DOI: 10.15514/ISPRAS-2025-37(4)-29

Применение динамической символьной интерпретации в гибридном фаззинге бинарного кода для архитектур Байкал-М и RISC-V 64

В. И. Логунова, ORCID: 0000-0002-3877-1906 <vlada@ispras.ru>

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

Аннотация. Гибридный фаззинг и динамическая символьная интерпретация активно применяются в жизненном цикле разработки безопасного программного обеспечения. В настоящее время доля кода программ, разрабатываемых для архитектур ARM и RISC-V, постоянно увеличивается вместе с потребностью в их эффективном анализе. Данная работа посвящена решению этой задачи путем разработки методов динамической символьной интерпретации и гибридного фаззинга для современных RISC-архитектур – отечественной «Байкал-М» (ARM/AArch64) и открытой RISC-V 64. Разработанные подходы, основанные на моделировании символьной семантики машинных инструкций, интегрированы в инструмент Sydr в составе фреймворка Sydr-Fuzz и нацелены на повышение эффективности гибридного фаззинга. Ключевые результаты включают алгоритмы обработки косвенных переходов для точного определения целевых адресов и реализацию поддержки набора целочисленных инструкций RISC-V в открытом фреймворке Triton, что предоставляет сообществу готовую основу для создания инструментов динамического анализа.

Ключевые слова: динамический анализ; динамическая символьная интерпретация; фаззинг; гибридный фаззинг; инструментация; анализ бинарного кода.

Для цитирования: Логунова В. И. Применение динамической символьной интерпретации в гибридном фаззинге бинарного кода для архитектур Байкал-М и RISC-V 64. Труды ИСП РАН, том 37, вып. 4, часть 2, 2025 г., стр. 235–250. DOI: 10.15514/ISPRAS–2025–37(4)–29.

Application of Dynamic Symbolic Execution in Hybrid Fuzzing of Binary Code for Baikal-M and RISC-V 64 Architectures

V. I. Logunova, ORCID: 0000-0002-3877-1906 <vlada@ispras.ru>

Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Hybrid fuzzing and dynamic symbolic execution have become a vital part of the secure software development lifecycle. Currently, the proportion of code being developed for ARM and RISC-V architectures is constantly increasing, making the task of their effective analysis a top priority. This work is dedicated to solving this task by developing methods for dynamic symbolic execution and hybrid fuzzing for modern RISC architectures – «Baikal-M» (ARM/AArch64) and RISC-V 64. Based on modeling symbolic semantics of machine instructions the developed approaches are integrated into the Sydr tool within the Sydr-Fuzz framework and aim to enhance the efficiency of hybrid fuzzing. Key results include algorithms for processing indirect branches with accurate target addresses determination and RISC-V integer instruction set support in open-source symbolic framework Triton that provides the community with a ready-made foundation for creating dynamic analysis tools.

Keywords: dynamic analysis; dynamic symbolic execution; fuzzing; hybrid fuzzing; instrumentation; binary code analysis.

For citation: Logunova V. I. Application of Dynamic Symbolic Execution in Hybrid Fuzzing of Binary Code for Baikal-M and RISC-V 64 Architectures. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 4, part 2, 2025, pp. 235-250 (in Russian). DOI: 10.15514/ISPRAS-2025-37(4)-29.

1. Введение

Программное обеспечение глубоко интегрировано в современную жизнь, от критической инфраструктуры до персональных гаджетов. Эта повсеместность, однако, сопряжена с риском, поскольку в процессе разработки неизбежно возникают программные дефекты в виде ошибок и уязвимостей. Поэтому в качестве стандарта для их детектирования и устранения используется жизненный цикл разработки безопасного программного обеспечения (ПО), включающий различные виды анализа программ [1]. Соответственно, одной из актуальных задач становится разработка универсальных автоматизированных инструментов, способных эффективно обнаруживать ошибки в коде.

Отличительной чертой динамического анализа является запуск исследуемой программы, за счёт чего снижается количество ложных сообщений об ошибках. Классическим методом в данном направлении, на текущий момент, является фаззинг-тестирование с обратной связью по покрытию (фаззинг "методом серого ящика") [2-3]. Ключевую роль для этого вида анализа играет метрика покрытия, достигнутого во время множественных запусков приложения с разнообразными входными данными. Генерация новых входных данных производится с помощью применения случайных мутаций к данным из предыдущих запусков, которые приоритизируются по приросту покрытия. Кроме того, существует фаззинг "методом белого ящика" [4] также известный как динамическая символьная интерпретация. Вместо получения метрики покрытия данный подход извлекает более подробную информацию о внутреннем устройстве программы для последующей генерации входных данных на основе аналитических вычислений. В первую очередь, это логические условия ветвления, зависящие от изначальных помеченных данных. Из таких условий составляется абстрактная (символьная) модель – предикат пути, отражающий зависимости потока управления от входных данных. Новые входные данные подбираются так, чтобы изменить выбор направления ветвления при будущем запуске. Чем больше информации о содержании кода извлекается при анализе, тем более точно можно указать на ошибку и тем больше возможных особенностей кода требуется учесть при создании анализатора для правильной

инструментации. К таким особенностям может относиться язык программирования в случае инструментации исходного кода или архитектура процессора при исследовании бинарного кода. Символьная интерпретация требует больше времени на один запуск по сравнению с обычным фаззингом. Несмотря на это, при совместной работе своевременный обмен входными данными между инструментами способен повышать общую эффективность анализа [5]. Данный подход называется гибридным фаззингом.

2. Создание абстрактной модели программы в динамической символьной интерпретации

В данной работе исследуется применение гибридного фаззинга для анализа бинарного кода архитектур Байкал-М и RISC-V. Как можно заметить, динамическая символьная интерпретация представляет собой достаточно низкоуровневый подход. Одним из важных аспектов является переносимость, то есть возможность анализа программ, разработанных для различных аппаратных платформ. Инструменты отличаются по требованию к наличию исходного кода и выбору представления, для которого формулируются символьные преобразования входных данных. Помимо этого, существуют различные варианты применения конкретного запуска в процессе анализа.

