DOI: 10.15514/ISPRAS-2025-37(6)-36

Интроспекция виртуальной машины на основе мониторинга системных вызовов и структур данных ядра

В.М. Степанов, ORCID: 0000-0003-2141-3333 <vladislav.stepanov@ispras.ru>
П.М. Довгалюк, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>
Н.И. Фурсова, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>

Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.
Новгородский государственный университет им. Ярослава Мудрого,
Россия, 173003, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41.

Аннотация. Семантический разрыв представляет собой одну из ключевых проблем в разработке решений полносистемного динамического анализа кода. Она заключается в том, что на уровне гипервизора инструмент имеет доступ только к низкоуровневым бинарным данным выполняемого кода, в то время как для анализа требуется высокоуровневая информация о состоянии объектов гостевой операционной системы. Данную проблему решают подходы интроспекции виртуальной машины. К сожалению, реализации существующих подходов сталкиваются с проблемами производительности и недостатка функционала, требуют от пользователя внедрять в образ виртуальной машины специальные агенты или иметь в наличии отладочные символы к ядру, а также оказываются заточенными под специфичные системы и архитектуры процессоров. В статье представлен ряд решений, помогающих снизить накладные расходы и добиться большей универсальности для инструмента анализа. Особенность разработанного подхода интроспекции заключается в том, что необходимую для анализа информацию он собирает в процессе запуска системы на эмуляторе, не требуя при этом каких-либо дополнительных действий со стороны пользователя.

Ключевые слова: виртуальные машины; динамический анализ; эмулятор QEMU; интроспекция.

Для цитирования: Степанов В.М., Довгалюк П.М., Фурсова Н.И. Интроспекция виртуальной машины на основе мониторинга системных вызовов и структур данных ядра. Труды ИСП РАН, том 37, вып. 6, часть 3, 2025 г., стр. 59–72. DOI: 10.15514/ISPRAS–2025–37(6)–36.

Благодарности: Исследование выполнено за счет гранта Российского научного фонда № 24-11-20022.

Virtual Machine Introspection Based on System Calls and Kernel Data Structures

V.M. Stepanov, ORCID: 0000-0003-2141-3333 <vladislav.stepanov@ispras.ru>
P.M. Dovgalyuk, ORCID: 0000-0003-2483-5718 <pavel.dovgalyuk@ispras.ru>
N.I. Fursova, ORCID: 0000-0001-6817-4670 <natalia.fursova@ispras.ru>

Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn str., Moscow, 109004, Russia.
Yaroslav-the-Wise Novgorod State University,
41, B. Sankt- Peterburgskaya st., Veliky Novgorod, 173003, Russia.

Abstract. The semantic gap is one of the key problems in developing solutions for full-system dynamic analysis. At the hypervisor level, tools have access only to low-level binary data, while analysis requires high-level information about the state of guest operating system objects. Virtual machine introspection approaches solve this problem. Unfortunately, implementations of existing approaches face performance issues and lack of functionality. They require the user to embed special agents into the virtual machine image or have debugging symbols for the OS kernel. They also turn out to work only for specific systems and processor architectures. The article presents a number of solutions that reduce overhead and increase the versatility of the analysis tool. The peculiarity of the developed introspection approach is that it does not require any additional actions from the user, collecting the information necessary for analysis during the OS boot on the emulator.

Keywords: virtual machine; dynamic analysis, QEMU; virtual machine introspection.
For citation: Stepanov V.M., Dovgalyuk P.M., Fursova N.I. Virtual Machine Introspection Based on System Calls and Kernel Data Structures. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, part 3, 2025, pp. 59-72 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-36.

Acknowledgements. The work was partially supported by the Russian Science Foundation (grant No. 24-11- 20022).

1. Введение

Технология интроспекции виртуальной машины имеет широкое применение в инструментах динамического анализа. Она реализует преобразование низкоуровневых бинарных данных, доступных на уровне гипервизора, в высокоуровневую информацию об объектах ОС. В инструменте Natch [1] с ее помощью собирается информация о поверхности атаки, т.е. об исполняемых файлах и функциях, которые отвечают за обработку данных, полученных из недоверенных источников. Это достигается за счет комбинирования интроспекции с отслеживанием помеченных данных.

Основная проблема существующих методов интроспекции виртуальных машин заключается в их узкой специализации и сложности применения. Архитектура операционных систем может сильно различаться в зависимости от семейства, версии ядра, флагов компиляции и других факторов. Поэтому построение универсального подхода, способного осуществлять мониторинг на любых гостевых ОС, является практически недостижимой целью. Большинство работ в данной сфере ограничиваются только реализацией интроспекции для ОС семейства Linux. Существующие инструменты также часто требуют наличия отладочных символов к анализируемому ядру [2-4], либо внедрения в это ядро специальных модулей, собирающих о нем информацию [4-6].

Мы предлагаем способ сбора необходимых для интроспекции данных о системе на основе простого запуска ОС на виртуальной машине. Это упрощает процесс подготовки образа виртуальной машины к анализу, ведь пользователю конечного инструмента потребуется только подождать, пока не закончится автоматическая настройка. По завершению такой настройки генерируется конфигурационный файл, называемый профилем интроспекции.

