Википедия определяет динамический анализ как это анализ программного обеспечения, выполняемый при помощи выполнения программ на реальном или виртуальном процессоре (в отличие от статического анализа). Утилиты динамического анализа могут требовать загрузки специ-

альных библиотек, перекомпиляцию программного кода. Некоторые утилиты могут инструментировать исполняемый код в процессе исполнения или перед ним. Для большей эффективности динамического анализа требуется подача тестируемой программе достаточного количества входных данных, чтобы получить более полное покрытие кода. Также требуется позаботиться о минимизации воздействия инструментирования на исполнение тестируемой программы (включая временные характеристики).

Одним из популярных подходов в реализации динамической интроспекции является использование программы-агента. Это специальное приложение, используемое для получения необходимых данных, как правило это: расположение в памяти структур, смещений для полей, адреса функций и т.д. Агенты могут запускаться один раз для каждой версии каждой операционной системы с целью составления профиля (набора необходимой для проведения анализа информации), а могут работать в системе в процессе анализа.

Профиль может быть представлен как в виде файла, загружаемого в анализатор, так и может быть зашит в коде анализатора. Как правило, для каждой версии каждой ОС требуется свой профиль. В зависимости от поставленной задачи профили содержат в себе различные данные. Например, для исследования процессов необходимо иметь информацию о расположении структур ядра (task_struct в Linux), смещениях полей в этих структурах. Если профиль представляет собой файл, то содержащиеся в нем данные были получены с помощью программы-агента. Такие профили нужны анализатору для того, чтобы в процессе анализа знать откуда и какие данные необходимо считывать.

Инструменты, реализующие динамическую интроспекцию также часто используют инструментирование, т.е. внедрение в код гостевой системы функций обратного вызова, которые выполняют определенные программи-

стом действия, например, передают в инструмент доступную на данный момент информацию о системе.

Инструмент PANDA

PANDA – это система, основанная на плагинах для разных архитектур, которые могут взаимодействовать между собой. Использование плагинов улучшает анализ кода, позволяют использовать код повторно, а так же упростить сложный анализ [8; 28].

PANDA является инструментом динамического анализа. Первым этапом ее работы является сбор информации, а именно захват событий и запись журнала выполнения системы, которую затем можно будет анализировать более детально. Код, который осуществляет анализ, представляется в виде плагинов. Также предусмотрены функции обратного вызова, которые PANDA вызывает в определенных точках, таких как перед выполнением блока гостевого кода и чтение виртуальной памяти. Эти плагины собирают данные и отдают их другим плагинам. Плагины обычно пишутся быстро и интерактивно, что позволяет запускать их с воспроизведением снова и снова для получения более глубокой и точной картины всех важных аспектов при работе системы. Инициирующий плагин может просто показать список выполняющихся процессов и какие ключевые события произошли. В следующем проходе фокус может быть переведен на определенную программу или фрагмент записанной работы. Дальнейшей итерацией может быть подключен анализ помеченных данных, которые вызвали интерес и позволяет проследить за ними во время выполнения системы.

PANDA относит себя к инструментам обратной инженерии и решает следующие задачи:

– Идентификация критических уязвимостей.

– Определение целей кода и действий, которые им предусмотрены.

PANDA разработана на эмуляторе QEMU версии 2.1.0.

Плагины, написанные для PANDA — это библиотеки, которые могут быть загружены в любой момент в процессе выполнения гостевого кода. Плагины завязаны на событиях, выполняют работу в зависимости от того, что происходит в гостевом коде. Плагины при загрузке вызывают инициализирующую функцию, при выгрузке – разинициализирующую. Внутри инициализирующей функции плагины определяют какие события они хотят инструментировать. Для инструментирования доступны некоторые точки в процессе трансляции гостевого кода и выполнения. Плагины могут быть задействованы при трансляции конкретной инструкции, также они могут до или после трансляции блока, и, если включен LLVM, анализировать или модифицировать код LLVM. В процессе выполнения плагины могут реагировать на функции обратного вызова до или после выполнения блока и обращения к памяти.

PANDA предоставляет API, которое позволяет плагинам пользоваться вместе одними и теми же данными.

Большинство плагинов зависит от общей функциональности. Чтобы избежать дублирования данных, в инструменте PANDA разработан механизм взаимодействия между плагинами. Плагины могут пользоваться общим API, вызывать функции обратного вызова других плагинов и получать уведомления от них. Плагинам видны функции, которые предоставляют другие плагины, а также они могут регистрировать свои.

Межплагинное взаимодействие позволяет осуществлять сложный анализ с помощью композиции плагинов.

PANDA является мощным инструментом, позволяющим быстро и качественно делать анализ. Из недостатков можно выделить требование загрузки программы-агента в исследуемую систему, а также необходимо наличие для каждой версии каждой ОС собственного файла с параметрами, который генерируется программой-агентом.

Инструмент RTKDSM

Инструмент RTKDSM является фреймворком с открытым исходным кодом, с своей работе использует возможности анализа Volatility, упрощает и автоматизирует анализ состояний выполняющейся виртуальной машины [10]. Система RTKDSM выполняет мониторинг состояния операционной системы в виртуальной машине в реальном времени. Она использует Xen как монитор виртуальной машины и некоторые плагины Volatility. Volatility специализируется на анализе дампов памяти. RTKDSM же анализирует память гостевой системы напрямую, не используя дампы.

Основанная на дампах архитектура Volatility не эффективна в режиме мониторинга, поскольку нужно анализировать целый дамп каждый раз, как только понадобится информация. Таким образом, RTKDSM использует Volatility только для поиска структур данных исследуемых операционных систем. После того как адреса структур получены, RTKDSM использует собственный агент мониторинга, который отслеживает изменения в этих структурах.

Интроспекция, реализованная в этом инструменте, обеспечивает возможность исследования виртуальных машин безагентным способом, сбор данных о состоянии системы осуществляется гипервизором. Анализ этих состояний происходит с помощью извлечения данных из работающей си-

стемы без загрузки агента в систему. В этой работе проблема семантического разрыва решается с помощью фреймворка Volatility. Он способен определять нужную информацию и его использование упрощает и автоматизирует анализ состояний виртуальной машины. RTKDSM расширяемый фреймворк, который предназначен для анализа конкретного приложения. Кроме того, RTKDSM может осуществлять мониторинг изменения состояний ВМ в режиме реального времени. Для уменьшения накладных расходов система использует некоторое количество оптимизаций.

Гипервизор имеет очень мало данных о системе-госте, у него нет структур ядра и другой семантической информации, поэтому его возможности сильно ограничены. Первый шаг в преодолении семантического разрыва заключается в сборе информации о гостевой ОС путем обнаружения и изучения внутренних структур данных ОС, используемых в системных вызовах или внутренних вызовах ядра. Этот шаг обычно труден, отнимает много времени и сопряжен с ошибками (исследуются захваченные страницы). Найти унифицированный способ определения этих данных для разных версий ОС является важной задачей.

RTKDSM предназначена для решения этой проблемы за счет автоматизации процесса извлечения структур данных ядра и для возможности отслеживать любые изменения выбранных структур в реальном времени. RTKDSM активно использует наработки Volatility. Кроме того, RTKDSM не требует от пользователей какой-либо информации об ОС, также не требуется доступ к исходным файлам ОС. Поскольку слежение за выделенными структурами ядра требует серьезных затрат производительности, фреймворк имеет возможность включить настраиваемую оптимизацию.

Требования, которые предъявили разработчики RTKDSM для инструмента:

- Отсутствие модифицирования гостевой системы и внедрения агентов в систему.
- Структуры данных ядра должны соответствовать известной семантической и синтаксической модели структур данных.
- Наиболее часто используемые структуры находятся в невыгружаемой памяти, поэтому данное исследование сконцентрировано на такой памяти.

RTKDSM предназначена для работы в двух режимах. В режиме идентификации и анализа приложения используют RTKDSM, чтобы обнаружить расположение конкретных структур ядра и значений определенных полей внутри этих структур, а затем выводят семантическое значение возвращаемых значений.

В режиме мониторинга приложения используют RTKDSM чтобы в реальном времени следить за изменениями выбранных структур данных ядра, а потом реагировать на эти изменения соответствующим образом.

RTKDSM обрабатывает запросы от приложений. Исходные данные для этих запросов получены с помощью Volatility.

RTKDSM состоит из двух агентов: агент для интроспекции и агент для мониторинга. Агент интроспекции собирает и анализирует структуры данных ядра в исследуемой ВМ, в то время как агент мониторинга обнаруживает попытки изменения контролируемых структур ядра.

Текущий прототип RTKDSM реализован на x86, который поддерживает технологию Intel VT, запускает гипервизор Xen и поддерживает гостевые ОС Windows и Linux.

RTKDSM — это мощный инструмент, который позволяет исследовать ВМ в реальном времени без участия агента. Он умеет следить за заранее найденными структурами данных и оповещать систему, если что-то изменилось. Это достигается с помощью Volatility.

Используя Volatility, система RTKDSM способна обнаруживать и анализировать структуры данных ядра в нескольких версиях ОС Windows и Linux без доступа к исходному коду.

В результате получается система, которая принимает запросы на высоком уровне, а потом автоматически получает локации в физической памяти структур ядра и те, что представляют интерес могут непрерывно контролироваться системой в режиме реального времени.

Инструмент TEMU

TEMU представляет собой расширяемую платформу, которая учитывает общие проблемы и требования в динамическом анализе и, таким образом, значительно облегчает создание на ее основе различных методов пользовательского анализа. Прежде всего, часто требуется представление всей системы, включая ядро ОС и все приложения, запущенные в системе. Такое полносистемное представление позволяет анализировать действия, происходящие в ядре ОС (например, вредоносное ПО ядра и уязвимости ядра) и взаимодействия между несколькими процессами[29].

Основная идея состоит в том, чтобы запустить всю операционную систему (включая общие приложения) внутри полносистемного эмулятора и выполнить двоичный код, представляющий интерес в этой эмулируемой среде. Во время выполнения двоичного кода отслеживается и анализируется его поведение на уровне инструкций. TEMU основан на полносистемного эмуляторе с открытым исходным кодом QEMU.

Для выполняющейся инструкции нужно знать к какому процессу, потоку и модулю она принадлежит. В некоторых случаях инструкции могут динамически генерироваться и выполняться в куче.

Для поддержания отображения между адресами в памяти и модулях требуется информация из гостевой операционной системы. Для Linux и Windows используется два разных подхода к извлечению информации о процессе и модулях.

Для Linux можно напрямую получать информацию о процессах и модулях, потому что соответствующие структуры ядра известны, а адреса соответствующих символов также экспортированы в файл `system.map`. Для того, чтобы поддерживать информация о процессе и модуле во время выполнения перехватываются несколько функций ядра, например `do_fork` и `do_exec`.

Для Windows доступа к исходному коду ядра нет. Хотя некоторые структуры данных ядра для некоторых версий Windows были получены с помощью обратной инженерии, это не является надежным способом извлечения семантики уровня ОС в разных версиях и пакетах обновления. Вместо этого для этой цели был разработан модуль ядра. Этот модуль загружается в гостевую операционную систему для сбора обновленной информации отображения в память. Этот модуль ядра регистрирует две функции обратного вызова. Первая функция обратного вызова вызывается всякий раз, когда процесс создается или удаляется. Вторая функция обратного вызова вызывается, когда загружается новый модуль, и определяет диапазон адресов в виртуальной памяти, который занимает новый модуль. Кроме того, модуль ядра получает значение регистра `CR3` для каждого процесса. Поскольку регистр CR3 содержит физический адрес таблицы страниц текущего процесса, он отличается (и уникален) для каждого процесса. Вся информация, описанная выше, передается в TEMU через предопределенный порт ввода-вывода. В настоящее время это оптимальное решение, лучшее решение этой проблемы рассматривается в качестве будущей работы.

TEMU получает текущий идентификатор потока, который важен для анализа многопоточных программ. В Windows структура данных для текущего потока отображается в известный виртуальный адрес, поэтому можно получить эту информацию напрямую. Для Linux также можно получить информацию о потоках с помощью аналогичного подхода. Однако текущая реализация TEMU получает информацию о потоках только в Windows, а Linux будет поддержан в будущем.

В бинарном модуле анализируется его заголовок и извлекаются экспортированные имена символов и смещения. После того как получено расположение всех модулей, можем определить абсолютный адрес каждого символа, добавив базовый адрес модуля и его смещение. Эта функция очень полезна, поскольку все API-интерфейсы в пользовательских библиотеках и API-интерфейсах ядра экспортируются их модулями. Информация о символах содержит важные семантические сведения: по имени функции можно определить для какой цели предназначена эта функция, какие входные аргументы она принимает и какие генерируются выходные аргументы и возвращаемое значение. Кроме того, информация о символах упрощает привязку функции — вместо указания фактического адреса функции можно указать имя его модуля и имя функции. Затем TEMU автоматически отобразит фактический адрес функции для пользователя. В текущей реализации TEMU может анализировать образы памяти для PE и ELF модулей и, таким образом, поддерживает обе операционные системы Windows и Linux.

Внешние вызовы функций. Сначала рассмотрим случай, когда анализируемая программа выполняет вызов внешней функции, тело функции которого находится в другом модуле кода (т.е. в библиотеке). Используется следующее наблюдение: всякий раз, когда программа выполняет вызов функции, значение указателя стека во время вызова функции должно быть

больше значения указателя стека в то время, когда точно выполняется вызов этой внешней функции. Это связано с тем, что при выполнении вызовов функций в стек должны быть вставлены один или несколько фреймов, а в архитектуре x86 стек растет к меньшим адресам.

Учитывая этот факт, используется следующий подход для определения выполнения внешней функции: всякий раз, когда выполнение переходит в анализируемый код, записывается текущее значение указателя стека вместе с текущим идентификатором потока. Когда выполнение выходит из этой области кода необходимо проверить есть ли записанный указатель стека для текущего идентификатора потока, и если да, то является ли это значение меньше текущего указателя стека. Если это так, запись удаляется, поскольку код больше не находится в стеке. Когда наблюдается интересное поведение, осуществляется проверка есть ли записанный указатель стека под текущим идентификатором потока. Если это так, считается, что выполнение происходит от имени анализируемого кода, потому что это означает, что анализируемый код находится в стеке вызовов.

TEMU способен обеспечить представление всей системы, выполнить анализ из коробки и сократить семантический разрыв между уровнями аппаратного уровня и уровня ОС. К ограничениям этого инструмента можно отнести использование структур ядра, что влечет за собой сложность реализации и плохую переносимость между различными ОС.

Инструмент DECAF

DECAF — это полносистемный фреймворк для бинарного анализа на основе QEMU [9; 30]. DECAF предлагает JIT (Just-in-time) интроспекцию, совмещенную с анализом помеченных данных на уровне инструкций.

Реализован в виде плагинов, имеет простой в использовании интерфейс. DECAF осуществляет точный контроль над инструкциями для выполнения оптимизаций «на лету». Представлено три платформо-независимых плагина:

– Трассировщик инструкций.
– Анализатор устройств ввода (клавиатура, мышь).
– Трассировщик API функций.

Инструмент DECAF является развитием TEMU, но теперь он предлагает более качественные результаты с большей гарантией правильности, чем в TEMU, а также более эффективное проведение анализа.

Проблемы, которые ставят перед собой разработчики DECAF:

– Как извне полностью восстановить семантическое представление уровня ОС (интроспекция).
– Как реализовать событийную парадигму, где одновременно будет и корректно и эффективно.
– Как точно и без потерь реализовать анализ помеченных данных, а именно саму пометку.
– Как реализовать поддержку кроссплатформенного анализа.

Ключевые компоненты DECAF:

– JIT интроспекция.
– Точный анализ помеченных данных.
– Событийный программный интерфейс.
– Динамическое управление инструментированием.

Как платформа для бинарного анализа, DECAF нуждается в переработке семантических знаний виртуальной машины уровня ОС: процессы, потоки, модули кода, экспортируемые символы.

Процессы: необходимо знать какой процесс работает в системе. Так как многие задачи анализа фокусируются на одном или нескольких пользовательских процессах, информация о процессах крайне важна.

Потоки: большинство программ многопоточные. Знания о том, какой поток работает в процессе так же важно для многих задач анализа.

Модули кода: в память процесса загружены основной исполняемый файл и несколько общих библиотек. Бинарный анализ часто нуждается в знаниях из какого модуля кода пришла инструкция.

Экспортируемые символы: общие библиотеки экспортируют список функций, чтобы другие модули когда могли динамически связаться друг с другом и вызывать эти экспортированные функции по имени. Получение этих экспортированных символов может сильно помочь в понимании поведения программы на уровне API.

