DOI: 10.15514/ISPRAS-2021-33(6)-4

Использование идентификации потоков выполнения при решении задач полносистемного анализа бинарного кода

[1] *И.А. Васильев, ORCID: 0000-0003-3824-2753 <vasiliev@ispras.ru>*
[2] *П.М. Довгалюк, ORCID: 0000-0003-2483-5718 <pavel.dovgaluk@ispras.ru>*
[1] *М.А. Климушенкова, ORCID: 0000-0001-6737-9092 <maria.klimushenkova@ispras.ru>*

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

Аннотация. При полносистемном анализе бинарного кода зачастую применяется динамический бинарный анализ, при котором предоставляемый аналитику объем данных представлен потоком выполняемых инструкций и содержимым оперативной памяти и регистров. Для обработки таких данных требуется глубокое понимание особенностей исследуемой системы, при этом трудозатраты на выполнение анализа и требования к технической осведомленности пользователя становятся очень велики. Для упрощения процесса анализа необходимо привести входные данные к более дружелюбному для пользователя виду, т.е. предоставить высокоуровневую информацию об исследуемом программном обеспечении. Такой высокоуровневой информацией является информация о потоке выполнения программы. Для восстановления потока выполнения программы важно иметь представление о вызываемых ею процедурах. Получить такое представление можно с помощью стека вызова функций для конкретного потока. Построение стека вызовов без информации о выполняемых потоках невозможно, т.к. каждому потоку однозначно соответствует один стек и vice versa. Помимо этого, само наличие информации о потоках повышает уровень знаний о системе, позволяет более тонко профилировать объект исследования и проводить узконаправленный анализ, применяя принципы выборочного инструментирования. Виртуальная машина не предоставляет напрямую такой информации, и строить предположения о работе исследуемой системы приходится, основываясь на доступных низкоуровневых данных (поток выполняемых виртуальным процессором инструкций и оперативная память виртуальной машины). Таким образом, существует необходимость в разработке метода для автоматической идентификации потоков в исследуемой системе, опирающегося на имеющемся объеме данных. В данной работе рассматриваются существующие подходы к реализации получения высокоуровневой информации при полносистемном анализе и предлагается метод для восстановления данных о потоках в условиях полносистемной эмуляции с низкой степенью ОС-зависимости. Также приводятся примеры практического использования данного метода при реализации инструментов анализа, а именно: восстановление стека вызовов, обнаружение подозрительных операций возврата и обнаружение обращений к освобожденной памяти в стеке. Приведенное в статье тестирование показывает, что накладываемое описанными алгоритмами замедление позволяет проводить работу с исследуемой системой, а сравнение с эталонными данными подтверждает корректность получаемых алгоритмами результатов.

Ключевые слова: полносистемный анализ; стек вызовов; обнаружение уязвимостей

Для цитирования: Васильев И.А., Довгалюк П.М., Климушенкова М.А. Использование идентификации потоков выполнения при решении задач полносистемного анализа бинарного кода. Труды ИСП РАН, том 33, вып. 6, 2021 г., стр. 51-66. DOI: 10.15514/ISPRAS–2021–33(6)–4

Using the identification of threads of execution when solving problems of full-system analysis of binary code

[1] *I.A. Vasiliev, ORCID: 0000-0003-3824-2753 <vasiliev@ispras.ru>*
[2] *P.M. Dovgalyuk, ORCID: 0000-0003-2483-5718 <pavel.dovgaluk@ispras.ru>*
[1] *M.A. Klimushenkova, ORCID: 0000-0001-6737-9092 <maria.klimushenkova@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia*
[2] *Yaroslav-the-Wise Novgorod State University, 41, B. Sankt-Peterburgskaya st., Novgorod, 173003, Russia*

Abstract. Dynamic binary analysis, that is often used for full-system analysis, provides the analyst with a sequence of executed instructions and the content of RAM and system registers. This data is hard to process, as it is low-level and demands a deep understanding of studied system and a high-skileed professional to perform the analysis. To simplify the analysis process, it is necessary to bring the input data to a more user-friendly form, i.e. provide high-level information about the system. Such high-level information would be the program execution flow. To recover the flow of execution of a program, it is important to have an understanding of the procedures being called in it. You can get such a representation using the function call stack for a specific thread. Building a call stack without information about the running threads is impossible, since each thread is uniquely associated with one stack, and vice versa. In addition, the very presence of information about flows increases the level of knowledge about the system, allows you to more subtly profile the object of research and conduct a highly focused analysis, applying the principles of selective instrumentation. The virtual machine only provides low-level data, thus, there is a need to develop a method for automatic identification of threads in the system under study, based on the available data. In this paper, the existing approaches to the implementation of obtaining high-level information in full-system analysis are considered and a method is proposed for recovering thread info during full-system emulation with a low degree of OS-dependency. Examples of practical use of this method in the implementation of analysis tools are also given, namely: restoring the call stack, detecting suspicious return operations, and detecting calls to freed memory in the stack. The testing presented in the article shows that the slowdown imposed by the described algorithms allows working with the system under study, and comparison with the reference data confirms the correctness of the results obtained by the algorithms.

Keywords: full-system instrumentation, call stack, vulnerabilities detection.

For citation: Vasiliev I.A., Dovgalyuk P.M., Klimushenkova M.A. Using the identification of threads of execution when solving problems of full-system analysis of binary code. Trudy ISP RAN/Proc. ISP RAS, vol. 33, issue 6, 2021, pp. 51-66 (in Russian). DOI: 10.15514/ISPRAS–2021–33(6)–4

1. Введение

Можно выделить принципиально разные подходы при получении данных о потоках, которые зависят от уровня проведения динамического бинарного анализа: уровень приложений и полносистемный уровень. Различные системы анализа уровня приложений (Pin [1], Valgrind [2]) легко получают доступ к информации об исследуемом процессе и потоках, так как выполняются в одной операционной системе с исследуемым приложением и могут, используя системные API, получить интересующую информацию от ОС. Но в этом случае инструментарий анализа делит одно адресное пространство с исследуемым приложением, а значит может быть обнаружен и скомпрометирован. По этой причине данный подход нельзя использовать при решении вопросов обеспечения безопасности.

При полносистемном анализе прямого доступа к высокоуровневой информации получить нельзя, поэтому различные инструменты предлагают решение этой проблемы. Например, Virtuoso [3] использует для реализации программу-агент, которая собирает интересующие данные, и впоследствии создает алгоритм на основании полученной трассы выполнения этой программы. DECAF [4,5] при анализе основывается на полях данных из структур ядра исследуемой операционной системы. PEMU [6] не использует ни программы-агенты, ни

структуры ядра, однако использует примитивные алгоритмы для определения высокоуровневой информации, дающие приблизительный результат и строго-ориентированные под архитектуру x86.