Существует ряд работ, в которых задача построения профиля интроспекции для системы решается на основе анализа снимков памяти с применением специально подготовленных разработчиком эвристик [7, 8]. В других исследованиях процесс построения эвристик автоматизируется на основе статического анализа бинарного кода [9], либо применения машинного обучения [10]. Есть также работы демонстрирующие эффективность применения для данной задачи динамического анализа, например подходы Katana [11] и ORIGEN [12].

В данной статье описывается подход интроспекции виртуальной машины для гостевых ОС Linux, Windows и FreeBSD. Он опирается на отслеживание изменений в структурах данных ядра на протяжении сеанса работы эмулятора. Для достижения универсальности подхода вводятся абстрактные типы структур данных ядра, которые обобщают реальные типы структур разных семейств ОС. Таким образом, становится возможной единая реализация алгоритмов интроспекции без привязки к конкретным архитектурам. С использованием этих абстрактных типов также реализуется набор тестов, проверяющих корректность извлекаемой из структур данных ядра информации. Эти тесты применяются для подбора смещений полей и определения размеров структур, которые по ходу настройки заносятся в генерируемый профиль интроспекции.

2. Обзор существующих решений

Инструменты TEMU [4], DECAF [5] и PANDA [6] реализуют задачи интроспекции виртуальной машины на базе эмулятора QEMU [13]. Во всех трех работах применяются внедряемые в гостевую ОС модули ядра. В TEMU такой модуль используется во время исполнения для извлечения высокоуровневых данных о состоянии Windows. В DECAF и PANDA внедряемые агенты применяются только для создания конфигурационного файла, применяемого затем в анализе без осуществления вмешательства в гостевую ОС.

Инструменты DRAKVUF [2] и RTKDSM [3] работают на основе эмулятора Xen, из-за чего ограничены в применении только архитектурой процессоров x86. Для своей работы они требуют наличия отладочных символов ядра, которые используются для извлечения параметров объектов ОС из структур данных в гостевой памяти. В DRAKVUF отладочные символы применяется также для перехвата системных функций, на основе которых инструмент отслеживает происходящие изменения в системе. В свою очередь RTKDSM выполняет мониторинг объектов ОС, отслеживая модификации страниц памяти со структурами данных ядра.

Табл. 1. Сравнение подходов интроспекции.
Table 1 Comparison of introspection approaches.

	TEMU	DECAF	PANDA	RTKDSM	Drakvuf
Внедряет гостевые драйвера	для исполнения (в Windows)	для настройки	для настройки	нет	нет
Требует наличия отладочных символов	да (для Linux)	нет	нет	да	да
Эмулятор	QEMU	QEMU	QEMU	Xen	Xen
Гостевые ОС	Windows, Linux	Windows, Linux	Windows, Linux, FreeBSD	Windows, Linux	Windows, Linux
Архитектуры процессоров	x86	x86, Arm	x86, Arm, Mips	x86	x86
Отслеживает системные вызовы	нет	нет	да	нет	да

При работе с вышеперечисленными инструментами необходимо иметь в наличии отладочные символы ядра или внедрять в ядро гостевой ОС специальные агенты. В первом случае может возникнуть проблема, когда разработчик дистрибутива не предоставляет такие символы к своей системе, или возможно даже не хранит их для старых версий. Что касается внедряемых агентов, то с некоторыми образами виртуальной машины произвести подобные модификации оказывается затруднительно или невозможно.

Ряд инструментов умеет выполнять задачу построения профиля интроспекции только на базе анализа бинарного кода. Например, подходы Katana[11] и RAMAnalyzer [13] для этого перехватывают вызовы определенных функций ядра, выполняющих доступ к искомым параметрам. При этом для распознавания функций ядра они извлекают отладочные символы Linux из механизма kallsyms. Проблема таких решений в том, что kallsyms может быть отключен в конфигурации ядра, а функции могут быть сильно изменены при переходе на новые версии. Для решения последней проблемы Katana применяет анализ исходного кода Linux, позволяющий автоматически определять функции ядра, на основе которых в ходе динамического анализа определяются смещения полей структур данных.

Подход ORIGEN [12] применяет сопоставление бинарного кода разных версий ядра ОС для переноса отладочной информации с одной системы на другую. Он использует динамический анализ для нахождения операций доступа к искомым полям структур данных в эталонном ядре, затем находит эквивалентные инструкции в целевом ядре, по которым восстанавливает смещения.

3. Natch

Далее обсудим полностью автоматизированный подход восстановления архитектуры ОС, работающий исключительно на основе анализа бинарного кода, а значит не требующий внедрения гостевых агентов и наличия отладочных символов ядра. Это решение легло в принцип работы инструмента определения поверхности атаки Natch.

3.1 Общая структура подхода

Реализация подхода представляет собой набор плагинов к эмулятору QEMU, которые осуществляют инструментирование выполняемого кода. В силу того, что такой способ сбора данных о работе гостевой системы замедляет работу эмулятора, плагины для мониторинга объектов ОС предпочтительно применять совместно с детерминированным воспроизведением. Это нужно для того, чтобы замедление от средств анализа не оказывало влияние на поведение приложений в гостевой системе.