DECAF реализует метод интроспекции и отслеживание потока данных, которое несет гораздо меньшие накладные расходы на производительность во время выполнения, чем на других платформах при полносистемном анализе. Он обеспечивает простой, управляемый событиями API-интерфейс для построения плагинов.

Инструмент Nitro

Nitro — инструмент для интроспекции, работающий на уровне гипервизора [31]. Основной целью является трассировка и мониторинг системных вызовов.

Nitro очень гибкая система, поддерживает все три механизма системных вызовов, используемых в архитектуре Intel x86. Производительность Nitro является высокой, позволяет проводить анализ в реальном времени.

Преодоление семантического разрыва осуществляется через знания об архитектуре аппаратного обеспечения гостевой ОС.

Мониторинг системных вызовов тут используется для обнаружения вредоносной активности.

Поддерживает архитектуру Intel, работает на операционных системах Windows/Linux 32/64-bit.

Nitro использует мониторинг и трассировку системных вызовов на уровне аппаратного обеспечения.

В Intel 32 и 64 не поддерживается перехват событий системных вызовов через гипервизор. Разработчики Nitro этого добились, используя системные прерывания (page fault) для которых перехват поддержан в Intel Virtualization Extensions (VT-x). Следовательно, можно создать эффективный механизм для перехвата системных вызовов даже когда это не поддерживается.

Все три механизма системных вызовов отличаются друг от друга и для каждого должен быть сделан отдельный механизм перехвата.

Гибкость решения достигается за счет самостоятельного конфигурирования получаемых данных.

Nitro — мощный и гибкий инструмент для трассировки и мониторинга системных вызовов. Он поддерживает все три механизма системных вызовов, предоставляемые архитектурой Intel 32-64. Он успешно трассирует системные вызовы с Windows, Linux 32 и 64, и разработчики уверены, что также хорошо он будет работать и с другими системами. Кроме того, доказано что реализация может собирать данные в реальном времени. И, наконец, все механизмы реализованы таким образом, что злоумышленникам их не обойти. Гибкий и безопасный Nitro позволяет использовать его для различных приложений, таких как машинное обучение к обнаруже-

нию вредоносных программ, мониторинг приманок (honeypot), а так же песочниц (sandbox).

Инструмент Virtuoso

Virtuoso представляет новый подход к интроспекции виртуальных машин с минимальным человеческим усилием [32]. Анализируя небольшие трассы гостевых программ, вычисляя нужную для интроспекции информацию, можно производить новые программы, которые получают ту же информацию извне виртуальной машины. Эффективность данного метода продемонстрирована на автоматической генерации 17 программ, которые получают информацию о безопасности в трех разных операционных системах. Функциональность не зависит от выбранной гостевой системы.

Данная методика позволяет инструментам интроспекции работать на нескольких платформах и позволяет разрабатывать приложения для обеспечения безопасности.

В настоящее время Virtuoso поддерживает множество полезных механизмов интроспекции на различных операционных системах, однако еще есть ряд областей, в которых возможно развитие.

Поддержка нескольких адресных пространств: возможно самым серьезным ограничением Virtuoso является его неспособность правильно обрабатывать трассы, в которых присутствует несколько адресных пространств (то есть трассы, в которых появляется несколько процессов). В простейшем случае, другие процессы не делают никакой работы, связанной с интроспекцией, но их трудно отфильтровать, потому что трасса будет содержать части планировщика кода для переключения на другой процесс и идентифицировать и удалить этот код автоматически сложно. В насто-

ящее время Virtuoso обходит эту проблему, запуская свои тренировочные программы с высоким приоритетом (используя start/realtime на Windows и chrt на Linux).

Сложнее выглядит проблема анализа и извлечения программ, которые используют межпроцессное взаимодействие. Решение этой проблемы особенно важно для интроспектирования микроядерной архитектуры, где многие задачи выполняются набором скооперированных процессов. Основная проблема в том, что хотя возможно извлечь соответствующий код для каждого работающего процесса, там будут данные из других процессов, которые должны быть прочитаны гостевой операционной системой во время выполнения. Сгенерированная программа нуждается в другом пути нахождения соответствующей кооперации процессов, взаимодействующей во время выполнения, чтобы прочитать эти данные, что требует значительных знаний в предметной области. Требуются дополнительные исследования, чтобы определить в какой степени это может быть автоматизировано.

Самомодифицирующийся код: Virtuoso не поддерживает исследование самомодифицирующегося кода. Это не вызывает проблем ни с одной из интроспекций (операционные системы редко используют самомодифицирующийся код), такая поддержка необходима для работы с вредоносным кодом, что является планом на будущее.

Перемещение и рандомизация размещения адресного пространства: на данный момент Virtuoso предполагает, что модули, содержащие данные, не были перенесены. Это предположение может не выполняться, однако, модули могут быть загружены с разных базовых адресов по различным причинам, включая безопасность (например, Address Space Layout Randomization). Если это происходит, то считанные данные из этих модулей могут быть найдены по разным адресам и интроспектирующая программа не в состоянии корректно определить их местонахождение. Хотя эта

проблема решаема путем включения большего количества знаний, но это усложнит поддержку новых операционных систем.

Virtuoso — это система для автоматической генерации инструментов интроспекции, с помощью которых из низкоуровневых источников данных можно извлечь семантически значимую информацию. Применяя техники полносистемного динамического слайсинга, Virtuoso решает задачи, которые требовали от часов до недель тяжелого труда от программистов, работающих в сфере обратной инженерии, за минуты с помощью небольших и простых трудозатрат рядовых программистов. Это все должно помочь сузить семантический разрыв, снять существенную часть проблем в расследовании вредоносных программ, безопасности, основанной на виртуализации и анализе вредоносного кода.

Инструмент PinOS

PinOS — это расширение динамического фреймворка Pin [33] для инструментирования системы, например, для использования как кода ядра, так и кода пользователя. Это достигается путем взаимодействия между исследуемой системой и аппаратным обеспечением с использованием технологий виртуализации. PinOS основа на мониторе виртуальной машины Xen с технологией Intel VT, чтобы обеспечить возможность использования немодифицированных ОС. Наследуя мощный инструментарий API от Pin, а также внедряя новый API для аппаратуры на системном уровне, можно использовать PinOS для написания системных инструментальных средств для таких задач, как анализ программ и архитектурные исследования. На сегодняшний день PinOS может загружать Linux на IA-32 в однопроцес-

сорном режиме и может исследовать сложные приложения, такие как базы данных и веб-серверы [34].

PinOS стремится наблюдать и обрабатывать каждую инструкцию, выполняемую в исследуемой системе, поэтому она должна вставляться между существующей операционной системой и основным оборудованием. Для внедрения слоя PinOS под уровнем гостевой системы используется технологи виртуализации. Модифицированная версию монитора виртуальной машины Xen [35] запускается на «голом» оборудовании. Для запуска немодифицированных гостевых операционных систем само оборудование должно быть оснащено поддержкой аппаратной виртуализации Intel на x86 (VT-x). В терминологии Xen виртуальные машины называются доменами. Существует одна привилегированная виртуальная машина, называемая доменом хоста или Dom0, а все другие виртуальные машины называются гостевыми доменами или DomU. Исследуемая система, включая ее саму и все ее приложения, запускаются в гостевом домене. Внутри этого гостевого домена PinOS прозрачно внедряется под исследуемую операционную систему, так что PinOS может наблюдать каждую инструкцию, когда-либо выполняемую в данной системе.

Xen был выбран по нескольким веским причинам: это надежный, широко применимый, простой и эффективный монитор виртуальной машины, он доступен через лицензию с открытым исходным кодом. Драйвер PinOS вводится внутрь Xen, изменяя его. В дополнение к Xen используется слой PinOS, который состоит из трех основных компонентов: «Code Cache», «Instrumentation Engine» и Pintool.

PinOS может быть использован для полносистемного анализа программ и исследования архитектуры с помощью профилировщика кода и эмулятора кэша, построенных поверх PinOS. Будущая работа включает

перенос PinOS на другие платформы, такие как 64-разрядные x86 и ОС Windows.

1.2.3 Сравнительный анализ инструментов, реализующих различные методы мониторинга объектов ОС

Классификация аспектов, относящихся к мониторингу объектов операционной системы:

- Вид интроспекции (Method - M)
  - Динамический анализ «на лету» (**MF**)
  - Динамический анализ по снимкам системы (**MD**)
  - Динамический анализ по записанным трассам (**MT**)
  - Статический анализ по снимкам системы (**MS**)
- Особнности метода (Feature - F)
  - Внедрение внешних модулей в систему (**FM**)
  - Наличие исходных кодов (**FS**)
  - ABI (**FI**)
- Добываемая информация (Information - I)
  - Объекты ядра (**IK**)
  - Структуры данных приложений (**IA**)
  - Содержимое памяти (**IM**)
  - Вызываемые функции API (**IL**)
  - Вызываемые системные функции (**IS**)

Таблица 1.1

Сравнительная таблица методов мониторинга объектов ОС

Название инструмента	Вид интроспекции				Особенности метода			Выходная информация				
	MF	MD	MT	MS	FM	FS	FI	IK	IA	IM	IL	IS
Volatility				+		+	+	+	+	+	+	+
FATKit				+		+		+	+	+		
InSight				+	+	+		+		+		
TEMU	+				+		+	+	+	+	+	+
PANDA			+		+	+	+	+	+	+	+	+
DECAF		+			+	+	+	+	+	+	+	+
RTKDSM	+	+				+	+	+	+	+	+	+
Nitro	+						+					+
Virtuoso			+		+	+		+	+	+		
PinOS	+					+	+			+		
Новый метод	+		+				+	+		+	+	+

1.3 Выводы по главе

В контексте этой работы интерес представляют только те инструменты, которые реализуют динамическую интроспекцию. Рассмотренные в этой группе инструменты имеют ряд ограничений, а именно:

1. Использование структур ядра. Данные о структурах ядра не всегда доступны, например, когда используются ОС с закрытым исходным кодом. Получение этой информации сопряжено со сложным анализом системы. Кроме того, в различных версиях эти структуры могут быть разными, следовательно, необходимо иметь профиль под каждую версию каждой операционной системы.
2. Невозможность работы с неизменяемыми образами исследуемых систем. В связи с использованием программ-агентов, рассмотренные методы не предполагают анализ систем такого рода.
3. Трудности сопровождения. Учитывая изменяемость операционных систем, а именно их внутреннего устройства, возникают трудности с поддержкой работоспособности инструмента анализа. Чем меньше изменений терпят затронутые части исследуемой системы в процессе развития, тем проще сопровождать продукт для ее анализа.

Таким образом, возникает задача разработки методов мониторинга объектов ОС, обеспечивающих построение анализатора без использования исходных кодов и настроек компиляции исследуемой системы. Анализатор не должен загружать в исследуемую систему инструментальный код, а также должен обеспечивать переносимость между ОС одного семейства.

Глава 2. Модель исследуемой системы

В этой работе предлагается модель исследуемой системы. Модель представляет собой набор сущностей, информацию о которых необходимо получить, и источника информации, с помощью которого будет происходить мониторинг.

Были выделены следующие сущности: файл, процесс, адресное пространство, отображение и модуль. Сущности характеризуются набором атрибутов и списком операций. В качестве источника информации выступает поток инструкций, из которого выделяются системные вызовы и API вызовы.

Если рассматривать абстрактную модель операционной системы, то все эти объекты в том или ином виде будут там присутствовать [36].

Ниже в подразделах приведено описание каждой сущности, а на рисунке 2.1 представлена структурная схема, определяющая взаимосвязь сущностей между собой.

2.1 Описание модели исследуемой системы

2.1.1 Файл

Одним из ключевых понятий, поддерживаемых практически всеми операционными системами, является файловая система. Основная функция операционной системы — скрыть специфику дисков и других устройств ввода-вывода и предоставить программисту удобную и понятную абстракт-

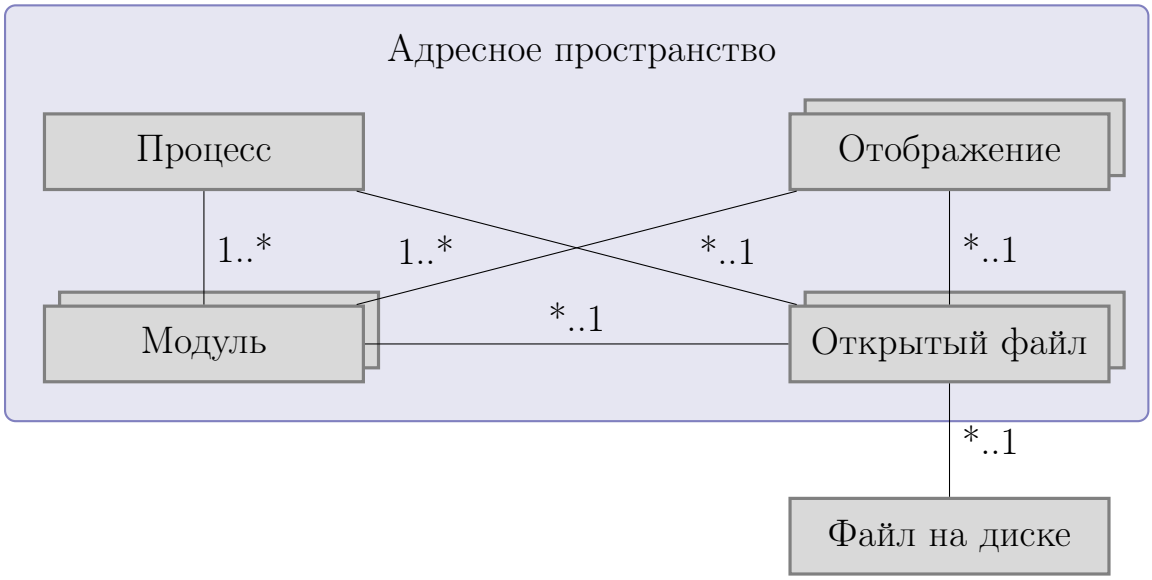

Рисунок 2.1 — Структурная схема модели

ную модель, состоящую из независимых от устройств файлов. Вполне очевидно, что для создания, удаления, чтения и записи файлов понадобятся системные вызовы. Перед тем как файл будет готов к чтению, он должен быть найден на диске и открыт, а после считывания — закрыт. Для проведения этих операций предусмотрены системные вызовы. Чтобы предоставить место для хранения файлов, многие операционные системы персональных компьютеров используют каталог как способ объединения файлов в группы.

Каждый файл, принадлежащий иерархии каталогов, может быть обозначен своим полным именем с указанием пути к файлу, начиная с вершины иерархии — корневого каталога. Этот абсолютный путь состоит из списка каталогов, которые нужно пройти от корневого каталога, чтобы добраться до файла, где в качестве разделителей компонентов служат символы косой черты.

Перед тем как с файлом можно будет работать в режиме записи или чтения, он должен быть открыт. На этом этапе происходит также проверка прав доступа. Если доступ разрешен, система возвращает целое число,

называемое дескриптором файла, который используется в последующих операциях. Если доступ запрещен, то возвращается код ошибки [36].

Под сущность **Файл** попадают все использующиеся в работе исследуемой системы файлы, как системные, так и пользовательские.

Файл обладает следующими атрибутами:

– Имя
– Дескриптор
– Права доступа

Также эта сущность наделена рядом операций, а именно:

– Создать
– Открыть
– Читать
– Писать
– Закрыть

В контексте этой работы все файлы логически разделены на две группы: существующие на диске и открытые в данный момент. В этой модели файлы представляют интерес и как самостоятельные сущности и как вспомогательные объекты для обнаружения других сущностей, а именно модулей. В качестве самостоятельных объектов они предоставляют информацию о своем поведении в системе (какие файлы были открыты, что было прочитано или записано и т.д.), в качестве вспомогательных — помогают отслеживать загрузку модуля в память.

2.1.2 Процесс

Другим ключевым понятием во всех операционных системах является **Процесс**. Процессом, по существу, является программа во время ее

выполнения. С каждым процессом связано его адресное пространство — список адресов ячеек памяти от нуля до некоторого максимума, откуда процесс может считывать данные и куда может записывать их. Адресное пространство содержит выполняемую программу, данные этой программы и ее стек. Кроме этого, с каждым процессом связан набор ресурсов, который обычно включает регистры (в том числе счетчик команд и указатель стека), список открытых файлов, необработанные предупреждения, список связанных процессов и всю остальную информацию, необходимую в процессе работы программы. Таким образом, процесс — это контейнер, в котором содержится вся информация, необходимая для работы программы [36].