При наличии исходного кода возможна компиляция в универсальное промежуточное представление, при создании которого архитектурно-зависимые компоненты будут оставлены за скобками за счет специфически модифицированного компилятора. Похожий подход применим также непосредственно к бинарному коду, что более трудоёмко из-за частичных потерь информации на этапе компиляции. Однако доступ к исходному коду программы есть не всегда, как, например, в случае использования сторонних динамических библиотек. Кроме того, интеграция анализатора, требующего доступ к исходному коду, предполагает внесение изменений либо непосредственно в код, либо затрагивает технологический стек в виде компилятора и систем сборки ПО, что также вносит ограничения на удобство и универсальность применения инструмента. Таким образом, в контексте динамической символьной интерпретации требование к наличию исходного кода выглядит избыточным, поэтому далее главным образом будут рассмотрены методы инструментации, применимые на уровне бинарного кода.

Наиболее распространенным подходом к динамической символьной интерпретации является так называемое конкретно-символьное исполнение, когда непосредственный запуск исследуемой программы комбинируется с символьной интерпретацией для получения более точной абстрактной модели исполнения программы. Первым инструментом с реализацией такого подхода для бинарного кода стал SAGE [4], где сначала производится конкретный запуск программы для записи трассы исполнения, а затем её преобразование в символьные ограничения. В современных инструментах конкретно-символьного исполнения эти два этапа совмещаются с помощью динамической инструментации, которая предоставляет текущий контекст конкретного исполнения программы (исполняемые инструкции, значения памяти, регистров и флагов) для символьной интерпретации. Например, QSYM [6] и Sydr [7] используют фреймворки динамической бинарной инструментации и выполняют символьную интерпретацию на уровне отдельных инструкций бинарного кода. Для них поддержка каждой новой платформы требует добавления символьного представления для анализируемого набора инструкций выбранной архитектуры. На уровне машинных инструкций, но без реального запуска проходит символьная интерпретация в TritonDSE [8] – инструмент эмулирует бинарный код для пошагового выполнения. При этом, запуск может присутствовать для верификации или сбора покрытия, как и в виртуальной символьной машине KLEE [9], эмулирующей инструкции байт-кода LLVM. Преимущество инструментации промежуточного представления также используется в SymQEMU [10], где перед запуском бинарный код транслируется с помощью QEMU в TCG ops и обратно.

Символьная инструментация производится на уровне этого представления, позволяя абстрагировать инструмент от деталей реализации отдельных аппаратных платформ.

Данная работа по применению символьного анализа бинарного кода архитектур Байкал-М (ARM/AArch64) и RISC-V в гибридном фаззинге выполнялась на основе динамического символьного интерпретатора Sydr, разработанного в ИСП РАН. Инструмент Sydr реализует конкретно-символьное исполнение бинарного кода архитектуры x86/x86-64 с использованием динамического бинарного инструментатора DynamoRIO и библиотеки символьного исполнения Triton.

3. Символьная интерпретация бинарного кода архитектуры Байкал-М (ARM/AArch64)

В предлагаемом методе для инструмента Sydr моделирование символьного контекста выполнения производится для целочисленных инструкций ARM/AArch64. Обработка операций с плавающей точкой при символьной интерпретации программы порождает довольно сложные символьные уравнения, для которых SMT-решатель обычно не дает решение за установленный лимит времени. Соответственно, для архитектуры AArch64 в предлагаемом методе применяется символьная интерпретация на основе следующих компонентов:

- набор регистров общего назначения, включающий регистры x0-x30 размером 64 бита, а также соответствующие им подрегистры w0-w30 размером 32 бита;
- набор специальных регистров, включающих регистр указателя текущей инструкции (pc), регистр стекового указателя (sp), регистр флагов, а также нулевые регистры xzr и wzr;
- операционная семантика инструкций с целочисленными операндами;
- механизмы вычисления адресных выражений;
- механизмы передачи управления и соглашение о вызовах.

Обновление символьного контекста происходит по мере обработки потока инструкций, выполняющихся под контролем DynamoRIO. Получив от инструментатора текущую инструкцию вместе с конкретными значениями, процесс обработки символьного представления дизассемблирует её операционный код и производит набор проверок на наличие символьных операндов и необходимость обновления символьного стека вызовов. Для AArch64 инструкциями вызова функций являются BL и BLR, а для возврата из функции используется RET, т.е. инструкция BR x30, которая передает управление на инструкцию, адрес которой хранится в регистре x30 (Procedure Link Register). Отслеживание данных инструкций необходимо для актуализации символьной модели памяти стека. Иначе при последующих операциях чтения или сохранения символьных данных в стековой памяти интерпретатор обнаружит несоответствие и подставит значение конкретных данных, то есть символьная пометка будет потеряна.

Для архитектуры ARM/AArch64 инструкции обращения к памяти (load/store) выделяются в отдельную группу. Размеры области памяти, к которой может обращаться инструкция, определяются семантикой и составляют 1, 2, 4 или 8 байтов. При реализации предлагаемого метода в абстрактном интерфейсе MemoryAccess библиотеки Triton была исправлена ошибка присвоения значения области памяти размером, тождественным размеру регистра для её временного хранения (4 либо 8 байтов). Среди прочего, для механизма адресации с индексированием в ARM/AArch64 присутствует несколько необычный формат применения коэффициента масштабирования к индексному регистру. Обычно данный коэффициент представляет собой константный числовой множитель. Например, 8 в выражении «Addr = Reg1 + Reg2 * 8». В AArch64 вместо него предусмотрена возможность применения операции

битового сдвига влево (LSL), которая может быть совмещена с операцией расширения индексного регистра (см. рис. 1).

Операция сдвига эквивалентна, по сути, умножению на степени двойки от 0 до 3, позволяя адресовать блоки памяти размером от 1 до 8 байтов. Она также может участвовать в адресации косвенных табличных переходов [11], в том числе, в контексте арифметической операции (например, инструкции сложения из последней строки на рис. 1), выполняемой уже после загрузки значения из памяти. Предлагаемый метод находит такие арифметические преобразования между инструкцией чтения из памяти и передачи управления с помощью алгоритма отслеживания регистров (слайсинга), на основе которого производится коррекция адресного выражения с учетом возможности битовых сдвигов. Само адресное выражение ветвления с несколькими вариантами перехода добавляется в предикат пути позднее, при появлении соответствующей инструкции передачи управления.

```
Инструкция                    Выполняемая операция
ldr  x0, [x1, x2, lsl 3]      x0 := Mem[x1 + x2 * 8]
ldr  x0, [x1, w2, uxtx 3]     x0 := Mem[x1 + ext(w2) * 8]
ldrb w0, [x1, w2, uxtx 3]     w0 := ext(Mem[x1 + x2 * 8])

add x0, x1, x2, lsl 3         x0 := x1 + x2 * 8
```

Рис. 1. Примеры применения битовых сдвигов к операндам инструкций AArch64.
Условные обозначения: := – операция присваивания значения, ext – операция расширения размера, Mem[addr] – значение памяти по адресу addr.

