DOI: 10.15514/ISPRAS-2025-37(6)-58

Обзор существующих методов анализа полносистемного покрытия кода

[1] *А.А. Иванов, ORCID: 0000-0003-2697-1449 <arkadiy.ivanov@ispras.ru>*
[1,2] *П.М. Довгалюк,* ORCID: 0000-0003-2483-5718 *<pavel.dovgaluk@ispras.ru>*
[1] *Н.И. Фурсова,* ORCID: 0000-0001-6817-4670 *<natalia.fursova@ispras.ru>*

[1] *Институт системного программирования РАН им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*

[2] *Новгородский государственный университет им. Ярослава Мудрого, 173003, Великий Новгород, ул. Большая Санкт-Петербургская, д. 41.*

Аннотация. С увеличением числа кибератак и несанкционированного доступа к данным растет популярность методологии безопасной разработки (SDL), направленной на интеграцию мер безопасности на всех этапах разработки ПО. Анализ покрытия кода – важный инструмент для изучения поведения программы в процессе выполнения, который помогает оценить эффективность методов выявления уязвимостей и непроверенного кода. Однако большинство существующих инструментов анализируют код в изоляции от реальной среды выполнения, что ограничивает точность анализа. Для его улучшения необходим комплексный подход, учитывающий динамические взаимодействия компонентов и особенности выполнения программы. В статье рассматриваются методы сбора покрытия кода в контексте их адаптации к полносистемной эмуляции, возможности корреляции с машинными инструкциями и связанные с этим затраты.

Ключевые слова: покрытие кода; динамический анализ; инструментарий natch; эмулятор qemu.

Для цитирования: Иванов А.А., Довгалюк П.М., Фурсова Н.И. Обзор существующих методов анализа полносистемного покрытия кода. Труды ИСП РАН, том 37, вып. 6, часть 4 , 2025 г., стр. 187–200. DOI: 10.15514/ISPRAS–2025–37(6)–58.

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

Review of Existing Methods for Analyzing Full-System Code Coverage

[1] *A.A. Ivanov, ORCID: 0000-0003-2697-1449 <arkadiy.ivanov@ispras.ru>*
[1,2] *P.M. Dovgalyuk,* ORCID: 0000-0003-2483-5718 *<pavel.dovgaluk@ispras.ru>*
[1] *N.I. Fursova,* ORCID: 0000-0001-6817-4670 *<natalia.fursova@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. With the rise of cyberattacks and unauthorized data access, the popularity of Secure Software Development Lifecycle (SDL) methodology has grown, aiming to integrate security measures at every stage of software development. Code coverage analysis is a key tool for studying program behavior during execution, helping assess the effectiveness of vulnerability detection methods and untested code. However, most existing tools analyze code in isolation from the actual execution environment, limiting analysis accuracy. A comprehensive approach is needed, considering dynamic component interactions and program execution context. This paper discusses code coverage collection methods, their adaptation to full-system emulation, the possibility of correlating with machine instructions, and associated costs.

Keywords: dynamic analysis; code coverage; natch; qemu.

For citation: Ivanov A.A., Dovgalyuk P.M., Fursova N.I. Review of existing methods for analyzing full-system code coverage. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 4, 2025. pp. 187-200 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-58.

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

1. Введение

С ростом количества кибератак и несанкционированного доступа к данным, всё больше набирает популярность методология безопасной разработки (SDL), целью которой является интеграция мер безопасности во все этапы разработки программного обеспечения. Анализ покрытия кода является важным инструментом для изучения поведения программы в процессе выполнения. Он позволяет оценить эффективность методов, применяемых для выявления уязвимостей и идентификации непроверенного кода.

Полносистемное покрытие кода представляет собой комплексный метод анализа, который исследует работу программного обеспечения как единого целого, а не отдельных изолированных компонентов. Такой анализ учитывает реальные условия выполнения программы: взаимодействие между различными модулями, работу с операционной системой и внешними сервисами. Это позволяет обнаруживать уязвимости и ошибки, которые проявляются только при комплексной работе всей системы и остаются незамеченными при изолированном тестировании отдельных компонентов. К сожалению, большинство современных инструментов проводят анализ кода в отрыве от фактической среды выполнения, что существенно снижает точность получаемых результатов.

Natch [1] – это инструмент для определения поверхности атаки, основанный на полносистемном эмуляторе QEMU [2], интегрирующий технологии анализа помеченных данных, интроспекции виртуальных машин [3]-[4] и детерминированного воспроизведения [5]. В контексте исследования покрытия кода существует два концептуальных подхода. Первый из них предполагает детальный анализ на уровне машинных инструкций, а второй базируется на интеграции данных о покрытии с отладочными символами, что обеспечивает более высокий уровень абстракции при исследовании. Принципиальным ограничением

существующего методов сбора покрытия кода является отсутствие возможности однозначной корреляции исходного кода с машинными инструкциями. Преодоление данного ограничения позволит расширить возможности Natch в следующих направлениях:

1) **Поиск поверхности атаки**: информация о частоте выполнения строк исходного кода позволит более точно определить потенциальные точки входа. Часто выполняемые функции, особенно те, которые обрабатывают чувствительные данные, могут представлять больший интерес для злоумышленников и требуют особого внимания с точки зрения безопасности.
2) **Оптимизация производительности**: выявление наиболее часто выполняемых участков кода, которые могут стать целью для оптимизации.
3) **Анализ безопасности**: определение редко выполняемых участков кода, которые могут содержать ошибки или уязвимости из-за недостаточного тестирования.