Описанным подходам свойственны недостатки, ограничивающие возможность их применения: невозможность использования в системах с закрытым исходным кодом, невозможность использования во встроенных системах, не поддерживающих загрузку новых программ, низкое качество получаемых данных, строгая направленность на определенные ОС.

Таким образом, для обеспечения универсальности и достаточной степени доверия к результатам работы конечного алгоритма, необходимо в первую очередь использовать методы анализа, функционирующие вне исследуемой системы, т.е. на уровне виртуальной машины, а также опираться на данные, являющиеся ОС-независимыми или находящиеся в прямом доступе.

Отсюда возникают требования, предъявляемые к методу идентификации потоков в ОС: во-первых, необходимо избежать загрузки программ-агентов в гостевую ОС, а значит весь функционал, обеспечивающий идентификацию потоков, должен располагаться на уровне виртуальной машины. Во-вторых, необходимо обеспечить универсальность метода, т.е. опираться либо на ОС-независимые данные, либо на данные из открытых официальных источников. В-третьих, метод должен быть реализуем для различных процессорных архитектур.

При реализации метода восстановления стека вызовов функций необходимо придерживаться описанных выше принципов. При этом также важно не опираться в реализации на соглашение о вызовах, так как оно не всегда содержит необходимую информацию об указателе фрейма, а значит не может универсально использоваться при восстановлении стека вызовов.

Соблюдение этих требований позволит получить метод, применимый как для открытых, так и для закрытых систем. При этом такой метод можно будет с минимальными изменениями портировать на ОС одного семейства и адаптировать для работы на новых архитектурах.

В последующих главах будет описан разработанный метод идентификации потоков в операционной системе на уровне виртуальной машины, без встраивания программ-агентов в исследуемую систему и без использования данных из структур ядра ОС. Идентификация потоков подразумевает установку надежного разделения между различными потоками с возможностью последующего использования этой информации для реализации алгоритмов анализа. Полученный метод переносим и адаптируем для новых конфигураций исследуемых систем.

Также в данной статье уделяется внимание практическому применению разработанного алгоритма для решения задач анализа, например, будет предложен метод для восстановления стека вызовов, не опирающийся на данные соглашения о вызовах. Также будут приведены примеры использования полученных данных для вопросов решения безопасности использования ПО (алгоритм обнаружения подозрительных возвратов из процедур и алгоритм обнаружения использования освобожденной памяти в стеке).

2. Существующие реализации восстановления данных

Среди существующих подходов нам будут интересны те, в которых решается задача идентификации потоков в ОС, и данная информация используется для дальнейшей реализации алгоритмов анализа.

Рассмотрим следующие показатели предлагаемых подходов:

- ОС-зависимости алгоритма;
- использование программ-агентов, загружаемых в систему;

- возможность обнаружения со стороны исследуемой системы;
- возможность использования в отсутствии исходного кода исследуемой системы.

Системы, реализующие анализ уровня процессов (такие как: Pin [1], Valgrind [2], DynamoRIO [7]), как правило представляют из себя виртуальные машины, внутри которых запускается рассматриваемое приложение. Им свойственен прямой доступ к любой интересующей информации о рассматриваемом процессе – легкость доступа к данным и их полнота обеспечивается за счет выполнения инструментария анализа в одной операционной системе с исследуемым приложением, а значит и наличии доступа к системному API. Однако в этом случае невозможно анализировать межпроцессное взаимодействие, анализировать ядро ОС. При этом инструменты характеризуются строгой ОС зависимостью и не обеспечивают необходимой в вопросах безопасности изоляции инструмента и предмета анализа.

Среди систем, реализующих полносистемный анализ, рассмотрим Virtuoso [3], TEMU [8], DECAF [4,5], PANDA [10], PyREBox [11], PEMU [6], IceBox [12].

Создатели инструмента Virtuoso [3] реализуют получение информации о системе извне, на уровне виртуальной машины, в которой запущена гостевая система. Однако для определения способа извлечения интересующей информации в данном инструменте применяется внедряемая программа-агент. Программа-агент определяет адреса интересующих данных в системе, что позволяет в последующем использовать их при выполнении анализа. Данный подход не может применяться при работе с самомодифицирующимся кодом, при работе с системами, реализующими ASLR, а само использования программ-агентов ограничивает область применения подхода до открытых систем.

TEMU [8] реализует динамическое бинарное полносистемное инструментирование на базе эмулятора QEMU [9]. Определенные события в гостевой системе вызывают функции обработки в подключаемых модулях, которые и выполняют анализ данных. Для получения информации о семантике уровня операционной системы используется “семантический извлекатель” - модуль, ответственный за извлечение информации о процессах, модулях, потоках, символьной информации и контексте выполнения. Данный модуль поддерживает операционные системы Windows XP, Windows 2000 и Linux, причем реализация получения данных из ОС для них отличается. Для Linux семантический извлекатель обращается напрямую к структурам данных ядра по предзаданным адресам. Для Windows используется специальный модуль ядра – он загружается в гостевую систему и выполняет функцию отслеживания специфических событий (запуск процессов, потоков, загрузка модулей) и передачи этих данных в саму инструментальную систему. Таким образом TEMU использует как строго ОС-специфичные данные, так и загрузку модуля в исследуемую систему, что является значительным минусом для инструмента.

DECAF [4,5] также является фреймворком динамического бинарного полносистемного инструментирования, причем его создатели при разработке опирались на опыт TEMU. Им удалось избавиться от необходимости использования загружаемого модуля, однако вся информация о процессах, потоках, модулях основывается на содержимом структур ядра исследуемой системы, а значит подход является строго ОС зависимым и для каждой итерации ОС и даже версии ядра будет требовать применения приемов обратной инженерии.

Схожий подход используют и фреймворки PANDA [10] и PyREBox [11]. Процесс анализа в обоих случаях базируется на конфигурационном файле, строго специфичном для ОС, который в случае с PANDA генерируется загружаемой в систему программой-агентом, а для PyREBox используются готовые конфигурационные файлы, заимствованные из Volatility [13]. При этом в результате для получения информации используется считывание данных из структур ядра исследуемой системы.

PANDA при этом предлагает также и способы условно ОС-независимые, но они дают результаты низкой точности: так, например, определение потоков по ASID дает ложные результаты, если одному процессу соответствует несколько потоков, а эвристический метод

основан на определении смены потока по резкой смене указателя стека, что не всегда соответствует действительности.

