На правах рукописи

Логунова Влада Игоревна

Разработка методов гибридного фаззинга для приложений процессорных архитектур Байкал-М и RISC-V 64

Специальность 2.3.5 —
«Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

АВТОРЕФЕРАТ

диссертации на соискание ученой степени

кандидата технических наук

Москва – 2025

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институт системного программирования им. В. П. Иванникова Российской Академии Наук.

Научный руководитель:	**Гетьман Александр Игоревич,** кандидат физико-математических наук
Официальные оппоненты:	**Шабанов Борис Михайлович**, доктор технических наук, член-корр. РАН, руководитель отделения Федерального государственного бюджетного учреждения «Национальный исследовательский центр «Курчатовский институт»
	Маркин Дмитрий Олегович, кандидат технических наук, сотрудник ФГКВОУ ВО «Академия Федеральной службы охраны Российской Федерации»
Ведущая организация:	Федеральное государственное учреждение "Федеральный исследовательский центр "Информатика и управление" Российской академии наук" (ФИЦ ИУ РАН)

Защита диссертации состоится 11 декабря 2025 г. в 14:45 на заседании диссертационного совета 24.1.120.01 при Федеральном государственном бюджетном учреждении науки Институт системного программирования им. В.П. Иванникова Российской академии наук по адресу: 115035, г. Москва, Садовническая улица, 41, ст.2.

С диссертацией можно ознакомиться в библиотеке и на официальном сайте Федерального государственного бюджетного учреждения науки Институт системного программирования им. В.П. Иванникова РАН.

Автореферат разослан «__» _____ 2025 г.

Ученый секретарь
диссертационного совета 24.1.120.01
кандидат физико-математических наук Д. Ю. Турдаков

Общая характеристика работы

Актуальность темы. Современный мир с каждым годом становится всё более удобным и технологичным, и вместе с тем менее понятным обычному человеку. Значительное количество критичных операций выполняется с применением ПО, порождая риски, связанные с проявлениями программных дефектов. Поэтому передовые технологии предлагают использование концепции жизненного цикла разработки безопасного ПО. Данный подход направлен на всестороннее рассмотрение каждого этапа разработки ПО с точки зрения качества и безопасности и включает тестирование ПО посредством автоматизированных статических и динамических анализаторов. Динамический анализ выполняется на основе запуска исследуемой программы, позволяя снизить число ложных сообщений об обнаруженных ошибках по сравнению со статическими подходами.

В роли инструментов динамического анализа, как правило, выступают средства фаззинг-тестирования с обратной связью по покрытию, которые генерируют данные для передачи на вход программе. Ориентируясь на увеличение покрытия, такие инструменты производят множественные запуски для различных входных данных, на которых могут проявиться ошибки. Другой подход для генерации тестовых входных данных под названием *динамическая символьная интерпретация* опирается на моделирование возможных путей выполнения программы за счет подробного анализа её потока управления. В первую очередь, это логические условия ветвлений, зависящие от внешних 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. Из-за этого символьная интерпретация требует больше времени на один запуск по сравнению с классическим фаззингом. Однако увеличение производительности символьной интерпретации с помощью различных оптимизаций (предложенных в работах К. Кадара, И. Юна, С. Поплау, Л. Борзаккьелло) позволяет сочетать работу современных символьных анализаторов с инструментами фаззинга. Данный подход называется *гибридным фаззингом* и в среднем показывает лучшие результаты в

сравнении с масштабированием обычного фаззинг-тестирования благодаря повышению точности анализа (например, за счет применения предикатов безопасности и разрешения косвенной адресации, обсуждавшихся в диссертационных работах А. В. Вишнякова и Д. О. Куца, соответственно).

Помимо применимости в гибридном фаззинге важными аспектами при внедрении инструментов динамической символьной интерпретации в технологический стек разработки ПО являются отсутствие требования наличия исходного кода и переносимость. Большинство современных символьных анализаторов бинарного кода, применимых в контексте гибридного фаззинга, представлена инструментами для целевой платформы x86(_64). Тем не менее, энергоэффективные процессорные RISC-архитектуры также довольно распространены, а значит для них необходимы качественные средства своевременного выявления и устранения программных дефектов. Наиболее известной RISC-архитектурой можно назвать ARM, на основе которой разработан имеющий государственную лицензию процессор Байкал-М, использующий стандарт ARMv8/AArch64 и применимый в ответственных системах высокого доверия. Как и x86(_64), ARM относится к проприетарным архитектурам, в отличие от RISC-V, популярность которой основана на свободно распространяемой лицензии, чем активно пользуется, в том числе, академическое сообщество. Современные символьные интерпретаторы, для которых легко возможна или уже реализована поддержка ARM и RISC-V, опираются при инструментации на трансляцию программы в упрощенное промежуточное представление. За счет временных затрат на трансляцию, такой подход обеспечивает универсальность, в то время как альтернативный вариант инструментации на уровне машинных инструкций предлагает больше вариативности техник анализа бинарного кода. Например, применение различных видов упомянутых выше предикатов безопасности для целенаправленного поиска ошибок, а также разрешение косвенной адресации табличных переходов. Преимуществом такого подхода является отсутствие временных затрат на создание промежуточного представления, что положительно сказывается в контексте использования анализатора в гибридном фаззинг-тестировании. В то же время, такой символьный интерпретатор должен обеспечивать поддержку символьной семантики набора инструкций (ISA) и прочих компонентов для каждой из исследуемых архитектур.

Целью данной работы является разработка методов динамической символьной интерпретации бинарного кода архитектур Байкал-М (AArch64)

и RISC-V 64, обеспечивающих возможность анализа косвенной адресации и применимых в контексте гибридного фаззинга.

Методы должны поддерживать соответствующие архитектурно-зависимые компоненты для отслеживания преобразований символьного контекста выполнения программы с целью инвертирования как бинарных логических ветвлений, так и косвенных табличные переходы. Разработанные для указанных архитектур методы должны быть применимы к бинарным программам, работающим под управлением ОС Linux, и не зависеть от наличия исходных кодов тестируемых программ.