Адресное пространство процесса является важной его характеристикой, поскольку зная идентификатор адресного пространства можно отличать процессы друг от друга во время работы системы.

В некоторых операционных системах процессы представляют собой иерархию, где четко определены родитель и потомок, в других как таковой иерархии нет, но объединяет разные модели то, что все процессы изначально зарождаются от главного процесса системы.

Так как процесс это запущенная программа, то можно сказать, что у него есть имя.

Процессы обладают такими атрибутами, как:

– Идентификатор процесса

– Имя загруженного образа

– Идентификатор адресного пространства

Несмотря на огромную значимость процесса в системе, операций над процессами можно выделить только две: создание процесса и его уничтожение. Имея возможность знать какой процесс работает в определенный

период времени возможности анализа сильно возрастают, ведь внутри процесса осуществляется работа и с файлами и с внешними модулями.

2.1.3 Адресное пространство

Следующей сущностью является **Адресное пространство**. Оно уже упоминалось при описании сущности **Процесс**, поскольку они связаны между собой. Это набор адресов, который выделен для каждого процесса и может быть им использован.

Каждый процесс в многозадачной операционной системе выполняется в своем виртуальном адресном пространстве. Соответствие между виртуальными и физическими адресами описывается с помощью таблицы страниц. Таблицы создает и заполняет ядро, а процессор обращается к ним при необходимости осуществить трансляцию адреса. Каждый процесс работает со своим набором таблиц. По умолчанию адресное пространство каждого процесса изолировано. Данные двух разных процессов, записанные по одному и тому же виртуальному адресу, оказываются в разных страницах физической памяти при помощи корректной работы системы трансляции адреса. В ряде случаев изоляция может быть частично снята (файлы, отображаемые в память; разделяемая память). В подобных случаях нужно отдельно обеспечить контроль доступа к области памяти, для чего создается отдельный объект (объект-секция или объект-раздел, section object), включающий атрибуты защиты [37].

Адресное пространство можно назвать неофициальным идентификатором процесса. Это сущность, которая связана с определенным процессом.

Например, в архитектуре x86 для идентификации адресного пространства выступает регистр `cr3`. Его значение уникально для каждого процесса.

Атрибутом адресного пространства является его идентификатор, а операцией — запрос этого идентификатора. Так как адресное пространство связано с процессом, то запросив идентификатор адресного пространства можно определить какой процесс на данный момент работает в системе.

2.1.4 Отображение

Вообще **Отображение** файла в память это механизм, идея которого состоит в том, что процесс может выдать системный вызов для отображения файла на какую-то часть его виртуального адресного пространства.

Отображаемые файлы предоставляют альтернативную модель для ввода-вывода. Суть заключается в том, что к файлу можно обращаться как к большому символьному массиву, находящемуся в памяти. В некоторых ситуациях эта модель является более удобной [36].

Другими словами отображение — это способ работы с файлами в некоторых операционных системах, при котором всему файлу или некоторой непрерывной его части ставится в соответствие определённый участок памяти. В контексте этой работы отображение является сущностью, наделенной атрибутами и операциями.

Атрибутами отображения являются:

– Имя отображенного файла

– Дескриптор файла

Операции, связанные с секциями:

– Создать

– Открыть

– Удалить

общие слова про маппинг. как в винде как в линуксе

Большинство современных операционных систем или оболочек поддерживают те или иные формы работы с файлами, отображенными на память. Например, функция `mmap`, создающая отображение для файла с данным дескриптором, начиная с некоторого места в файле и с некоторой длиной — является частью спецификации POSIX. Таким образом, огромное количество POSIX-совместимых систем, таких как UNIX, Linux, FreeBSD, Mac OS X или OpenVMS, поддерживают общий механизм отображения файлов. ОС Microsoft Windows также поддерживает определённый API для этих целей, например, `CreateFileMapping` [38].

2.1.5 Модуль

Под **Модулем** понимается бинарный код, который загружается в систему динамически [39]. В контексте этой работы модулями являются библиотеки (*.dll, *.so) и исполняемые файлы (*.exe), однако в текущей реализации используются только библиотеки. Модули, как правило, предоставляют информацию, которая была в них заложена, например, отладочная информация или список экспортируемых функций (явно предоставляемых самой библиотекой для «внешнего мира»).

Модули строятся по определенным форматам (для Windows это `Portal Executable (PE)`, для Linux `Executable and Linkable Format (ELF)`). Как правило в формат входит заголовок, в котором определено что и по каким адресам находится в библиотеке.

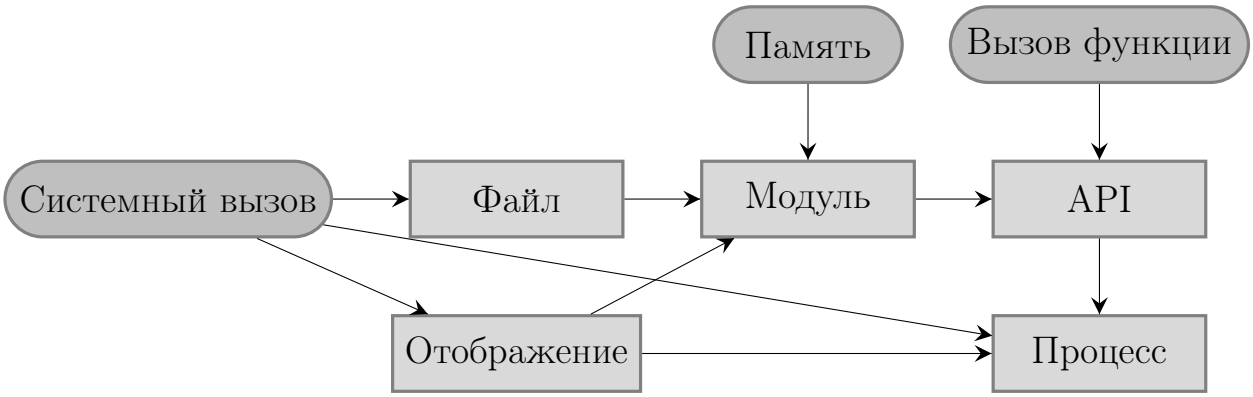

Рисунок 2.2 — Схема потоков данных

В привычном виде для модулей не определены атрибуты и операции, однако модуль характеризует его имя, к операциям можно отнести извлечение полезной информации, например список экспортируемых функций или отладочные символы.

2.1.6 Источники информации

Основным материалом для исследования является поток данных. Важное место в этом потоке занимают инструкции. Именно инструкции позволяют определить, что произошел системный вызов и только по адресу инструкции распознается нужный API вызов.

Помимо системных и API вызовов в модель должно быть включено и исследование памяти, являющееся своего рода источником информации. Память анализируется при рассмотрении модулей.

Поток данных в представленной модели изображен на рисунке 2.2.

Системный вызов

В любой традиционной операционной системе поддерживается некоторый механизм, который позволяет пользовательским программам обра-

щаться за услугами ядра ОС. Такие средства называются системными вызовами.

Системные вызовы (system calls) — интерфейс между операционной системой и пользовательской программой. Они создают, удаляют и используют различные объекты, главные из которых процессы и файлы. Пользовательская программа запрашивает сервис у операционной системы, осуществляя системный вызов. Имеются библиотеки процедур, которые загружают машинные регистры определенными параметрами и осуществляют прерывание процессора, после чего управление передается обработчику данного вызова, входящему в ядро операционной системы. Цель таких библиотек сделать системный вызов похожим на обычный вызов подпрограммы.

Основное отличие состоит в том, что при системном вызове задача переходит в привилегированный режим или режим ядра (kernel mode). Поэтому системные вызовы иногда еще называют программными прерываниями в отличие от аппаратных прерываний, которые чаще называют просто прерываниями.

В этом режиме работает код ядра операционной системы, причем он исполняется в адресном пространстве и в контексте вызвавшей его задачи. Таким образом, ядро операционной системы имеет полный доступ к памяти пользовательской программы, и при системном вызове достаточно передать адреса одной или нескольких областей памяти с параметрами вызова и адреса одной или нескольких областей памяти для результатов вызова [40].

Если рассматривать системный вызов как сущность, то можно выделить следующие свойства:

– Идентификатор системного вызова

– Имя системного вызова

– Параметры

– Возвращаемое значение

С системными вызовами связаны две операции:

– Начать выполнение (например, инструкция `Sysenter`)

– Закончить выполнение (например, инструкция `Sysexit`)

Реализация механизма системных вызовов в разных ОС может отличаться. В современных ОС на архитектуре x86 наиболее часто встречаются реализации через инструкции процессора `Sysenter/Sysexit` или `Syscall/Sysexit`, однако иногда встречается реализация через прерывания `int 80h/iret`.

Системные функции предоставляют такую информацию, как: сам факт вызова функции (можно фиксировать в журнале и отслеживать поведение системы), входные параметры, выходные параметры и возвращаемое значение.

Вызов функций из динамических библиотек

Следующим источником информации является API вызов. По смыслу он похож на системный вызов, однако этот источник находится уровнем выше и для его использования необходимо провести ряд подготовительных действий.

API вызов является вызовом функции из библиотеки, поэтому прежде чем его использовать для получения информации об объектах, необходимо получить информацию об этой функции. Такой информацией является:

– Имя функции

– Адрес функции

– Параметры

Перехватываются API функции с помощью адреса функции, то есть в потоке инструкций определяется инструкция, отвечающая за вызов функции (например, `call`). Параметром таких инструкций является адрес, на

который в дальнейшем осуществляется переход. Так как адреса интересующих функций известны, можно извлекать информацию. Окончание выполнения следует искать в стандартных функциях возврата, например, `ret`. Как и системные функции, они предоставляют такую информацию, как: сам факт вызова функции, входные параметры, выходные параметры и возвращаемое значение.

2.2 Выводы по главе

В этой главе представлена разработанная модель исследуемой системы, представляющая собой набор взаимосвязанных сущностей.

Для достижения поставленной в этой работе цели вышеупомянутый набор сущностей достаточен, однако, при необходимости она может быть расширена и другими компонентами систем, например, потоками, объектами синхронизации и др.

То, что выделенных сущностей достаточно, подтверждается опытом разработчиков инструментов динамического анализа, описанных в главе 1.

Глава 3. Мониторинг объектов ОС

3.1 Описание метода мониторинга событий виртуальной машины для получения информации об объектах ОС

Ключевыми отличиями существующих методов являются затрагивание внутренних структур ядра, а также использование программ-агентов. Первое требует доступа к исходным кодам исследуемой системы (что иногда проблематично), второе же не может быть использовано при анализе встроенных систем. Разрабатываемый метод должен обходить эти недостатки: использовать только те знания о системе, которые есть в открытом доступе, и не прибегать к программам-агентам.

Поэтому использование недокументированных структур данных ядра системы использовать не будет, вместо них задействован ABI (Application Binary Interface). ABI — это набор соглашений для доступа приложения к операционной системе и другим низкоуровневым сервисам, спроектированный для переносимости исполняемого кода между машинами, имеющими совместимые ABI. ABI можно рассматривать как набор правил, позволяющих компоновщику объединять откомпилированные модули компонента без перекомпиляции всего кода. Он регламентирует: использование регистров процессора, состав и формат системных вызовов и вызовов одного модуля другим, формат передачи аргументов и возвращаемого значения при вызове функции [41].

Для реализации подхода с использованием модели, которая была описана в главе 2 необходимо иметь набор данных, предоставляемый ABI: как происходит системный вызов, идентификаторы системных вызовов, имена и параметры системных функций, размер указателя на область памяти,

возвращаемое значение. Например, для архитектуры x86 данные выглядят следующим образом [42]: [`sysenter, sysexit`] — инструкции для входа в системный вызов и выхода из него, [`esp, cr3`] — регистры, определяющие контекст выполнения, [`eax`] — регистр, хранящий номер системного вызова на входе и возвращаемое значение функции на выходе, параметры функций передаются слева направо, коды системных вызовов и параметры функций определены документацией.

Идея метода заключается в том, чтобы получать высокоуровневую информацию из низкоуровневой, имея минимальные знания об исследуемой системе. Имеется поток инструкций, которые генерирует эмулятор QEMU, из этих инструкций отфильтровываются те, что отвечают за системные вызовы. Для этой операции необходимо знать опкоды для начала и окончания системного вызова, идентификатор адресного пространства и значение указателя стека для верного сопоставления пары начало–конец, а так же идентификаторы системных вызовов.

Системный вызов предполагает вызов системной функции, которая имеет параметры и возвращаемое значение. Далее из потока системных вызовов выбираются те, что представляют интерес, а именно вызовы, связанные с файлами, процессами, отображениями. Список рассматриваемых вызовов может быть расширен по мере необходимости. При входе в системную функцию уже доступны входные параметры функций, но так как функция еще не выполнилась, то выходные параметры и возвращаемые значения сразу узнать не представляется возможным. Для их получения необходим перехват инструкции окончания системного вызова. Требуется сопоставить начало и конец выполнения функции, после чего можно считать выходные данные. Информация, полученная в результате работы системных функций (название функции, значения параметров и возвращаемых значений), собирается в журналы и может быть использована как

самостоятельная единица анализа, как отправная точка для дальнейших исследований более сложных элементов системы, а также может быть передана другим анализаторам (например, анализу помеченных данных).

Для анализа некоторых частей системы, например, файловых операций, достаточно только системных функций. Для других же, например, процессов, системные функции не всегда дают тот объем информации, который может быть использован для анализа. В этом случае необходимо исследовать библиотечные функции. Идея остается прежней — перехват интересующих функций из общего потока, однако теперь из общего потока необходимо выделять инструкцию вызова функции (например, `call`) и идентифицировать функции по адресу.

Таким образом можно получить высокоуровневую информацию об объектах системы, например, журнал файловых операций, список запущенных процессов, список загруженных модулей, вызываемых функций и так далее.

Метод мониторинга событий виртуальной машины для получения информации об объектах ОС [11; 12; 14—16] предполагает извлекать информацию о сущностях, описанных в разделе 2.1.

На рисунке 3.1 представлена схема процесса мониторинга объектов операционной системы. В следующих подразделах главы описан каждый шаг этой схемы.

3.1.1 Входные данные для работы метода

Набор данных, необходимых для работы метода, может быть получен с помощью документации или, в случае ее отсутствия, с помощью дизассемблирования.

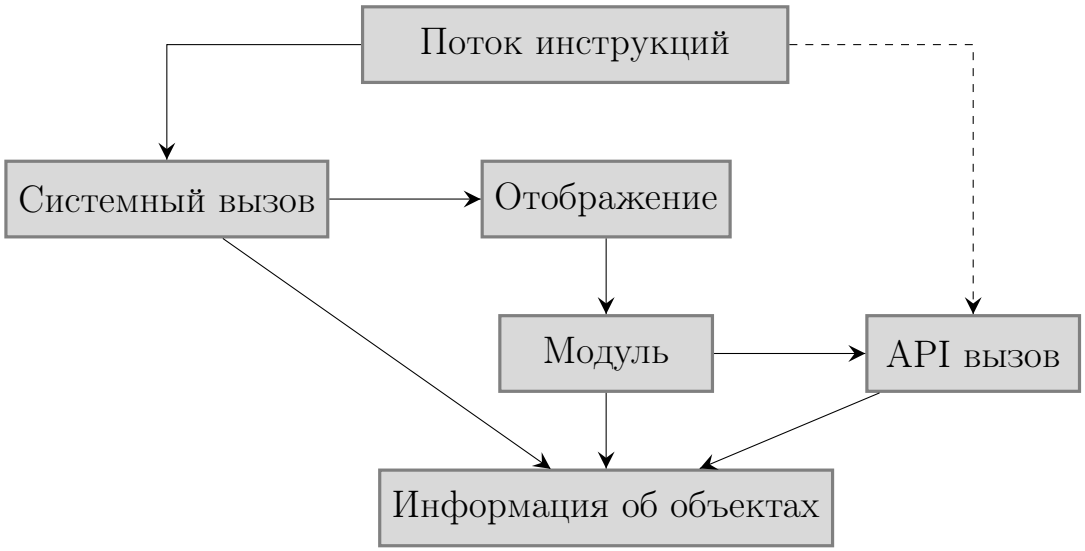

Рисунок 3.1 — Схема процесса мониторинга объектов ОС

Основным источником данных о системе является ABI. Из этого интерфейса необходимо получить следующую информацию: набор системных вызовов (количество системных вызовов, их идентификаторы, описание системных функций (параметры и возвращаемые значения)), соглашения о вызовах.

Помимо ABI необходимо определить инструкции для реализации системных вызовов, идентификаторы адресного пространства, форматы динамических библиотек, а также набор библиотечных функций.