Исходя из этого, сценарий работы с инструментом включает в себя следующие шаги:

1) Подготовка образа и запуск гостевой ОС для построения профиля интроспекции.
2) Запись сценария работы с объектами оценки средствами эмулятора QEMU.
3) Воспроизведение записанного сценария со сбором данных о состоянии объектов гостевой ОС.

На рис. 1 можно увидеть, что за чтение структур данных во время мониторинга и за создание профиля интроспекции отвечают два отдельных плагина. Они опираются на общую библиотеку, описывающую для каждого семейства ОС алгоритмы извлечения параметров структур данных. Плагины мониторинга процессов и файлов в своей работе взаимодействуют с плагинами трассировки системных вызовов и чтения структур данных ядра. Другие плагины опираются на мониторинг объектов для выполнения некоторых специфичных задач, таких как отслеживание исполняемых модулей, пометка файлов и построение графов распространения помеченных данных. По завершению мониторинга объектов гостевой ОС формируются лог-файлы и графы, которые затем могут быть приведены к более удобному для аналитика формату с помощью средств визуализации.

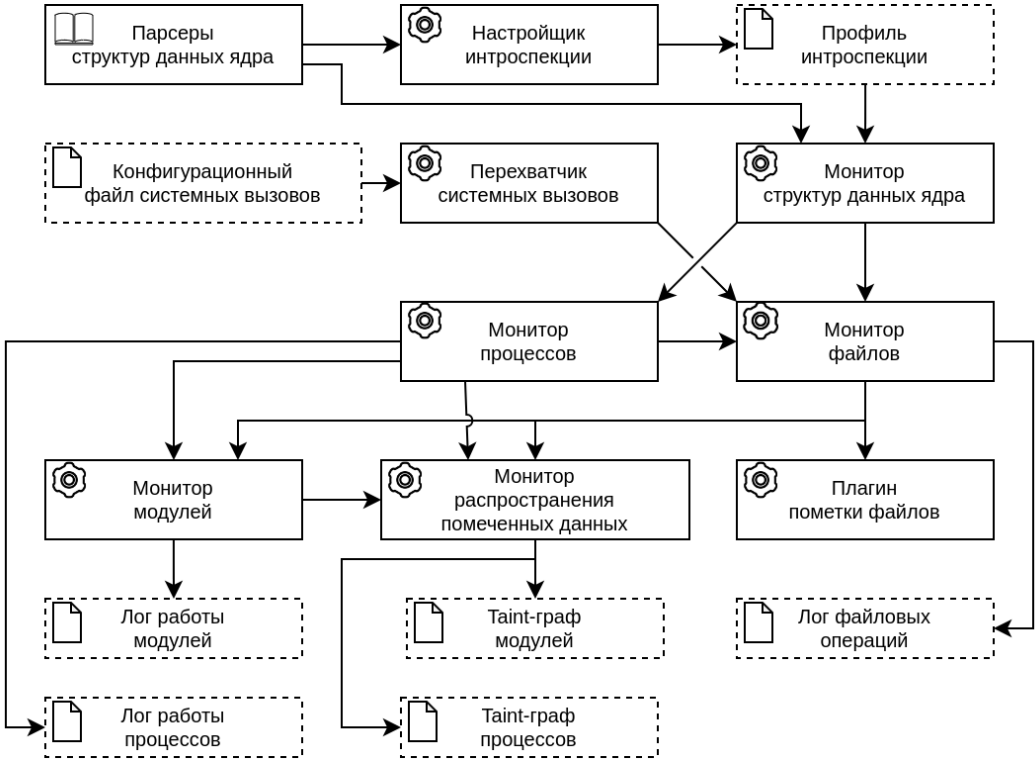

Рис. 1. Схема взаимодействий плагинов интроспекции.
Fig. 1. Introspection plugin interaction diagram.

3.2 Мониторинг объектов ОС

К основным объектам ОС, которые представляют интерес для отслеживания, относятся процессы, потоки, файлы, каталоги, системные вызовы. Задача интроспекции ВМ предполагает преобразование низкоуровневых данных, таких как потоки инструкций и снимки памяти в высокоуровневую информацию об этих объектах.

Свойства объектов описываются в специальных структурах данных ядра, называемых дескрипторами. Для нахождения этих дескрипторов в памяти используется профиль интроспекции анализируемой ОС. Также существует способ отслеживания информации об объектах без использования структур ядра. Он заключается в перехвате системных функций с обработкой их входных и выходных данных. Системные вызовы, как правило, менее изменчивы по сравнению со структурами данных, поэтому одна реализация их трассировки может работать на большом диапазоне версий ядра ОС [13].

Объекты ОС взаимосвязаны и иногда дублируют одни и те же данные. Например, имена пользовательских процессов могут быть извлечены из дескриптора процесса, из дескриптора исполняемого файла, а также из входных данных системной функции execve. Благодаря этому свойству можно проверять корректность параметров на специальном наборе тестов, в ходе которых сопоставляются полученные из разных мест данные. Такие наборы тестов позволяют удостовериться в соответствии профиля интроспекции анализируемой гостевой ОС.