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

1. Разработать метод динамической символьной интерпретации бинарного кода программ архитектуры Байкал-М (AArch64).
2. Разработать метод символьной интерпретации набора целочисленных инструкций архитектуры RISC-V, включающего сокращенные инструкции и псевдоинструкции.
3. Разработать метод динамической символьной интерпретации бинарного кода программ архитектуры RISC-V 64.
4. Реализовать алгоритмы определения возможных направлений косвенных переходов для бинарного кода архитектур Байкал-М (AArch64) и RISC-V 64.
5. Реализовать разработанные методы и оценить их эффективность.

Научная новизна. В работе получены следующие результаты, обладающие научной новизной:

1. Разработан метод символьной интерпретации набора целочисленных инструкций архитектуры, RISC-V, включающего сокращенные инструкции и псевдоинструкции. Разработанный метод реализует символьную семантику инструкций, посредством конструирования SMT-формулы, описывающей преобразование символьного контекста на уровне выполнения отдельной инструкции RISC-V.
2. Разработанные методы динамической символьной интерпретации бинарного кода архитектур Байкал-М (AArch64) и RISC-V 64 предоставляют возможность точного определения границ для множественных условных переходов, что позволяет выявлять соответствующие целевые адреса исполняемого кода в процессе динамической символьной интерпретации и ускорять исследование новых путей выполнения при применении разработанных методов в контексте гибридного фаззинг-тестирования.

Теоретическая и практическая значимость. Теоретическая значимость работы заключается в разработанных методах динамической символьной интерпретации бинарного кода процессорных архитектур Байкал-М (AArch64) и RISC-V 64, а также методе символьной интерпретации набора целочисленных инструкций RISC-V на основе конструирования абстрактных синтаксических деревьев их операционной семантики. Кроме того, для вышеуказанных архитектур была проведена экспериментальная оценка эффективности применения методов динамической интерпретации в контексте фаззинг-тестирования.

Практическая значимость работы заключается в том, что предложенные методы анализа бинарного кода архитектур Байкал-М (AArch64) и RISC-V 64 были реализованы в разрабатываемом ИСП РАН инструменте динамической символьной интерпретации Sydr, входящем в фреймворк гибридного фаззинга Sydr-Fuzz. В рамках данных инструментов предложенные методы применяются в Центре доверенного искусственного интеллекта ИСП РАН, а также они были внедрены в процессы безопасной разработки ООО "Фобос-НТ" и ООО "Лаборатория безопасности". Метод символьной интерпретации набора целочисленных инструкций архитектуры RISC-V, включающего сокращенные инструкции и псевдоинструкции, был интегрирован в открытую символьную библиотеку Triton, доступную широкому кругу разработчиков для дальнейших изысканий в области символьного анализа программ и создании инструментов динамического анализа.

Методология и методы исследования. Результаты научно-квалификационной работы получены с использованием методов динамической символьной интерпретации программ. Математическую основу исследования составляют теория алгоритмов, теория множеств и математическая логика.

Основные положения, выносимые на защиту:

1. Метод динамической символьной интерпретации бинарного кода архитектуры Байкал-М (AArch64).
2. Метод символьной интерпретации набора целочисленных инструкций архитектуры RISC-V, включающего сокращенные инструкции и псевдоинструкции.
3. Метод динамической символьной интерпретации бинарного кода архитектуры RISC-V 64.

4. Реализация предложенных методов динамической символьной интерпретации в инструменте Sydr, входящем в фреймворк гибридного фаззинга Sydr-Fuzz.

Апробация работы. Основные результаты работы обсуждались на научно-практической конференции OS DAY в Москве в 2023 и 2025 гг., а также были представлены на Международной конференции «Иванниковские чтения» в Иркутске в 2025 г.

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

Публикации. По теме работы опубликовано 5 научных работ, получено 1 свидетельство о государственной регистрации программы для ЭВМ [1]. Три работы [2–4] изданы в периодических научных журналах с индексацией Web of Science и Scopus. Другие две из опубликованных статей [5–6] изданы в журналах, включенных в перечень рецензируемых научных изданий ВАК при Минобрнауки РФ. В работе [2] автором были описаны базовые принципы работы динамической символьной интерпретации и выполнен литературный обзор существующих инструментов. Также автором был разработан метод моделирования семантики функций стандартной библиотеки, представленный в работе [3]. В совместных работах [4, 6] автором было выполнено исследование существующих решений для реализации концепции непрерывного фаззинг-тестирования и описана последовательность применения инструментов для повышения эффективности динамического анализа с использованием гибридного фаззинга.

Объем и структура работы. Научно-квалификационная работа состоит из введения, 5 глав, и заключения. Полный объём работы составляет 117 страниц, включая 20 рисунков, 3 таблицы, 1 алгоритм и 3 приложения. Список литературы содержит 91 наименование.

Содержание работы

Во **введении** приводится обоснование актуальности исследований, проводимых в рамках данной диссертационной работы, ставятся цели и задачи работы, излагается научная новизна, теоретическая и практическая значимость представляемой работы, а также приводятся основные положения, выносимые на защиту.

Первая глава посвящена обзору предметной области и исследовательских работ по теме диссертации. Глава начинается с описания подходов для проведения динамической символьной интерпретации на основе эмуляции и конкретно-символьного исполнения бинарного кода. Далее раскрываются основные принципы работы, оптимизации и техники анализа символьных данных, а также приводится описание существующих инструментов в контексте применимости для различных архитектур.