PEMU [6] является полносистемным продолжателем идей Pin – предоставляет тот же API, но расширяет доступную для исследования область применения. При этом разработчики предоставляют возможность получения данных о процессах и потоках из системы, т.к. без этой информации значительная часть API функций, переносимых из Pin не смогла бы функционировать. Однако решение строго привязано к специфике процессора x86 и для потоков является только ориентировочным: поток идентифицируется по значению регистра esp с наложенной обнуляющей маской на нижние 12 бит.

IceBox [12] – инструмент для интроспекции виртуальной машины в VirtualBox, позволяющий трассировать и отлаживать пользовательские процессы и процессы ядра в исследуемой системе. Основывается на Winbagility [14] от которой и наследует недостатки – строгая зависимость от инструментария WinDBG, возможность работы только с ОС семейства Windows и использование для анализа данных из структур ядра исследуемой системы.

Таким образом, были определены основные недостатки рассмотренных в обзоре инструментов.

- Использование данных структур ядра. Для систем с закрытым исходным кодом получение данной информации сопряжено или со сложным анализом системы, или вовсе невозможно. Также данное решение является строго ОС-специфичным, а значит требует генерации новой конфигурации для каждой из исследуемых систем, вплоть до версий и сервис паков.
- Использование программ-агентов. Многие из рассмотренных инструментов используют программы-агенты для получения данных из системы напрямую, либо для генерации файлов-конфигураций, используемых при последующем анализе. Использование таких внедряемых программ невозможно для неизменяемых образов исследуемых систем. В случае, если внедрение возможно, исследуемая система может изменить свое поведение. Кроме того, генерация файлов конфигураций будет бесполезна в случае, если расположение структур данных определяется случайным образом при каждой загрузке системы.
- Сложность портирования и сопровождения в связи с ориентированностью решения на определенную операционную систему.

3. Предлагаемый метод восстановления данных

Предлагаемый нами подход для восстановления информации о потоках лишен недостатков, обнаруженных при рассмотрении других инструментов, и позволяет получать информацию при полносистемном динамическом анализе, не используя программы-агенты. Принципы, на которых построен разработанный метод, также обеспечивают простоту дальнейшей поддержки и адаптации для новых систем.

Суть предлагаемого метода идентификации потоков заключается в следующем. Известно, что в системе выполняются процессы, и каждому процессу соответствует один или более потоков [15]. Для организации виртуальной памяти система выделяет для каждого процесса новое значение записи в таблице трансляции виртуальных адресов в физические. Процессор выделяет отдельный регистр для хранения этого значения (для x86 это CR3, для ARM - CP15 и т.п.). Таким образом, наблюдая за значением этого регистра, можно однозначно отличать друг от друга запущенные процессы. В рамках процесса может выполняться несколько потоков, и каждый из потоков использует свои локальные переменные и оперирует своими адресами возврата.

Для организации работы потоков система выделяет каждому потоку отдельный стек. Значение верхушки стека (SP, stack pointer) хранится в определенном регистре процессора

(x86 - ESP, ARM - R13). Таким образом, идентифицировать поток можно по процессу, к которому он принадлежит, и по выделенному ему стеку. Однако, определение выделенного стека является нетривиальной задачей, в отличие от определения процесса. Для идентификации процесса достаточно отслеживать состояние регистра, содержащего значение записи в таблице трансляции. Изменения же в регистре, содержащем значение вершины стека, не всегда говорят о смене потока, так как стек постоянно изменяется и при нормальной работе потока: на него помещаются значения переменных, значения адресов возврата, он постоянно расширяется и сужается. При этом очевидно, что раз у каждого потока стек должен быть свой, то и значение регистра с вершиной стека для каждого потока будет изменяться только в рамках определенного диапазона и не будет пересекаться между разными потоками.

Отсюда следует вывод, что для однозначной идентификации потока необходимо и достаточно рассматривать пару “идентификатор процесса” и “диапазон значений стека”. На рис. 1 визуально отображено соотношение идентификатора потока и системной информации.

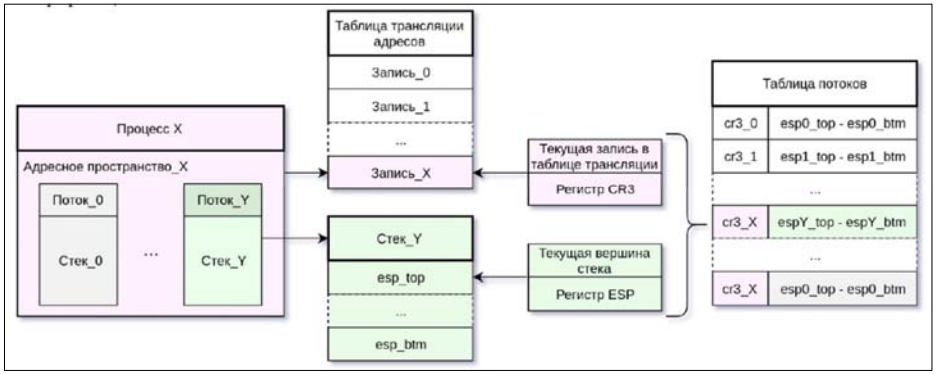

Рис. 1. Визуализация предлагаемого метода идентификации потоков
Fig. 1. Visualisation of proposed method of threads identification

Поскольку задача определения процесса является тривиальной, сложность метода сводится к задаче определения диапазона значений стека – как понять, относится ли очередное значение SP к тому же потоку или созданному новому? Предлагаемый метод основывается на предположении, что в системе должен присутствовать определенный набор событий, предшествующих операции создания нового стека и связанного с ним потока. Другие же события, связанные с изменением стека, служат для расширения границ диапазонов уже обнаруженных стеков.

Реализация разработанного метода строится на базе эмулятора с открытым исходным кодом QEMU [9]. Для внесения дополнительного функционала эмулятор поддерживает систему плагинов – динамических загружаемых библиотек, написанных на языке C. При этом источником данных для обработки плагином может являться как эмулятор, так и другие плагины. Это позволяет организовать комплексные алгоритмы анализа, где одни плагины опираются на данные, восстановленные другими плагинами.

Типичный процессор имеет ряд инструкций явно и косвенно взаимодействующих со стеком, а значит изменяющих указатель на его вершину. Например, к явно изменяющим относятся инструкции, использующие SP как операнд, к косвенно – выполняющие функционал push и pop (помещение значения на стек, и получение значения с вершины стека), инструкции вызова и возврата из функций.

Анализируя выполняющиеся при работе исследуемой системы инструкции и изменение значения SP, можно выделить подмножество инструкций и событий, появление которых свидетельствует о создании нового потока, и подмножество инструкций, расширяющих

существующий диапазон. Данные подмножества являются архитектурно-зависимыми, однако, как было упомянуто выше, имеют схожий вид для различных процессорных архитектур. При этом такой способ определения потоков является ОС-независимым. Применение данного метода обеспечивает легкость переноса на новые архитектуры и новые операционные системы.