Между семействами ОС Linux, Windows и FreeBSD существует множество различий с точки зрения способа хранения метаданных объектов в памяти. Для упрощения работы с ними мы определяем абстрактные типы структур данных, которые обобщают некоторые структуры

ядра разных ОС. Для извлечения параметров объектов задается общий интерфейс, каждая функция которого реализуется отдельно под каждую архитектуру. В свою очередь плагины эмулятора, выполняющие мониторинг за объектами, описываются на более высоком уровне с использованием абстрактных типов, поэтому оказываются абстрагированы от особенностей реализации гостевых систем.

Как показано на рис. 2, мониторинг процессов может отслеживать их имена, идентификаторы, состояние и иерархию. Дескриптор процесса представляет собой структуру *task_struct* в ядре Linux, *EPROCESS* в Windows, *Process* в FreeBSD. Определить местоположение дескриптора текущего выполняемого потока или процесса в каждом случае возможно по значению регистра GS для платформы x86, либо *TPIDR_EL1* для ARM. Через параметры текущего процесса удается выполнить переход к структурам ядра, описывающим все остальные объекты системы.

Рис. 2. Диаграмма структур данных ядра.
Fig. 2. Kernel data structures diagram.

Разбор таблицы объектов позволяет определить метаданные открытых процессом файлов. Кроме обычных файлов в той же таблице могут быть найдены сокеты и каналы, используемые для осуществления взаимодействий между процессами. Операции с файлами, такие как чтение и запись, отслеживаются по вызовам системных функций. Для средств взаимодействия также определяются параметры объектов, с которыми происходит соединение и обмен данными.

Дескрипторы областей виртуальной памяти организуются в виде деревьев поиска. По ним могут быть определены диапазоны адресов всей выделенной процессу памяти. В том числе можно найти исполняемый код процесса, стек, динамические библиотеки, разделяемую память и многое другое. Разбор этих структур ядра применяется в мониторинге исполняемых модулей, определении командной строки запуска процесса и отслеживании способов передачи при распространении помеченных данных (рис. 3).

3.3 Теневые структуры данных

Чтение данных из гостевой памяти является достаточно ресурсоёмкой операцией, вследствие чего ее частое применение может приводить к сильному замедлению работы виртуальной машины. Для ускорения анализа, а также корректной обработки изменений, все извлекаемые метаданные объектов следует кэшировать. На старте анализа выполняется сбор информации обо всех запущенных процессах и их ресурсах. Далее на протяжении сеанса работы виртуальной машины отслеживаются изменения в состояниях объектов ОС, в соответствии с чем обновляются данные кэша.

Изменения, как правило, сопровождаются системными вызовами. При этом в соответствии с типом системного вызова и его аргументами легко определить, в каких конкретно

объектах ОС что-то поменялось, и обновить только их. Файловый монитор, например, может обходиться совсем без чтения структур данных ядра, извлекая имена файлов из аргументов вызова open, а параметры сокетов из вызовов socket, bind и connect. Тем не менее часть изменений в объектах может происходить в отсутствие системных вызовов, что приводит к необходимости применения альтернативного метода отслеживания их состояний.

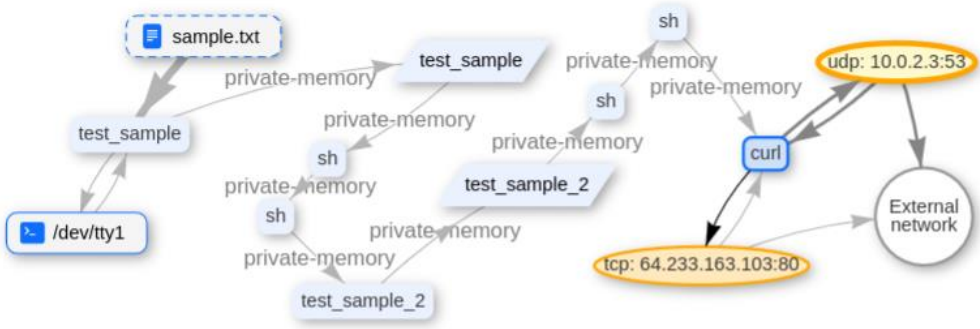

Рис. 3. Граф распространения помеченных данных по процессам.
Fig. 3. Taint graph for processes.

Другой метод реагирования на изменения опирается на перехват операций записи в структуры ядра. Так как запись каких-либо данных в память является крайне часто встречающейся операцией, инструментирование всех операций записи очевидно приводит к большим временным издержкам. Значительно облегчает задачу наличие в эмуляторе механизма точек наблюдения. Этот механизм позволяет инструментировать запись в определенные диапазоны памяти. При этом замедлению поддаются только те операции записи, которые направлены на физические страницы памяти с такими точками наблюдения.

Точки наблюдения удобно применять для отслеживания состояний процессов. Когда происходит завершение процесса, в его поле состояния записывается флаг смерти. Таким же образом может отслеживаться переход процесса в состояние зомби. Аналогичная идея: определять необходимость удаления файлового дескриптора из кэша по перехвату записи нулевого значения в его счетчик указателей.