Идентификатор адресного пространства важен, поскольку применяется для сопоставления начала и конца системной или библиотечной функции. В качестве этого идентификатора могут выступать значения некоторых регистров процессора.

Информацию из функций можно получить из двух источников: входные параметры и выходные параметры с возвращаемым значением. Некоторые функции предоставляют всю необходимую информацию во входных параметрах, а некоторые содержат важные данные только в выходных параметрах. В общем случае перехватывается выход, где можно получить информацию из обоих источников, однако, существуют функции, которые

не возвращают управление (`execve, ExitProcess`), поэтому их отслеживать нужно только на входе. Как правило, такие функции предоставляют всю необходимую информацию во входных параметрах.

Все механизмы метода работают, начиная с перехвата системных вызовов. Следующим шагом мониторинга объектов ОС является перехват API вызовов.

3.1.2 Перехват системных вызовов

Перехватчик системных вызовов — это один из двух основных механизмов метода. Он выделяет из потока инструкций виртуального процессора те, которые связаны с реализацией системных вызовов.

Системный вызов — это запрос из пользовательского приложения к ядру. Для этих запросов обычно предусмотрены специальные инструкции (например, `sysenter/syscall` для x86/64 или `svc #0` для ARM). Эти инструкции переводят процессор в режим ядра и выполняют соответствующий системному вызову код.

Перехват системных вызовов происходит в процессе выполнения кода виртуальной машины. Перехватчик сканирует поток выполняющихся инструкций с целью найти инструкции, реализующие системные вызовы. Когда они обнаружены, осуществляется переход в обработчик системных вызовов, соответствующий его идентификатору.

В процессе перехвата системных вызовов появляется задача сопоставления начала и конца функции. Зачастую инструкции начала и конца, последовательно встретившиеся в потоке, могут не быть парой, потому что архитектуры современных систем многопоточные и в разных потоках вызываются разные функции. Поэтому нельзя ориентироваться только на оп-

код (часть машинного языка, называемая инструкцией и определяющая операцию, которая должна быть выполнена), для сопоставления необходимо использовать дополнительные данные.

Определить что инструкция начала и конца соответствуют одной функции можно с помощью идентификатора адресного пространства, заранее определив что он будет из себя представлять (например, это может быть значение регистра, указывающего на каталог виртуальных страниц памяти).

Таким образом, на входе в системную функцию необходимо сохранить идентификатор адресного пространства. Когда в потоке появляется инструкция окончания системного вызова из существующих входов по вышеописанным параметрам выбирается соответствующий. Когда пара восстановлена, можно получать выходные данные и возвращаемое значение функции.

С помощью перехвата системных вызовов можно получить информацию о таких объектах, как файлы, процессы и отображения.

3.1.3 Получение информации об отображениях

Отображение в этом методе имеет характер вспомогательной сущности, промежуточного звена между потоком инструкций и модулем. Информация об отображении получается с помощью цепочки системных вызовов, то есть применяется механизм, описанный в подразделе 3.1.2.

В контексте этой работы рассматриваются случаи отображения динамических библиотек. Результатом отображения этих файлов должен стать адрес загрузки файла в память виртуальной машины и размер загруженного образа.

Библиотека или исполняемый файл представляются в системе в виде файлов, поэтому для них работают обычные файловые операции. В случае с обнаружением библиотеки цепочка системных вызовов в общем случае может выглядеть как последовательность двух вызовов: открытие файла и отображение в память (маппинг). В некоторых системах могут присутствовать промежуточные шаги (например, в ОС Windows для файлов такого рода создаются секции и только потом происходит отображение).

3.1.4 Получение информации о модулях

Библиотечные функции являются важным источником данных для мониторинга объектов ОС. Библиотечные функции работают с данными уровня приложений, а не уровнем ядра операционной системы, в отличие от системных вызовов, поэтому в ряде случаев могут предоставить более полную информацию о некоторых частях системы, например, о процессах.

Динамическая библиотека — файл, содержащий машинный код. Загружается в память процесса загрузчиком программ операционной системы либо при создании процесса, либо по запросу уже работающего процесса, то есть динамически.

Библиотечные функции вызываются только когда в память загружена соответствующая динамическая библиотека. Поэтому начать исследование функций нужно с определения момента загрузки библиотеки в память. Библиотека — это файл, поэтому открывается и загружается она с помощью функций работы с файлами, которые необходимо перехватить. Важными характеристиками библиотеки являются адрес загрузки, имя и размер загруженного образа.

Для того, чтобы отследить загрузку динамической библиотеки и получить адрес необходимо отследить отображение библиотеки в память.

Например, для Windows необходимо выполнить несколько шагов:

1. Отследить системный вызов открытия файла с именем интересующей библиотеки.
2. По дескриптору файла перехватить создание секции.
3. По дескриптору секции перехватить отображение секции в память.
4. Получить данные о библиотеке после выполнения функции отображения.

В Linux же отсутствует понятие секции и отображение происходит сразу после открытия файла.

Однако, полученный адрес загрузки еще не означает, что библиотека загрузилась в память, этот момент необходимо проверять. Например, зная диапазон адресов библиотеки, можно отслеживать эти адреса. Гипотеза: если осуществляется переход на адрес, принадлежащий библиотеке, то она уже загрузилась и можно попробовать получить таблицу экспорта. Как только адрес из указанного диапазона появляется в счетчике команд, осуществляется проверка загрузки заголовка библиотеки. В случае успеха необходимо перейти в таблицу экспорта и извлечь данные о функциях, а именно имена и адреса.

Для того, чтобы получить информацию о функциях, содержащихся в модулях, необходимо провести ряд действий. Во-первых, получить информацию от отображения про базовый адрес загрузки модуля и его размер. Во-вторых, определить момент загрузки библиотеки в память. В-третьих, считать нужные секции библиотеки и получить адреса и имена экспортируемых функций.

Первый шаг описан в подразделе 3.1.3. Второй этап заключается в проверке загрузки необходимых частей файла. Дело в том, что после отоб-

ражения файл не сразу и не полностью загружается в память, он подгружается частями, когда они необходимы для работы системы. Как правило, для извлечения информации о функциях весь файл не требуется, необходимо прочитать заголовок и с его помощью определить какие секции библиотеки содержат нужную информацию. Чтобы удостовериться что заголовок файла уже загружен, необходимо дождаться, когда управление перейдет на адрес из диапазона адресов библиотеки. Этот диапазон выглядит следующим образом: [адрес загрузки: адрес загрузки + размер образа].

Когда заголовок загружен и можно получить адреса секций, где хранится информация об экспортируемых функциях, можно пробовать запрашивать память по этим адресам. В случае успешной загрузки можно получить список функций и адресов. Если память не загружена, то эта операция повторяется до тех пор, пока нужные секции не будут загружены. Данные о вызываемых из этой библиотеки функций не утрачиваются, потому что если секция не загрузилась, значит и функции из нее не вызывались.

Таким образом может быть получена информация о загруженных в систему модулях и экспортированных функциях.

3.1.5 Перехват API вызовов

Механизм перехвата API вызовов практически идентичен механизму перехвата системных вызовов. Разница заключается в том, что для перехвата API вызовов необходимо предварительно получить информацию об адресах и именах функций, содержащихся в библиотеках. Когда эта информация определена и сохранена, из потока инструкций выделяются инструкции для вызова функции и для возврата управления (например, для Intel x86 это `call` и `ret`). С помощью информации об адресном про-

странстве происходит сопоставление этих пар, как при работе с системными вызовами. Параметром инструкции вызова функции является адрес, по которому и осуществляется переход в соответствующий обработчик. Данные, которые можно получить от работы функций — входные и выходные параметры и возвращаемые значения.

С помощью механизма перехвата API вызовов можно осуществлять трассировку функций. Когда адреса и имена функций известны, можно определить что произошел вызов из библиотеки, проверив его по таблице. При совпадении адресов станет ясно, что началось выполнение библиотечной функции.

Такой метод трассировки функций подходит под концепцию этой работы, поскольку необходимые входные данные, так же как и в случае с системными вызовами, редко изменяются, легко доступны и не требуют внешних вмешательств. Форматы библиотек определены и документированы.

В качестве примера можно рассмотреть библиотеку для Windows `kernel32.dll`. Это одна из базовых библиотек Windows, которая отвечает за обработку памяти, операций ввода-вывода и прерываний. В этой работе из библиотеки `kernel32.dll` выбираются функции, связанные с работой процессов. Например, это `GetCurrentProcessId`, `CreateProcessW`, `CreateProcessAsUser`, `TerminateProcess`.

С помощью API вызовов можно получить информацию о процессах и вызываемых функциях.

3.2 Метод вызова системных функций по запросу анализатора для получения заданных атрибутов объектов ОС

В процессе работы вышеописанного метода мониторинга событий могут возникать ситуации, когда невозможно определить необходимую информацию, например, сопоставить между собой все полученные данные о процессах. В этой ситуации обычно не хватает системного вызова запроса идентификатора процесса, который возникает по запросу от какого-либо приложения, работающего в системе. Таким образом, нужно иметь возможность вызывать некоторые системные вызовы, когда это необходимо для анализа.

3.2.1 Инструмент Crosscut

Инструмент Crosscut [43] является наиболее близким аналогом для предлагаемого метода вызова системных функций для получения заданных атрибутов объектов ОС.

Crosscut — решение для детерминированного воспроизведения работы системы. В нем есть возможность выделения журналов (подтрасс) недетерминированных событий отдельных приложений из общего журнала работы системы при воспроизведении. Это позволяет анализировать работу отдельных приложений без дополнительных затрат на воспроизведение остальных приложений и системных компонентов во время проведения анализа. Кроме выделения трасс для отдельных приложений на уровне процессорных команд, в Crosscut есть возможность работы с приложениями более высокого уровня. Для выделения трассы во время воспроизведения

на определенном шаге сохраняется состояние машины и выполняется гостевой трассировочный код, соответствующий этому шагу. После этого состояние машины восстанавливается и происходит переход к следующему шагу.

3.2.2 Описание метода вызова системных функций

Идея метода заключается в том, что в основной поток инструкций эмулятора должны быть встроены инструкции, выполняющие нужные в данный момент действия. Если осуществлять встраивание на работающей в обычном режиме виртуальной машине, то это грозит изменением поведения системы. Допускать такую ситуацию неприемлемо, поэтому этот метод лучше использовать в сочетании с детерминированным воспроизведением. Детерминированное воспроизведение [44] — это технология, позволяющая записывать и воспроизводить сценарии работы исследуемой системы. Полученный сценарий можно запускать многократно, используя различные виды анализа. В этом случае анализ не сможет повлиять на систему, например, нарушив ее работу замедлением [45]. С его помощью становится возможным вызывать необходимые инструкции, не нарушая обычной работы эмулятора.

Встраивать можно любые инструкции, которые могут привести к получению необходимой на данный момент информации. В контексте этой работы встраиваются системные вызовы, потому что в процессе работы эмулятора не всегда возникают те вызовы, с помощью которых можно составить полную картину, например, о процессах, работающих в системе. Возникает необходимость вызывать такие функции принудительно, встраивая свой код в общий поток инструкций. После того, как сценарий ра-

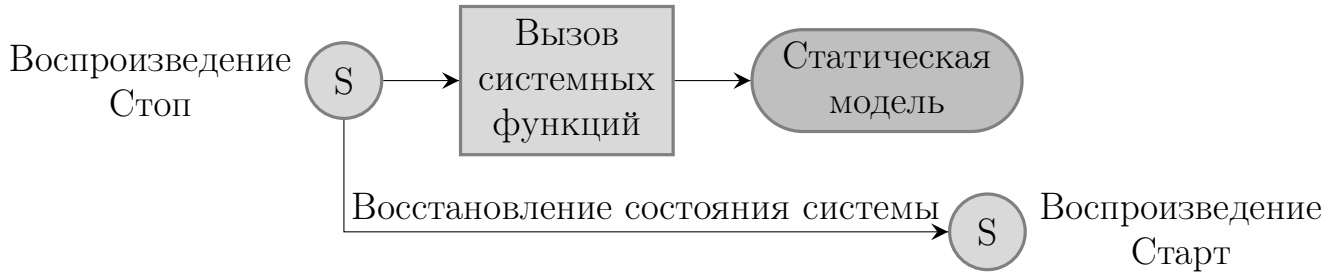

Рисунок 3.2 — Вызов системных функций по запросу анализатора

боты записан, воспроизведение можно останавливать и вызывать нужные функции, предварительно сохранив состояние виртуальной машины, затем получать информацию и продолжать воспроизведение с сохраненного состояния. Таким образом возможно дополнить недостающие данные и получить более полную картину. Этот процесс представлен на рисунке 3.2.

3.2.3 Практическое применение метода

Метод был применен для получения смещений полей `pid` и `tid` в структуре `task_struct` для ОС Android [46].

Информация о процессах и потоках в операционных системах хранится в специальных структурах, доступ к которым извне проблематичен, но возможен.

В процессе работы эмулятора не всегда возникают системные вызовы, с помощью которых можно составить полную картину, например, вызов `getpid`, позволяющий получить идентификатор текущего процесса. Возникает необходимость вызывать такие функции сознательно, встраивая свой код в общий поток инструкций. Однако при обычном выполнении эмулятора это делать нельзя, потому что поменяется ход событий и анализ будет неточным. Эту проблему решает детерминированное воспроизведение [44].

С его помощью становится возможным вызывать необходимые функции, не нарушая обычной работы эмулятора. После того, как журнал записан, воспроизведение можно останавливать и вызывать нужные функции, получать информацию и запускать воспроизведение снова. Таким образом, возможно дополнить недостающие данные и получить более широкую картину.

Для встраивания собственной функции в поток выполнения был опеределен алгоритм. В соответствии с ним необходимо выполнить следующие шаги:

1. Сохранить состояние системы.
2. Переключить процессор в режим ядра.
3. Записать в регистр, использующийся для передачи номера системной функции, идентификатор нужного системного вызова.
4. Подменить регистр IP на инструкцию, реализующую системный вызов.
5. Изменить адрес возврата.
6. Интерпретировать результат.

Таким способом был встроен системный вызов `getpid()` в эмулятор операционной системы Android. Он использовался для определения смещений в структуре `task_struct`, по которым записаны идентификаторы процесса и потока. Высчитать эти смещения один раз не является решением, потому что от версии к версии в Linux эти смещения менялись.

Операционная система Android и, соответственно, ее эмулятор работают на процессоре ARM. Необходимые данные для работы метода следующие:

– `svc#0` — команда, вызывающая системный вызов.

– `r7` — регистр, в который записывается номер системной функции.

– `r0` — регистр, в который записывается результат системного вызова.

Интерпретация результата заключается в следующем: после вызова функции getpid становится известным идентификатор процесса, который находится в структуре `task_struct`. Адрес, с которого начинается структура, известен по документации Linux. Задача состоит в том, чтобы найти в известном фрагменте памяти значение идентификатора процесса и вычислить его смещение относительно начального адреса структуры.

Так как идентификатор процесса представляет собой четырехбайтовое число, то велика вероятность, что оно неоднократно встретится в выбранном фрагменте памяти, поэтому встраивание вызова происходит несколько раз. При каждом запуске все вхождения этого числа в память сохраняются до тех пор, пока не будет однозначно ясно, что смещение соответствует полю идентификатора процесса.

В структуре `task_struct` сразу после идентификатора процесса располагается идентификатор потока, поэтому, узнав смещение `pid`, можно получить и смещение `tid = pid + 4`.

Эмулятор Android хранит свои основные настройки в конфигурационном файле. Туда же была добавлена информация о смещении идентификатора процесса, что позволило записывать значение `pid` в трассу [3].

3.3 Выводы по главе

Разработан новый метод мониторинга событий виртуальной машины для получения информации об объектах ОС, лишенный ограничений, описанных в подразделе 1.3. В качестве преимуществ над существующими методами можно выделить:

1. Метод не использует внутренние структуры ядра, как источник информации о системе. Благодаря этому инструмент позволяет исследовать любую версию любой системы без ее предварительной настройки. Например, если ABI разных систем совпадает, то они обе могут быть исследованы без дополнительных настроек.
2. Представленный метод обладает высокой производительностью. Она исследуется в главе 5. Это связано с небольшим потоком данных между гостевой ОС и анализатором.
3. Для работы метода не требуется загрузка внешних программ или модулей в исследуемую систему, что позволяет анализировать встроенные системы.
4. Совмещение разработанного метода с детерминированным воспроизведением снимает ограничения по времени, что важно для некоторых приложений, работающих с интернет трафиком.