При конструировании символьных предикатов безопасности [12], проверяющих возможность ошибки целочисленного переполнения, учитываются как знаковые, так и беззнаковые переполнения для инструкций сложения, вычитания, умножения, а также битовых сдвигов. Для этого производится развертывание ACT-выражения инструкции, и из него извлекаются подвыражения, соответствующие операндам инструкции. К примеру, выражение инструкции вычитания с битом переноса содержит две операции сложения битовых векторов, где значение исходного операнда вычитаемого будет взято с отрицанием в виде дополнительного узла в дереве. Поскольку инструкция фактически включает две арифметические операции, допускающие возможность переполнения, будет сконструировано два предиката безопасности.

3.1 Метод обнаружения косвенных переходов в бинарном коде

Ветвления с множественным выбором (в виде конструкции switch) предполагают наличие нескольких вариантов выбора цели передачи управления. При использовании оптимизаций компилятора такие переходы производятся с помощью организации целевых вариантов исполняемого кода в виде таблицы, состоящей из последовательно расположенных базовых блоков, соответствующих различным направлениям ветвления. Целевые адреса из такой таблицы исполняемого кода последовательно хранятся в памяти, а выбор нужной ячейки памяти происходит посредством добавления смещения к базовому адресу, где располагается начало таблицы указателей на исполняемый код. На рис. 2 выделен светлой областью базовый блок, в котором происходит чтение адреса целевого перехода из памяти и передача управления, которая в данном случае представлена вызовом функции и осуществляется с помощью инструкции BLR x8. Как можно заметить, для вычисления базового адреса таблицы здесь используются три различные константы (0x420000, 0x28 и 0x180), а также значение регистра x8, которое перед попаданием в данный базовый блок было прочитано из памяти в инструкции LDRB по адресу 0x4006c0. Поскольку первые восемь регистров x0-x7 в AArch64

используются для хранения аргументов в соответствии с соглашением о вызовах, при взгляде на предыдущую инструкцию можно также догадаться, что в регистре x8 представлено потенциально символьное значение, указатель на адрес которого передается в качестве аргумента функции *main*. Таким образом, в зависимости от данного аргумента будет осуществлен вызов одной из функций *func0-func5*.

```
0000000000400664 <func5>:
  400664:       90000000        adrp    x0, 400000 <_init-0x4e0>
  400668:       911ee000        add     x0, x0, #0x7b8
  40066c:       17ffffb9        b       400550 <puts@plt>

0000000000400670 <func4>:
  400670:       90000000        adrp    x0, 400000 <_init-0x4e0>
  400674:       911f0c00        add     x0, x0, #0x7c3
  400678:       17ffffb6        b       400550 <puts@plt>

000000000040067c <func3>:
  40067c:       90000000        adrp    x0, 400000 <_init-0x4e0>
  400680:       911f3800        add     x0, x0, #0x7ce
  400684:       17ffffb3        b       400550 <puts@plt>

0000000000400688 <func2>:
  400688:       90000000        adrp    x0, 400000 <_init-0x4e0>
  40068c:       911f6400        add     x0, x0, #0x7d9
  400690:       17ffffb0        b       400550 <puts@plt>

0000000000400694 <func1>:
  400694:       90000000        adrp    x0, 400000 <_init-0x4e0>
  400698:       911f9000        add     x0, x0, #0x7e4
  40069c:       17ffffad        b       400550 <puts@plt>

00000000004006a0 <func0>:
  4006a0:       90000000        adrp    x0, 400000 <_init-0x4e0>
  4006a4:       911fbc00        add     x0, x0, #0x7ef
  4006a8:       17ffffaa        b       400550 <puts@plt>

00000000004006ac <main>:
  4006ac:       a9bf7bfd        stp     x29, x30, [sp, #-16]!
  4006b0:       910003fd        mov     x29, sp
  4006b4:       7100081f        cmp     w0, #0x2
  4006b8:       54000101        b.ne    4006d8 <main+0x2c>  // b.any
  4006bc:       f9400428        ldr     x8, [x1, #8]
  4006c0:       39400108        ldrb    w8, [x8]
  4006c4:       f100bd1f        cmp     x8, #0x2f
  4006c8:       54000108        b.hi    4006e8 <main+0x3c>  // b.pmore
  4006cc:       90000000        adrp    x0, 400000 <_init-0x4e0>
  4006d0:       911fe800        add     x0, x0, #0x7fa
  4006d4:       14000003        b       4006e0 <main+0x34>
  4006d8:       90000000        adrp    x0, 400000 <_init-0x4e0>
  4006dc:       91203400        add     x0, x0, #0x80d
  4006e0:       97ffff9c        bl      400550 <puts@plt>
  4006e4:       14000007        b       400700 <main+0x54>
  4006e8:       90000109        adrp    x9, 420000 <__libc_start_main@GLIBC_2.17>
  4006ec:       9100a129        add     x9, x9, #0x28
  4006f0:       8b080d28        add     x8, x9, x8, lsl #3
  4006f4:       d1060108        sub     x8, x8, #0x180
  4006f8:       f9400108        ldr     x8, [x8]
  4006fc:       d63f0100        blr     x8
  400700:       2a1f03e0        mov     w0, wzr
  400704:       a8c17bfd        ldp     x29, x30, [sp], #16
  400708:       d65f03c0        ret
```

Рис. 2. Пример организации таблицы переходов в ассемблерном коде.

Детектирование таких таблиц в предлагаемом методе происходит на основе эвристики в виде присутствия в базовом блоке следующих составляющих:

- в качестве последней инструкции используется безусловная инструкция передачи управления по регистровому значению;
- соответствующий регистр хранит значение, прочитанное из памяти, либо является его производным;
- в базовом блоке присутствует одна из инструкций загрузки фиксированного адресного значения ADR/ADRP.