Еще один способ применения точек наблюдения – это отслеживание записи в примитивы синхронизации структур данных ядра. Поля структур данных ядра часто оказываются защищены от возможного «состояния гонки» с помощью спин-блокировок, мьютексов или семафоров. Соответственно момент разблокировки таких примитивов может послужить для инструмента анализа сигналом к повторному чтению метаданных объекта ОС из гостевой памяти и обновлению теневых структур данных ядра.

3.4 Построение профиля интроспекции

Задача нахождения смещений полей структур данных в нашей работе решается на основе набора тестов, позволяющих удостовериться в корректности извлекаемых из гостевой памяти данных. Тесты организуются таким образом, чтобы проверки, требующие наименьшего количества известных смещений полей, выполнялись в первую очередь. Для каждого теста при этом выполняется перебор всех возможных значений смещений проверяемых им полей. Большинство извлекаемых из ядра параметров последовательно проверяются сразу на нескольких тестах, в результате чего отсеиваются наборы смещений, прошедшие первые проверки по ошибке.

Что представляет собой тест? Это небольшой алгоритм, выполняемый в некотором остановленном состоянии виртуальной машины. Как правило, в ходе выполнения теста

происходит попытка извлечения из гостевой памяти параметров ядра. Корректность этих параметров проверяется по области допустимых значений, либо по совпадению значений с некоторыми известными из другого источника данными.

Тесты можно поделить на две категории: обязательные к прохождению и эвристические. Первые всегда должны выдавать положительный результат при корректно подобранных смещениях полей. За счет этого они позволяют отсеивать все варианты значений, с которыми результат проверок отрицательный. С эвристическими проверками предполагается, что даже при корректно подобранных смещениях полей результат может оказаться отрицательным. Поэтому такие проверки нужно выполнять несколько раз на разных состояниях виртуальной машины, пока не удастся найти набор смещений, с которым они будут пройдены.

Тесты связываются между собой в соответствии с их зависимостями от полей структур данных и передаваемых параметров. Из этих связей формируется дерево. Корневым узлом, например, становится проверка, в ходе которой извлекается адрес текущего дескриптора процесса. Тесты, проверяющие отдельные параметры процесса, такие как имя, родитель, уникальный идентификатор, оказываются связаны с корнем. За счет этого они используют уже извлеченный адрес дескриптора процесса из предыдущей выполненной проверки, а не читают его из гостевой памяти вновь. По итогу, при удачном выполнении всех связанных тестов удается подтвердить, что проверяемая структура данных содержит все те поля, которые должны присутствовать в дескрипторе текущего процесса, а значит наиболее вероятно она им и является.

Некоторые проверки требуют своего выполнения на определенных состояниях виртуальной машины. Например, тест идентификатора процессов выполняется в момент завершения системного вызова `getpid` (В случае Linux и FreeBSD). В ходе его работы извлекаемый из структуры ядра идентификатор `pid` сравнивается с возвращаемым значением системной функции. Другой пример, смещения параметров файловых структур данных подбираются при перехвате вызова `open`. Здесь проверяется, что извлекаемое из структур ядра имя открываемого файла совпадает с тем, которое передается в системную функцию в качестве аргумента. Смещения параметров областей виртуальной памяти в свою очередь подбираются при перехвате вызова `mmap`. Для этой задачи реализуется тест, в ходе которого находится структура данных ядра, описывающая создаваемое отображение файла в память.

Основная идея подхода генерации профиля интроспекции заключается в последовательном нахождении смещений всех параметров структур ядра на базе ранее найденных. Так подбор смещений для файловых структур данных выполняется только после нахождения указателя на дескриптор текущего процесса. А тесты для нахождения дескрипторов областей виртуальной памяти и дескрипторов сокетов работают на базе ранее найденных файловых структур. Эти особенности выражаются в виде связей между тестами.

Зависимости, наборы перебираемых смещений и передаваемые между тестами параметры описываются в декларативной форме представления. За счет этого, при разработке алгоритмов генерации профиля интроспекции удается абстрагироваться от точной последовательности шагов по подбору смещений и описывать по большей части только сами проверки, включающие в себя извлечение из гостевой памяти искомых параметров. Также используемый формат организации тестов открывает возможность проверять с их помощью соответствие гостевой ОС уже готовому профилю интроспекции, либо начинать подбор смещений с середины, то есть на базе какого-то незаконченного профиля. Этапы переборов смещений в этих случаях просто пропускаются, т.к. их значения уже известны.

На рис. 4 в виде схемы представлен пример декларативного описания проверок. Узлы, обозначенные в виде скругленного прямоугольника, представляют собой точки останова, в которых выполняются следующие по цепочке проверки. В данном случае это системные вызовы `getpid`, `open` и `mmap`. Обычные прямоугольники описывают извлечение параметров из структур ядра и выполнение тестов с ними. С помощью стрелок указывается

последовательность выполнения, а также направление передачи всех вычисленных переменных, которые могут поступать на вход функциям проверки. Стрелки с пунктирной линией обозначают передачу найденных наборов смещений с одного узла на другой. Ромбы представляют собой операцию конъюнкции между разными ветвями тестов.