Разработанный метод мониторинга событий виртуальной машины для получения информации об объектах ОС имеет следующие ограничения:

1. Полная информация о работе системы для некоторого вида анализа (API вызовы, процессы) может быть получена только в том случае, если система и анализ будут запущены одновременно. В противном случае загрузка гостевых модулей может быть упущена и трассировка по ним осуществляться не будет.
2. Не предусмотрена работа с микроядерными системами.
3. Метод работает с учетом того, что ядро системы работает согласно документации, следовательно, измененное поведение ядра не рассматривается.

Глава 4. Практическое применение методов мониторинга объектов ОС

На основе разработанных методов мониторинга объектов ОС была реализована инструментальная среда.

Инструментальная среда представляет собой набор плагинов для эмулятора QEMU [47; 48]. Плагины — это динамические библиотеки, которые могут быть загружены при запуске эмулятора или подключены к нему в процессе работы.

Плагин может работать как самостоятельно, так и взаимодействовать с другими плагинами. Для взаимодействия плагинов существует механизм событий-подписок, который заключается в том, что каждый плагин может генерировать события и подписываться на события других плагинов. Например, плагин генерирует события о файловых операциях, а другому плагину нужно знать когда происходит открытие файла. Второй плагин подписывается на событие открытия файла, которое генерируется первым плагином, и получает всю информацию [1; 13].

Как описывалось в главе 3, системные и API вызовы являются источниками информации для мониторинга объектов системы, поэтому для каждой операционной системы разработаны платформо-зависимые плагины для перехвата системных вызовов (`syscalls_название ОС`) и перехвата API функций (`api_monitor_название ОС`).

Плагин для перехвата системных вызовов получает оповещения от QEMU после трансляции каждой инструкции, при этом на этапе выполнения инструментируются только системные вызовы. Плагин выступает как промежуточное звено между QEMU и плагинами для анализа информации

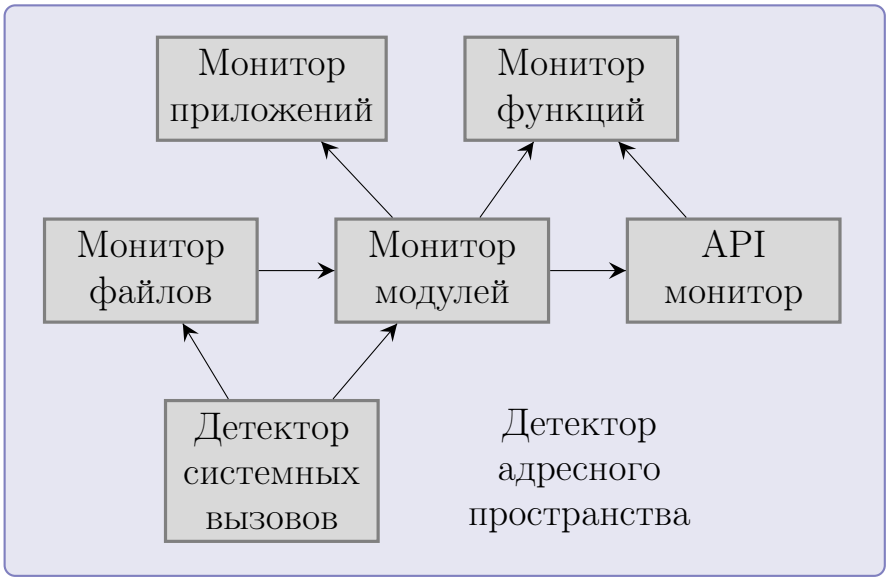

Рисунок 4.1 — Схема разработанного инструмента

на более высоком уровне [13]. Пример подобного взаимодействия приведен выше для плагина, анализирующего работу с файлами.

Кроме плагинов, события генерирует и эмулятор. Примерами событий являются прерывание, трансляция инструкции, выполнение инструкции, получение сетевого пакета, чтение/запись ячеек памяти и так далее. Все эти события также могут быть использованы плагинами.

Инструментальная среда разработана для операционных систем Windows, Linux, FreeBSD, работающих на архитектуре x86. На рисунке 4.1 представлена общая схема разработанных модулей.

Отдельно следует выделить плагин «Детектор адресного пространства». Он не зависит от гостевой системы и связан со всеми разработанными плагинами. Основной его задачей является отслеживание значения регистра процессора, хранящего указатель на каталог страниц, для рассматриваемой архитектуры `Intel x86` им является регистр `cr3`. Этот плагин называется `contexts`, он генерирует события при создании нового адресного пространства, а также позволяет запрашивать информацию о текущем адресном пространстве. В следующих разделах будет использовано собы-

тие `new_context`, которое как раз и является событием создания нового адресного пространства.

Далее в главе рассмотрены реализации мониторинга объектов для операционных систем Windows [49], Linux [50] и FreeBSD [51].

4.1 Реализация инструмента для ОС Windows

Для построения плагинов необходимо определить описанные в подразделе 3.1.1 входные данные. Набор получается следующим:

- Инструкции для реализации системных вызовов `sysenter/sysexit`.
- Адресное пространство представляется в виде регистра указателя на каталог страниц (`cr3`).
- Идентификатор системных вызовов передается через регистр `eax`.
- Параметры функций передаются слева направо, находятся по адресу `edx + 8`.
- Возвращаемое значение системной функции записывается в регистр `eax`.
- Формат динамических объектов PE (Portable Executable).
- Идентификаторы системных вызовов, а также описания системных и API функций находятся в свободном доступе.

На рисунке 4.2 представлен механизм работы метода на примере получения информации о файлах и процессах для ОС Windows. Аналогичным образом он работает и для других систем.

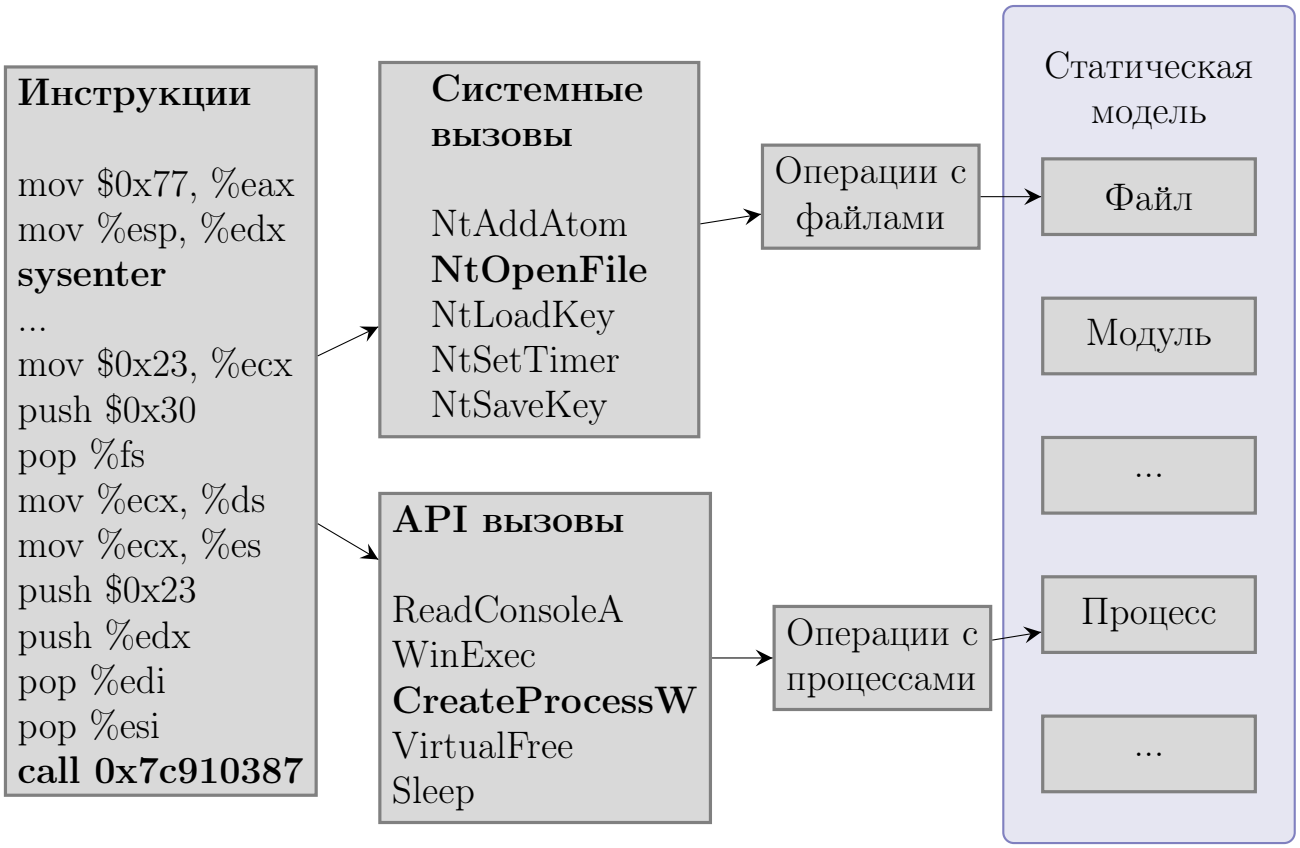

Рисунок 4.2 — Механизм работы метода на примере получения информации о файлах и процессах

4.1.1 Мониторинг файлов

Мониторинг файлов и файловых операций реализован с помощью механизма перехвата системных вызовов, описанном в разделе 3.1.2, поскольку вся необходимая информация о файлах (дескриптор, имя, тип доступа, буфер прочитанных/записанных данных) может быть получена на уровне системных функций.

При работе с файлами этот плагин отфильтровывает системные вызовы, связанные с файлами и передает их в плагин `file_monitor` для непосредственной работы с системными функциями.

Для мониторинга файлов и файловых операций для всех рассматриваемых ОС разработан один плагин (`file_monitor`), так как операции

с файлами в этих операционных системах организованы схожим образом. Он не зависит от гостевой операционной системы и гостевого аппаратного обеспечения.

Для анализа файловых операций в Windows перехватываются следующие функции: `NtCreateFile, NtOpenFile, NtReadFile, NtWriteFile, NtClose`.

Когда плагин `syscalls_windows` обнаруживает платформо-зависимый системный вызов, он преобразовывает параметры и возвращает данные системного вызова в платформо-независимой форме.

В абстрактном виде из операций над файлами извлекаются следующие данные:

- Создание файла: дескриптор, имя и права доступа.
- Открытие файла: дескриптор, имя и права доступа.
- Чтение файла: дескриптор, адрес буфера, количество считанных байт.
- Чтение файла: дескриптор, адрес буфера, количество записанных байт.
- Закрытие файла: дескриптор.

Так как работа производилась для операционных систем Windows, Linux и FreeBSD было необходимо свести параметры системных функций каждой из ОС к общему знаменателю. Например, операция чтения файла в Linux/FreeBSD:

```
read(int handle, void *buffer, int nbyte);
```

В Windows операция чтения выглядит так:

```
NtReadFile(HANDLE  FileHandle, HANDLE Event, PIO\_APC\
   _ROUTINE ApcRoutine, PVOID ApcContext, PIO\_STATUS\_BLOCK
    IoStatusBlock, PVOID Buffer, ULONG Length, PLARGE\
   _INTEGER ByteOffset, PULONG Key);
```

В Windows гораздо больше параметров для этой функции, однако, для преследуемой цели некоторые из них можно опустить. На выходе получается абстрактная функция чтения файла:

```
Abstract_read(file handle, address of the buffer, number of
   bytes to read);
```

Такое же преобразование было проведено со всеми функциями для работы с файлами. Ниже представлен фрагмент полученного абстрактного журнала файловых операций.

```
open(name \SystemRoot\bootstat.dat, mode READ | WRITE,
handle 0x1c)
read(handle 0x1c, buffer 0x15f88c, length 0x4)
write(handle 0x1c, buffer 0x15f8cb, length 0x1)
```

В результате работы плагина `file_monitor` можно получить информацию о файлах в системе: имена и дескрипторы, буферы чтения и записи, операции, которые с ними происходили. Информация может быть записана в виде журнала для изучения поведения файлов, либо по мере необходимости нужные файлы могут быть задействованы для проведения анализа, например, анализа помеченных данных.

4.1.2 Мониторинг модулей

Для мониторинга модулей был разработан плагин `api_monitor_windows`. Одним из его практических применений является трассировка вызовов функций из динамических библиотек.

В этом разделе рассматриваются только динамические библиотеки. Динамические библиотеки в Windows — это файлы с расширением `dll` и

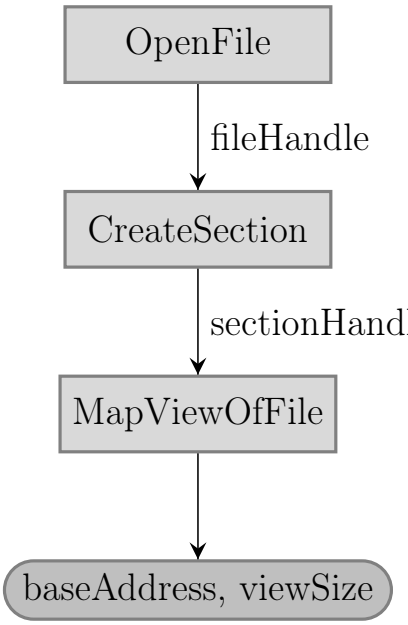

Рисунок 4.3 — Первый вариант получения базового адреса загрузки библиотеки

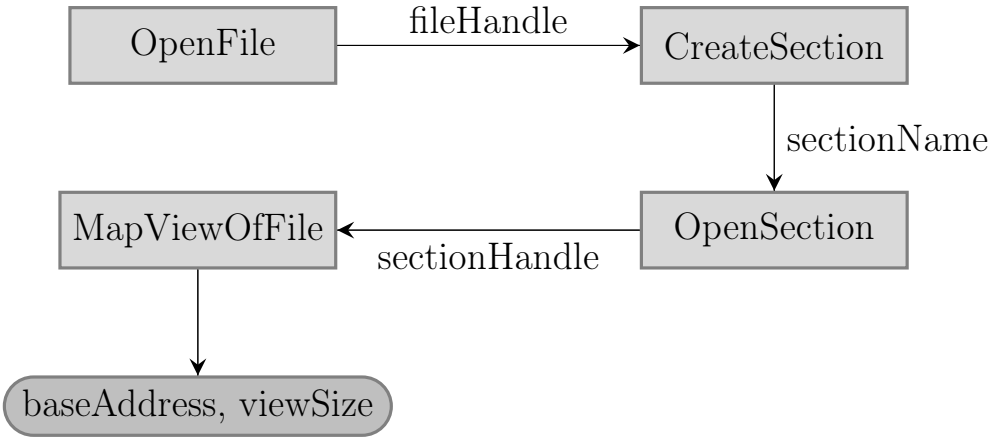

Рисунок 4.4 — Второй вариант получения базового адреса загрузки библиотеки

построенные в соответствии со стандартизированным форматом PE [52; 53].

Факт загрузки модуля в систему может быть использован для анализа, однако библиотеки больше интересны своим содержимым, а именно экспортированными функциями. Формат PE описывает что и по каким адресам находится в `DLL`- файле. В этой работе были задействованы та-

кие составляющие формата PE как заголовок, необязательный заголовок, секции, таблицы адресов и имен.

Задача, решаемая этим плагином состоит в том, чтобы обнаружить библиотеку и получить адреса и имена экспортируемых функций.

Логику работы API монитора можно представить в виде алгоритма, состоящего из следующих шагов:

1. Получение имен библиотек и адресов их загрузки в память.
2. Чтение заголовка файла.
3. Проверка загрузки нужной секции в память.
4. Получение данных о функциях (имена и адреса).
5. Трассировка функций по полученным адресам.

Для получения данных из библиотеки необходимо определить ее устройство. Она включает в себя заголовок и различные секции, одна из которых содержит информацию об экспортируемых функциях. Эти функции могут вызываться приложениями, загружающими библиотеку. На этом этапе необходимо получение адресов и имен экспортируемых функций для каждого загруженного модуля.

Важными данными при обнаружении библиотеки являются имя, базовый адрес загрузки ее в память и размер образа. Находить эти сведения можно с помощью перехвата ряда системных функций. Динамическая библиотека это прежде всего файл, который в определенный момент открывается с помощью системной функции `NtOpenFile`. На этом этапе становится известным имя файла и его дескриптор. Следующий шаг, который проделывается с библиотекой, это создание секции, а именно вызывается функция `NtCreateSection`. Во входных параметрах этой функции присутствует дескриптор файла, для которого будет создана секция. На выходе может быть два варианта: в параметрах секции присутствует имя секции или нет. Помимо прочего, функция возвращает дескриптор секции.