Описанная конструкция может быть дополнительно усложнена, если вместо целевого адреса перехода требуется прочитать из памяти смещение относительно начала таблицы в исполняемом коде. Таким образом, адрес в таблице с исполняемыми базовыми блоками также разбивается на базовый адрес её начала и смещение. В свою очередь, хранение значений отступов аналогично организуется в виде хранящейся в памяти таблицы смещений, для которой осуществляется операция чтения с помощью инструкции загрузки данных (load) тоже на основе индексируемого базового адреса. Размеры блоков памяти для хранения отступов составляют от 1 до 8 байтов. Поскольку ресурс значения смещения для кодирования номера инструкции или записи, отвечающей за конкретный базовый блок, ограничен, при добавлении значения смещения в адресное выражение, оно умножается на соответствующий размер. Как было упомянуто в ранее, в случае архитектуры AArch64 для подобных вычислений используется встроенный механизм арифметических инструкций в виде операции битового сдвига на константное значение.

На рис. 3 представлен базовый блок с передачей управления, основанной на вычислении целевого адреса с помощью отступа, прочитанного из таблицы смещений. Первые две инструкции содержат константы для определения базового адреса таблицы отступов, а третья загружает адрес начала таблицы в исходном коде. Благодаря тому, что присутствие таблицы смещений обуславливает наличие двух базовых адресов вместо одного, можно отличить данный вариант организации таблицы от предыдущего (как на рис. 2) по второй инструкции загрузки адресного значения.

```
4006c0:    54000241    b.ne    400708 <main+0x54>  // b.any
4006c4:    f9400428    ldr     x8, [x1, #8]
4006c8:    39400108    ldrb    w8, [x8]
4006cc:    51018908    sub     w8, w8, #0x62
4006d0:    7100211f    cmp     w8, #0x8
4006d4:    54000428    b.hi    400758 <main+0xa4>  // b.pmore
4006d8:    90000009    adrp    x9, 400000 <_init-0x520>
4006dc:    91206129    add     x9, x9, #0x818
4006e0:    1000008a    adr     x10, 4006f0 <main+0x3c>
4006e4:    3868692b    ldrb    w11, [x9, x8]
4006e8:    8b0b094a    add     x10, x10, x11, lsl #2
4006ec:    d61f0140    br      x10
4006f0:    90000000    adrp    x0, 400000 <_init-0x520>
```

Рис. 3. Базовый блок косвенной передачи управления на основе таблицы смещений.

Для составления предиката пути, способного учитывать все доступные варианты, требуется реализовать не только метод распознавания подобных конструкций, но и определения размеров таблицы. Предлагаемый метод определения границ таблицы переходов основан на поиске инструкции сравнения *CMP* в предыдущем логическом базовом блоке. Для этого применяется механизм кэширования выполненных базовых блоков. Путем извлечения константного значения для инструкции сравнения определяются целевые ячейки читаемой памяти. Для их содержимого вычисляются адреса возможных переходов, которые верифицируются на принадлежность к областям исполняемого кода.

4. Символьная интерпретация бинарного кода архитектуры RISC-V

Открытая процессорная архитектура RISC-V появилась сравнительно недавно. Особенностями данного архитектурного стандарта являются свободная лицензия, модульность и минималистичность набора инструкций, что делает его применимым для широкого класса приложений.

4.1 Конструирование символьной семантики целочисленных инструкций RISC-V

Предлагаемый метод реализован на базе фреймворка Triton [13]. Метод способен конструировать в терминах битовых векторов символьные AST-выражения, отображающие операционную семантику инструкций RISC-V, для которых осуществляется символьная интерпретация. Соответствующие функциональные методы и генерируемые ими символьные выражения в данной работе также называются "символьной семантикой".

Набор инструкций архитектуры RISC-V имеет модульную структуру из подключаемых расширений, обозначаемых латинскими буквами. К базовому I-расширению относятся инструкции передачи управления, обращений к памяти, основные арифметические операции (кроме принадлежащих M-расширению инструкций умножения и деления). Помимо того, в базовом наборе содержатся инструкции организации прерываний и барьеров памяти (fence-инструкции), а также инструкции для доступа к служебному регистру CSR, но эти группы инструкций не включаются в предлагаемый метод. Значительная часть набора интерпретируемых методом 64-битных инструкций представляет собой пересечение с 32-битной версией архитектуры, отличаясь только размером используемых регистровых операндов и адресов. По данной причине, метод охватывает как набор модульных расширений RV64IMC, так и RV32IMC, где буквой C обозначается набор сокращенных (compressed) инструкций.

Для представления выбранной архитектуры с помощью инфраструктуры абстрактных интерфейсов Triton были добавлены соответствующие спецификации, позволяющие организовывать взаимодействие с внешним дизассемблером Capstone и производить моделирование символьного состояния программы. После создания спецификаций и идентификаторов в предложенном методе были реализованы методы интерфейсов riscv64Cpu и riscv32Cpu, с помощью которых обрабатываются архитектурно-зависимые компоненты на основе абстрактных универсальных классов (например, triton::arch::Register). Данные интерфейсы позволяют получать и обновлять символьные выражения и конкретные значения регистров и областей памяти, осуществлять доступ к регистрам служебного назначения (sp, pc), а также содержат функции для преобразования контекста дизассемблированных посредством Capstone инструкций в абстрактный класс Triton triton::arch::Instruction.

На этапе конструирования символьной семантики с помощью идентификатора инструкции выбирается соответствующий функциональный метод. В первую очередь, он извлекает операнды инструкции и получает для них битвекторные выражения символьного контекста выполнения. Таким выражением может быть как ранее построенное абстрактное дерево, так и битовый вектор, соответствующий значению константы для заданного размера операнда. Далее для полученных выражений в соответствии с операционной семантикой инструкции строится новое выражение, содержащее их в виде поддеревьев. Сконструированное выражение ассоциируется с целевым операндом, после чего увеличивается символьное значение счётчика инструкций, если иное не предусмотрено семантикой инструкции. Предлагаемый метод обеспечивает для архитектуры RISC-V формирование семантических выражений в терминах битовых векторов для 62 инструкций стандартного размера, их 19 дополнительных вариаций в виде псевдоинструкций, а также 33 сокращенных инструкций.

