Федеральное государственное бюджетное учреждение науки
Институт системного программирования им. В.П. Иванникова
Российской академии наук

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»

Диссертация на соискание ученой степени

кандидата технических наук

Научный руководитель:
Гетьман Александр Игоревич
к.ф.-м.н.

Москва – 2025

Оглавление

Введение

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 программных продуктов, создаваемых интеллектуальным трудом многочисленных разработчиков ПО. К сожалению, в жизни всегда есть место человеческим ошибкам, и не всегда код работает в соответствии с виденьем автора. Зачастую подобное случается, когда автор забывает предусмотреть граничные случаи или явно прописать предполагаемые ограничения. Чем позднее будет выявлен и устранен программный дефект, тем больше потенциального ущерба он сможет нанести [1]. Поэтому современные технологии предлагают использование концепции жизненного цикла безопасной разработки ПО [2-4]. Данный подход направлен на всестороннее рассмотрение каждого этапа разработки ПО с точки зрения качества и безопасности. Обязательным является внедрение процессов разнопланового тестирования, включающих, например, анализ результатов работы программного продукта при взаимодействии с источниками заранее неизвестных внешних данных. Как правило, для подобного вида тестирования ПО применяются автоматизированные статические и динамические анализаторы. Динамические средства анализа выполняют запуск исследуемой программы и наблюдают за ходом её выполнения. Это позволяет избежать большинства ложных сообщений об обнаруженных ошибках, что является распространенным явлением для статических инструментов. В роли инструментов для динамического анализа чаще всего используются средства фаззинг-тестирования [5], которые генерируют данные для передачи на вход программе. Как правило, такие инструменты производят множественные запуски для различных входных данных, на которых могут проявиться ошибки.

Количество информации, собираемой анализатором о запуске целевого ПО с целью поиска ошибок может быть различным. Наиболее простым вариантом является отслеживание наличия аварийных завершений. Так как информация о внутреннем состоянии выполняющейся программы не собирается, подобный вид тестирования называется фаззингом “методом черного ящика“. Более продвинутые инструменты [6-8] замеряют для входных тестовых данных метрику покрытия кода, который был выполнен в ходе их обработки. Такой подход называется фаззингом с обратной связью по покрытию или фаззингом “методом серого ящика“ и является наиболее распространенным. Метрика покрытия отображает структурные элементы исследуемой программы, которые были задействованы во время запуска для выбранных входных данных. Такими структурными элементами могут быть инструкции, строки исходного кода программы, логические ветвления, а также функциональные методы. Для сбора покрытия, как правило, на этапе компиляции применяется инструментация, когда каждому такому элементу сопоставляется бит информации, который принимает истинное значение при выполнении соответствующего кода. Обратная связь по покрытию означает, что входные данные ранжируются инструментом фаззинга по добавленному покрытию относительно объединенного покрытия из предыдущих запусков. Таким образом, с помощью метрики покрытия фаззинг может определить области кода, выполненного хотя бы на одном протестированном пути выполнения. Однако генерация входных данных, приводящих на путь выполнения, который проходил бы через ранее не покрытые области кода, в таком подходе остается близкой к задаче случайного перебора. Тем не менее, также существует более трудозатратный, с точки зрения инструментации, фаззинг “методом белого ящика“ [9]. Он используется для поиска новых зависящих от входных данных путей на основе подробного анализа определяемого логическими ветвлениями потока управления внутри программы. Данный вид анализа также известен как *динамическая символьная интерпретация*. При запуске целевого приложения входные данные из

“недоверенного“ внешнего источника помечаются как символьные. Такая пометка фиксирует размер, но не конкретное значение входных данных, и распространяется на их дальнейшие производные, получаемые в результате выполнения инструкций программы. Например, после парсинга помеченного содержимого json-файла полученные переменные с полями, как строковыми, так и целочисленными, тоже будут рассмотрены как символьные. Каждое преобразование символьных данных описывается с помощью абстрактной формулы (AST-представления). Последовательность выполненных преобразований отражает связь между исходными данными, которые передаются на вход программе, и переменными, регистрами и значениями памяти в каждой точке пути выполнения. Поток управления для заданного пути определяется условиями логических ветвлений, часть из которых может зависеть от символьных данных. При поиске новых путей символьный интерпретатор пытается изменить выбор направления каждого символьного ветвления относительно конкретного запуска. Это реализуется с помощью вычисления подходящих входных данных на основе составленных формул преобразований символьных данных.

Легко заметить, что инструментация для отслеживания символьных преобразований требует заметно больше ресурсов как памяти, так и времени, нежели просто получение списка выполненных инструкций как в случае “серого“ фаззинга. По отдельности эти подходы обладают недостатками в виде более низкой производительности символьной интерпретации и отсутствия у фаззинга с обратной связью по покрытию аналитического способа генерации входных данных для открытия новых путей. Поэтому для повышения эффективности анализа существует *гибридный фаззинг*, сочетающий одновременную работу этих динамических анализаторов при помощи обмена данными. Таким образом, решаются взаимодополняющие задачи: символьный интерпретатор вычисляет сложные комбинации входных данных, а фаззер

генерирует их многочисленные вариации, быстро осваивая новые области покрытия кода.