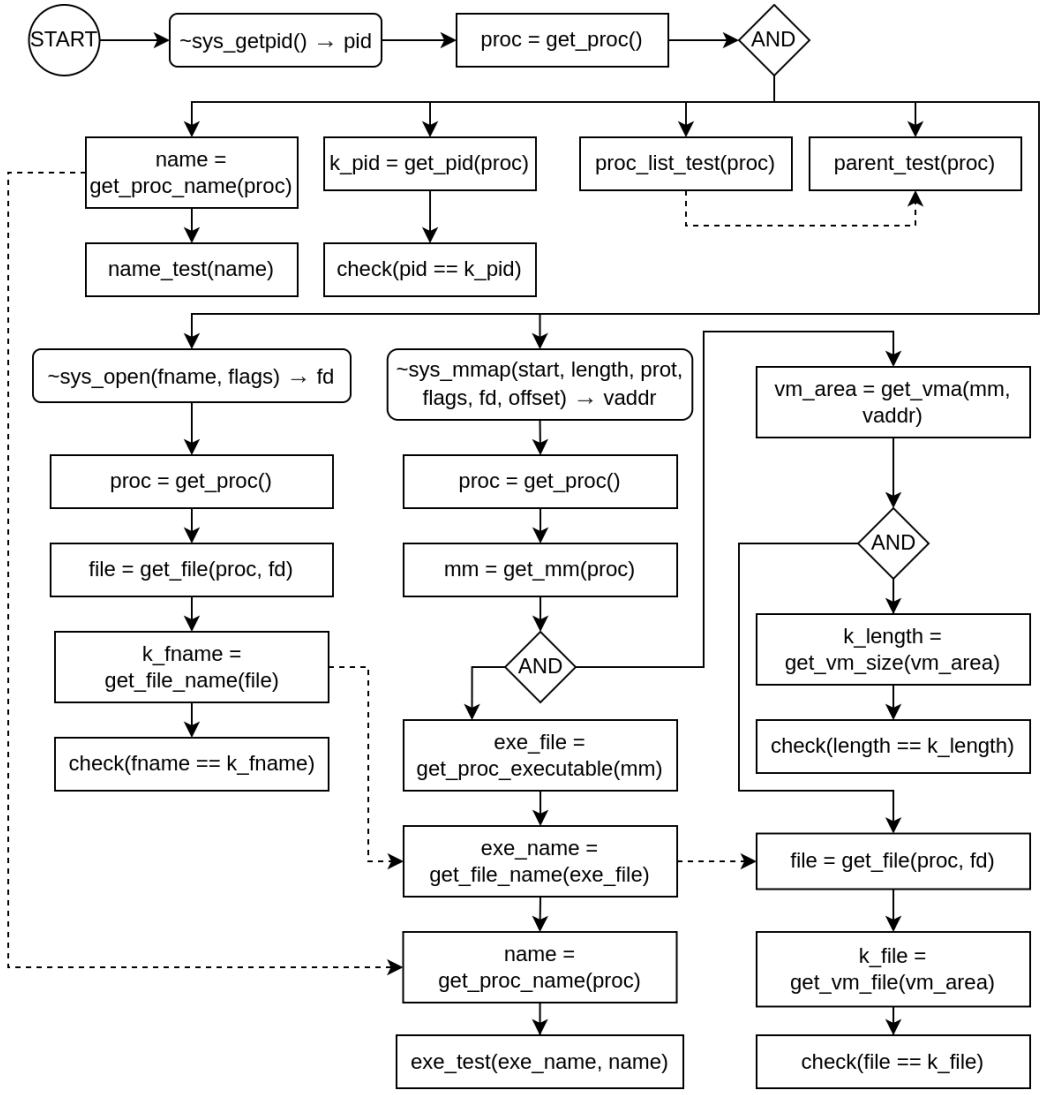

Рис. 4. Схема проверок для генерации профиля интроспекции.
Fig. 4. Scheme of checks for generating an introspection profile.

Тесты разделяются на разные ветви в зависимости от того, какие наборы смещений они проверяют и какие переменные они принимают на вход. Если тесты не зависят от проверяемых смещений друг друга, то их следует располагать на разных ветвях. За счет этого общее количество переборов смещений может значительно сокращаться.

Связанные между собой тесты и переборы проверяемых ими значений выполняются в порядке обхода дерева в глубину. При этом дочерние ветви узлов с переборами обрабатываются повторно на каждое выдаваемое значение. При прохождении тестов, расположенных на листовых узлах дерева, определяются конечные наборы смещений

параметров. В случае корректно составленных тестов такой набор смещений должен быть только один. В обратном же случае необходимы дополнительные тесты, которые уточнят местоположение искомых параметров. Для этой задачи был реализован механизм, позволяющий перед любым тестом в графе выполнить перебор наборов смещений с какого-либо другого узла. Таким образом, на более поздних этапах генерации профиля удается дополнительно протестировать ранее найденные параметры и отсечь некорректные варианты.