В разделе 1.1 обсуждается задача решения булевых формул в теориях, на которой основана работа SMT-решателей для символьных выражений. В динамической символьной интерпретации SMT-решатели являются одним из основных элементов анализа и выполняют запросы на проверку математической модели исполнения программы и генерацию новых входных данных. Решение задачи SMT сводится к решению классической NP-полной задачи выполнимости (SAT). В худшем случае решение таких задач занимает экспоненциальные вычислительные ресурсы относительно длины входа, поскольку соответствующие алгоритмы производят перебор всех возможных значений с различными оптимизациями. Из-за значительного объема памяти для хранения символьных ограничений и количества запросов, достигающего порой сотен тысяч, работа SMT-решателей может становиться «узким местом» всего процесса анализа. Это обуславливает необходимость применения набора описанных в разделе 1.2 дополнительных эвристик и оптимизаций конкретно-символьного исполнения перед отправкой формульных выражений в качестве запроса решателю. Во-первых, устанавливается фиксированный лимит времени на решение одного запроса. Чтобы как можно больше запросов укладывались в это ограничение, динамический символьный интерпретатор проводит различные оптимизации SMT-формул: синтаксические упрощения выражений, инкрементальные решения, кэширование запросов, конкретизацию части данных и слайсинг предиката пути. Вероятность получения решения запроса также повышается при устранении избыточной помеченности, возникающей при включении в формулу условий, которые не представляют интерес для анализа. Для реализации данной задачи применяются конкретизация части данных (например, аргументов функций логгирования), готовые формулы для моделирования семантики библиотечных функций, а также слайсинг предиката пути и «оптимистичные» решения, когда в запрос включается единственное условие предиката пути, соответствующее целевому переходу.

Тем не менее, ключевое преимущество динамической символьной интерпретации состоит в высокой точности анализа. Поэтому отдельного

внимания заслуживает возникновение недостаточной помеченности, когда зависимости интересной для анализа части программы перестают отслеживаться символьными переменными. Как правило, это связано с увеличением сложности процесса моделирования символьного контекста выполнения. Одной из причин такой конкретизации полезных символьных выражений можно назвать взаимодействие тестируемого приложения со средой исполнения. Частичная нейтрализация данного явления возможна с помощью отслеживания системных и библиотечных вызовов. Другая причина кроется в использовании символьных выражений для вычисления адресов при операциях обращения к памяти. Моделирование косвенной адресации позволяет решить эту проблему, увеличивая вариативность при поиске новых путей. Однако символьная адресация приводит к включению в выражения объёмных участков памяти, содержимое которых также может оказаться символьным. В таких ситуациях приходится искать баланс между производительностью и точностью, ограничивая потребление ресурсов анализатором с помощью конкретизации.

Раздел 1.3 направлен на рассмотрение способов обнаружения программных дефектов, проявление которых сопряжено с обработкой внешних данных. Динамическая символьная интерпретация предлагает средства целенаправленного поиска ошибок и уязвимостей в виде предикатов безопасности. Они представляют собой набор дополнительных символьных ограничений, содержащих условия возникновения той или иной ошибки, например, деления на нуль. Такие условия добавляются к предикату пути и отправляются в качестве запроса к решателю. При запуске на успешно сгенерированных для такого запроса данных, проявление ошибки может приводить к повреждениям памяти, например, в случае выхода за границу массива. Поэтому при верификации наличия ошибки на тестовых данных используются специальные обработчики-санитайзеры, изначально придуманные для фаззинг-тестирования. При выявлении ошибки санитайзер останавливает запуск и выводит отчет о находке. В отличие от предикатов безопасности проверки санитайзеров нацелены, в первую очередь, не на поиск ошибки, а на подтверждение её присутствия, например, когда не было аварийного завершения. Поиск ошибок предикатами безопасности впервые был реализован при анализе символьных аргументов форматной строки и счетчика инструкций в динамическом символьном интерпретаторе Mayhem. Аналогичную функциональность символьной интерпретации для других типов ошибок можно найти в инструментах IntScope и SAVIOR. Однако все три инструмента работают только с архитектурой x86(_64).

Раздел 1.4 предоставляет ретроспективу наиболее значимых этапов развития динамической символьной интерпретации на основе описания соответствующих инструментов и воплощенных в них подходов, сделавших возможным применение символьного анализа в гибридном фаззинг-тестировании. Первые инструменты для динамической символьной интерпретации предназначались для автоматизированного анализа программ, написанных на языке Си. Уже тогда их можно было разделить на два направления. Предпосылки для создания конкретно-символьного подхода были заложены в использующем трассу конкретного выполнения инструменте DART и его улучшенной версии CUTE. Предобработка символьных выражений из CUTE (в виде объединения общих выражений, инкрементальных решений вложенных предикатов, проверки отсутствия взаимоисключения отрицания целевого условия и остального предиката пути) перед отправкой запроса решателю сейчас является стандартной практикой. Логическим продолжением DART и CUTE стал инструмент SAGE, первый символьный анализатор бинарного кода архитектуры x86, применявшийся для промышленного тестирования продукции компании Microsoft. Кроме того, в своей статье авторы SAGE впервые проводят аналогию между конкретно-символьным выполнением и фаззинг-тестированием, обозначая свой подход как "метод белого ящика". Для исследования глубоких путей в масштабных приложениях, получающих на вход большие объемы данных, ими был предложен алгоритм "генерационного поиска", основанный на приоритизации входных файлов. С его помощью SAGE успешно находил ошибки для Windows-приложений. Другое направление символьной интерпретации было предложено разработчиками из Стенфорда, которые реализовали исследование нескольких путей внутри одного и того же запуска. Прототипы EGT и EXE для языка программирования Си выполняли обход исследуемых путей на основе алгоритма поиска в глубину при помощи клонирования (fork) символьного контекста в точках ветвления программы. Тот же принцип множественных символьных состояний был унаследован едва ли не самым известным из актуальных инструментов символьной интерпретации – виртуальной машиной KLEE, представленной ещё в 2008 году. В организации символьных процессов для различных выборов условных переходов KLEE выполняет обязанности распределения ресурсов и переключения контекста, что отчасти похоже на работу операционной системы. Нововведением KLEE в части инструментации стала трансляция исходного кода в промежуточное представление LLVM IR, виртуальный

набор инструкций которого интерпретатор преобразует в символьные выражения. Благодаря подробному погружению в устройство механизмов для решения SMT-запросов тот же коллектив сумел разработать качественный набор оптимизаций выражений на основании тождественных преобразований, кэширования предыдущих запросов, а также выделения подмножеств ограничений, независимых по переменным.