Нами был разработан плагин, реализующий разбиение значений стека на диапазоны и идентификацию потоков. Логика работы разработанного плагина отображена на рис. 2 и заключается в следующем: при трансляции инструкции QEMU передает сигнал в плагин, где по опкоду определяется её тип и принадлежность к группе выделенных событий. При возникновении события, потенциально расширяющего текущий диапазон, происходит сравнение нового значения SP с границами текущего диапазона, и при необходимости его границы расширяются соответствующим значением. При возникновении события, потенциально создающего новый диапазон, происходит проверка вхождения нового значения вершины стека в один из существующих диапазонов. Если ни к одному из существующих диапазонов новое значение не принадлежит, то это свидетельствует о создании нового диапазона с полученным значением SP. Так как возможно ложное создание нового диапазона, то также предусмотрен механизм объединения, если в процессе расширения диапазона произошло наложение на соседний. В таком случае, один из диапазонов удаляется, а второй расширяется до объединенных границ. При этом происходит генерация сигнала об объединении, сопровождаемого идентификаторами потоков, чтобы позволить опирающимся на данные о потоках алгоритмам анализа провести соответствующее обновление данных.

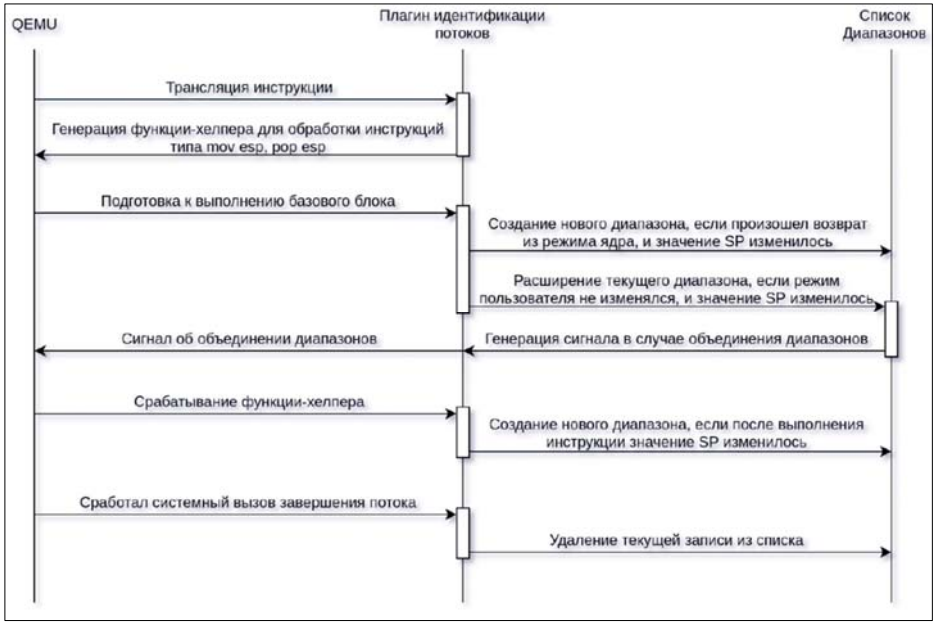

Рис. 2. Логика работы плагина идентификации потоков
Fig. 2. How the thread identification plugin works

Для плагина идентификации потоков были определены следующие условия для создания и расширения диапазонов (подробнее процесс определения условий описан в [16]). Для потенциального создания нового диапазона в i386 и x86_64: возникновение инструкции pop

esp, mov esp и возвращение из режима ядра в пользовательский режим, при этом новый диапазон будет создан, только если значение SP не входит в один из существующих. Отдельно стоит заметить, что при работе с ОС семейства Windows приходится также учитывать существование системного вызова NtContinue, который изменяет значение SP в режиме ядра, однако данное событие соответствует или расширению текущего потока, или переключению в другой существующий.

Для создания нового диапазона в mips64: возникновение инструкций move sp, lw sp, lui sp, ld sp, add sp, или инструкций addu sp, addiu sp, sub sp, subu sp, где один из аргументов при выполнении арифметической операции является нулем, или инструкций типа uld и uwd.

Описанный в [16] алгоритм был технически доработан для ускорения работы. В первую очередь использование тяжеловесного capstone было заменено на самописный дизассемблер под конкретные задачи для интересующих архитектур. Также хранение диапазонов значений указателя стека было оптимизировано. Ранее диапазоны объединялись в случае пересечения только при запросе данных у плагина. Так как для работы других плагинов анализа эта информация запрашивается с высокой периодичностью, то возникла потребность объединять диапазоны сразу в момент расширения.

Для упрощения этого процесса хранение диапазонов было приведено к упорядоченному списку, где в случае расширения и наложения диапазонов достаточно выполнить объединение с соседними, и не проверять остальные диапазоны. Дополнительно был изменен и момент обработки событий – ранее обработка выполнялась сразу после возникновения выделенных инструкций, теперь же значение SP проверяется только перед выполнением очередного блока, но с учетом произошедших в прошлом блоке событий. Суммарно введенные усовершенствования изменили время, проводимое в обработчике событий в плагине, с 20% до 0,5% от всего времени выполнения эмулятора (замер производился с помощью perf).

Отдельно стоит отметить, что для потоков существует необходимость в отдельной обработке их завершения (что также отображено на рис. 2). В общем случае, каждый поток однозначно идентифицируются описанной выше парой значений: “идентификатор процесса” и “диапазон значений стека”. Этих данных достаточно, чтобы отличать потоки между собой и реализовывать на основании этой информации дальнейшие алгоритмы анализа и инструментирования. Однако после завершения работы потока существует вероятность, что при создании нового потока система использует ту же область памяти, что и для завершенного потока. Для некоторых алгоритмов различие этих потоков будут критичны (тонкие алгоритмы профилирования, отладка потоков). В таком случае требуется ввести обработку системных вызовов.

Известно, что при завершении потока ОС генерирует соответствующий системный вызов, и определить это событие можно отслеживая поток выполняемых инструкций и значения регистров. Обнаружив соответствующий системный вызов, текущий диапазон необходимо отметить “завершенным” и оповестить об этом все зависимые алгоритмы анализа. Информация о системных вызовах является открытой и широко доступной для большинства операционных систем. Таким образом, если возникает необходимость в использовании системных вызовов, это добавляет методу фактор ОС-зависимости, однако позволяет сохранить простоту портирования за счёт открытости рассматриваемых данных.