Далее может быть два варианта развития событий:

1. Имя секции не заполнено и сразу после `NtCreateSection` вызывается функция `NtMapViewOfFile`, определяющая файл по дескриптору секции. Эта функция возвращает искомые данные: адрес загрузки библиотеки в память и размер образа (`baseAddress` и `viewSize`).
2. Когда в `NtCreateSection` присутствует имя секции, это означает, что необходимо сделать дополнительный шаг. Это вызов функции `NtOpenSection`. Секция открывается по имени и возвращает дескриптор секции, по которому теперь может быть выполнена `NtMapViewOfFile`.

Вышеописанные варианты обнаружения адреса загрузки библиотеки изображены на рисунках 4.3 и 4.4.

После того как адрес загрузки библиотеки известен можно пробовать читать файл. Для того, чтобы быть уверенным, что хотя бы заголовок файла уже загружен, необходимо дождаться, когда регистр `PC` будет содержать адрес из диапазона загрузки библиотеки. Этот диапазон начинается с адреса загрузки и заканчивается адресом загрузки, увеличенным на размер образа (`baseAddres, baseAddress + viewSize`).

Когда известны адреса таблиц секции экспорта, можно пробовать запрашивать память по этим адресам. В случае успешного чтения памяти можно получить список функций и адресов, сохранить их переходить к трассировке. Если память не загружена, то необходимо дождаться пока она будет загружена. Это действие повторяется до тех пор, пока функции не будут считаны. Считается, что данные не упускаются, потому что если секция не загрузилась, значит и функции из нее не вызывались. По этой же причине проверки осуществляются только в определенном диапазоне адресов, что увеличивает попадание и уменьшает накладные расходы.

Таблица 4.1

Фрагмент журнала вызовов API функций в Windows

Имя модуля	**Имя функции**
gdi32.dll	SelectPalette
gdi32.dll	DeleteDC
user32.dll	ReleaseDC
gdi32.dll	GdiReleaseDC
kernel32.dll	WaitForSingleObject
kernel32.dll	WaitForSingleObjectEx
user32.dll	PeekMessageW

Библиотечные функции более высокоуровневые, чем системные и параметры у них зачастую более полезные и понятные человеку. Так, например, с помощью функции getCurrentProcessId можно точно получить идентификатор процесса в текущем адресном пространстве. Помимо полезных параметров, отслеживая вызывающиеся функции, можно делать выводы о работе исследуемой системы. В таблице 4.1 приведен фрагмент журнала вызовов API функций.

Основной задачей извлечения данных о модулях в этой ОС стало получение данных о работе процессов. Помимо этого модули предоставляют сведения и о других частях системы.

4.1.3 Мониторинг процессов

Процессы являются одной из важнейших абстракций в операционной системе. Каждый процесс — это запущенная программа, которая выполняется в определенном адресном пространстве. Для анализа работы системы крайне важно иметь информацию о существующих процессах, переключении между ними, завершении работы процессов. Различать процессы между собой можно по идентификатору, который назначается им при создании,

тогда же выделяется адресное пространство, в котором будет работать процесс.

Отслеживание процессов является очень важной задачей, потому что процесс является практически основной единицей в исследовании работы приложений и систем. Определив идентификатор и адресное пространство процесса, можно связывать между собой остальные исследуемые объекты, например, файлы, получая таким образом наиболее полную и понятную картину происходящего в системе.

В этой работе рассматривается два способа получения информации о процессах: с помощью системных вызовов и с помощью API функций.

Системные вызовы для мониторинга процессов

Для отслеживания работы процессов в помощью системных вызовов было решено использовать ряд системных функций: `NtCreateProcess`, `NtCreateThread`, `NtQueryInfoProcess`, `NtTerminateProcess`.

Функция `NtCreateProcess` создает процесс. В каждом процессе существует как минимум один поток, поэтому `NtCreateProcess` вызывает функцию `NtCreateThread`. Необходимо изучить параметры этих функций, чтобы понять какую информацию можно из них получить.

Функция `NtCreateProcess` не содержит в списке своих аргументов идентификатор процесса (pid), оперируя параметром handle (дескриптор). Может показаться, что дескриптор способен заменить pid, однако это не так. Сразу после создания процесса, назначенный ему дескриптор закрывается, и он больше не ассоциируется с процессом. Такой вариант для преследуемых целей не подходит.

NtCreateThread имеет более полезные параметры, а именно структуру `ClientId`, полями которой являются две переменные - `UniquePid` и `UniqueTid`. Таким образом, перехватив создание потока в процессе, можно получить pid созданного процесса.

В данном подходе есть явная проблема — pid получен, но привязать его к адресному пространству не представляется возможным, потому что новый процесс запускается из текущего, соответственно в регистре `cr3` находится указатель на его адресное пространство, а не на вновь созданного. Тем не менее, эту информацию можно сохранить и использовать в будущем для разметки процессов.

Важно не только найти pid, но и быть внутри процесса, для которого этот pid определен. Функция `NtQueryProcessInfo` может осуществить эту задачу при выполнении двух условий: она должна быть вызвана для текущего процесса и запрашивать базовую информацию о процессе. Если условия выполнены, то функция вернет структуру `ClientId`.

Таким образом, сопоставив `cr3` и `UniquePid` можно определить процесс. Используя этот метод нельзя сказать, что процесс будет определен в начале своей работы, но тем не менее это точечное сопоставление дает основу для разметки работы системы по процессам.

Было выделено четыре компонента, необходимых для получения информации о процессе: адресное пространство процесса (Context), адресное пространство родительского процесса (Parent context), идентификатор процесса (PID) и имя загруженного образа (Image name).

В таблице 4.2 приведено какие данные можно получить с помощью какой системной функции.

Как видно из таблицы, получить полную запись о процессе можно лишь с помощью функции `NtQueryInfoProcess`, которая вызывается по инициативе запущенных приложений или системы. Также в этой реализа-

Таблица 4.2

Информация о процессах, предоставляемая системными функциями

	Context	**Parent context**	**PID**	**Image name**
New_context	+			
NtCreateProcess		+		+/-
NtCreateThread		+	+	
NtQueryInfoProcess	+		+	

ции имеются проблемы с именем загруженного образа, поскольку он формально присутствует в параметрах функции `NtCreateProcess`, но фактически не был обнаружен.

Из вышеописанного можно сделать вывод, что этот способ обладает хорошим быстродействием из-за использования только системных вызовов, однако не предоставляет полный набор необходимой информации для получения информации о процессах.

API функции для мониторинга процессов

Для Windows подход с системными вызовами оказался не применим, потому что на уровне системых вызовов эта ОС поставляет не те сведения, которые требуются для получения информации о процессе. Поэтому в этой ОС используется второй вариант получения информации — с помощью API вызовов.

Чтобы получить информацию о процессах таким образом необходимо воспользоваться методом извлечения модулей, описанным в разделе 4.1.2.

Модули в этом случае нужны не сами по себе, а лишь информация, которая в них содержится, а именно список экспортированных функций (их адреса и имена).

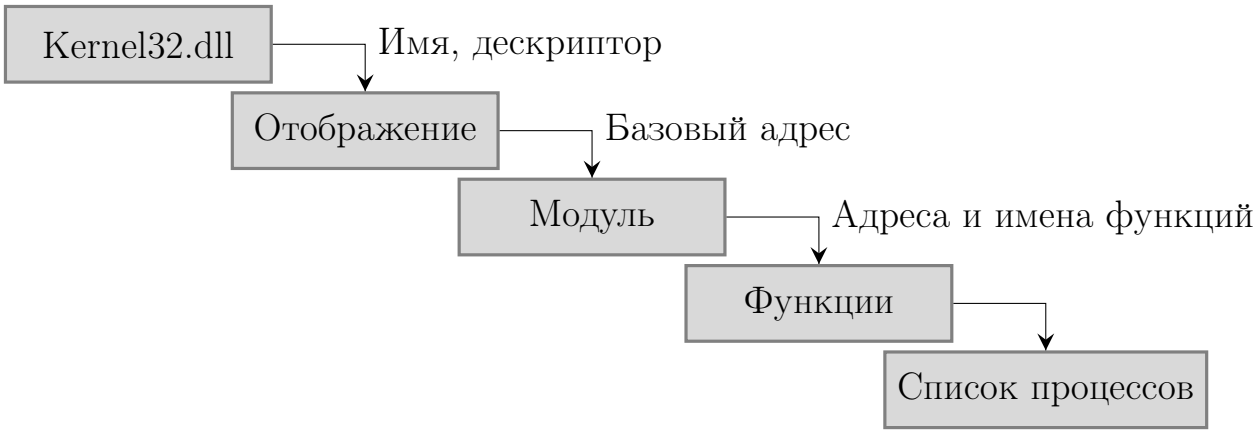

Рисунок 4.5 — Получение списка процессов

Процессы в Windows создаются с помощью API функций семейства `CreateProcess`, которые в ОС Windows находятся в библиотеке `kernel32.dll`. Функции из этой библиотеки и будут извлекаться.

Для получения сведений о процессах используются функции `CreateProcessW`, `CreateProcessAsUser`, `GetCurrentProcessId` и `TerminateProcess`.

Библиотечные функции более высокоуровневые, чем системные и их параметры обладают большей информативностью.

В Windows выделение памяти для процесса происходит без помощи отдельного системного вызова, поэтому из одной функции создания процесса можно получить информацию об идентификаторе процесса, имени образа и родительском контексте. Остается определить только адресное пространство созданного процесса, для чего используется функция `GetCurrentProcessId`, которая всегда вызывается после создания процесса, поэтому проблем с соединением информации о процессе не происходит.

На рисунке 4.5 представлен процесс получения списка процессов. Из рисунка видно, что для получения информации о процессах необходимо иметь информацию о модулях. Когда она получена, то адреса и имена функций передаются в перехватчик API вызовов, где происходит обработка библиотечных функций.

Таблица 4.3

Фрагмент списка процессов

Parent context	**Context**	**PID**	**Image name**
0x4533000	0x4888000	0x218	services.exe
0x4533000	0x4893000	0x22c	lsass.exe
0x4888000	0x4cc4000	0x2cc	svchost.exe
0x4888000	0x61d7000	0x52c	svchost.exe
0x6331000	0x65fe000	0x538	explorer.exe
0x65fe000	0x252f000	0x6bc	notepad.exe
0x65fe000	0x5fa4000	0xd4	calc.exe

С помощью описанных методов получаем разметку по процессам, а также возможность в любой момент запрашивать список работающих в системе процессов, что полезно если при анализе используется детерминированное воспроизведение.

Список процессов, полученный таким образом, соответствует списку процессов, отображенных системной утилитой Windows «Диспетчер задач».

Пример информации о процессах, полученной с помощью плагина `process_monitor_windows` представлен в таблице 4.3.

Функция `TerminateProcess` отслеживается для того, чтобы определить что процесс завершился. Это необходимо для поддержания актуального списка процессов, работающих в системе.

Отслеживание этой функции отличается от других тем, что перехватывать ее необходимо при входе в функцию, а не при ее завершении.

4.2 Реализация инструмента для ОС Linux

Необходимый набор данных для Linux выглядит следующим образом:

– Инструкции для реализации системных вызовов `sysenter/sysexit` или `int 0x80/iret` в зависимости от версии (в старых версиях используется прерывание).
– Адресное пространство представляется в виде регистра указателя на каталог страниц (`Cr3`).
– Идентификатор системного вызова передается через регистр `eax`.
– Параметры функций располагаются на регистрах `ebx, ecx, edx, esi, edi`, если их меньше пяти, в противном случае в `ebx` передается укзатель на массив аргументов.
– Возвращаемое значение системной функции записывается в регистр `eax`.
– Формат динамических объектов ELF (Executable and Linkable Format).
– Идентификаторы системных вызовов можно получить из исходных кодов системы, а именно из файла `unistd.h`.
– Описания системных и API функций находятся в свободном доступе.

4.2.1 Мониторинг файлов

Плагин для получения информации о файлах является платформонезависимым и описан в подразделе 4.1.1 про операционную систему Windows.

Для анализа файловых операций в Linux перехватываются следующие функции: creat, open, read, write, close.

4.2.2 Мониторинг процессов

Для отслеживания работы процессов, как и для файлов, существует набор системных вызовов. В операционной системе Linux к нему относятся такие вызовы, как создание процесса, выделение адресного пространства, получение идентификатора процесса, завершение процесса.

В Linux информация о процессах получается с помощью системных вызовов. Применить API вызовы в случае с этой операционной системе не представляется возможным, потому что системные функции вызываются из кода напрямую, минуя оболочки библиотечных функций.

Запрашивается такая же как и в Windows информация: идентификатор процесса, адресное пространство созданного процесса, адресное пространство родительского процесса и имя загруженного образа.

Как и в работе с файлами, из каждой операции требуется получить определенные сведения:

- Создание процесса: идентификатор, адресное пространство родителя.
- Выделение адресного пространства: имя загруженного образа, текущее адресное пространство.
- Получение идентификатора процесса: идентификатор, текущее адресное пространство.
- Завершение процесса: идентификатор.

Для мониторинга процессов разработан плагин `process_monitor_linux`.

Хранение информации о процессах организовано в виде списка, элементом которого является структура данных. Добывается необходимая информация из нескольких источников:

– Создание нового адресного пространства (`new_context`). Плагин, отвечающий за адресное пространство, описанный в начале главы 4, может генерировать сигнал о появлении нового адресного пространства, что означает создание нового процесса. Эти сигналы перехватываются плагином, отвечающим за процессы, где все новые адресные пространства записываются в список. Подразумевается что адресное пространство родительское, потому что создание нового процесса осуществляется из адресного пространства родителя.
– Системные вызовы `clone/fork`. Если происходит один из этих системных вызовов, значит появляется новый процесс, следовательно, в структуру добавляется новая запись, которую можно будет дополнить новой информацией. Так как системные вызовы для создания процессов выполняются из родительского адресного пространства, то можно связать родительский контекст и идентификатор текущего процесса, который вернется вызовами `fork/clone`.
– Системный вызов `execve`. С помощью этого системного вызова можно получить имя загруженного образа. Так как `execve` работает в текущем адресном пространстве, то сопоставить его с имеющимися в списке записями сразу не получится, потому что в них отсутствует информация о текущем адресном пространстве.
– Системный вызов `getpid`. Этот системный вызов вызывается по запросу системы или приложений. Невозможно предположить когда он произойдет и произойдет ли вообще. Однако, если он все же возникает, то надо воспользоваться тем полезным, что он может дать. Вызов `getpid` позволяет получать адресное пространство текущего процесса. Если бы `getpid` всегда вызывался из текущего процесса для него же, значит, можно было бы связать несколько

Таблица 4.4

Информация о процессах, предоставляемая системными функциями

	Context	**Parent context**	**PID**	**Image name**
new_context	+			
clone		+	+	
execve	+			+
getpid	+		+	

Таблица 4.5

Фрагмент журнала процессов

Context	**Parent context**	**PID**	**Image name**
0x359b8000			/sbin/mount
	0x359b8000	0x29	
0x359dd000			
0x359b8000			/sbin/mkdir
	0x359b8000	0x2a	
0x359b8000			/sbin/kmod
	0x359b8000	0x2b	

уже имеющихся записей об этом процессе и получить полную информацию.

В большинстве случаев после сбора этих данных можно получить адресное пространство родительского процесса и идентификатор процесса или текущее адресное пространство и имя образа. Возможна ситуация, что кроме адресного пространства в этой записи не будет больше ничего. Возможность получения данных о процессах из системных вызовов показана в таблице 4.4. То, что можно получить — окончательный результат, который предоставляется при реальной работе эмулятора. Результат работы плагина представлен в таблице 4.5.

Чтобы расширить и дополнить эту структуру необходимо подключать такие вещи как детерминированное воспроизведение и, возможно, встраиваемые системные вызовы.

4.2.3 Мониторинг модулей

В Linux не было цели использовать модули для получения информации о процессах, потому что в этой ОС имеется практически однозначная связь между системными вызовами (например `read`) и библиотечными функциями (с той же `read`), используемыми для обращения к системным вызовам. Иными словами, для каждого системного вызова обычно существует одна библиотечная функция, чаще всего одноименная, вызываемая для обращения к нему [36]. Таким образом, получить из библиотечных функций какую-то дополнительную информацию не представляется возможным.

Несмотря на это, информация о модулях в Linux собирается, потому что эту информацию можно использовать, например, для трассировки библиотечных функций.