Развитие символьных анализаторов продолжилось в формате многофункциональных фреймворков анализа бинарного кода, опиравшихся на использование промежуточного представления. Например, фреймворк автоматизированного исследования путей S2E, ориентированный на моделирование полносистемного окружения, предоставляет опции символьного анализа взаимодействия ПО с внешними библиотеками и ядром ОС с помощью выбора модели целостности выполнения и выборочной символьной интерпретации. Данное решение, как и многие другие, основано на переиспользовании инструмента KLEE и включает целых два этапа трансляции: сначала бинарный код транслируется в представление QEMU, которое затем преобразуется в LLVM IR. Хотя подобная конструкция обладает существенным преимуществом универсальности, оно достигается в ущерб производительности анализатора.

Создатели следующего фреймворка angr позиционировали его как дружелюбную к пользователю платформу создания собственных инструментов исследования бинарного кода на основе интерфейсов для его обработки и трансляции в абстрактное представление VEX IR. На данный момент, модули фреймворка полноценно поддерживают ряд архитектур, среди которых есть ARM, а также возможна символьная интерпретация отдельных инструкций RISC-V, для которой пока не реализованы анализ неявных переходов и соглашения о вызовах. Недостатком, как и в случае S2E, является замедление, поскольку angr написан на удобном для разработчиков языке Python. Примечателен данный фреймворк и тем, что для демонстрации его возможностей авторы впервые реализовали концепцию гибридного фаззинга в инструменте Driller. Driller комбинирует подходы символьной интерпретации и мутационного фаззинга, попеременно применяя символьные техники фреймворка angr и открытый инструмент фаззинга AFL для анализа набора входных данных на основе метрики прироста покрытия.

Фундаментальной идеей и отличием вышедшего в 2018 г. анализатора бинарного кода Qsym стало качественное ускорение производительности символьного интерпретатора, достаточное для совместного применения с

фаззером в параллельном режиме. Для экономии времени его разработчики отказались от трансляции в промежуточное представление, а также внедрили выборочную динамическую инструментацию машинных инструкций архитектуры x86(_64) только для символьных базовых блоков. Помимо конкретизации символьного потока данных при внешних вызовах, были предложены оптимистичные решения, которые даже без корректного инвертирования условного перехода при решении запроса могли принести пользу после применения мутаций фаззером. Архитектура QSym в свое время стала образцом качественного базового символьного интерпретатора, превратив его в инструмент-бенчмарк для сравнения при разработке следующего поколения скоростных concolic-анализаторов.

В том же 2018 году был представлен инструмент Angora, в котором для решения символьных ограничений в отличие от классических интерпретаторов вроде QSym используются не SMT-решатели, а комбинацию анализа помеченных данных (taint tracking) и градиентных методов. Такой подход позволяет быстро генерировать новые входные данные для расширения тестового покрытия при совместной работе с фаззером. Поскольку Angora опирается не на бинарный код, а на инструментацию LLVM, то теоретически возможна поддержка различных архитектур, однако на практике анализируются только бинарные Linux программы для x86_64 .

Опубликованный в статье 2019 г. инструмент Manticore отчасти близок к фреймворку angr, так как он написан на Python и предоставляет удобный пользовательский API. Manticore умеет работать с бинарными программами Linux в формате ELF, а также со смарт-контрактами Ethereum (EVM) и байт-кодом WASM. При этом в данном инструменте реализована инструментация на уровне машинного кода для платформ x86/x86_64, AArch64и ARMv7. Но использование интерпретируемого языка ограничивает его применимость в рамках гибридного фаззинга из-за падения производительности. К тому же, разработчики прекратили поддержку данного инструмента, и его дальнейшее развитие не планируется.

Возвращаясь к инструментам конкретно-символьного направления, заданного QSym, следует упомянуть работы коллектива С. Поплау, SymCC и SymQEMU, вышедшие в 2020 и 2021 годах. Их основная идея состояла в предварительной перекомпиляции изучаемой программы для вставки кода инструментации в будущий исполняемый файл. Такой подход похож на принципы работы санитайзеров, но чтобы миновать специфические особенности программы (язык программирования или архитектурные

зависимости) авторы вновь обращаются к промежуточному представлению. Для SymCC, требовавшего наличие исходного кода, это биткод LLVM, а в SymQEMU это ограничение снимается путем трансляции машинных инструкций в представление QEMU TCG, которое после инструментации вновь рекомпилируется в исполняемый код. При этом оба анализатора переиспользуют функциональность интерпретатора QSym для обработки символьных выражений, а также имеют достаточную производительность для применения в гибридном фаззинг-тестировании.

Конкретно-символьный исполнитель SymFit, основанный на SymQEMU, вводит принцип "оптимизации для общего случая", фокусируясь на ускорении интерпретации конкретных инструкций, которые составляют преобладающую часть исследуемого кода. В отличие от SymQEMU, где для конкретных инструкций все равно проводится дополнительная символьная обработка, SymFit минимизирует накладные расходы, позволяя большинству инструкций выполняться без лишнего инструментирующего кода с помощью эффективного переключения между символьным и конкретным режимами исполнения. За счет использования промежуточного представления QEMU TCG, SymFit тоже может быть портирован для анализа бинарного кода архитектур RISC-V и ARM. Его предшественник, инструмент SymSan использует инструментацию LLVM на этапе компиляции и больше напоминает SymCC. SymSan интегрирован с DFSan (DataFlowSanitizer), который отслеживает поток данных при выполнении программы. Как и в SymFit за счет переключения между символьными и конкретными состояниями SymSan достигает гораздо большей эффективности, чем SymCC или SymQEMU. SymSan анализирует программы на уровне LLVM IR, поэтому может поддерживать различные архитектуры, но по умолчанию он предназначен только для x86_64 программ.