В данной статье рассматриваются различные методы сбора покрытия кода в контексте их адаптации к полносистемной эмуляции, возможности однозначной корреляции исходного кода с машинными инструкциями, а также связанных с этим затрат.

2. Определение проблемы

2.1 Корреляция покрытия с исходным кодом

Отладочная информация обычно хранится в формате DWARF и предоставляет данные о том, к какому диапазону адресов в машинном коде принадлежит каждая строка исходного кода. Эти диапазоны адресов затем делятся на базовые блоки (ББ) машинных инструкций. В некоторых случаях одной строке исходного кода может соответствовать несколько несмежных диапазонов адресов в машинном коде. Это явление особенно проявляется в процессе оптимизации циклов, когда компилятор может разделять операции инициализации переменной цикла и ее инкремента, размещая их в разных участках машинного кода.

В качестве демонстрации данного тезиса обратимся к конкретному случаю. Допустим, в исходном коде на строке 225 находится цикл. При декодировании отладочной информации в формате DWARF получаем, что указанная строка связана с двумя непересекающимися диапазонами машинных адресов: 0x12a3 → 0x12a7 и 0x1438 → 0x145b (рис. 1).

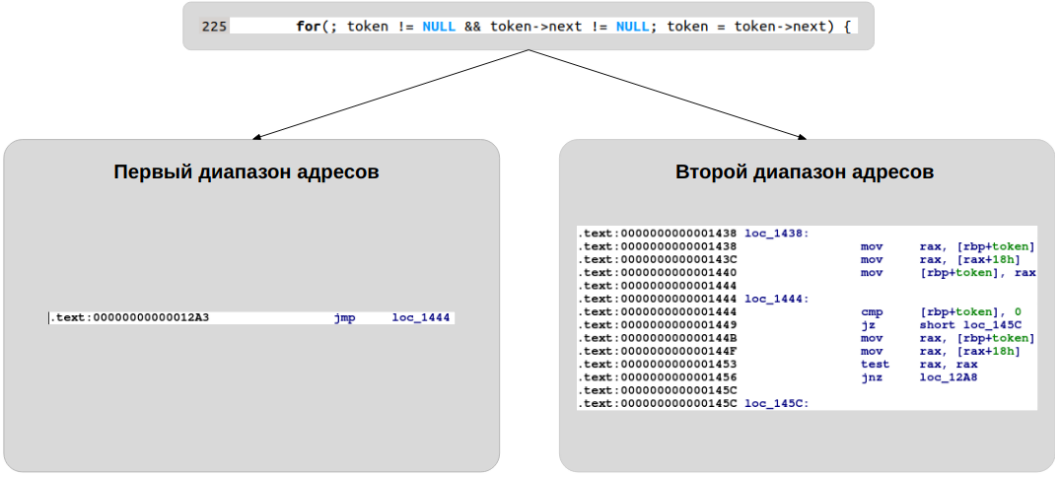

Рис. 1. Интерпретация отладочной информации DWARF. Разбиение строки на диапазоны адресов.
Fig. 1. DWARF debug information interpretation. Address range splitting.

Для определения количества выполнений строки 225 необходимо подсчитать количество выполнений каждого из этих интервалов и затем суммировать результаты.

Мы рассмотрим два метода вычисления количества выполнений: первый основан на анализе покрытия, учитывающем только частоту выполнения отдельных базовых блоков (рис. 2), в то время как второй метод включает отслеживание цепочки выполнения, что позволяет учесть порядок исполнения базовых блоков (рис. 3).

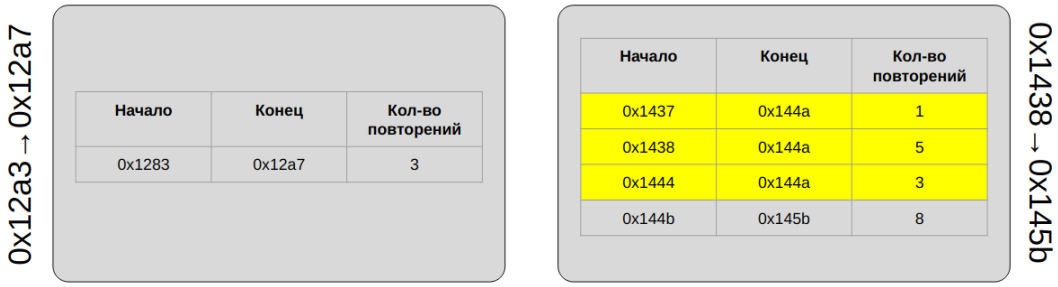

Рис. 2. Интерпретация отладочной информации DWARF. Соответствие строк и базовых блоков без информации о порядке их выполнения.

Fig. 2. DWARF debug information interpretation. Line and basic block mapping without execution order.

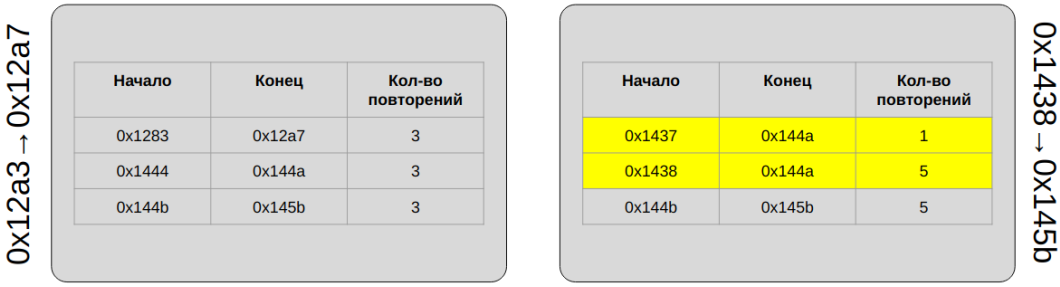

Рис. 3. Интерпретация отладочной информации DWARF. Соответствие строк и базовых блоков с порядком их выполнения.