4. Практическое применение восстановленных данных

4.1 Стек вызовов

Имея данные о потоках, восстановленные описанным выше плагином, можно приступить к реализации механизмов для анализа исследуемой системы. Одним из широко применяемых при анализе и отладке инструментов является стек вызовов. В данном случае речь ведется о

восстановлении стека вызовов в условиях полносистемного бинарного анализа, а значит алгоритм должен быть ОС-независимым и также работать как для функций пользовательского уровня, так и для уровня ядра. При реализации алгоритма важно не опираться на соглашение о вызовах, т.к. информация об указателе фрейма не всегда описана в соглашении, а значит и восстановить стек вызова с его помощью в этом случае не удастся.

Логика работы алгоритма восстановления стека вызовов заключается в следующем: из выполняемых в гостевой машине инструкций выделяются те, что относятся к вызову функции, и те, что свидетельствуют об операции возврата. Для каждого из потоков сохраняется свой список адресов, по которым произошел вызов, а при возникновении возврата соответствующая запись из списка вызовов удаляется.

Изначально в качестве анализируемой данным плагином единицы выступал блок трансляции. На этапе трансляции в блоке проверялась последняя инструкция, и блок помечался, если это была инструкция типа “вызов” или “возврат”. На этапе выполнения после блоков “вызов” в стек вызовов заносился адрес последней инструкции и ожидаемый адрес возврата. Перед выполнением блока, идущего за блоком “возврат” происходила проверка соответствия адреса первой инструкции в блоке верхнему адресу возврата в текущем стеке вызовов, и происходило удаление записи, в случае совпадения.

Однако из-за присутствующей в QEMU оптимизации, при которой идущие подряд блоки объединяются в цепь для последовательного связанного выполнения, событие “после выполнения блока” вызывается лишь для последнего блока в цепи. По этой причине логика плагина была изменена таким образом, что ключевыми событиями при анализе стали “трансляция инструкции” для обнаружения интересующих вызовов и возвратов, и “перед выполнением блока” для уточнения дополнительных данных о вызове и удаления записей после успешного возврата.

В результате алгоритм принял следующий вид: на этапе трансляции плагин получает сигнал от QEMU с информацией о транслируемой инструкции и определяет её тип по опкоду с помощью специального разборщика. Выделяются инструкции типа call и ret. Если была обнаружена одна из таких инструкций, то для нее создается специфическая функция-хелпер – call_helper и ret_helper. Данные функции-хелперы вызываются на этапе выполнения кода перед соответствующей инструкцией. Вызов перед выполнением, а не после является вынужденной мерой, так как особенности эмулятора QEMU не позволяют размещать хелперы после крайней инструкции в блоке трансляции, каковыми чаще всего и являются call и ret.

В call_helper происходит запрос текущего идентификатора из плагина идентификации потоков, и по данному идентификатору в таблице стеков вызовов обновляется соответствующий стек – добавляется новая запись с адресом вызова функции равным адресу инструкции call и соответствующим адресом на системном стеке, где будет сохранено значение адреса возврата после выполнения этой функции (хоть сам адрес возврата на этом этапе еще не был записан в стек, его расположение в стеке можно однозначно предсказать уже на этом этапе). Также происходит переключение глобальной переменной was_call, информация о чем будет использоваться на этапе обработки выполнения следующего блока. Для ret_helper происходит переключение глобальной переменной was_ret.

На этапе выполнения также происходит обработка сигнала от QEMU “перед выполнением блока трансляции”. Если была установлена отметка о произошедшем в конце предыдущего блока call, то на данном этапе к соответствующей записи в стеке вызовов происходит добавление данных об адресе вызванной процедуры (адрес первой инструкции текущего блока) и адрес возврата из текущей функции, извлекаемый из стека. Затем происходит происходит проверка на наличие устаревших записей и их последующее удаление. Такие записи возникают из-за того, что библиотеки операционной системы зачастую пользуются нестандартными способами оперирования со стеком и процессом возврата из функции.

Например, адрес возврата может быть перезаписан на предустановленное значение, или может произойти мнимый “возврат” без фактического использования инструкций ret. В этом случае на вершине стека могут оказаться записи, возврат по которым уже произошел, но не был пойман алгоритмом, которые необходимо удалить. Проверка на то, что запись является неактуальной, выполняется следующим образом: в начале блока считывается текущее значение указателя на вершину стека. Затем, данное значение сверяется с записанными на этапе вызова адресами в стеке, содержащими адреса возврата из функций, так как для их использования они все еще должны быть доступны. Если текущая вершина стека располагается выше (для растущих вниз стеков) или ниже (для растущих вверх стеков), чем адрес возврата для текущей записи, то эта запись удаляется с вершины стека вызовов, так как этот адрес возврата.

Дальнейшая обработка происходит только при установленном переключателе was_ret, так как выполняет функции обработки возврата и приведения стека вызовов в актуальное состояние. На этом этапе происходит сравнение текущего адреса базового блока с адресами возврата, записанными в стеке вызовов. При этом важно учитывать, что возврат может осуществляться через несколько записей в стеке вызовов, эмпирическим путём было получено значение в 10 верхних записей, которые достаточно проверять чтобы покрыть такой случай. Если адрес блока совпал с адресом возврата, то эта и все вышестоящие записи из стека вызовов удаляются.

Отдельная ситуация, которая нуждается в специфической обработке – инструкция типа jmp иногда может выступать в качестве call. Определить это можно по началу следующего блока трансляции. Если он принимает классическую форму пролога функции с записью фрейма стека в стек, то это свидетельствует о произошедшем вызове. Чтобы учесть данную ситуацию, на этапе трансляции помимо инструкций типа call и ret необходимо также устанавливать и переключатель was_jmp для jmp инструкций. На это выполнения блока при установленном переключателе was_jmp происходит проверка первых инструкций данного блока на ожидаемую последовательность, и в случае совпадения происходит добавление соответствующей записи в стек вызовов.

Также необходимо учитывать, что идентификатор стека устанавливается в соответствии с информацией, полученной от плагина идентификации потоков. Как было упомянуто в предыдущем разделе, работа идентификатора потоков подразумевает, что диапазоны после создания могут объединяться, а значит это может привести к созданию нескольких стеков вызовов вместо одного в рассматриваемом плагине. Для того, чтобы избежать этой ситуации, плагин отслеживает возникновение сигнала “объединение диапазонов”, сопровождаемого идентификаторами диапазонов, что позволяет объединить соответствующие стеки вызовов в соответствие с направлением роста стека в данной системе.