Псевдоинструкции представляют собой альтернативное прочтение стандартных инструкций при использовании специфических комбинаций операндов. Чаще всего в качестве такого операнда выступает всегда равный нулю регистр x0. Также используется регистр x1 и константные операнды. Одна и та же инструкция может иметь несколько вариаций в виде псевдоинструкций в зависимости от выбора операндов. Например, инструкция передачи управления JALR rd, rs, offset имеет три варианта псевдоинструкций, для каждой из которых значение численного операнда offset приравниваются к нулю, в то время как rd обязательно принимает значение x0 либо x1. В случае комбинации JALR x0, x1, 0 получается инструкция возврата из вызова RET. Из-за особенностей работы Capstone, поля предопределяющих семантику операндов не заполняются при дизассемблировании. К примеру, абстрактный класс, отображающий псевдоинструкцию RET не имеет ни одного операнда. С точки зрения построения символьной семантики, это означает, что перед получением битвекторных выражений необходимо определить, является ли интерпретируемая инструкция стандартной или представляет собой псевдоинструкцию. Для этого в предлагаемом методе применяется разбор на основе количества операндов и текстового описания инструкции.

Сокращенные инструкции образуют набор базовых операций в виде передачи управления, обращений к памяти и простых арифметических операций. Основное их отличие заключается в том, что на их кодирование отводится 16 бит вместо 32, что следует учитывать при обновлении символьного значения счетчика инструкций. Кроме того, из-за отсутствия корректной трансляции для инструкций обращения к памяти при разработке предлагаемого метода создание операнда класса MemoryAccess для таких инструкций производится в соответствующем функциональном методе для создания символьной семантики с помощью регистровых операндов.

4.2 Динамическая символьная интерпретация бинарного кода архитектуры RISC-V 64

Разработанный метод динамической символьной интерпретации опирается на метод конструирования символьной семантики инструкций RISC-V 64, описанный в предыдущем подразделе. Для моделирования символьного контекста архитектуры RISC-V 64 с помощью предлагаемого метода для бинарного кода задействованы следующие архитектурные компоненты:

- набор 64-битных регистров общего назначения x0-x31, среди которых имеют служебное назначение регистр x1 (регистр возврата) и x2 (sp);
- указатель счетчика инструкций (pc);
- операционная семантика инструкций с целочисленными операндами;
- механизмы передачи управления, соглашения о вызовах и режимы адресации.

В отличие от AArch64 инструкции архитектуры RISC-V обладают свойством лаконичности производимых операций. При этом сходство прослеживается в разграничении инструкций обращения к памяти и арифметических операций. В RISC-V не используются флаговые регистры, но, тем не менее, выполнение условных переходов возможно для набора соответствующих инструкций передачи управления после сравнения их операндов. Кроме того, для правильного моделирования потока управления и корректной обработки символьной памяти стека необходимо детектировать псевдоинструкции для инструкций JAL и JALR.

Наиболее значимое отличие наблюдается для косвенных ветвлений в результате присутствия для архитектуры RISC-V технического ограничения в работе используемого инструментатора DynamoRIO. Для AArch64 задача распознавания косвенных табличных переходов решается на уровне анализа базового блока, полностью загружаемого инструментатором перед обработкой входящих в него инструкций. Однако в ходе работы над

методом интерпретации бинарного кода RISC-V 64 оказалось, что фактически при появлении базового блока размером превышающем 5 инструкций, инструментатор разбивает его на несколько блоков. При этом из-за минималистичности, присущей компактному набору инструкций данной архитектуры, базовые блоки в среднем имеют более крупный размер по сравнению, например, с AArch64 и, тем более x86. Таким образом, базовые блоки, содержащие неявные табличные переходы могут быть разбиты на 2 или 3 части, каждая из которых содержит не больше 5 инструкций. Предикат же для инвертирования табличных переходов создается при обработке инструкции загрузки данных (load), но добавляется к общему предикату пути в процессе интерпретации инструкции передачи управления. Если две эти связанные инструкции попадают при разбиении в различные блоки, то адрес перехода заранее неизвестен. Поэтому вместо него записывается фиктивный адрес, а именно адрес целевой инструкции загрузки данных, увеличенный на единицу. Этот адрес является нечетным, так что из-за выравнивания он не сможет совпасть к каким-либо другим адресом инструкции, для которой не требуется добавление предиката, даже если это будет сокращенная инструкция. По нему инструкция перехода из другой части базового блока сможет определить нужный предикат.

Подробное объяснение механизма работы таблиц переходов с чтением из памяти как целевых адресов в исходном коде, так и относительных смещений приведено в аналогичном разделе для AArch64. Для архитектуры RISC-V 64 о присутствии в базовом блоке табличного перехода свидетельствуют следующие три составляющие:

- паттерн из трёх инструкций, которые могут располагаться в различном порядке: auipc – операция загрузки адресного значения, операция битового сдвига (например, slli), а также одна из инструкций сложения;
- инструкция чтения из памяти, которая располагается после всех трех инструкций, образующих паттерн;
- инструкция передачи управления для регистрового значения.

На рис. 4а-4в представлены варианты содержащих табличные переходы базовых блоков, которые будут подвергнуты разным разбиениям с точки зрения необходимых составляющих. На основе эвристических наблюдений можно сказать, что три образующие паттерн инструкции попадают в первый блок, в который также может попасть целевая инструкция чтения из памяти.

В следствие возможности разбиения базового блока на три части, каждая из которых содержит только одну из составляющих, характеризующих наличие косвенного ветвления, при появлении нового блока инструкций статус стадии поиска табличного перехода может быть различным. Для его хранения используется специально выделенная область памяти, которая проверяется и заполняется по мере появления составляющих.