Fig. 3. DWARF debug information interpretation. Line and basic block mapping by execution order.

Методика подсчета выполнений каждого интервала предусматривает два сценария: при совпадении адресных пространств базовых блоков (выделены желтым цветом) их частота выполнения суммируются, иначе, берется максимальное значение повторений.

Тогда конечная формула принимает следующий вид:

$$N_{total} = \sum_{i=1}^{n} \left(\begin{cases} \sum_{j=1}^{m_i} N_{ji}, \text{ББ разделяют одно и то же адресное пространство} \\ \max\left(N_{i1}, \dots, N_{im_i}\right), \text{ББ разделяют разное адресное пространство} \end{cases} \right)$$

N_{total} – общее количество выполнений строки

N_{ij} – количество выполнений j – го базового блока в интервале i

n – количество интервалов

m_i – количество базовых блоков в интервале i

После подсчёта числа выполнений, были получены результаты для двух методов: 12 и 9 соответственно. Из чего делаем вывод, что без учета порядка выполнения базовых блоков достичь точного результата довольно сложно.

2.2 Подход Natch и его ограничения

Подход, применяемый в Natch, базируется на одноименном плагине QEMU [6], предназначенном для генерации данных о покрытии в формате drcov [7], который был адаптирован для обеспечения полносистемного анализа и обработки чувствительных данных. Однако, в связи с необходимостью подсчета количества выполненных строк исходного кода, возникла проблема: формат drcov предоставляет информацию только о факте выполнения базового блока, без возможности восстановить цепочку выполнения. В связи с этим возникла необходимость в поиске альтернативного метода сбора покрытия.

2.3 Требования к подходу

При выборе метода сбора покрытия кода часто возникает компромисс между точностью и производительностью. Важно отметить, что Natch использует динамическое бинарное инструментирование (DBI) за пределами виртуальной машины (Out-of-VM) [8], что сопряжено с высокими накладными расходами, обусловленными виртуализацией и преодолением семантического разрыва. В связи с этим производительность становится менее критичным фактором, что позволяет сосредоточиться на точности сбора покрытия кода.

При этом важно помнить, что поставленная цель не заключается в различении путей выполнения программы, а лишь в соотнесении базовых блоков с соответствующими строками исходного кода и подсчете их выполнения. Следовательно, использование методов, основанных на путях или N-граммах, является избыточным и приведет к лишнему замедлению и увеличению объема хранимых данных.

Кроме того, будущий метод должен обеспечивать возможность хранения дополнительных данных. В частности, для полносистемного покрытия важно иметь возможность сопоставить выполненные базовые блоки с модулями и процессами, в которых они исполнялись.

Наконец, необходимо оставить возможность анализа покрытия двумя методами: на уровне машинных инструкций и с использованием отладочных символов для более высокого уровня абстракции.

3. Обзор существующих решений

3.1 AFL-подобные инструменты

Рассмотрим подход к сбору покрытия кода, применяемый в фаззерах, которые упорядочивают и выбирают входные данные на основе информации о покрытии. Одним из первопроходцев в применении подобных техник является American Fuzzy Lop (AFL) – фаззер, ориентированный на обеспечение безопасности [9]. В своей основе он использует специальные хеш-таблицы, где в качестве ключа используется хеширование кортежа "предыдущий блок – текущий блок". Индекс в такой таблице вычисляется как результат побитовой операции XOR между текущим и предыдущим адресами. Для обеспечения квазиравномерного распределения значений, адреса предварительно обрабатываются специальной хеш-функцией. Поскольку операция XOR симметрична, направление между двумя базовыми блоками A и B неразличимо ($A \oplus B \equiv B \oplus A$). Чтобы сохранить различие направления переходов ($A \rightarrow B$), применяется сдвиг местоположения на один бит вправо, что позволяет отличать направление $A \oplus B$ от $B \oplus A$. В хеш-таблице сохраняется статистика, обычно отражающая количество выполнений рёбер. В AFL используется грубая статистическая оценка, распределённая по восьми диапазонам: 1, 2, 3, 4-7, 8-15, 16-31, 32-127 и 128+ [10].

Одной из причин рассмотрения AFL-подобных инструментов является наличие помимо статического инструментирования исходного кода с помощью GCC и Clang ещё и динамического бинарного инструментирования на основе пользовательского режима QEMU.

В AFL++ [11] используются специальные хелперы – функции, которые встраиваются в процесс трансляции исходного кода в промежуточное представление. Эти функции интегрируются в поток промежуточного кода и позволяют регистрировать в реальном времени переходы между базовыми блоками посредством динамического инструментирования.

3.1.1 Коллизии

Фаззер AFL использует хеш-функцию для сопоставления пары базовых блоков с индексом в карте покрытия. Однако, из-за ограниченного размера карты разные пары блоков могут хешироваться в одно и то же место. Это называется коллизией. В результате AFL видит два разных пути как один и тот же, что снижает точность покрытия. Исследования, представленные в статье [12], показали, что в некоторых приложениях до 75% переходов могут пересекаться с другими.