4.2 Обнаружение подозрительных операций возврата

Также, на основе выше описанных алгоритмов был разработан и реализован метод обнаружения подозрительных операций возврата. Одним из используемых злоумышленниками способов повлиять на выполнение программы является передача управления на вредоносный код. Рассмотрим пример, когда функция после окончания выполнения совершает возврат, но попадает в заданный злоумышленником участок кода. Происходить это может по причине того, что значение адреса возврата в стеке было перезаписано с использованием ошибок в программе. Ключевым механизмом для реализации уязвимостей такого рода является использование переполнения буфера для подмены реального адреса возврата на ссылающийся на зловредный код (рис. 3). Такие уязвимости эксплуатируются как ROP-цепочками (цепочка "гаджетов" в коде, традиционно из нескольких инструкций, где каждый гаджет завершается инструкцией ret), так и используются для загрузки готового шеллкода в память для непосредственного выполнения

этих инструкций по нужному адресу. В любом случае, первым шагом в эксплуатации такой уязвимости является подмена адреса возврата, и именно на обнаружение такой манипуляции и направлен реализованный механизм.

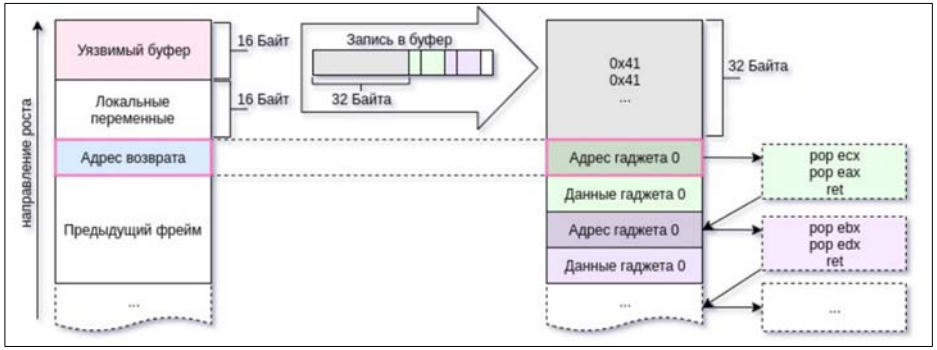

Рис. 3. Пример эксплуатации уязвимого буфера с помощью ROP-цепочки
Fig. 3. An example of exploitation of a vulnerable buffer with a help of a ROP-chain

При работе механизма пользователь получает оповещение каждый раз, когда после возврата из процедуры выполнение программы переходит не к ожидаемому адресу возврата, т.е. происходят так называемые “подозрительные операции возврата”. Функционал данного плагина строится на информации, получаемой из плагина восстановления стека вызовов. После каждой инструкции типа ret происходит генерация соответствующего сигнала, что позволяет описываемому плагину запросить информацию о текущем стеке вызовов и сравнить адрес вызванного базового блока с верхними записями в стеке вызовов.

Если совпадения обнаружено не будет, это является свидетельством неожиданного поведения программы и перехода в подозрительную часть кода. Это может может являться признаком зловредного характера происходящих действий в программе, поэтому в данной ситуации плагин выводит для пользователя информацию об адресе выполненной инструкции ret, ожидаемом адресе возврата и фактическом адресе возврата. Обладая этой информацией, пользователь может проверить вызванный код, чтобы определить характер и потенциальную опасность происходящей операции.

Так как функционал алгоритма построен на принципах полносистемного бинарного анализа, это позволяет выполнять обнаружение зловредного воздействия даже на уровне ядра, что недоступно широко используемым программам для автоматического анализа пользовательских приложений (таким как Valgrind). Однако во время анализа инструкций, относящихся к выполнению системных библиотек, было обнаружено, что ОС может намеренно использовать модификацию адреса возврата в стеке для прямого вмешательства в поток выполнения кода и перехода на некоторые утилитарные функции.

С точки зрения разработанного механизма данные переходы также будут выглядеть как “подозрительные” возвраты. По этой причине сообщения, предоставляемые плагином аналитику, должны использоваться не для совершения выводов о характере поведения программы, а в качестве указателя на участки кода, которые пользователь сможет проанализировать и в дальнейшем самостоятельно дать им характеристику.

Отдельно учитывается такая ситуация, как использование инструкции ret в качестве инструкции jmp – программа помещает в стек значение адреса для последующего перехода, затем вызывает ret и таким образом осуществляет безусловный переход, при этом фактически использование ret в данном случае не является операцией возврата. Для того, чтобы подобный механизм не вызывал срабатывание алгоритма плагина, была добавлена дополнительная проверка – при выполнении инструкции вызова запоминается адрес в стеке,

по которому записан адрес возврата. При последующем выполнении инструкции ret проверяется адрес в стеке, из которого было извлечено значение для возврата. Если этот адрес отличается от записанного при вызове, то это является свидетельством использования инструкции ret в качестве jmp.

Таким образом подобные инструкции не учитываются алгоритмом и удается избежать ложного оповещения аналитика. Данный механизм был включен в плагин восстановления стека вызовов, что позволяет поддерживать корректность восстановленных данных и генерировать сигнал о событии (инструкции возврата) только для фактических операций возврата.

В данном плагине поддерживается возможность указания идентификатора потока, для анализа только конкретной интересующей программы, а не всего процесса работы системы. Это может быть полезно для сепарации системных библиотек, содержащих корректные “подозрительные” возвраты, от интересующих аналитика программ. Такой подход позволяет минимизировать замедление работы исследуемой системы и упростить последующий процесс анализа за счет первичного отсеивания полезных данных.

4.3 Обнаружение обращений к освобожденной памяти

Также, базируясь на восстановленных данных о потоках, был реализован плагин, который отслеживает обращения к памяти в стеке за пределами текущих границ стека (частный случай use after free). Алгоритм функционирует следующим образом: QEMU генерирует сигналы о работе с памятью, и на каждый сигнал, соответствующий загрузке из памяти в данном плагине назначается функция-обработчик. В данной функции происходит запрос информации о текущем потоке от плагина восстановления данных о потоках. Из полученной информации интерес представляют границы возможных значений SP для выполняющегося потока. Также происходит запрос значения регистра, содержащего текущее значение SP, от QEMU.

Адрес чтения из памяти сравнивается с полученными значениями. Если данный адрес располагается в сторону роста стека, относительно значения вершины стека, но при этом попадает в диапазон адресов, принимаемых SP, для текущего потока, то можно сделать однозначный вывод о попытке загрузки информации из адресов, потерявших актуальность. Сообщение о такой загрузке будет выведено в лог и представлено пользователю. При такой загрузке есть большая вероятность получить непредсказуемое поведение программы, что может как быть источником трудно отлаживаемых ошибок, так и уязвимым местом для зловредного воздействия на программу.