Еще в одном похожем на SymQEMU и применимом в гибридном фаззинге инструменте FUZZOLIC акцент был сделан на более тесной интеграции с инфраструктурой QEMU и внедрении новых методов оптимизации символьных выражений. Вместо отдельных инструкций FUZZOLIC обрабатывает предварительно оптимизированные базовые блоки после трансляции QEMU, что позволяет значительно снизить издержки на трассировку и повысить производительность символьного анализа. Кроме того, для преодоления узкого места при решении SMT-запросов трассировщик и решатель вынесены в разные процессы и взаимодействуют через разделяемую память. Разработчики интегрировали FUZZOLIC с собственным решателем Fuzzy-Sat, ориентированным на приблизительные

решения, что позволяет быстро исследовать новые пути исполнения без существенного роста вычислительных затрат. FUZZOLIC поддерживает символьную интерпретацию архитектурно-зависимых TCG-helpers, что расширяет охват специфичных для архитектуры инструкций по сравнению с SymQEMU, где упор делался на архитектурно-независимые операции. Это повышает точность анализа для сложных бинарных программ под x86/x86-64, но в то же время и ограничивает его применение для других архитектур. Так что несмотря на то что FUZZOLIC построен на базе QEMU, он не поддерживает архитектуры ARM и RISC-V.

Фреймворк Triton предоставляет примитивы для проведения динамического анализа исполняемого кода на основе отслеживания помеченных данных, создания символьных выражений и модификации ограничений предиката пути, сохранения контекста символьных состояний, интерфейса для передачи запросов SMT-решателю, а также конструкторов операционной семантики инструкций для архитектур x86, x86_64, ARM32 и AArch64 в терминах битовых векторов. Для работы с интерфейсами фреймворка можно использовать язык программирования C++ либо биндинги на языке Python. В качестве динамических символьных интерпретаторов, созданных на основе Triton, можно назвать TritonDSE и Sydr, которые входят в комплексы оркестрации гибридного фаззинга PASTIS и Sydr-Fuzz, соответственно. Конкретный запуск в написанном на Python инструменте TritonDSE применяется для предварительной записи трассы выполненных инструкций посредством динамического бинарного инструментатора QBDI, который также используется для сбора покрытия. Символьное состояние программы после инициализации начального состояния конкретными данными полностью эмулируется. Для загрузки бинарного кода и соответствующих динамических библиотек в TritonDSE задействован компонент распознавания форматов исполняемых файлов из фреймворка angr. За символьную часть отвечает функциональность, предоставляемая интерфейсами Triton, поэтому TritonDSE поддерживает такой же набор процессорных архитектур. Символьное состояние программы полностью эмулируется, однако требуется предварительная инициализация начального состояния конкретными данными.

Конкретно-символьный интерпретатор Sydr использует фреймворк DynamoRIO для динамической бинарной инструментации машинного кода x86(_64). Архитектура Sydr построена на параллельной работе двух процессов – конкретном и символьном вычислителях, что впервые было предложено в инструменте Mayhem. Базовый сценарий работы инструмента

предполагает перенаправление потока инструкций от динамического инструментатора в символьный исполнитель, который применяет интерфейсы Triton для конструирования символьных выражений Однако при написании дополнительного конфигурационного файла символьный процесс также поддерживает режим эмуляции. В Sydr реализовано большое количество оптимизаций и техник символьного анализа, включающих предикаты безопасности, оптимистичные решения и их продвинутую версию с использованием алгоритма слайсинга, анализ символьных указателей, моделирование семантики функций стандартной библиотеки, контекстное кэширование переходов. В качестве SMT-решателя можно использовать инструменты Z3 и Bitwuzla. Комплекс динамического анализа Sydr-Fuzz ориентирован на проведение гибридного фаззинг-тестирования с использованием Sydr и таких инструментов фаззинга с обратной связью по покрытию как libFuzzer, AFL++ и honggfuzz. Описание нужных инструментов и их параметров для фаззинг-сессии задается с помощью конфигурационного файла. Фреймворк предоставляет удобный интерфейс командной строки, позволяющий запускать фаззинг, минимизировать полученный корпус входных данных, применять режим анализа интерпретатора с генерацией символьных предикатов, собирать данные о покрытии и создавать отчеты о найденных аварийных завершениях.

В разделе 1.5 приведено сравнение с классической CISC-архитектурой x86(_64) основных учитываемых в процессе символьной интерпретации особенностей бинарного кода для целевых RISC-архитектур Байкал-М (AArch64) и RISC-V. В первую очередь, это набор интерпретируемых инструкций архитектуры (ISA), который может быть дополнен подключаемыми расширениями. Далее обобщается и дополняется список символьных инструментов, поддерживающих анализ архитектур Байкал-М (AArch64) и RISC-V. При рассмотрении которых делается наблюдение, что большинство из таких анализаторов не обладают практической применимостью в гибридном фаззинге. На основе чего в разделе 1.6 делается вывод об актуальности задачи разработки соответствующих методов гибридного фаззинга для бинарных программ архитектуры Байкал-М (AArch64) и RISC-V.

Вторая глава посвящена предлагаемому методу динамической символьной интерпретации бинарного кода архитектуры Байкал-М (AArch64), основанному на конкретно-символьном подходе с применением динамической бинарной инструментации и реализованному в анализаторе Sydr. В разделе 2.1 освещаются особенности моделирования символьного

контекста архитектуры AArch64 для указанного подхода, в то время как раздел 2.2 содержит подробное описание используемых представленным методом механизмов обнаружения косвенных переходов в бинарном коде.