В ситуации, когда производится первичное сканирование сначала проверяется, что последняя инструкция блока не относится к условным инструкциям передачи управления. Далее с конца осуществляется поиск последней инструкции загрузки данных (load) в блоке, при появлении которой будет использован статус хранилища относительно наличия паттерна в предыдущем блоке. Для первичного сканирования, при котором текущий инспектируемый блок действительно является началом базового блока, очевидно, статуса присутствия паттерна быть не может, поэтому требуется далее искать комбинацию из трех инструкций, что происходит и при отсутствии инструкции чтения в текущем блоке. При обнаружении данного паттерна сохраняется местоположение последней входящей в него инструкции для возможного сравнения с адресом найденной инструкции чтения. Это требуется для исключения случаев, когда в том же блоке разбиения присутствует нецелевая инструкция обращения к памяти, как на рис. 4а и рис. 4в. Если оба компонента успешно найдены, но последняя инструкция не передает управление по значению регистра (и, соответственно, вообще не является инструкцией передачи управления), то соответствующий статус вместе с

фиктивным адресом целевой инструкции чтения сохраняется в хранилище. При сканировании следующей части базового блока в результате обращения к хранилищу, которое происходит перед поиском инструкции чтения, будет возвращен адрес для добавления предиката. Если при первичном сканировании был найден только паттерн, то информация об этом будет также сохранена. Для разбиения на три части обновление статуса хранилища будет произведено дважды. При получении нужного фиктивного адреса целевой инструкцией передачи управления в символьном исполнителе будет заново создан предикат с правильным адресом.

```
6f6:   47a5              li      a5,9
6f8:   0ae7e463          bltu    a5,a4,7a0 <main+0xfc>
6fc:   fec46783          lwu     a5,-20(s0)
700:   00279713          slli    a4,a5,0x2
704:   00000797          auipc   a5,0x0
708:   19c78793          addi    a5,a5,412 # 8a0 <_IO_stdin_used+0x78>
70c:   97ba              add     a5,a5,a4
70e:   439c              lw      a5,0(a5)
710:   0007871b          sext.w  a4,a5
714:   00000797          auipc   a5,0x0
718:   18c78793          addi    a5,a5,396 # 8a0 <_IO_stdin_used+0x78>
71c:   97ba              add     a5,a5,a4
71e:   8782              jr      a5
720:   4785              li      a5,1
```

Рис. 4а. Базовый блок с разбиением на 3 части.

```
718:   12b00713          li      a4,299
71c:   00d77463          bgeu    a4,a3,724 <main+0x80>
720:   33f0106f          j       225e <main+0x1bba>
724:   00279713          slli    a4,a5,0x2
728:   00003797          auipc   a5,0x3
72c:   ea078793          addi    a5,a5,-352 # 35c8 <_IO_stdin_used+0x12e0>
730:   97ba              add     a5,a5,a4
732:   439c              lw      a5,0(a5)
734:   0007871b          sext.w  a4,a5
738:   00003797          auipc   a5,0x3
73c:   e9078793          addi    a5,a5,-368 # 35c8 <_IO_stdin_used+0x12e0>
740:   97ba              add     a5,a5,a4
742:   8782              jr      a5
744:   fe042223          sw      zero,-28(s0)
```

Рис. 4б. Базовый блок с разбиением, отделяющим инструкцию перехода.

```
800:   4795              li      a5,5
802:   00e7da63          bge     a5,a4,816 <main+0x6e>
806:   00000517          auipc   a0,0x0
80a:   10a50513          addi    a0,a0,266 # 910 <_IO_stdin_used+0x80>
80e:   e13ff0ef          jal     ra,620 <puts@plt>
812:   4781              li      a5,0
814:   a821              j       82c <main+0x84>
816:   00001717          auipc   a4,0x1
81a:   7f270713          addi    a4,a4,2034 # 2008 <jump_table>
81e:   fec42783          lw      a5,-20(s0)
822:   078e              slli    a5,a5,0x3
824:   97ba              add     a5,a5,a4
826:   639c              ld      a5,0(a5)
828:   9782              jalr    a5
```

Рис. 4в. Базовый блок с разбиением, отделяющим блок с паттерном и нецелевой load-инструкцией от блока с целевой load-инструкцией и инструкцией передачи управления.

Определение размеров таблицы переходов для RISC-V 64 производится с помощью выбора максимального из сравниваемых аргументов инструкции условного выполнения, с помощью которой поток управления попал в блок, содержащий паттерн. Для выделенных возможных целевых адресов передачи управления также проверяется, что они относятся к исполняемому коду.

5. Экспериментальная оценка предлагаемых методов динамической символьной интерпретации

5.1 Байкал-М (ARM/AArch64)

Для апробации предложенного метода динамической символьной интерпретации были разработаны тестовые наборы на основе синтетических тестов и реальных приложений Байкал-М (AArch64). Метод был реализован на базе инструмента Sydr. Наиболее близким к нему по принципу работы и набору поддерживаемых символьных техник анализа среди открытых современных аналогов, поддерживающих архитектуру ARM/AArch64, можно назвать динамический интерпретатор SymQEMU. С целью оценки разработанного метода было проведено экспериментальное сравнение двух данных символьных анализаторов. Сравнение проводилось с помощью набора открытых приложений, запускаемых на процессоре Байкал-М. В качестве ограничения времени для каждого приложения был использован лимит в 20 минут. Результаты сравнения представлены в табл. 1.

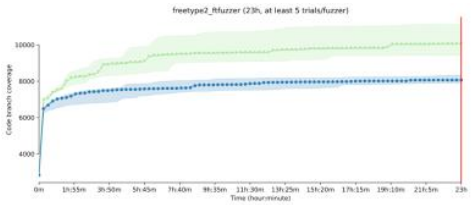
Табл. 1. Сравнение динамических символьных интерпретаторов Sydr и SymQEMU для набора приложений архитектуры AArch64.

Приложение	Sydr				SymQEMU			
	Покрытие	Уник.	Вх.файлы	Время	Покрытие	Уник.	Вх.файлы	Время
freetype2	**98.93%**	227	1645	20m	97.99%	120	1287	20m
jsoncpp	**100%**	176	907	11,4s	75.35%	0	193	4,2s
lcms	**99.78%**	5	103	9,7s	99.64%	3	422	16m35s
libpng	**97.60%**	76	317	20m	97.47%	72	734	20m
libxml2	96.94%	12	971	20m	**99.85%**	249	1736	20m
openthread	**99.96%**	12	285	1m26s	99.89%	5	486	2m51s
re2	**93.37%**	219	47	5,6s	89.64%	140	81	36,1s
woff2	**100%**	163	239	20m	93.52%	0	408	20m

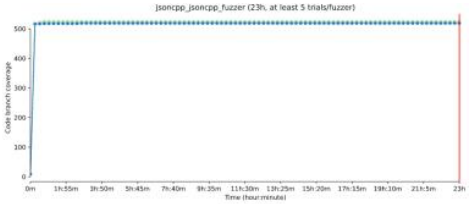
Результаты работы для каждого инструмента включают количество успешно сгенерированнных тестов (обозначенных как "Вх. Файлы"), а также число уникальных строк покрытия, достигаемого для всего набора тестов (столбец "Уник."), которые были получены от обоих инструментов. Процент метрики покрытия считается для тестовых данных отдельного инструмента относительно общего достигнутого покрытия. Как можно заметить, для подавляющего большинства проектов (за исключением libxml2), предлагаемый метод символьной интепретации продемонстрировал лучшие результаты.