Данный алгоритм не всегда применим для системных библиотек, так как при их работе было может встретиться преднамеренное использование данных из стека, лежащих выше текущей вершины стека. Этот подход нельзя назвать хорошей практикой, однако он не является редкостью и неоднократно встретился при анализе работы различных ОС. Авторы ПО в данном случае предполагают, что они имеют полный контроль над стеком, знают его содержимое в любой момент и обращение такого характера к данным не может вызвать непредвиденных ошибок. Однако, даже несмотря на подобные “ложные” срабатывания, результат работы плагина (т.е. лог обращений к адресам за пределами стека) может использоваться для анализа работы программ, как элемент отладки, а также для улучшения надежности разрабатываемого ПО.

В результате был реализован ряд связанных взаимодействующих плагинов (рис. 4), где каждый из последующих плагинов применяет данные, полученные в результате работы предыдущего.

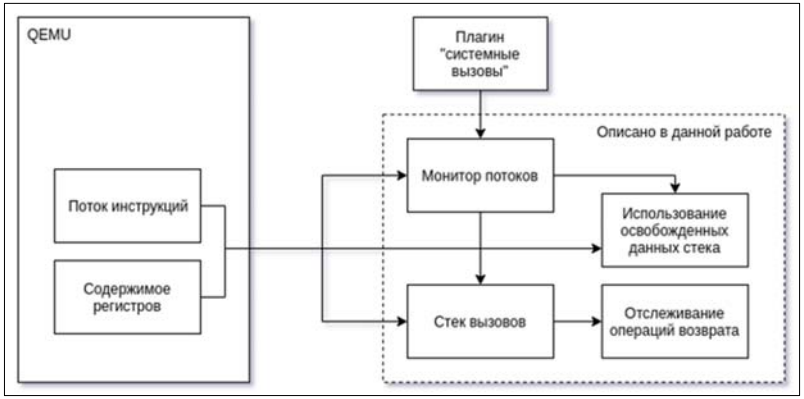

Рис. 4. Схема взаимосвязи реализованных плагинов и эмулятора
Fig. 4. The diagram of the relationship between the implemented plugins and the emulator

5. Оценка замедления и корректности

Для проведения сравнения производительности и функциональности в данной главе рассматриваются следующие комбинации архитектур и операционных систем: i386 и Windows XP, x86_64 и Windows 7, mips64 и операционная система реального времени (ОС РВ).

Для оценки накладываемого замедления были рассмотрены следующие сценарии:

- QEMU без подключения разработанных плагинов;
- предыдущий сценарий с дополнительно подключенным плагином для идентификации потоков thread_monitor;
- предыдущий сценарий с дополнительно подключенным плагином для восстановления стека вызовов call_stack.

При этом во время тестирования в системах выполнялись следующие операции:

- для i386 + Windows XP рассматривалась загрузка операционной системы, запуск и выполнение программы Solitaire.exe;
- для x86_64 + Windows 7 рассматривалась загрузка операционной системы, запуск и выполнение программы Paint.exe;
- для Mips64 + ОС РВ рассматривалась загрузка операционной системы, и многократное выполнение операций show processes и stacktrace.

Для каждого сценария проводилось не менее пяти замеров времени выполнения, после чего крайние значения были отброшены, а по оставшимся было вычислено среднее время выполнения сценария и полученное замедление.

Измерение времени производилось с помощью встроенной в Ubuntu утилиты time, при этом для достижения более точного результата замер времени производился не при непосредственной работе системы, а при воспроизведении записанного журнала выполнения. Таким образом каждый из запусков сценарий был полностью идентичен, что позволят говорить о корректном сравнении времени выполнения. Результаты исследования приведены в табл. 1. Наибольшее замедление во всех сценариях соответствует использованию плагина по восстановлению данных стека вызовов, что оправданно, так как данный случай помимо собственных вычислений включает и два предыдущих сценария. При этом самое большое среднее замедление было показано в исследуемой системе x86_64 + Windows 7 (55%). Данное замедление является ощутимым, однако оно не мешает работе с системой, комфортному

использованию большинства приложений и позволяет осуществлять практически любые необходимые при анализе операции. Также можно заметить, что данное замедление является относительно небольшим, в сравнении с аналогичными фреймворками анализа, базирующимися на инструментировании. Например, авторы PEMU в своей работе [6] приводят следующие данные по замедлению, накладываемому различными плагинами: минимальное в 149%, максимальное в 922%, со средним значением в 433%.

Табл. 1. Замедление, оказываемое разработанными плагинами на работу QEMU
Table 1. Slowdown caused by implemented plugins

Исследуемая система	QEMU, мин	thread_monitor, мин	call_stack, мин
i386 + WindowsXP	3:40	3:44	3:53
Замедление		1,8%	5,9%
x86_64 + Windows7	4:43	5:59	7:19
Замедление		27%	55%
mips64 + ОС РВ	1:21	1:24	1:33
Замедление		3,7%	15%

Для определения корректности получаемых результатов для разных систем и плагинов были использованы различные методы.

Для проверки корректности работы алгоритма идентификации потоков в операционных системах семейства Windows было выполнено сравнение распознанных потоков с содержимым системной структуры TEB, а именно с полем UniqueThread. Адреса необходимых структур были получены средствами обратной инженерии. Для 32-битной Windows XP адрес структуры TEB располагается по смещению 0x18 относительно регистра FS, а поле UniqueThread располагается в TEB по смещению 0x24. Для 64-битной Windows 7 адрес структуры TEB располагается по смещению 0x30 относительно регистра GS, а поле UniqueThread располагается в TEB по смещению 0x48.

При сравнении было выявлено, что каждому идентифицированному потоку в плагине thread_monitor соответствует один идентификатор из системной структуры, при этом не возникает ситуаций, когда один идентификатор из структуры соответствует нескольким идентифицированным потокам. В редких случаях, когда система завершает работу потока без использования соответствующего системного вызова, возникает ситуация с одним идентифицированным потоком и соответствующими несколькими системными идентификаторами. Зачастую это не играет роли для дальнейшего анализа (например, это не сказывается на корректности возвращаемых данных плагином восстановления стека вызовов), однако необходимо учитывать эту особенность при дальнейшей проектировке плагинов анализа.

Для определения корректности работы алгоритма восстановления стека вызовов в ОС семейства Windows были собраны тестовые программы, к которым удаленно подключался отладчик GDB. Далее выполнялось сравнение адресов, возвращаемых отладчиком по команде backtrace, с адресами, записанными в восстановленном плагином стеке. Полученные результаты показали, что плагин позволяет получить корректные данные.

Для системы mips64 с ОС РВ при определении корректности результата работы алгоритмов была возможность использовать системные команды: команда вывода списка потоков используется для сравнения с результатами идентификации потоков; команда вывода списка вызванных внутри потока функций используется для проверки восстановленного стека вызовов. В обоих случаях были получены корректные данные, восстановленный стек вызовов полностью соответствовал возвращаемому системой.