CollAFL [12] решает проблему коллизий хешей в AFL, добиваясь того, чтобы каждое ребро в целевой программе имело уникальный хеш, что позволяет AFL различать любые два ребра. Для этого предложен новый способ хеширования для уменьшения коллизий при отслеживании покрытия программы. Алгоритм состоит из трёх частей: для блоков с одним предшественником, для блоков с несколькими предшественниками и общей схемы предотвращения коллизий. Чтобы его использовать, программу нужно предварительно проанализировать с помощью статического анализатора или компилятора, чтобы разделить базовые блоки на указанные категории и рассчитать оптимальный размер таблицы покрытия, которая будет фиксировать все переходы между блоками. Как отмечается в самой статье, такое решение позволяет избавиться от коллизий, для всех найденных ребер.

В статье [13] отмечается, что с точки зрения концепции алгоритмы хеширования CollAFL близки к покрытию без коллизий, достигаемому с помощью LTO-инструментирования в AFL++. При традиционной компиляции программа разбивается на отдельные единицы (объектные файлы), и компилятор видит только часть программы, что затрудняет присвоение уникальных идентификаторов, так как полная структура программы доступна только при линковке. В результате базовым блокам могут присваиваться псевдослучайные идентификаторы, что требует применения хеш-функций для определения индексов в таблице покрытия. С другой стороны, при использовании оптимизации на этапе линковки (LTO) компилятор имеет возможность видеть все базовые блоки одновременно, что позволяет назначать уникальные идентификаторы. Это, в свою очередь, даёт точное определение количества базовых блоков и позволяет заранее вычислить оптимальный размер таблицы. В режиме без LTO AFL++ использует побитовые операции для вычисления индексов в таблице покрытия, что вносит небольшие накладные расходы, тогда как с LTO они почти исчезают благодаря применению предсказуемых индексов. Таким образом AFL++ предлагает использовать интерфейс LTO для получения информации о всех базовых блоках программы на этапе линковки. Это позволяет создать хеш-таблицу, которая свободна от статических коллизий и каждый базовый блок получает уникальный индекс, что минимизирует вероятность коллизий при доступе к таблице покрытия.

3.1.2 Арифметические погрешности

Помимо коллизий, AFL сталкивается с проблемой арифметических погрешностей при отслеживании частоты выполнения ребер. Эта проблема связана с тем, что карта покрытия использует ограниченные по размеру счетчики для каждого ребра.

Причины погрешностей:

1) **Переполнение счетчиков**: когда программа многократно проходит один и тот же путь (например, в циклах), счетчики в карте покрытия могут переполниться. Например, если размер ячейки для хранения счетчика составляет один байт, то при

достижении 255 выполнений счетчик переполнится и обнулится. В результате теряется информация о точном количестве проходов по этому ребру.

2) **Диапазонные счетчики**: до этого момента обсуждались простые счетчики, которые увеличиваются на единицу при каждом выполнении ребра, однако в AFL используется альтернативный метод – так называемые «диапазонные счетчики». Этот подход группирует частоты выполнения в определенные интервалы и хранит лишь информацию о том, в какой интервал попадает частота (например, с использованием логарифмических счетчиков). Основная идея диапазонных счетчиков заключается в том, что для больших частот небольшие различия считаются несущественными. Однако, диапазонные счетчики теряют точность на высоких частотах выполнения. Например, в стандартной реализации AFL различие между 200 и 2000 выполнений пути не фиксируется, что приводит к потере детальной информации.

Наиболее критичный сценарий возникает при сочетании проблемы переполнения с коллизиями в хеш-таблице. Когда два ребра отображаются на один и тот же индекс, происходит коллизия, приводящая к тому, что оба ребра увеличивают один счётчик частоты, что существенно снижает его информативность.

Для предотвращения проблемы сброса счётчика в AFL++ он не только увеличивается на единицу, но и получает специальный бит переноса, благодаря чему переполненные счётчики сбрасываются до единицы [14]. Однако даже переполненные счётчики, которые не равны нулю, могут приводить к пропущенным состояниям из-за высокой частоты их выполнения (например, в циклах). Интересно, что авторы AFL++ пытались реализовать насыщенные счётчики, которые фиксируются на максимальном значении перед переполнением. Однако из-за дополнительного ветвления накладные расходы на производительность оказались слишком высокими [14].

3.1.3 Анализ применимости метода

Для получения приемлемой точности AFL-подобных инструментов, необходимо предварительно производить статический анализ объекта оценки. Однако при полносистемном сборе покрытия возникают следующие проблемы:

1) **Распакованный код**

   Разработчики используют различные техники упаковки – от простого сжатия до сложных алгоритмов шифрования – с целью воспрепятствовать обратной разработке, усложнить анализ вредоносного кода или просто уменьшить размер дистрибутива. Каждый слой упаковки добавляет новый уровень абстракции. Структура такого кода скрыта за слоями шифрования или сжатия, что делает программу нечитаемой до тех пор, пока она не будет распакована в процессе исполнения.

2) **Динамически сгенерированный код**

   В современных системах динамическая генерация кода стала не просто техническим приемом, а важной парадигмой программирования: компиляция "на лету" (JIT), препроцессоры и среды выполнения создают новый код в процессе работы программы. Это делает невозможным прогнозирование поведения программы до её исполнения.

3) **Неопределенности** межмодульных взаимодействий

   В контексте полносистемного анализа приложения не всегда возможно заранее определить все потенциальные модули и их взаимодействия. Современные системы характеризуются высокой степенью модульности и распределенности, где модули

могут подключаться и взаимодействовать динамически во время выполнения приложения.