Для архитектуры ARM/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 динамического бинарного инструментатора DynamoRIO. Символьный исполнитель дизассемблирует операционный код инструкции и производит набор проверок на наличие символьных операндов и необходимость обновления символьного стека вызовов. Для ARM/AArch64 инструкциями вызова функций являются BL и BLR, а для возврата из функции используется RET, то есть инструкция BR x30, которая по сути передает управление на инструкцию, адрес которой хранится в регистре x30 (Procedure Link Register). Отслеживание данных инструкций необходимо для актуализации символьной модели памяти стека, которая также должна поддерживать выравнивание. 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. При реализации предлагаемого метода в интерфейсе библиотеки Triton для работы с символьной памятью была выявлена и исправлена ошибка назначения размера области памяти, тождественного размеру регистра для хранения значения (то есть 4 либо 8 байтов), вместо размера, определяемого семантикой инструкции. Помимо этого, для механизма адресации с индексированием в ARM/AArch64 вместо коэффициента в виде константного числового множителя внутри инструкции обращения к памяти предусмотрена возможность применения операции

битового сдвига, которая может быть совмещена с операцией расширения индексного регистра. Примеры таких инструкций приведены на Рисунке 1.

```
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 + ext(w2) * 8])

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

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

При анализе косвенных табличных переходов участвующий в адресных вычислениях константный множитель на основе битового сдвига может быть упущен из виду. Это связано с тем, что, как правило, после загрузки значения из памяти возникают дополнительные арифметические операции, которые также могут содержать битовый сдвиг (например, в последней строке на Рисунке 1 приведена такая инструкция сложения). При символьной интерпретации подобных ветвлений с несколькими вариантами передачи управления требуется заранее сохранить адресное выражение загружаемого из памяти значения. Добавление его в виде условий в предикат пути происходит позднее при появлении соответствующей инструкции передачи управления. Соответственно, требуется отслеживание арифметических преобразований между инструкцией чтения из памяти и передачи управления. В предлагаемом методе символьной интерпретации данная задача решается посредством анализа последовательности регистровых операндов на основе алгоритма слайсинга, при помощи которого производится коррекция адресного выражения с учетом возможности битовых сдвигов. Детектирование таблиц косвенных переходов в предлагаемом методе происходит на основе эвристики 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*.

Случай дополнительного разбиения адреса целевого перехода на базовый адрес начала таблицы переходов и относительное смещение можно отличить по наличию второй инструкции загрузки адресного значения. Определение границ таблицы переходов основано на поиске инструкции сравнения *CMP* в предыдущем логическом базовом блоке. Для этого применяется механизм кэширования выполненных базовых блоков. С помощью извлечения константного значения для инструкции сравнения определяются целевые ячейки читаемой памяти, для содержимого которых вычисляются адреса, которые верифицируются на принадлежность к областям исполняемого кода. Глава заканчивается разделом 2.3, содержащим выводы о полученных при разработке метода результатах.

В **третьей главе** представлен метод символьной интерпретации набора целочисленных инструкций архитектуры RISC-V, включающего сокращенные инструкции и псевдоинструкции. В разделе 3.1 определяются границы применимости предлагаемого метода с точки зрения поддерживаемых расширений инструкций, а именно *RV64IMC* и *RV32IMC*. Далее в разделе 3.2 раскрываются детали реализации метода в рамках модуля разрешения архитектурных зависимостей символьной библиотеки Triton. Данный модуль предназначен для преобразования машинных инструкций в абстрактные классы Triton. Для представления выбранной архитектуры с помощью инфраструктуры абстрактных интерфейсов Triton были добавлены соответствующие спецификации архитектурных компонентов, позволяющие организовывать взаимодействие с Capstone, внешним инструментом дизассемблирования, и производить моделирование символьного состояния программы. Помимо конструирования символьных выражений операционной семантики инструкций, в модуле разрешения архитектурных зависимостей производится выделение памяти двойного набора переменных регистров для получения конкретного и символьного контекста выполнения во время анализа, а также устанавливается соответствие идентификаторов наборов регистров и инструкций.

После создания спецификаций и идентификаторов в описанном модуле 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 поля абстрактного класса Triton *triton::arch::Instruction*. На этапе конструирования символьной семантики вызывается общий метод *riscvSemantics::build_semantics*, который с помощью идентификатора инструкции выбирает и вызывает соответствующий функциональный метод для отдельной инструкции. По аналогии с архитектурами x86 и x86_64 для RISC-V метод *riscvSemantics::build_semantics* является универсальным и содержит одни и те же конструкторы символьных выражений операционной семантики инструкций 32-битной и 64-битной версий архитектуры, различие которых заключается в используемых размерах регистровых операндов.

Программная реализация описанного метода в библиотеке символьной интерпретации Triton составляет порядка 7,5 тысяч строк исходного кода (с учетом скриптов сборки и тестирования порядка 10,7 тысяч строк кода).

В разделе 3.3 разъясняются особенности моделирования символьной семантики для вариаций стандартных инструкций в виде псевдоинструкций и для сокращенных инструкций, размер которых вдвое меньше, чем у стандартных, и составляет 2 байта. Раздел 3.4 посвящен апробации предлагаемого метода с помощью синтетического тестового набора, а также демонстрации возможностей символьной интерпретации бинарного кода архитектуры RISC-V с помощью Triton на основе символьной эмуляции примера для задачи «захвата флага» (CTF). Выводы, касающиеся проделанной при создании метода работы, отражены в разделе 3.5.