Приведем пример. В начале настройки наш алгоритм может найти несколько возможных вариантов смещения для параметра имени процесса. На этом этапе он проверяется исключительно по области допустимых значений и служит подтверждением того, что структура данных, в которой он находится, действительно является дескриптором процесса. Позже, когда становятся известны смещения в файловых структурах, выполняется подбор параметра исполняемого файла процесса. В ходе проверки все возможные варианты имени процесса сопоставляются с вариантами полного пути к исполняемому файлу. Как итог, при их совпадении получается подтвердить местоположение обоих искомых параметров.

Для повышения надежности подбора смещений также используется сбор статистики. Если проверки иногда дают ложно положительный результат, но работают корректно в большинстве случаев, то достаточно выполнить их несколько раз и выбрать тот набор смещений, который будет встречаться наиболее часто.

Говоря о подборе смещений полей, следует отметить, что в большинстве случаев их значения достаточно перебирать в рамках диапазона максимального размера структуры данных с определенным шагом, обычно равным размеру параметра. Когда вариантов немного, их бывает удобнее перебирать по некоторому списку возможных значений. Также есть параметры, смещения которых следует вычислять по формуле вместо каких-либо переборов. В качестве примера последних можно привести поле состояния процесса. Для его нахождения выполняется перехват системного вызова exit. Далее отслеживаются операции записи в область памяти дескриптора процесса. Выполняемый тест проверяет, что записываемое такой операцией значение представляет собой флаг смерти процесса, и в случае успеха вычисляет из адреса записи смещение искомого поля.

Для ускорения подбора смещений и построения профиля интроспекции в целом реализуется ряд оптимизаций. Они затрагивают механизмы обхода графа и чтения памяти, но не сами реализации проверок. Опишем некоторые из них.

Во время переборов смещений тесты могут обращаться множество раз к одним и тем же адресам гостевой памяти на одном и том же состоянии виртуальной машины. В связи с этим в рамках каждой остановки эмулятора выполняется кэширование ранее прочитанных из памяти значений, за счет чего параметры читаются из памяти значительно быстрее.

При поиске в структуре данных указателя на другую структуру может возникать ситуация, когда попадается несколько указателей с одним и тем же значением. Это может приводить к тому, что ряд последующих тестов, рассчитанных на поиск и проверку полей той другой структуры, будет повторно выполняться на одних и тех же данных, что потенциально может приводить к большому количеству лишних вычислений. В связи с этим алгоритм перебора отслеживает такие ситуации и выполняет последующие тесты только один раз. Для таких случаев также реализуются дополнительные проверки, которые определяют какой из дублирующихся указателей является истинным искомым параметром.

В тестах для извлечения параметров из гостевой памяти применяется тот же интерфейс с абстрактными дескрипторами объектов, что используются и для задач мониторинга. Это приводит к тому, что вызываемые функции извлечения параметров могут читать сразу несколько разных полей структур данных за раз. В ряде случаев при чтении какого-то определенного поля структуры бывает очевидно, что его смещение некорректно. Например, указатель на структуру может хранить неправильный адрес памяти. Тогда оказывается, что в

переборе значений последующих смещений полей нет смысла, ведь при всех их возможных значениях будет возникать одна и та же ошибка. В качестве решения проблемы для попыток извлечения параметров была реализована обработка ошибок, в ходе которой определяется какое конкретно поле структуры данных некорректно. Смещение этого поля при этом переключается на следующее его возможное значение, пропуская переборы других смещений. Во множестве случаев такая оптимизация сильно сокращает общее количество перебираемых комбинаций значений.

4. Тестирование

Проверка работоспособности реализации предлагаемого подхода интроспекции виртуальной машины проводилась с различными гостевыми ОС. В качестве таких систем использовались дистрибутивы Linux (Ubuntu, Debian, Fedora, Centos и др.) с версиями ядра от 2.4 до 6.8, Windows с версиями ядра NT от 6.1 до 10.0 и FreeBSD с версией 13.1. Тестирование во всех случаях выполнялось с эмуляцией архитектуры x86_64. Интроспекция Linux также проверена для платформы AArch64.

Результаты построения профиля интроспекции сравнивались с наборами смещений, извлекаемыми из отладочных символов. Эти наборы смещений сильно варьируются в зависимости от версии и опций компиляции ядра, и тестирование показало полное соответствие найденных значений эталонным в каждом случае. Корректность кэширования данных тестировалась на различных состояниях виртуальной машины путем автоматического сопоставления метаданных объектов в кэше с данными, читаемыми из гостевой памяти. Выполняемые в ходе тестов сценарии включали в себя запуск на гостевой ОС программ, работающих с файлами и сетью. Лог файлы, получаемые путем мониторинга объектов, проверялись вручную с использованием инструмента визуализации этих данных.

В ходе измерения производительности было установлено, что совместная работа плагинов мониторинга в среднем замедляет воспроизведение записанного на эмуляторе QEMU сценария примерно в 2,5 раза. Применение точек наблюдения за примитивами синхронизации и переменными состояния оказывает замедление на систему в пределах 5% по сравнению с подходом, где обновление кэша выполняется только по системным вызовам. При этом точки наблюдения позволили повысить точность мониторинга, т.к. с их помощью успешно отслеживаются изменения в объектах ОС, которые происходят в обход системных вызовов.