Поскольку у нас нет информации обо всех базовых блоках, используемых в анализируемой системе, и отсутствует возможность провести статический анализ, невозможно определить оптимальный размер хеш-таблицы для сбора покрытия ветвей без коллизий.

Кроме того, существенным недостатком AFL-подобных инструментов, основанных на покрытии ребер, в нашем случае является использование в качестве вершин графа ключа, который содержит только информацию о двух адресах базовых блоков, между которыми происходил переход. Поскольку в нашем случае у нас нет данных о самих базовых блоках, это делает невозможным не только построение покрытия, но и соотнесение выполненных базовых блоков с конкретным модулем и процессом.

Стоит отметить, что afl-cov [15], используемый для отображения покрытия, вместо упрощенной битовой карты AFL требует от пользователя скомпилировать анализируемое приложение с инструментированием gcov для сбора покрытия. Это позволяет создать HTML-отчет в формате lcov [16]. В данном случае от AFL используются только входные данные, сгенерированные в процессе фаззинга.

3.2 NQC²

Одной из недавних разработок в области покрытия кода является NQC² [17]. Авторы усовершенствовали механизм Execution Trace (etrace) от Xilinx [18], который собирает трассировки во время выполнения и сохраняет их в файл журнала выполнения (elog) на хостовой машине. Основные недостатки оригинального подхода заключались в зависимости от модифицированной версии QEMU и необходимости отключения цепочек трансляции блоков (TB chaining). В NQC² эти проблемы были решены благодаря использованию QEMU API, что обеспечивает универсальность и совместимость с любыми версиями QEMU.

3.2.1 Сбор покрытия кода

Плагин NQC² использует механизм обратных вызовов QEMU [19] для регистрации событий, таких как трансляция и выполнение блоков трансляции, сохраняя соответствующую информацию о покрытии в специализированном буфере. При заполнении этого буфера данные записываются в файл формата elog. Этот файл имеет бинарную структуру и состоит из блоков, каждый из которых включает заголовок и сегмент данных. Заголовок определяет тип и длину данных, а сегменты содержат адреса выполненных инструкций и время их исполнения, что позволяет производить детальный анализ покрытия кода на уровне базовых блоков.

Во время операции записи данных из буфера в файл происходит приостановка QEMU, что негативно сказывается на производительности. Для решения этой проблемы в NQC² была внедрена множественная буферизация, что не только улучшает производительность, но и позволяет объединять последовательные блоки трансляции, тем самым уменьшая размер файла elog.

3.2.2 Преобразование покрытия в формат LCOV

Полученное с помощью NQC² покрытие кода можно конвертировать в формат lcov, используя инструмент qemu-etrace [20].

Рассмотрим основные этапы алгоритма более подробно:

1) **Разбиение базовых блоков на машинные слова**
   Цель данного этапа – анализ файла elog, в ходе которого выполняется считывание базовых блоков и их сегментация на машинные слова длиной 4 байта, что

соответствует стандартному размеру инструкции для большинства современных архитектур. Процесс осуществляется следующим образом: выполняется итерация по адресному пространству ББ с шагом 4 байта. На каждой итерации полученный адрес делится на 4 для получения слова. Таким образом, слова, принадлежащие данному базовому блоку, вычисляются поэтапно, начиная с его начального адреса и продолжаются до тех пор, пока адрес не превысит значение конца блока, увеличенное на единицу.

2) **Формирование таблицы частотности**

   После разбиения формируется таблица частотности, в которой записывается количество выполнений каждого слова.

3) **Извлечение отладочных символов из ELF-файла**

   С помощью утилиты dwarfdump извлекаются отладочные символы из анализируемого ELF-файла, содержащие информацию о том, к какой строке исходного кода соответствует тот или иной адрес.

4) **Корреляция таблицы частотности с отладочной информацией**

   На этом этапе таблица частотности сопоставляется с извлечёнными отладочными символами, что позволяет отфильтровать только те слова, которые представляют интерес.

5) **Подсчёт выполнений строки исходного кода**

   Если значение счетчика выполнения строки исходного кода меньше счетчика соответствующего машинного слова, к нему прибавляется количество исполнений этого слова, обновляя счетчик строки.

3.2.3 Анализ применимости метода

Если говорить о формате в целом, то он включает множество данных, которые нам не нужны, например, время выполнения базового блока. В то же время, в нем отсутствует важная для нас информация о модуле и процессе для каждого базового блока.

Особый интерес представляет алгоритм конвертации покрытия из формата elog в формат lcov, и для его тестирования был выбран пример подсчета количества выполнений строки 5 (рис. 4).

```c
#include <stdio.h>

int main() {
    for (int i = 5; i > 0; i--) {
        for (int j = i; j < 2; j++) {
            printf("Iteration %d\n", j);
        }
    }
    return 0;
}
```

Рис. 4. Пример на языке C для тестирования метода NQC².
Fig. 4. C example for testing the NQC² method.

При декодировании отладочной информации в формате DWARF получаем, что указанная строка связана с двумя непересекающимися диапазонами машинных адресов: 0x115e → 0x1165 и 0x117c → 0x1185, в рамках которых находятся три базовых блока. На рис.5 показан процесс формирования таблицы частотности в ходе которого были выявлены следующие проблемы:

1) Для базового блока с адресами 0x115e → 0x1165 алгоритм вычисления упускает важную информацию. Следуя вышеописанному алгоритму, к данному ББ относятся

два слова: 0x457 и 0x458. Однако при делении конечного адреса (0x1165) нацело на 4 получается слово 0x459, которое не учитывается в анализе. В то же время, в отладочной информации для строки номер 5 присутствует точка 0x1164, соответствующая слову 0x459. При текущем подходе эта точка систематически пропускается.