В **четвертой главе** обсуждается разработанный метод динамической символьной интерпретации бинарного кода архитектуры RISC-V 64, который для осуществления конкретно-символьного анализа посредством инструмента Sydr, опирается на метод, представленный в предыдущей главе. Раздел 4.1 содержит краткую характеристику описываемого метода, после которой в разделе 4.2 рассматриваются потребовавшиеся для его реализации доработки в динамическом бинарном инструментаторе DynamoRIO.

В качестве средств для моделирования символьного контекста бинарного кода архитектуры RISC-V 64 посредством представленного данной главой метода задействованы следующие архитектурные компоненты:

- набор 64-битных регистров общего назначения x0-x31, среди которых имеют дополнительное 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 в зависимости от результатов сравнения их операндов. Кроме того, для правильного моделирования потока управления и корректной обработки символьной памяти стека необходимо детектировать псевдоинструкции для инструкций *JAL* и *JALR*. Размеры обращения к памяти, определяемые семантикой инструкций, для данного метода поддерживаются аналогично архитектуре AArch64.

Одним из основных механизмов, используемых при анализе, является перехват вызовов функций и возвратов из них. В DynamoRIO данная функциональность реализуется с помощью модуля *drwrap*, который позволяет произвести инструментацию вызова произвольной функции анализируемой программы путем вставки вызовов функций-анализаторов до и после вызова целевой функции:

1. Вызов функции-обертки *pre_func()*. Данная функция предоставляет пользователю информацию обо всех аргументах вызываемой функции.

2. Выполнение анализируемой функции.

3. Вызов функции-обертки *post_func()*. Эта функция вызывается сразу после завершения работы анализируемой функции и предоставляет пользователю информацию о возвращаемом значении.

DynamoRIO также предусматривает возможность обмена произвольными пользовательскими данными между *pre* и *post* обертками. На момент разработки метода интерпретации поддерживалась вставка и вызов только pre-оберток, поскольку механизм подмены адреса возврата из функции в DynamoRIO был реализован только для архитектур семейства x86 и ARM. Поэтому в рамках данной работы в DynamoRIO была поддержана работа указанного механизма с регистрами, соответствующими машинному контексту RISC-V 64, что обеспечило работоспособность поддерживаемых инструментом Sydr современных техник анализа в виде моделирования символьной семантики функций и предикатов безопасности для бинарных программ данной архитектуры.

Раздел 4.3 посвящен анализу косвенных табличных переходов в условиях неполной информации о списке инструкций текущего базового блока. Помимо вышеописанного механизма перехвата вызовов функций, к используемым в Sydr возможностям DynamoRIO относится предварительная загрузка базового блока инструкций, готового к исполнению, после чего осуществляется получение контекста конкретного запуска для каждой исполняемой инструкции. Данный механизм удачно согласуется с задачей распознавания косвенных табличных переходов для AArch64, поскольку она решается на уровне анализа всего базового блока.

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 архитектуры RISC-V, 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.

При наличии адреса инструкции передачи управления, то есть последней инструкции в базовом блоке, предикат для инвертирования переходов в зависимости от читаемого из памяти значения создается для load-инструкции, но добавляется к общему предикату пути в процессе интерпретации инструкции передачи управления при совпадении адреса. Если две эти связанные инструкции попадают при разбиении в различные блоки, то адрес перехода заранее неизвестен. Поэтому для RISC-V 64 вместо него в предикат записывается фиктивный адрес, а именно адрес целевой load-инструкции, увеличенный на единицу. По нему при получении инструкции перехода из другой части базового блока символьный исполнитель сможет определить сохраненный ранее соответствующий предикат. Следует отметить, что используемый фиктивный адрес оказывается нечетным. Благодаря механизму выравнивания он не сможет совпасть к каким-либо другим адресом инструкции, для которой не требуется добавление предиката, даже если это будет сокращенная инструкция.

В случае машинного кода 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 рисунке 2 можно видеть пример ассемблерного кода для содержащего табличный переход базового блока, который будет разбит на 3 части.

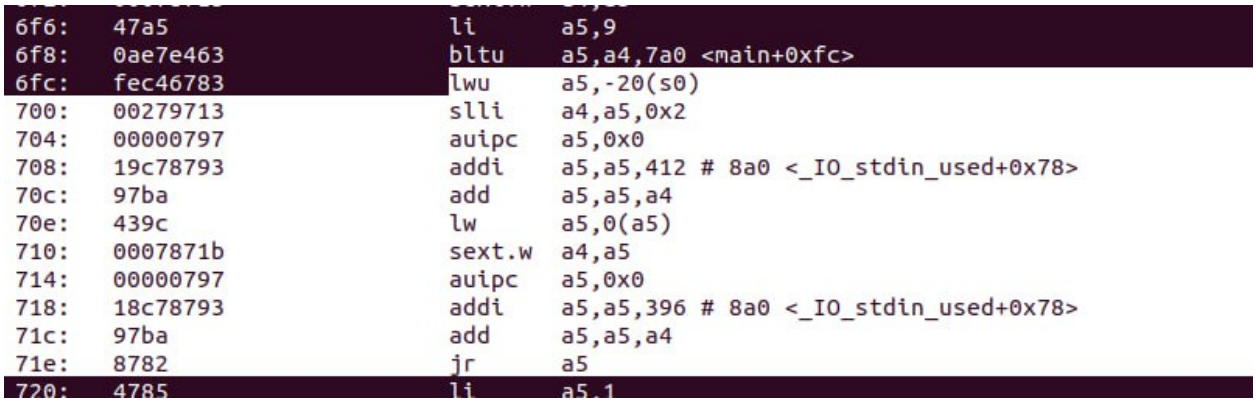

Рисунок 2. Базовый блок с разбиением на 3 части.

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, предоставленного DynamoRIO, статус стадии поиска табличного перехода может быть различным Для его хранения используется специально выделенная область памяти, которая проверяется и заполняется по мере появления составляющих с помощью функция *jt_cut_bb* Алгоритма 1. Методы вида *get_something* содержат вызовы API DynamoRIO, в том числе, в виде цикла перебора инструкций базового блока *bb*.

В ситуации, когда производится первичное сканирование сначала проверяется, что последняя инструкция блока не относится к условным инструкциям передачи управления. Далее происходит обращение к хранилищу для получения адреса инструкции обращения к памяти, которое возможно было сохранено ранее. В ситуации, когда такое значение отсутствует, с конца базового блока осуществляется поиск последней load-инструкции в блок. При обнаружении такой инструкции 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 первая инструкция выделенного базового блока на Рисунке 2.

Алгоритм 1 Алгоритм поиска косвенного перехода в базовом блоке

Входные данные: bb — набор инструкций анализируемого базового блока

Возвращаемое значение: $load\_addr$ — адрес целевой load-инструкции либо 0.