Время генерации профиля интроспекции зависит от скорости загрузки гостевой ОС на эмуляторе. В большинстве случаев все параметры определяются до момента начала авторизации пользователя. В случае некоторых особенно легковесных дистрибутивов Linux системных событий во время загрузки может не хватать на восстановление всех искомых смещений. С такими системами требуются дополнительные действия со стороны пользователя, такие как ввод логина и пароля, запуск каких-либо программ или перезагрузка ОС. Как правило, весь процесс генерации профиля для любой гостевой ОС занимает всего несколько минут.

5. Заключение

По итогам данной работы был разработан и проверен ряд новых решений для задачи интроспекции виртуальной машины. Представлен подход построения профиля интроспекции, реализованный на базе полносистемного динамического анализа. Он работает с гостевыми ОС семейств Linux, Windows и FreeBSD на архитектурах x86 и ARM. Для его функционирования не требуются отладочные символы ядра и какие-либо подготовительные работы с образом виртуальной машины со стороны пользователей.

Разработанные алгоритмы автоматической настройки также упрощают расширение функционала мониторинга за объектами гостевой системы. Реализация новых возможностей

осуществляется через добавление функции чтения нового параметра из гостевой памяти и набора тестов для его проверки.

Кэширование метаданных объектов ОС сыграло важную роль в реализации задач мониторинга, а точки наблюдения позволили более корректно и быстро отслеживать изменения в структурах данных ядра, после чего своевременно обновлять кэш.

Список литературы / References

[1]. Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. 2014. Scalability, fidelity and stealth in the DRAKVUF dynamic malware analysis system. In Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14). Association for Computing Machinery, New York, NY, USA, 386–395. DOI: 10.1145/2664243.2664252.

[2]. Jennia Hizver and Tzi-cker Chiueh. 2014. Real-time deep virtual machine introspection and its applications. In Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE '14). Association for Computing Machinery, New York, NY, USA, 3–14. DOI: 10.1145/2576195.2576196.

[3]. TEMU: The BitBlaze dynamic analysis component. Available at: http://bitblaze.cs.berkeley.edu/temu.html, accessed 07.10.2025.

[4]. Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014). Association for Computing Machinery, New York, NY, USA, 248–258. DOI: 10.1145/2610384.2610407.

[5]. B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, R. Whelan. Repeatable Reverse Engineering with PANDA. 5th Program Protection and Reverse Engineering Workshop, Los Angeles, California, December 2015.

[6]. Richard Golden, Andrew Case, and Lodovico Marziale. 2010. Dynamic Recreation of Kernel Data Structures for Live Forensics. Digital Investigation 7(2010), pp. 32–40.

[7]. Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing system-wide information flow for malware detection and analysis. In Proceedings of the 14th ACM Conference on Computer & Communications Security. pp. 116–127.

[8]. Shuhui Zhang, Xiangxu Meng, and Lianhai Wang. 2016. An adaptive approach for Linux memory analysis based on kernel code reconstruction. EURASIP Journal on Information Security 2016, 1 (2016), p. 14.

[9]. Fellicious, Christofer & Reiser, Hans & Granitzer, Michael. (2025). Bridging the Semantic Gap in Virtual Machine Introspection and Forensic Memory Analysis. Available at: 10.48550/arXiv.2503.05482, accessed 07.10.2025.

[10]. Fabian Franzen, Tobias Holl, Manuel Andreas, Julian Kirsch, and Jens Grossklags. 2022. Katana: Robust, Automated, Binary-Only Forensic Analysis of Linux Memory Snapshots. In 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus. ACM, New York, NY, USA, 18 p.

[11]. Qian Feng, Aravind Prakash, Minghua Wang, Curtis Carmony, and Heng Yin. 2016. Origen: Automatic extraction of offset-revealing instructions for cross-version memory analysis. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. pp. 11–22.

[12]. Qemu. A generic and open source machine emulator and virtualizer. Available at: https://www.qemu.org/, accessed 07.10.2025.

[13]. Shuhui Zhang, Xiangxu Meng, and Lianhai Wang. 2016. An adaptive approach for Linux memory analysis based on kernel code reconstruction. EURASIP Journal on Information Security 2016, 1 (2016), 14.

Информация об авторах / Information about authors

Владислав Михайлович СТЕПАНОВ – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Vladislav Mikhailovich STEPANOV is a software developer. Research interests: debugging, introspection and instrumentation of virtual machines, dynamic analysis of binary code, emulators.

Павел Михайлович ДОВГАЛЮК – кандидат технических наук, инженер. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, отладчики, эмуляторы.

Pavel Mikhailovich DOVGALYUK – Cand. Sci. (Tech.), engineer. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, debuggers, emulators.

Наталья Игоревна ФУРСОВА – кандидат технических наук, инженер. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, эмуляторы.

Natalia Igorevna FURSOVA – Cand. Sci. (Tech.), engineer. Research interests: virtual machines introspection and instrumentation, dynamic analysis of code, emulators.