2) Механизм суммирования счетчиков исполнений при объединении данных о покрытии в данном примере работает некорректно. Синим цветом, отмечены слова, которые содержаться в отладочной информации DWARF и учувствуют в формировании конечного результата. Итак, количество выполнений первого интересующего нас слова равно 5, значит количество выполнений строки равно 5, для следующего слова количество его выполнений равно 6. Алгоритм складывает 5 и 6, получая 11, но это ошибка, потому что правильный результат для этого примера по данным gcov – 6.

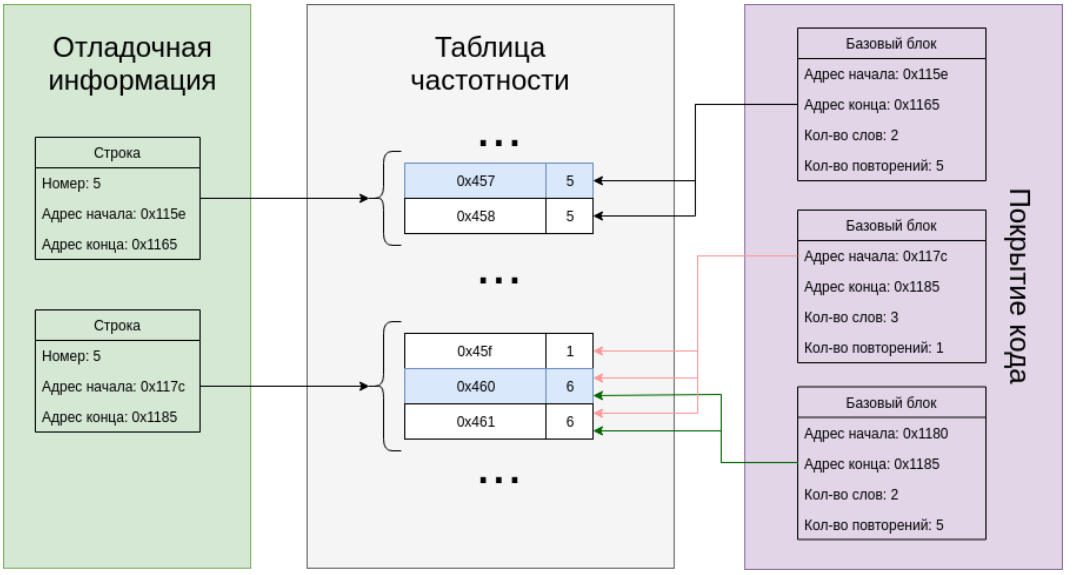

Рис. 5. Визуализация алгоритма преобразования покрытия из формата elog в lcov.
Fig. 5. Visualization of the elog to lcov conversion.

Также стоит отметить, что отсутствует корректная обработка инструкций, размер которых не кратен размеру машинного слова (4 байта).

3.3 Panda

В области анализа ПО Panda [21] представляет собой решение с открытым исходным кодом, предназначенное для архитектурно-нейтрального динамического анализа. Архитектура Panda базируется на модульной системе плагинов, в основе которой лежит эмулятор QEMU. В рамках данной статьи будет рассмотрен специализированный плагин для анализа покрытия кода, реализующий три режима работы: osi-block, asid-block и edge mode [22].

3.3.1 Режимы osi-block и asid-block

Оба режима фокусируются на сборе покрытия на основе базовых блоков и предоставляют следующую информацию о каждом из них:

- признак выполнения в режиме ядра;
- адрес блока;

- размер блока.

Информация о покрытии сохраняется в файл в формате CSV. В первой строке находится заголовок, а далее для каждого базового блока на новой строке представлена информация, разделенная запятыми. По умолчанию базовый блок сохраняется только при первом его появлении, однако существует опция, позволяющая сохранять запись каждый раз при ее генерации.

Основное отличие между режимами osi-block и asid-block заключается в наличии интроспекции (OSI). Это позволяет в режиме osi-block получать дополнительную информацию о процессе, включая его имя, идентификатор и поток, в котором он выполнялся.

3.3.2 Режим edge

В режиме edge реализован принципиально иной подход: покрытие кода фиксируется на основе переходов между блоками, при этом плагин осуществляет регистрацию начальных и конечных блоков при переключении выполнения. В основе данного режима лежит метод покрытия ребер с использованием N-грамм. При N=0 N-граммное покрытие сводится к покрытию блоков, в то время как при $N \to \infty$ оно эквивалентно покрытию путей. Подробности выбора оптимального значения N можно найти в исследованиях [23][24].

На первом этапе собирается трасса выполнения программы: для каждого идентификатора адресного пространства (ASID) записывается последовательность адресов выполненных базовых блоков, которая сохраняется в хеш-таблице.

После завершения выполнения программы собранная трасса анализируется для формирования статистики покрытия. Этот процесс включает в себя создание новой структуры данных, которая агрегирует информацию о частоте выполнения различных последовательностей базовых блоков. Ключевым параметром в этом анализе является N - максимальная длина последовательности базовых блоков, которую мы хотим анализировать. В этой структуре для каждого ASID хранится информация о частоте встречаемости различных последовательностей PC длиной до N. Ключами во внутренней хеш-таблице являются последовательности адресов PC, а значениями - количество их повторений в трассе выполнения.

Наконец, формируется отчет в формате PandaLog, который оптимизирован для обработки больших объемов данных, генерируемых в процессе выполнения программы. Главная его особенность заключается в его структуре, основанной на чанках – фиксированных блоках данных, каждый из которых сжимается отдельно.