```
bb_start ← bb.get_start_addr()
last_instr ← bb.get_last_instr()
if last_instr.is_cbr() then return 0 // Пропуск блока с условной передачей управления
// Проверка текущего статуса сканирования из хранилища
(load_addr, has_pattern) ← jt_cut_bb(bb_start, 0, check_mode)
if load_address ≠ 0 then
    if last_instr.is_mbr() then return load_addr
    else return 0
// В хранилище сохраненный ранее адрес целевой load-инструкции отсутствует
load_addr ← get_last_load(bb, last_instr)
if load_address ≠ 0 and has_pattern then
    if last_instr.is_mbr() then return load_addr
    else jt_cut_bb( bb.get_last_addr(), load_addr, save_mode)
// Пропуск закончившегося блока, если нет целевой load-инструкции
if load_addr = 0 and last_instr.is_mbr() then return 0
(pattern_addr, has_pattern) ← bb.find_pattern()
if has_pattern and pattern_address < load_addr then
    jt_cut_bb( bb.get_last_addr(), load_addr, save_mode); return load_addr
else if has_pattern then jt_cut_bb( bb.get_last_addr(), 0, save_mode)
return 0
```

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. 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. Выводы для метода, описанного в данной главе приводятся в разделе 4.4.

Пятая глава содержит экспериментальную оценку разработанных методов динамической символьной интерпретации бинарного кода архитектур Байкал-М (AArch64) и RISC-V 64. В разделе 5.1 детально рассматривается проведение экспериментов для приложений архитектуры AArch64, а в разделе 5.2 для архитектуры RISC-V 64. Оценка опирается на метрику достигнутого покрытия и проводится двумя способами.

Во-первых, производится сравнение реализации предложенных методов в динамическом символьном интерпретаторе Sydr с его открытым аналогом SymQEMU при анализе набора реальных приложений целевой архитектуры. Количественные результаты сравнения для архитектуры AArch64 и архитектуры RISC-V 64 представлены в Таблице 1 и Таблице 2, соответственно.

Результаты для каждого инструмента включают количество сгенерированнных входных данных (столбец «*Вх. Файлы*»), которые соответствуют успешным (SAT) запросам на инвертирование условных переходов. Столбец «*Уник.*» содержит число уникальных покрытых строк исходного кода для набора входных данных, полученного от отдельного инструмента. Например, при тестировании скомпилированного на процессоре Байкал-М приложения *freetype2* инструмент Sydr смог достигнуть 227 строк кода, которые не удалось покрыть инструменту SymQEMU. Данная метрика показывает разницу между двумя инструментами с точки зрения фактической пользы от генерируемых ими наборов данных. В столбце «*Покрытие*» приведен процент всех покрытых отдельным инструментом строк кода относительно объединенного достигнутого двумя инструментами покрытия. Данная метрика показывает, какую долю от общего тестового покрытия удалось обнаружить инструменту. В столбце «*Время*» указано время, которое инструменты потратили на исследование программы.

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

Таблица 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
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

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

Во-вторых, для оценки применимости предложенных методов в контексте гибридного фаззинг-тестирования сравнивается парная работа Sydr и инструмента мутационного фаззинга с двумя параллельно работающими процессами такого же фаззера. На основе полученных данных в разделе 5.3 делается вывод о том, что представленные методы обеспечивают для целевых архитектур как высокую эффективность символьного анализа, так и достаточную производительность для их применимости в гибридном фаззинг-тестировании реальных приложений.

В **заключении** подытоживаются основные полученные результаты и перечисляются задачи, выполненные во время подготовки научно-квалификационной работы:

1. Был разработан метод динамической символьной интерпретации бинарного кода программ архитектуры Байкал-М (AArch64).

2. Был разработан метод символьной интерпретации набора целочисленных инструкций архитектуры RISC-V, включающего сокращенные инструкции и псевдоинструкции.

3. Был разработан метод динамической символьной интерпретации бинарного кода программ архитектуры RISC-V 64.

4. Были реализованы алгоритмы определения возможных направлений косвенных переходов для бинарного кода архитектур Байкал-М (AArch64) и RISC-V 64.

5. Разработанные методы были реализованы в инструментах Triton и Sydr, а также была проведена оценка их эффективности.

Список публикаций автора по теме диссертационной работы

1. Программа для ЭВМ «Инфраструктура доверенных фреймворков машинного обучения», свидетельство № 2022685212 от 22.12.2022.

2. *Vishnyakov, A.* Sydr: Cutting edge dynamic symbolic execution [Текст] / A. Vishnyakov [и др.] // 2020 Ivannikov ISPRAS Open Conference (ISPRAS). — IEEE. 2020. — С. 46—54.

3. *Vishnyakov A. Symbolic Security Predicates: Hunt Program Weaknesses [Текст] / A. Vishnyakov [и др.] // 2021 Ivannikov Ispras Open Conference (ISPRAS). —IEEE. 2021. — С. 76—85.*

4. *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.

5. Логунова В.И. Применение динамической символьной интерпретации в гибридном фаззинге бинарного кода для архитектур Байкал-М и RISC-V 64 [Текст]. Труды Института системного программирования РАН, том 37, вып. 4, часть 2, 2025, стр. 235-250. DOI: 10.15514/ISPRAS-2025-37(4)-29.

6. Вишняков А.В. Sydr-Fuzz: непрерывный гибридный фаззинг и динамический анализ для жизненного цикла безопасной разработки [Текст] / Вишняков А.В., Куц Д.О., Логунова В.И. [и др.] // Труды Института системного программирования РАН, том 37, вып. 4, часть 2, 2025, стр. 251-270. DOI: 10.15514/ISPRAS-2025-37(4)-30.