Более того, метод успешно продемонстрировал свою эффективность для применения в контексте гибридного фаззинг-тестирования. Результаты в виде графиков метрики покрытия представлены на рис. 5. При проведении сравнения использовалась парная работа динамического символьного интерпретатора Sydr с инструментом фаззинга с обратной связью по покрытию (на графиках отмечены зеленым цветом). Пара состязалась с тем же фаззером, работающим в два потока (синий цвет). При осуществлении экспериментального сравнения для гибридного фаззинга применялась инфраструктура оценки фаззинг-инструментов FuzzBench. Ограничение времени, отведенного на фаззинг-тестирование, составляет 23 часа. По результатам для двух различных инструментов фаззинга libFuzzer и AFL++ (графики для которых на изображении относятся к левому и правому столбцам, соответственно), применение гибридного фаззинга вместо масштабирования приводило к увеличению достигнутой метрики покрытия более чем для половины исследуемых приложений.

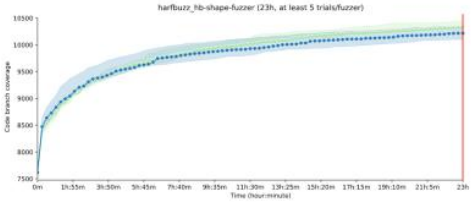
К факторам оценки работоспособности предложенного метода можно также отнести несколько выявленных и исправленных ошибок в интерфейсах символьного фреймворка Triton для работы с архитектурой ARM/AArch64.



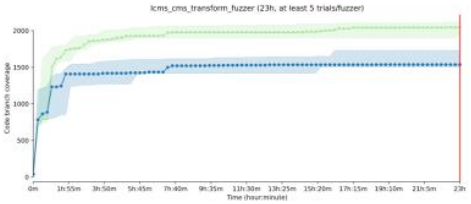
(a) freetype2



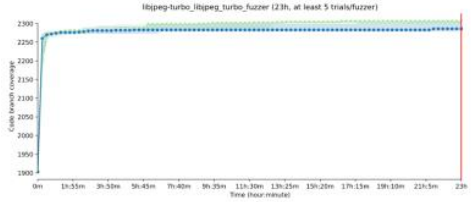
(a) jsoncpp



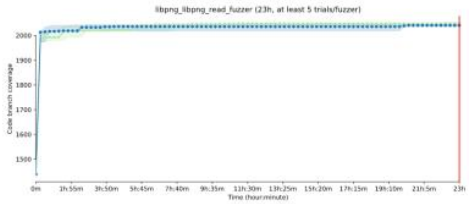
(b) harfbuzz



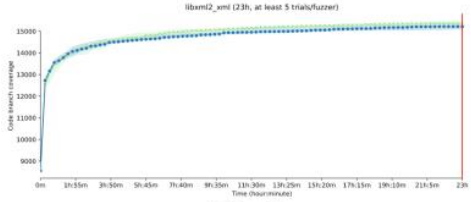
(b) lcms



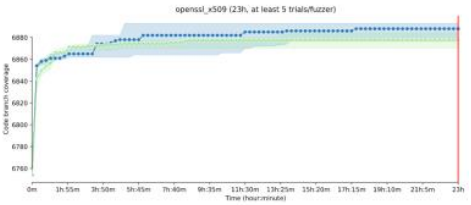
(c) libjpeg_turbo



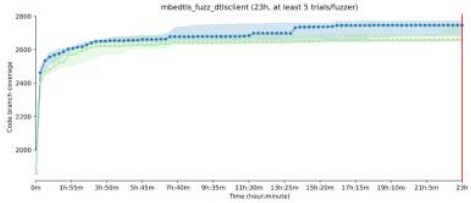
(c) libpng



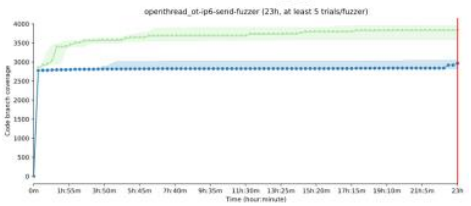
(d) libxml2



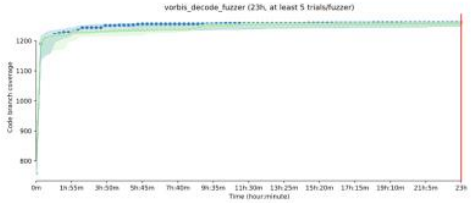
(d) openssl



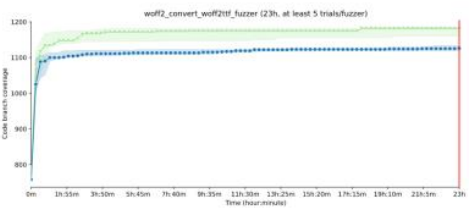
(e) mbedtls



(e) openthread



(f) vorbis



(f) woff2

Рис. 5. Графики достигнутых метрик покрытия FuzzBench.

5.2 RISC-V 64

Для оценки работоспособности предложенного метода динамической символьной интерпретации, внедренного в инструмент Sydr, были созданы синтетические тесты и тестовый набор на основе реальных приложений RISC-V 64. Для данного метода также было проведено экспериментальное сравнение Sydr с динамическим интерпретатором SymQEMU.

Результаты сравнения представлены в табл. 2. Сравнение проводилось с помощью набора открытых приложений, запускаемых с помощью виртуальной машины, в связи с чем лимит времени для каждого приложения был увеличен до 60 минут. Метод проведения сравнения и условные обозначения для уникальных строк покрытия и процентного соотношения покрытия отдельного инструмента относительно общего достигнутого числа строк такие же как в предыдущем разделе. Из табл. 2 можно видеть, что для двух третей анализируемых приложений Sydr достигает лучших показателей достигнутого покрытия, а в остальных случаях показывает равные результаты для данной метрики.

Табл. 2. Сравнение динамических символьных интерпретаторов Sydr и SymQEMU для набора приложений архитектуры RISC-V 64.