Формат для описания динамических библиотек, поддерживаемый ОС Linux, называется ELF (Executable and Linkable Format) [54].

Как и в Windows, библиотека сначала определяется с помощью механизма перехвата системных вызовов. В Linux их два: открытие файла (`open`) и отображение в память (`mmap/mmap2`). Вызов, реализующий отображение в память, предоставляет информацию об адресе загрузки библиотеки в память и размере этой библиотеки. Далее происходят аналогичные Windows вещи, то есть проверка загрузки, разбор библиотеки согласно заголовку и сохранение информации об экспортированных функциях.

Фрагмент полученного журнала API вызовов представлен в таблице 4.6.

Таблица 4.6

Фрагмент журнала вызовов API функций в Linux

Имя модуля	**Имя функции**
libc.so	realloc
libc.so	__libc_malloc
libc.so	__open64
libc.so	__fxstat64
libc.so	__errno_location
libc.so	sigprocmask
libc.so	__gettimeofday
libc.so	__read

4.2.4 Дополнительные плагины для ОС Linux

В процессе работы с различными прошивками возникала необходимость создавать вспомогательные плагины. Для операционной системы Linux был разработан ряд дополнительных модулей.

Дамп файлов виртуальной машины (file_dump)

Этот плагин копирует создаваемые в виртуальной машине файлы в хостовый каталог `/tmp/file_dump`.

Файлы, которые открываются на чтение или дозапись не копируются. Если использовалась операция seek, результат тоже будет некорректным.

Лог сообщений ядра (printk_log)

Плагин `print_log` перехватывается функцию ядра `printk` и записывает в отладочный журнал QEMU (ключ plugins) сообщения, выводимые с помощью этой функции.

Лог вывода на консоль (stdout_log)

Записывает в отладочный журнал QEMU (ключ plugins) сообщения, выводимые в стандартные потоки `stdout` и `stderr`, а также на устройство `/dev/console`.

4.3 Реализация инструмента для ОС FreeBSD

Необходимый набор данных для FreeBSD выглядит следующим образом:

- Инструкции для реализации системных вызовов `int 0x80/iret` .
- Адресное пространство представляется в виде регистра указателя на каталог страниц (`Cr3`).
- Идентификатор системного вызова передается через регистр `eax`.
- Параметры функций располагаются на стеке `esp + 4`.
- Возвращаемое значение системной функции записывается в регистр `eax`.
- Формат динамических объектов ELF (Executable and Linkable Format).
- Идентификаторы системных вызовов можно получить из исходных кодов системы, а именно из файла `syscall.c`.
- Описания системных и API функций находятся в свободном доступе.

4.3.1 Мониторинг файлов

Плагин для получения информации о файлах является платформонезависимым и описан в подразделе 4.1.1 про операционную систему Windows.

Для анализа файловых операций в FreeBSD перехватываются следующие функции: open, read, write, close.

4.4 Конфигурирование трассировки системных вызовов

В связи с существованием множества операционных систем и архитектур процессоров приходится учитывать интерфейсы взаимодействия с ядром для каждого случая и создавать отдельный плагин перехватчик системных вызовов.

Был разработан один плагин для перехвата системных вызовов, работающий с конфигурационным файлом. Он предусматривает различные варианты осуществления системных вызовов в разных системах и подставляет нужные данные. В файле должны быть описаны входные данные, представленные в подразделе 3.1.1, относящиеся к системным вызовам.

Помимо широко известных операционных систем в мире существуют и редко используемые неизвестные системы. Однако иногда возникает необходимость проанализировать и их. Такие системы могут быть зашиты не только в компьютеры, но и в роутеры, телефоны и другую технику. Чтобы не разрабатывать для каждой существующей ОС отдельный плагин был реализован этот механизм, позволяющий организовывать перехват системных вызовов с заданными настройками.

Конфигурационный файл является входными данными для работы плагина. При реализации структуры конфигурационного файла стоит учитывать все особенности трассировки системных вызовов. В связи с этим структуру конфигурационного файла было решено разбить на несколько модулей.

Первый модуль должен содержать следующие данные:

– Название операционной системы.

– Версия операционной системы.

– Разрядность операционной системы.

Второй модуль содержит в себе перечисление всех доступных в данной операционной системе механизмов реализации системных вызовов и необходимую для этих механизмов информацию. Описание механизмов должно идти парами, например, `Syscall` — `Sysexit`, `Sysenter` — `Sysexit`. Для инструкций выполнения системного вызова необходимы следующие данные:

- Опкод системного вызова, представленный в виде шестнадцатеричного или десятичного числа.
- Маска для опкода системного вызова, представленная в виде шестнадцатеричного или десятичного числа.
- Выражение, представляющее адрес текущего контекста исполнения.
- Выражение, представляющее адрес текущего уровня стека.

Для инструкций возврата из системного вызова требуются такие данные:

- Опкод возврата из системного вызова, представленный в виде шестнадцатеричного или десятичного числа.
- Маска для опкода возврата из системного вызова, представленная в виде шестнадцатеричного или десятичного числа.
- Выражение, представляющее адрес текущего контекста исполнения.
- Выражение, представляющее адрес текущего уровня стека.

Составленный по этой структуре конфигурационный файл должен содержать хотя бы одни механизм реализации системного вызова.

Третий модуль конфигурационного файла должен содержать:

- Выражение, представляющее идентификатор системного вызова.
- Выражение, представляющее адрес возврата из системного вызова.

Последний модуль представляет собой список отслеживаемых системных вызовов, каждый элемент которого представлен следующими значениями:

- Выражение, представляющее идентификатор системного вызова.
- Строка — наименование системного вызова.
- Список аргументов системного вызова, каждый элемент которого представлен следующими значениями:
  - Строка — входной параметр или возвращаемое значение.
  - Строка — типа переменной.
  - Строка — описание текущего аргумента

Результатом работы этого плагина является трасса выполнявшихся системных вызовов с их параметрами и возвращаемыми значениями. Также этот плагин имеет возможность генерировать события, чтобы взаимодействовать с другими плагинами.

Конфигурационный файл делает трассировщик более гибким и настраиваемым. Он позволяет составлять собственные списки системных вызовов, которые необходимо трассировать, кроме того, нет необходимости пересобирать плагин для перехода на исследование другой ОС. Так же такой механизм облегчает возможность добавления поддержки новых операционных систем и архитектур процессора.

4.5 Выводы по главе

По описанному в главе 3 методу мониторинга объектов ОС был разработан набор плагинов для трех семейств операционных систем: Windows, Linux и FreeBSD. Далее описаны реализованные варианты использования полученных в процессе работы плагинов данных.

Вся полученная информация записывается в журналы и может быть использована как самостоятельный инструмент анализа, например, изучить какие файлы открывались и что с ними происходило, по списку процессов проанализировать работу системы или отследить поведение по журналу выполняемых API функций.

Помимо этого разработанные механизмы могут служить источниками информации для разного вида динамического анализа, такого как анализ помеченных данных, поведенческий анализ. Также полученные данные могут быть использованы для разметки трассы.

Анализ помеченных данных в реализации ИСП РАН [1] использует плагин, перехватывающий системные вызовы работы с файлами для внесения пометок.

Для разметки трассы был применен метод, описанный в разделе 3.2. Получив адреса смещений идентификаторов процессов и потоков, эти данные (идентификаторы) сразу стали записываться в трассу, облегчив дальнейший разбор этой трассы.

Инструмент может применяться для анализа встроенных систем. Благодаря поддержке семейств ОС можно осуществлять анализ, не имея точной информации о версии исследуемой операционной системы. В качестве примера можно привести уже решенную задачу. Имелось встроенное ПО коммуникационного оборудования, необходимо было запустить его на эмуляторе QEMU. ПО не запускалось по неизвестной причине. Разработанный инструмент позволил получить вывод сообщений ядра исследуемого ПО, из которых стало ясно каких модулей не хватает QEMU для успешной загрузки этого ПО. После доработки эмулятора задача была успешно выполнена.

Глава 5. Тестирование и оценка разработанного инструмента

В этой главе представлены результаты оценки разработанного инструмента.

Разработанный инструмент был протестирован, в результате чего была оценена его производительность и сложность конфигурирования, а также проведена проверка на достоверность получаемых результатов.

5.1 Производительность

Изначально планировалось провести сравнение производительности разработанного инструмента с аналогичными инструментами, например, DECAF. В процессе подготовки и проведения тестов оказалось, что такое сравнение будет не вполне корректно по причине использования разных кодовых баз (оба инструмента основаны на QEMU, но на разных его версиях), а также механизмы реализации плагинов и их функциональность оказались очень разными. Поэтому сравнить напрямую скорость работы DECAF и разработанного инструмента является нетривиальной задачей.

Было решено проводить сравнение скорости работы разработанного инструмента и эмулятора QEMU без инструментирования и плагинов.

Используемый для тестирования компьютер обладает следующими характеристиками: четырехядерный процессор Intel Core i3-3220 3.3ГГц, объем оперативной памяти 3072 Мб, объем жесткого диска 223 Гб, установленная операционная система Debian GNU/Linux 8 (jessie) 64-bit.

Для тестирования использовались операционные системы Windows XP SP2, Ubuntu 9.1 и Arch Linux 4.2.

Чтобы протестировать работу инструмента были выбраны наиболее часто используемые плагины, а именно перехватчик системных вызовов (название в таблице `Qemu_syscalls`) и плагин для отслеживания процессов (название в таблице `Qemu_process`). Перехватчик системных вызовов является основным плагином и предназначен для отслеживания и трассировки системных вызовов. Таким образом, в процессе работы осуществляется запись журнала произошедших системных вызовов. Для корректной работы необходима загрузка плагина для определения адресного пространства. Плагин для отслеживания процессов также работает в связке с другими плагинами, а именно: перехватчик системных вызовов, определитель адресного пространства, трассировщик API вызовов (для Windows). Плагин для отслеживания процессов ведет учет запуска и уничтожения процессов в системе.

Замерялась скорость работы QEMU, QEMU с включенным перехватчиком системных вызовов и QEMU с плагином для отслеживания процессов. Рассматривались такие ситуации, как время загрузки операционной системы, время скачивания файла из интернета (файл объемом 472Мб), время архивирования файла (использовался фрагмент скачанного файла размером в 50Мб).

Результаты измерений, а также замедление работы QEMU с плагинами представлены в таблице 5.1. Из нее видно, что замедление работы с применением разработанных механизмов составляет от 3 до 14% в зависимости от ОС и операции и является не существенным.

Для сравнения в таблице 5.2 приведены данные о производительности DECAF, представленные в статье [9]. Из таблицы видно, что замедление по сравнению с QEMU составляет от 8 до 24 %, что превышает показатели разработанного инструмента.

Таблица 5.1

Сравнение скорости работы QEMU и разработанного инструмента

	Qemu, мин	Qemu_syscalls, мин	Qemu_process, мин
Windows XP SP2			
Загрузка ОС Замедление	2:31	2:44 9%	2:51 13%
Ubuntu 9.1			
Загрузка ОС Замедление	1:11	1:21 14%	1:22 14%
Скачивание файла Замедление	1:37	2:41 4%	2:46 9%
Arch Linux 4.2			
Загрузка ОС Замедление	0:27	0:29 7%	0:30 11%
Скачивание файла Замедление	1:14	1:16 3%	1:18 5%
Архивирование файла Замедление	2:08	2:13 4%	2:14 5%

Таблица 5.2

Сравнение скорости работы QEMU и инструмента DECAF

	XUbuntu	WinXP SP3	Debian Squeeze (ARM)
DECAF + VMI	3:26	1:04	2:50
QEMU 1.0.1	2:46	0:53	2:36
DECAF + VMI Замедление %	24.14	21.91	8.72

5.2 Конфигурирование

Сложность конфигурирования разработанного инструмента определялась сравнением его с инструментом DECAF. Как уже говорилось ранее, DECAF использует сгенерированные программой-агентом профили для каждой операционной системы. Профили эти включают адреса и сме-

щения структур, необходимых для работы этого инструмента. Также, при необходимости, для плагинов могут быть написаны дополнительные конфигурационные файлы, например, для плагина трассировщика API вызовов. Основной конфигурационный файл приведен в приложении A.1, фрагмент конфигурационного файла для трассировки API вызовов представлен в приложении A.2.

В случае с разработанным инструментом конфигурационной информацией будут выступать входные данные, описанные в разделе 3.1.1. Если представить ее в виде конфигурационного файла для Linux, то получится следующее:

```
Syscall mechanism:
    Sysenter Sysexit
    int 80h iret

Address space ID:
    cr3

Format of shared objects:
    ELF

Syscall info:
    syscall ID: eax
    params: ebx, ecx, edx, edi, esi
    return value: eax

Syscall function info:
    name
    ID
    parameters

API function info:
    name
```

```
    address
25  parameters
```

Предлагаемые в этой работе профили применимы к семействам ОС (не зависят от версии системы), в то время как DECAF вынужден иметь конфигурационный файл для каждой версии. Также конфигурирование разработанного инструмента требует меньше параметров и может быть осуществлено вручную. Профили DECAF генерируются автоматически программой-агентом и при ее отсутствии составить такой профиль вручную является весьма трудной задачей. Кроме того, не в каждую систему можно загрузить свою программу. Также могут возникать ситуации с некорректным определением версии гостевой системы, вследствие чего результата может не быть или смещения окажутся неверными, что сделает анализ невозможным.

Что касается трассировки API, в этом случае инструменты практически идентичны, потому что для отслеживания конкретной библиотечной функции необходимо иметь о ней всю информацию, как показано в приложении А.2. Для разработанного инструмента ситуация выглядит также.

5.3 Достоверность получаемых результатов

Достоверность получаемых с помощью инструмента данных является важным аспектом. Проверить это можно с помощью утилит, доверие к которым не ставится под сомнение.

Для проверки в операционной системе Windows использовался вывод утилиты «Диспетчер задач». Она запускалась несколько раз в процессе работы системы, в это время открывались и закрывались различные приложения. Параллельно работал плагин отслеживания процессов и запра-

шивался список процессов. На каждом этапе полученный список процессов соответствовал выводу утилиты «Диспетчер задач».

Для ОС Linux корректность метода определять по трассировке системных вызовов. В Linux есть специальная утилита «strace», отслеживающая системные вызовы. Для теста запускался плагин перехватчик системных вызовов и утилита «strace», затем выводы сравнивались. Результаты оказались идентичными.

5.4 Выводы по главе

В главе представлена оценка разработанного инструмента по трем параметрам. По результатам оценки выяснилось, что инструмент не дает большого замедления при своей работе (от 3 до 14% в зависимости от гостевой операционной системы и задачи), прост при поддержке новых гостевых ОС, а также предоставляет достоверные результаты.

Заключение

В результате проведенной работы были получены следующие научные результаты:

1. Разработан метод мониторинга состояния виртуальной машины для восстановления объектов операционной системы без внедрения инструментального кода в исследуемую систему.
2. Разработан метод принудительного вызова системных функций в виртуальной машине для восстановления атрибутов объектов операционной системы при детерминированном воспроизведении.

Также в работе был получен практический результат:

1. На основе предложенных автором методов разработана и реализована инструментальная среда, поддерживающая три семейства операционных систем общего назначения (Windows, Linux, FreeBSD).

Разработанное средство оформлено в виде набора библиотек, которые интегрированы как подключаемые модули в эмулятор QEMU. Разработанные модули расширяют возможности анализа исполняемого кода и поведения исследуемой операционной системы и приложений. Некоторые части разработанного средства являются машинно-независимыми, что позволяет использовать их на всех платформах, поддерживаемых эмулятором QEMU.

От существующих инструментов разработанное средство отличает отсутствие использования внутренних структур ядра исследуемой системы и вторжений в систему, а также для его работы требуется небольшой объем выходных данных, находящихся в свободном доступе. Перечисленные свойства позволяют анализировать не только настольные ОС, но и встроенные

системы. Инструмент можно использовать в связке с детерминированным воспроизведением, решая проблему замедления при работе в виртуальной среде.

Эффективность и достоверность полученных практических выводов подтверждена тестировочными испытаниями и сравнениями с системными утилитами операционных систем.

Научные результаты исследования были опубликованы в нескольких выпуках журнала «Труды ИСП РАН», сборнике с конференции в Италии, журнале «Системы высокой доступности» и «Программирование». Результаты выносились на обсуждение на конференциях:

- Международная конференция РусКрипто, Солнечногорск, Россия, 25-28 марта, 2014.
- Открытая конференция по компиляторным технологиям, Москва, Россия, 2 декабря, 2015.
- SAC '16 Proceedings of the 31st Annual ACM Symposium on Applied Computing, Пиза, Италия, 4-8 апреля, 2016.
- Международная конференция РусКрипто, Солнечногорск, Россия, 21-24 марта, 2017.
- Международная Ершовская конференция по информатике PSI–2017, Москва, Россия, 27–29 июня 2017.