Следует отметить, что данный режим требует поддержки интроспекции и в текущей реализации применим исключительно для архитектуры X86.

3.3.3 Анализ применимости метода

Как упоминалось ранее (п. 2.3), использование покрытия ребер с применением N-грамм для наших целей является избыточным. Это означает, что наиболее подходящим вариантом остается использование режима osi-block. Данное решение основано на сохранении полной трассы выполнения программы, что, несмотря на повышенные требования к дисковому пространству, обеспечивает доступ как к последовательности выполнения базовых блоков, так и к их содержимому. Однако существенным ограничением является отсутствие информации о модуле, в контексте которого осуществлялось выполнение кода. Помимо этого, запись информации в CSV-файл осуществляется перед трансляцией базового блока, без учета возможных исключений, способных изменить его размер. Это приводит к потенциальным неточностям в собранных данных о покрытии. Наконец, формат хранения данных без сжатия не является оптимальным для полносистемного анализа.

4. Заключение

В результате проведенного анализа существующих решений было установлено, что ни один из рассмотренных инструментов полностью не удовлетворяет установленным выше критериям оценки покрытия кода. Проведенное исследование позволило сделать вывод о необходимости применения гибридного подхода к измерению покрытия кода, сочетающего в себе как покрытие на основе базовых блоков, так и реберное покрытие. Такой подход не требует статического инструментирования для получения информации о базовых блоках и предоставляет достаточно данных для соотнесения базовых блоков с отладочной информацией.

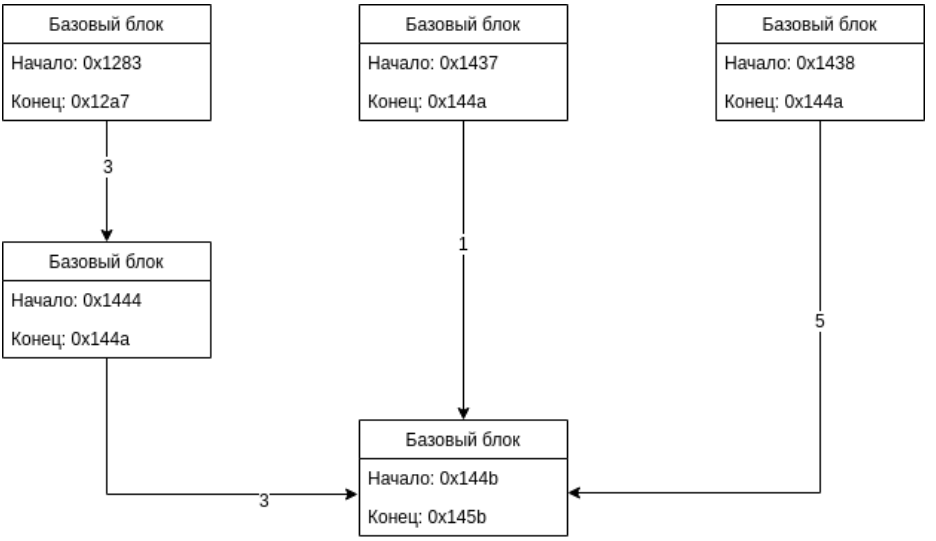

Рис. 6. Соответствие строк и базовых блоков на основе ребер.
Fig. 6. Line and basic block mapping based on edges.

Результаты показали, что подсчет выполнений строк работает корректно и не требует сложных вычислений – в нашем примере счетчик зафиксировал 9 выполнений, что полностью соответствует ожиданиям.

4.1 Перспективы развития исследования

В рамках дальнейшего развития системы оценки покрытия кода рассматриваются несколько ключевых направлений исследования. Первостепенной задачей является оптимизация механизма хранения данных о покрытии, для которой предложены два подхода. Первый подход основан на адаптации методологии NQC[2] и Panda в режиме osi-block, предполагающий хранение последовательности выполнения базовых блоков. Однако, в отличие от реализации в Panda, необходимо учитывать возможность изменения размеров блоков из-за исключений, что делает невозможным прямую запись в файл. Для работы с большими объемами данных может потребоваться внедрение системы множественной буферизации, аналогичной решению, описанному в [17], или реализация механизма динамического сжатия данных покрытия перед записью.

Альтернативный подход предполагает раздельное хранение информации о рёбрах и базовых блоках. В этом случае предлагается расширить формат drcov дополнительной хеш-таблицей, где будет фиксироваться статистика выполнений для каждого ребра, представленного парой идентификаторов базовых блоков. Выбор оптимального подхода будет осуществляться на основе сравнительного анализа производительности и ресурсоемкости предложенных методов в различных сценариях использования.

Важным направлением развития является расширение функциональности формата данных путем включения таблицы модулей с расширенной информацией о процессе. Это позволит обеспечить более детальный анализ работы ПО и повысить точность получаемых результатов.

Отдельное внимание необходимо уделить разработке методологии интеграции данных о покрытии кода с результатами анализа чувствительных данных. На текущий момент система успешно идентифицирует функции, взаимодействующие с помеченными данными, и отслеживает обращения к ним. Следующим этапом является создание эффективного метода наложения и визуализации этих двух типов данных для обеспечения комплексного ПО.