Приложение	Sydr				SymQEMU			
	Покрытие	Уник.	Вх.файлы	Время	Покрытие	Уник.	Вх.файлы	Время
ffmpeg	**100%**	0	28	2m11s	100%	0	197	5m11s
freetype2	**98.99%**	445	2215	60m	96.09%	115	1042	60m
jsoncpp	**99.50%**	9	183	1m20s	98.88%	4	211	14s
lcms	**100%**	0	98	1m33s	100%	0	325	22m1s
openjpeg	**100%**	12	140	1m28s	99.77%	0	340	49m46s
openssl	**99.86%**	47	169	60m	99.46%	12	167	25m44s
re2	100%	0	8	1m31s	**100%**	0	4	5s
woff2	**99.82%**	111	475	57m27s	95.93%	5	645	60m
zlib	**100%**	698	160	45m16s	66.25%	0	77	41m58s

Сравнительный запуск для гибридного фаззинга посредством инфраструктуры FuzzBench не проводился по причине использования виртуальной машины. Для данной архитектуры было проведено сравнение в режиме ручного тестирования между гибридным фаззингом пары Sydr и libFuzzer, а также двух параллельно работающих процессов libFuzzer. В Таблице 3 для каждого проекта представлены усредненные значения достигнутого покрытия по строкам среди 5 запусков, на каждый из которых отводилось по 24 часа. Как можно заметить, для 3 из 4 проектов гибридный фаззинг достиг более высоких значений метрики покрытия.

Табл. 3. Усредненные значения метрики покрытия по строкам для гибридного фаззинга и фаззинга «методом серого ящика».

Проект	Sydr + libFuzzer	2 x libFuzzer
lcms	4527	4535,6
openjpeg	9257,4	9100,6
woff2	3170	2991,6
zlib	2246,4	2155,2

6. Заключение

В данной работе были представлены разработанные методы динамической символьной интерпретации бинарного кода Байкал-М (AArch64) и RISCV64, основанные на конструировании символьных выражений для операционной семантики машинных инструкций и применимые в контексте гибридного фаззинга. Данные методы были реализованы в инструменте Sydr, входящем в фреймворк динамического анализа Sydr-Fuzz [14]. Методы позволяют обнаруживать косвенные табличные переходы и определять соответствующие целевые адреса исполняемого кода. Помимо этого, был разработан метод

символьной интерпретации набора целочисленных инструкций архитектуры RISC-V, включающего сокращенные инструкции и псевдоинструкции. Указанный метод был реализован в открытом фреймворке символьной интерпретации Triton и может быть использован сообществом разработчиков для создания новых инструментов динамического анализа.

Список литературы / References

[1]. ГОСТ Р 58412-2019: Защита информации. Разработка безопасного программного обеспечения. Угрозы безопасности информации при разработке программного обеспечения. — Национальный стандарт РФ, 2019.

[2]. Serebryany, K. Continuous Fuzzing with libFuzzer and AddressSanitizer [Текст] / Kosta Serebryany // 2016 IEEE Cybersecurity Development (SecDev) / IEEE. 2016, с. 157.

[3]. Fioraldi, A. AFL++: Combining Incremental Steps of Fuzzing Research [Текст] / A. Fioraldi, D. Maier, H. Eißfeldt, M. Heuse // 14th USENIX Workshop on Offensive Technologies (WOOT 20). 2020, с. 10.

[4]. Molnar, D. Automated whitebox fuzz testing [Текст] / D. Molnar, P. Godefroid, M. Levin // Network and Distributed System Security Symposium, NDSS. 2008, с. 416-426.

[5]. FuzzBench (Google). DSE+Fuzzing Experiment Report. 2021. [Электронный ресурс]. — URL: https://www.fuzzbench.com/reports/experimental/2021-07-03-symbolic/index.html (доступ 23.09.2025).

[6]. Yun I. QSYM: A practical concolic execution engine tailored for hybrid fuzzing [Текст] / I. Yun [и др.] // 27th USENIX Security Symposium (USENIX Security 18). 2018, с. 745-761.

[7]. Vishnyakov, A. Sydr: Cutting edge dynamic symbolic execution [Текст] / A. Vishnyakov [и др.] // 2020 Ivannikov ISPRAS Open Conference (ISPRAS). IEEE. 2020, с. 46-54.

[8]. David, R. From source code to crash test-cases through software testing automation [Текст] / Robin David, Jonathan Salwan, Justin Bourroux // CESAR 2021: Automation in Cybersecurity. 2021.

[9]. Cadar C. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. [Текст] / C. Cadar, D. Dunbar, D. R. Engler [и др.] // OSDI. Т. 8. 2008, с. 209-224.

[10]. Poeplau, S. SymQEMU: Compilation-based symbolic execution for binaries. [Текст] / S. Poeplau, A. Francillon // NDSS. 2021.

[11]. Kutz D.Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution [Текст] / D. Kuts // 2021 Ivannikov Memorial Workshop (IVMEM). IEEE. 2021, с. 42-49.

[12]. Vishnyakov A. Symbolic Security Predicates: Hunt Program Weaknesses [Текст] / A. Vishnyakov [и др.] // 2021 Ivannikov Ispras Open Conference (ISPRAS). IEEE. 2021, с. 76-85.

[13]. Saudel, F. Triton: A Dynamic Symbolic Execution Framework [Текст] / Florent Saudel, Jonathan Salwan // Symposium sur la sécurité des technologies de l'information et des communications. SSTIC. 2015, с. 31-54.

[14]. Vishnyakov, A. Sydr-Fuzz: Continuous Hybrid Fuzzing and Dynamic Analysis for Security Development Lifecycle [Текст]/ A. Vishnyakov, D. Kuts, V. Logunova, D. Parygina, E. Kobrin, G. Savidov, A. Fedotov // 2022 Ivannikov ISPRAS Open Conference (ISPRAS). IEEE, 2022, с. 111-123.

Информация об авторах / Information about authors

Влада Игоревна ЛОГУНОВА – сотрудник отдела компиляторных технологий Института системного программирования. Сфера научных интересов: динамический анализ, анализ бинарного кода, динамическая символьная интерпретация, гибридный фаззинг.

Vlada Igorevna LOGUNOVA – Research Fellow at the Department of Compiler Technologies at the Institute for System Programming. Research interests: dynamic analysis, binary code analysis, dynamic symbolic execution, hybrid fuzzing.