Все научные и практические результаты получены автором самостоятельно.

Дальнейшее развитие исследований может идти в следующи направлениях:

- Поддержка других процессорных архитектур.
- Расширение списка восстаналиваемых объектов операционной системы.
- Извлечение отладочной информации из исполняемых файлов.

Список литературы

1. *Климушенкова М.* [и др.] О некоторых ограничениях полносистемного анализа помеченных данных // Труды Института системного программирования РАН. — 2016.
2. *Portokalidis G., Slowinska A., Bos H.* Argos: An Emulator for Fingerprinting Zero-day Attacks for Advertised Honeypots with Automatic Signature Generation // SIGOPS Oper. Syst. Rev. — New York, NY, USA, 2006. — Апр. — Т. 40, № 4. — С. 15—27. — DOI: 10.1145/1218063.1217938. — URL: http://doi.acm.org/10.1145/1218063.1217938.
3. *Падарян В.* [и др.] Методы и программные средства, поддерживающие комбинированный анализ бинарного кода // Труды Института системного программирования РАН. — 2014. — Т. 1, № 26. — С. 251—276.
4. *Dovgalyuk P.* Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging // Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering. — New York, NY, USA, 2012. — С. 553—556.
5. *Hizver J., Chiueh T.-c.* Real-time Deep Virtual Machine Introspection and Its Applications // Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. — Salt Lake City, Utah, USA : ACM, 2014. — С. 3—14. — (VEE '14). — ISBN 978-1-4503-2764-0. — DOI: 10.1145/2576195.2576196. — URL: http://doi.acm.org/10.1145/2576195.2576196.

6. *Xu M.* [и др.] ReTrace: Collecting Execution Trace with Vitrual Machine Deterministic Replay // Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. — New York, NY, USA, 2008. — С. 121—130.

7. *Oliveira D.* [и др.] Ianus: Secure and Holistic Coexistence with Kernel Extensions - a Immune System-inspired Approach // Proceedings of the 29th Annual ACM Symposium on Applied Computing. — Gyeongju, Republic of Korea : ACM, 2014. — С. 1672—1679. — (SAC '14). — ISBN 978-1-4503-2469-4. — DOI: 10.1145/2554850.2554923. — URL: http://doi.acm.org/10.1145/2554850.2554923.

8. *Dolan-Gavitt B.* [и др.] Repeatable Reverse Engineering with PANDA // Proceedings of the 5th Program Protection and Reverse Engineering Workshop. — Los Angeles, CA, USA : ACM, 2015. — 4:1—4:11. — (PPREW-5). — ISBN 978-1-4503-3642-0. — DOI: 10.1145/2843859.2843867. — URL: http://doi.acm.org/10.1145/2843859.2843867.

9. *Henderson A.* [и др.] Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform // Proceedings of the 2014 International Symposium on Software Testing and Analysis. — San Jose, CA, USA : ACM, 2014. — С. 248—258. — (ISSTA 2014). — ISBN 978-1-4503-2645-2. — DOI: 10.1145/2610384.2610407. — URL: http://doi.acm.org/10.1145/2610384.2610407.

10. *Hizver J.*, *Chiueh T.-c.* Real-time Deep Virtual Machine Introspection and Its Applications // Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. — Salt Lake City, Utah, USA : ACM, 2014. — С. 3—14. — (VEE '14). — ISBN 978-1-4503-2764-0. — DOI: 10.1145/2576195.2576196. — URL: http://doi.acm.org/10.1145/2576195.2576196.

11. *Фурсова Н. И.*, *Довгалюк П.*, *Васильев И. А.* Использование ABI для интроспекции виртуальных машин // Труды Института системного программирования РАН. — 2015. — № 27. — С. 159—168.

12. *Фурсова Н. И.* [и др.] Легковесный метод интроспекции виртуальных машин // Programming and Computer Software. — 2017. — № 5. — С. 307—313.

13. *Васильев И.* [и др.] Модули для инструментирования исполняемого кода в симуляторе QEMU // Проблемы информационной безопасности. Компьютерные системы. — 2015. — № 4. — С. 194—203.

14. *Фурсова Н. И.* Интроспекция виртуальных машин // Ученые записки Новгородского государственного университета имени Ярослава Мудрого. — 2015. — № 3. — С. 1—3.

15. *Fursova N.* Introspection of the Virtual Machines with System Calls Monitoring: Student Research Abstract // Proceedings of the 31st Annual ACM Symposium on Applied Computing. — Pisa, Italy : ACM, 2016. — С. 1582—1583. — (SAC '16). — ISBN 978-1-4503-3739-7. — DOI: 10.1145/2851613.2852008. — URL: http://doi.acm.org/10.1145/2851613.2852008.

16. *Dovgalyuk P.* [и др.] QEMU-based Framework for Non-intrusive Virtual Machine Instrumentation and Introspection // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. — Paderborn, Germany : ACM, 2017. — С. 944—948. — (ESEC/FSE 2017). — ISBN 978-1-4503-5105-8. — DOI: 10.1145/3106237.3122817. — URL: http://doi.acm.org/10.1145/3106237.3122817.

17. *Chen P. M.*, *Noble B. D.* When Virtual Is Better Than Real // Proceedings of the Eighth Workshop on Hot Topics in Operating Systems. — Washington, DC, USA : IEEE Computer Society, 2001. —

С. 133—. — (HOTOS '01). — URL: http://dl.acm.org/citation.cfm?id=874075.876409.

18. *Fu Y.*, *Lin Z.* Bridging the Semantic Gap in Virtual Machine Introspection via Online Kernel Data Redirection // ACM Trans. Inf. Syst. Secur. — New York, NY, USA, 2013. — Сент. — Т. 16, № 2. — 7:1—7:29. — DOI: 10.1145/2505124. — URL: http://doi.acm.org/10.1145/2505124.

19. *Pfoh J.*, *Schneider C.*, *Eckert C.* A formal model for virtual machine introspection // Proceedings of the 1st ACM workshop on Virtual machine security. — ACM. 2009. — С. 1—10.

20. *Schneider C.*, *Pfoh J.*, *Eckert C.* A Universal Semantic Bridge for Virtual Machine Introspection // Proceedings of the 7th International Conference on Information Systems Security. — Kolkata, India : Springer-Verlag, 2011. — С. 370—373. — (ICISS'11). — ISBN 978-3-642-25559-5. — DOI: 10.1007/978-3-642-25560-1_25. — URL: http://dx.doi.org/10.1007/978-3-642-25560-1_25.

21. *More A.*, *Tapaswi S.* Virtual machine introspection: towards bridging the semantic gap // Journal of Cloud Computing. — 2014. — Т. 3, № 1. — С. 1.

22. *Савицкий В.*, *Сидоров Д.* «Ленивый» анализ исходного кода на языках c и c++ // Труды Института системного программирования РАН. — 2012. — Т. 23.

23. *Савицкий В.* Инкрементальный анализ исходного кода на языках C/C++ // Труды Института системного программирования РАН. — 2012. — Т. 22.

24. *Вартанов С.*, *Герасимов А.* Динамический анализ программ с целью поиска ошибок и уязвимостей при помощи целенаправленной генера-

ции входных данных // Труды Института системного программирования РАН. — 2014. — Т. 26, № 1.

25. Volatility Foundation [Электронный ресурс]. — URL: https://www.volatilityfoundation.org (visited on 12/27/2016).

26. *Petroni N. L.* [и др.] FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory // Digital Investigation. — 2006. — Т. 3.

27. InSight project website [Электронный ресурс]. — URL: https://code.google.com/p/insight-vmi/ (visited on 12/27/2016).

28. *Stamatogiannakis M.*, *Groth P.*, *Bos H.* Decoupling Provenance Capture and Analysis from Execution // 7th USENIX Workshop on the Theory and Practice of Provenance (TaPP 15). — Edinburgh, Scotland : USENIX Association, июль 2015. — URL: https://www.usenix.org/conference/tapp15/workshop-program/presentation/stamatogiannakis.

29. *Yin H.*, *Song D.* TEMU: Binary Code Analysis via Whole-System Layered Annotative Execution: тех. отч. / EECS Department, University of California, Berkeley. — Янв. 2010. — UCB/EECS-2010-3. — URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-3.html.

30. *Chow J.* [и др.] Understanding data lifetime via whole system simulation // Proceedings of the 13th USENIX Security Symposium (Security'03). — 2004.

31. *Pfoh J.*, *Schneider C.*, *Eckert C.* Nitro: Hardware-based System Call Tracing for Virtual Machines // Proceedings of the 6th International Conference on Advances in Information and Computer Security. — Tokyo, Japan : Springer-Verlag, 2011. — С. 96—112. — (IWSEC'11). — ISBN 978-3-642-25140-5. — URL: http://dl.acm.org/citation.cfm?id=2075658.2075669.

32. *Dolan-Gavitt B.* [и др.] Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection // Proceedings of the 2011 IEEE Symposium on Security and Privacy. — Washington, DC, USA : IEEE Computer Society, 2011. — С. 297—312. — (SP '11). — ISBN 978-0-7695-4402-1. — DOI: 10.1109/SP.2011.11. — URL: http://dx.doi.org/10.1109/SP.2011.11.

33. *Luk C.-K.* [и др.] Pin: building customized program analysis tools with dynamic instrumentation // Acm sigplan notices. Т. 40. — ACM. 2005. — С. 190—200.

34. *Bungale P. P.*, *Luk C.-K.* PinOS: A Programmable Framework for Whole-system Dynamic Instrumentation // Proceedings of the 3rd International Conference on Virtual Execution Environments. — San Diego, California, USA : ACM, 2007. — С. 137—147. — (VEE '07). — ISBN 978-1-59593-630-1. — DOI: 10.1145/1254810.1254830. — URL: http://doi.acm.org/10.1145/1254810.1254830.

35. *Barham P.* [и др.] Xen and the art of virtualization // ACM SIGOPS operating systems review. Т. 37. — ACM. 2003. — С. 164—177.

36. *Таненбаум Э. С.*, *Бос Х.* Современные операционные системы. — Питер, 2015. — 1120 с.

37. Академия Microsoft: Основы организации операционных систем Microsoft Windows [Электронный ресурс]. — URL: http://www.intuit.ru / studies / courses / 1089 / 217 / lecture / 5601 ? page = 2 (дата обр. 27.08.2017).

38. Отображение файла в память (Операционные Системы) [Электронный ресурс]. — URL: http://ru.bmstu.wiki (дата обр. 27.08.2017).

39. *Касперски К.* ПК: решение проблем. — BHV, 2003. — 560 с.

40. Академия Intel: Основы операционных систем [Электронный ресурс]. — URL: http://www.intuit.ru/studies/courses/2192/31/lecture/968?page=3 (дата обр. 27.08.2017).

41. Application Binary Interface [Электронный ресурс]. — URL: https://en.wikipedia.org/wiki/Application_binary_interface (дата обр. 02.08.2017).

42. Intel® 64 and IA-32 Architectures Software Developer's Manual [Электронный ресурс]. — URL: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf (дата обр. 02.08.2017).

43. *Chow J.* [и др.] Multi-stage Replay with Crosscut // Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. — Pittsburgh, Pennsylvania, USA : ACM, 2010. — С. 13—24. — (VEE '10). — ISBN 978-1-60558-910-7. — DOI: 10.1145/1735997.1736002. — URL: http://doi.acm.org/10.1145/1735997.1736002.

44. *Довгалюк П.* Детерминированное воспроизведение процесса выполнения программ в виртуальной машине // Труды Института системного программирования РАН. — 2011. — № 21. — С. 123—132.

45. *Wang G.* [и др.] Hypervisor Introspection: A Technique for Evading Passive Virtual Machine Monitoring // 9th USENIX Workshop on Offensive Technologies (WOOT 15). — Washington, D.C. : USENIX Association, 2015. — URL: https://www.usenix.org/conference/woot15/workshop-program/presentation/wang.

46. *Фурсова Н.* [и др.] Способы обратной отладки мобильных приложений // Материалы XVI международной научно-практической конференции «РусКрипто'2014». — 2014.

47. QEMU the FAST! processor emulator [Электронный ресурс]. — URL: https://www.qemu.org/ (дата обр. 03.08.2017).

48. QEMU [Электронный ресурс]. — URL: https://en.wikipedia.org/wiki/QEMU (дата обр. 03.08.2017).

49. *Руссинович М.*, *Соломон Д.* Внутреннее устройство Microsoft Windows. — Питер, 2013. — 800 с.

50. *Уорд Б.* Внутреннее устройство Linux. — Питер, 2016. — 384 с.

51. *МакКузик М. К.*, *Невилл-Нил Д. В.* FreeBSD: архитектура и реализация. — Москва, 2006. — 800 с.

52. x86 Disassembly/Windows Executable Files [Электронный ресурс]. — URL: https://en.wikibooks.org/wiki/X86_Disassembly/Windows_Executable_Files#PE_Header (дата обр. 03.08.2017).

53. *Слинкин В.* PE (Portable Executable): На странных берегах [Электронный ресурс]. — 2015. — URL: https://habrahabr.ru/post/266831/ (дата обр. 03.08.2017).

54. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification [Электронный ресурс]. — URL: http://refspecs.linuxbase.org/elf/elf.pdf (дата обр. 03.08.2017).

Список рисунков

Список таблиц

Приложение А

Конфигурационные файлы инструмента DECAF

А.1 Основной конфигурационный файл

```
;Ubuntu 12.04 3.5.0-23-generic
strName = 3.5.0-23-generic
init_task_addr  = 3246813760
init_task_size  = 3244
ts_tasks        = 440
ts_pid          = 520
ts_tgid         = 524
ts_group_leader = 556
ts_thread_group = 612
ts_real_parent  = 532
ts_mm           = 468
ts_stack        = 4
module_name     = 12
module_size     = 228
module_init     = 220
module_list     = 4
ts_real_cred    = 732
ts_cred         = 736
ts_comm         = 740
cred_uid        = 4
cred_gid        = 8
cred_euid       = 20
cred_egid       = 24
mm_mmap         = 0
mm_pgd          = 36
```

```
mm_arg_start    = 152
mm_start_brk    = 140
mm_brk          = 144
mm_start_stack  = 148
vma_vm_start    = 4
vma_vm_end      = 8
vma_vm_next     = 12
vma_vm_file     = 80
vma_vm_flags    = 28
vma_vm_pgoff    = 76
file_dentry     = 12
file_inode      = 32
dentry_d_name   = 20
dentry_d_iname  = 36
dentry_d_parent = 16
ti_task         = 0
inode_ino   = 40
proc_fork_connector  = 3241872240
proc_exit_connector  = 3241873696
proc_exec_connector  = 3241872480
vma_link  = 3239231168
vma_adjust  = 3239231456
remove_vma  = 3239229792
modules     = 3246901080
trim_init_extable = 3238241600
```

A.2 Фрагмент конфигурационного файла для трассировщика API вызовов

```
# Sample configuration file to hold the api that need to be
   traced.

api {
```

```
modname   = ntdll.dll
apiname   = NtCreateFile
numberargs  = 11
iskernel  = no
returntype  = u32
returnstr = NTSTATUS
returnhandler   = default
convention  = STDCALL
arg {
  name  = FileHandle
  type  = u32
  spec  = out
  handler = default
  size  = 4
}
arg {
  name  = DesiredAccess
  type  = u32
  spec  = in
  handler = default
  size  = 4
}
arg {
  name  = ObjectAttributes
  type  = u32
  spec  = in
  handler = default
  size  = 4
}
arg {
  name  = IoStatusBlock
  type  = u32
  spec  = out
  handler = default
```

```
    size  = 4
  }
  arg {
    name  = AllocationSize
    type  = IGNORE
    handler = default
    size  = 4
  }
  arg {
    name  = FileAttributes
    type  = u32
    spec  = in
    handler = default
    size  = 4
  }
  arg {
    name  = ShareAccess
    type  = u32
    spec  = in
    handler = default
    size  = 4
  }
  arg {
    name  = CreateDisposition
    type  = u32
    spec  = in
    handler = default
    size  = 4
  }
  arg {
    name  = CreateOptions
    type  = u32
    spec  = in
    handler = default
```

```
    size  = 4
  }
  arg {
    name  = EaBuffer
    type  = u32
    spec  = in
    handler = default
    size  = 4
  }
  arg {
    name  = EaLength
    type  = u32
    spec  = in
    handler = default
    size  = 4
  }
}
```