Список литературы

[1]. Довгалюк П.М., Климушенкова М.А., Фурсова Н.И., Степанов В.М., Васильев И.А., Иванов А.А., Иванов А.В., Бакулин М.Г., Егоров Д.И. Natch: Определение поверхности атаки программ с помощью отслеживания помеченных данных и интроспекции виртуальных машин. Труды ИСП РАН, том 34, вып. 5, 2022 г, стр. 89-110. DOI: 10.15514/ISPRAS-2022-34(5)-6. / Dovgalyuk P. M., Klimushenkova M. A., Fursova N. I., Stepanov V. M., Vasiliev I. A., Ivanov A. A., Ivanov A. V., Bakulin M. G., Egorov D. I. Natch: using virtual machine introspection and taint analysis for detection attack surface of the software. Trudy ISP RAN/Proc. ISP RAS, 2022, vol. 34, issue 5, pp. 89-110 (in Russian). DOI: 10.15514/ISPRAS-2022-34(5)-6

[2]. Qemu. A generic and open source machine emulator and virtualizer. URL: https://www.qemu.org/.

[3]. Dovgalyuk P., Fursova N. et al. QEMU-based Framework for Non-Intrusive Virtual Machine Instrumentation and Introspection. In Proc. of the 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 944-948.

[4]. Степанов В. М., Довгалюк П. М., Фурсова Н. И. Декларативный подход к задаче интроспекции виртуальной машины. Труды ИСП РАН, том 36, вып. 3, 2024 г, стр. 123-138. / Stepanov V. M., Dovgalyuk P. M., Fursova N. I. Declarative approach to the problem of virtual machine introspection. Trudy ISP RAN/Proc. ISP RAS, 2024, vol. 36, issue 3, pp. 123-138 (in Russian). DOI: 10.15514/ISPRAS-2024-36(3)-9

[5]. Dovgalyuk P. Deterministic Replay of System's Execution with Multi-target QEMU Simulator for Dynamic Analysis and Reverse Debugging. In Proc. of the 2012 16th European Conference on Software Maintenance and Reengineering, 2012, pp. 553-556.

[6]. Available at: https://github.com/qemu/qemu/blob/master/contrib/plugins/drcov.c, accessed 21.10.2024.

[7]. DrCov File Format, Available at: https://www.ayrx.me/drcov-file-format/, accessed 21.10.2024.

[8]. Zeng, Y., Fu, Y., Lin, Z. "Pemu: A pin highly compatible out-of-vm dynamic binary instrumentation framework." Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. 2015. pp. 147-160.

[9]. Google AFL. Available at: https://github.com/google/AFL, accessed 02.12.2024.

[10]. Technical "whitepaper" for afl-fuzz, Available at: https://lcamtuf.coredump.cx/afl/technical_details.txt, accessed 21.10.2024.

[11]. AFL++. README.llvm.md: Source code coverage through instrumentation, Available at: https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.llvm.md#8-source-code-coverage-through-instrumentation, accessed 21.10.2024.

[12]. Gan S., Zhang C., Qin X., Tu X., Li K., Pei Z., & Chen Z. Collafl: Path sensitive fuzzing. In 2018 IEEE Symposium on Security and Privacy (SP), 2018, pp. 679-696. IEEE.

[13]. Sarafov V., Markvica D., Berlakovich F., Bernad M., & Brunthaler S. Understanding and Improving Coverage Tracking with AFL++ (Registered Report). In Proceedings of the 3rd ACM International Fuzzing Workshop, 2024, pp. 80-89.

[14]. Fioraldi A., Maier D., Eißfeldt H., & Heuse M. {AFL++}: Combining incremental steps of fuzzing research. In 14th USENIX Workshop on Offensive Technologies (WOOT 20), 2020.

[15]. afl-cov - AFL Fuzzing Code Coverage, Available at: https://github.com/mrash/afl-cov, accessed 21.10.2024.

[16]. Lcov - a graphical GCOV front-end. Available at: https://man.archlinux.org/man/lcov.1.en, accessed 21.10.2024.

[17]. Bosbach N., Salama A., Jünger L., Burton M., Zurstraßen N., Pelke R., & Leupers R. NQC²: A Non-Intrusive QEMU Code Coverage Plugin. In Proceedings of the 16th Workshop on Rapid Simulation and Performance Evaluation for Design, 2024, pp. 16-21.
[18]. Xilinx. QEMU: an emulator supporting Xilinx architectures, Available at: https://github.com/Xilinx/qemu, accessed 21.10.2024.
[19]. QEMU. TCG Plugins API: documentation for developing plugins for QEMU, Available at: https://www.qemu.org/docs/master/devel/tcg-plugins.html#api, accessed 21.10.2024.
[20]. QEMU E-Trace: a tracing tool for QEMU, Available at: https://github.com/edgarigl/qemu-etrace, accessed 21.10.2024.
[21]. PANDA: a platform for dynamic analysis and tracing, Available at: https://github.com/panda-re/panda?tab=readme-ov-file, accessed 21.10.2024.
[22]. PANDA Coverage Plugins, Available at: https://github.com/panda-re/panda/tree/dev/panda/plugins/coverage, accessed 21.10.2024.
[23]. Wang, J., Duan, Y., Song, W., Yin, H., & Song, C. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In Proc. of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), 2019, pp. 1-15.
[24]. Peng, X., Jia, P., Fan, X., & Liu, J. ENZZ: Effective N-gram coverage assisted fuzzing with nearest neighboring branch estimation. In Information and Software Technology, 2024, Article 107582.

Информация об авторах / Information about authors

Аркадий Алексеевич ИВАНОВ – разработчик программного обеспечения. Сфера научных интересов: отладка, интроспекция и инструментирование виртуальных машин, динамический анализ бинарного кода, эмуляторы.

Arkady Alekseevich IVANOV 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.