Для оценки работы плагина для обнаружения подозрительных операций возврата были рассмотрены два примера: синтетический, написанный для доказательства потенциальной

пользы использования алгоритма, и реальный, использующий существующее приложение с уязвимостью и описанный на агрегаторе уязвимостей (сайте exploit-db.com) зловредный код.

Синтетический пример представлял из себя тестовую программу, выполняющую ряд вычислений, но сопровождаемую ассемблерной вставкой, которая напрямую подменяет значение адреса возврата в стеке и выполняет инструкцию ret. Данный пример был успешно обнаружен плагином, что свидетельствует о потенциальной возможности его использования для защиты от зловредного кода.

Для реального примера обнаружения злонамеренной подмены адреса возврата была рассмотрена программа с уязвимостью gAlan, для 32-битных ОС Windows. В данной программе при загрузке пользовательских файлов не выполняется проверка размера, что может вызвать переполнение буфера, а значит идеально подходил для атаки злоумышленника. В качестве примера зловредного кода был использован код под номером 10345 из базы данных уязвимостей (exploit-db.com) – это скрипт на языке Python, который формирует пользовательский файл с необходимым содержимым, в результате загрузки которого в gAlan происходит переполнения буфера и выполняется запуск программы "калькулятор". Содержимым, записываемым скриптом в файл, помимо прочего, является мусор, чтобы заполнить стек вплоть до адреса возврата, новый адрес возврата, роль которого выполняет адрес инструкции call esi из библиотеки glib-1_3, распространяемой в комплекте с уязвимой программой, и шеллкод, который будучи помещенным в память и выполненным, вызовет запуск системной программы “калькулятор”.

При запуске данного зловредного кода плагин обнаружения подозрительных операций возврата благополучно оповестил аналитика о потенциальной атаке, при этом отобразив адрес инструкции возврата и адрес, на который произошел возврат. Данной информации достаточно, чтобы при дальнейшем анализе сделать выводы о зловредности по участку кода, на который осуществляется переход, а также чтобы определить, что изначально являлось источником уязвимости. Таким образом, плагин обнаружения подозрительных операций возврата показал свою действенность и в условиях реальной уязвимости и использования её злоумышленником.

6. Заключение

В данной работе был предложен метод восстановления данных о потоках и описана его реализация. Данный метод лишен ряда недостатков, присутствующих в аналогичных фреймворках, и может использоваться для широкого ряда операционных систем за счёт обеспечения низкой ОС-зависимости. Основная часть работы сконцентрирована на описании способов применения полученных данных о потоках для реализации более комплексных инструментов для аналитика: плагин восстановления стека вызовов, плагин, отслеживающий подозрительные операции возврата, плагин мониторинга обращения к освобожденной памяти стека. Полученные результаты были проверены на корректность сравнением с эталонными данными, что доказало возможность применения разработанных плагинов для решения реальных задач анализа. Замедление, накладываемое плагинами, значительное, однако всё ещё позволяющее работать с исследуемой системой.

В дальнейшем разработка будет сосредоточена на реализации новых инструментов анализа, базирующихся на результатах описанной в данной статье работы.

Список литературы / References

[1]. Luk C.-K.. Cohn R. et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. ACM SIGPLAN Notices, vo. 40, issue 6, 2005, pp 190-200.

[2]. Nethercote N., Seward J. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In Proc. of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007, pp. 89-100.

[3]. Dolan-Gavitt B., Leek T. et al. Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection. In Proc. of the IEEE Symposium on Security and Privacy, 2011, pp. 297-312.

[4]. Henderson A., Prakash A. et al. Make It Work, Make It Right, Make It Fast: Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform. In Proc. of the International Symposium on Software Testing and Analysis. 2014, pp. 248-258.

[5]. Henderson A., Yan L.K. et al. DECAF: A Platform-Neutral Whole-System Dynamic Binary Analysis Platform. IEEE Transactions on Software Engineering, vol. 43, no. 2, 2017, pp. 164-184.

[6]. Zeng J., Fu Y., Lin Z. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework. In Proc. of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2015, pp. 147-160.

[7]. Bruening D., Duesterwald E., Amarasinghe S. Design and implementation of a dynamic optimization framework for Windows. In Proc. of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4).

[8]. Song D., Brumley D. et sl. BitBlaze: A new approach to computer security via binary analysis. In Proc. of the International Conference on Information Systems Security, 2008, pp. 1-25.

[9]. Bellard F. QEMU, a Fast and Portable Dynamic Translator. In Proc. of the Annual Conference on USENIX Annual Technical Conference, 2005, pp. 41-46.

[10]. Dolan-Gavitt B., Leek T. et al. Tappan Zee (North) Bridge: Mining Memory Accesses for Introspection. In Proc. of the 2013 ACM SIGSAC Conference on Computer & Communications Security, 2013, pp. 839-850.

[11]. Python scriptable reverse engineering sandbox, a virtual machine instrumentation and inspection framework based on qemu. Available at: https://github.com/Cisco-Talos/pyrebox, accessed 24.08.2021.

[12]. Icebox. Available at: https://github.com/thalium/icebox, accessed 24.08.2021.

[13]. Volatility framework – volatile memory extraction utility frame-work. Available at: https://github.com/volatilityfoundation/volatility, accessed on 24.08.2021.

[14]. Winbagility. Available at: https://winbagility.github.io/, accessed on 24.08.2021.

[15]. Tanenbaum A.S., Bos H. Modern operating systems. 4th edition. Pearson, 2014, 1136 p.

[16]. Vasiliev I., Dovgalyuk P., Klimushenkova M. Selective Instrumentation Mechanism and its Application in a VirtualMachine. In Proc. of the Ivannikov Memorial Workshop (IVMEM), 2019, pp. 72-76.

Информация об авторах / Information about authors

Иван Александрович ВАСИЛЬЕВ – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Ivan Aleksandrovich VASILIEV is a software developer. Research interests: debugging, introspection and instrumentation of virtual machines, dynamic analysis of binary code, emulators.

Павел Михайлович ДОВГАЛЮК – старший научный сотрудник, доцент, кандидат технических наук. Сфера научных интересов: интроспекция и инструментирование виртуальных машин, динамический анализ кода, эмуляторы.

Pavel Mikhailovich DOVGALYUK – Senior Researcher, Associate Professor, Candidate of Technical Sciences. Research interests: introspection and instrumentation of virtual machines, dynamic code analysis, emulators.

Мария Анатольевна КЛИМУШЕНКОВА – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Maria Anatolyevna KLIMUSHENKOVA is a software developer. Research interests: debugging, introspection and instrumentation of virtual machines, dynamic analysis of binary code, emulators.