Исторически символьная интерпретация появилась на полтора десятилетия раньше фаззинга и принадлежала к классу статических методов анализа. Статические методы выполняли исследование теоретически доступных путей, из-за проблемы экспоненциального роста числа которых потреблялось чрезмерно большое количество памяти. Из-за этого и других сложностей при отслеживании преобразований над входными данными, методы символьной интерпретации развивались медленно, несмотря на заложенный в них аналитический потенциал. Постепенно количество техник и оптимизаций увеличивалось, позволяя значимо сокращать число программных ошибок за счет внедрения в процессы тестирования и отладки разрабатывающих ПО корпораций. Среди таких техник возникла идея использования конкретных данных, для которых можно сузить зону поиска до единственного пути выполнения, имея при этом набор конкретных значений переменных в качестве опоры для вычислений. Так появилась динамическая символьная интерпретация, которая в свою очередь обзавелась рядом оптимизаций, ставших на данный момент стандартом современных инструментов. Такие инструменты значительно продвинулись в скорости работы, благодаря чему стало возможно их совместное использование с инструментами фаззинг-тестирования, генерирующими данные посредством случайных мутаций.

Существенная часть современных анализаторов представлена инструментами для архитектуры x86(_64). Данная архитектура занимает центральное положение в сфере персональных компьютеров и серверных решений благодаря многолетнему развитию обширной экосистемы. Она стала стандартом для большинства рабочих станций, а также для облачных серверов, используемых в центрах обработки данных. Тем не менее, более энергоэффективные RISC-архитектуры, такие как ARM и RISC-V, создают серьезную конкуренцию архитектуре x86. Одной из причин для этого можно

назвать наличие конкуренции между многочисленными производителями процессоров, которые создают решения на основе RISC-архитектур, в то время как x86 является закрытой архитектурой и производится всего двумя корпорациями. Как и накопленная кодовая база, требующая сохранения обратной совместимости, это ограничивает возможности для инноваций и разнообразия в экосистеме x86, в то время как более гибкие и открытые RISC-архитектуры позволяют большему числу компаний разрабатывать и внедрять новые технологии, что, в свою очередь, способствует их популяризации и распространению на рынке.

Для сохранения ресурсов и расширения возможностей развития экосистем программного обеспечения RISC-архитектурам необходимы качественные инструменты анализа для своевременного выявления и устранения программных дефектов. Особенную актуальность подобное направление работы приобретает благодаря отечественным процессорным разработкам. Например, имеющий государственную лицензию процессор Байкал-М, использующий стандарт ARMv8/AArch64, представляет интерес с точки зрения импортозамещения в ответственных системах высокого доверия. Для будущего проектирования новых процессоров перспективным вариантом предстает открытый стандарт RISC-V, для которого в отличие от ARM права на разработку распространяются свободно.

Как можно заметить, динамическая символьная интерпретация представляет собой достаточно низкоуровневый подход и предполагает сильную зависимость от архитектурных особенностей при исследовании. Инструменты отличаются по уровню требований к наличию исходного кода целевого приложения и представления, для которого формулируются символьные преобразования. При наличии исходного кода возможна компиляция в универсальное промежуточное представление, при создании которого архитектурно-зависимые компоненты будут оставлены за скобками. Таким образом за счет специфически модифицированного компилятора

обеспечивается универсальность этапа отслеживания символьных преобразований с данными. Похожий подход применим также непосредственно к бинарному коду, что более трудоёмко из-за частичных потерь информации на этапе компиляции. Однако не всегда присутствует доступ к исходному коду программы, как, например, в случае использования сторонних динамических библиотек. В подобной ситуации альтернативой генерации промежуточного представления из бинарного кода будет инструментация на уровне машинных инструкций. Преимуществом последнего подхода является отсутствие временных затрат на создание промежуточного представления, что положительно сказывается в контексте использования анализатора в гибридном фаззинг-тестировании. В то же время, 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 научно-квалификационной работе результаты получены лично автором.

Публикации. По теме работы опубликовано 5 научных работ, получено 1 свидетельство о государственной регистрации программы для ЭВМ [10]. Три работы [11–13] изданы в периодических научных журналах с индексацией Web of Science и Scopus. Другие две из опубликованных статей [14–15] изданы в журналах, включенных в перечень рецензируемых научных изданий ВАК при Минобрнауки РФ. В работе [11] 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 [12]. В совместных работах [13, 15] 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.

Глава 1. Динамическая символьная интерпретация

Применение абстрактных, или символьных, входных данных, передающихся в программу извне, впервые было описано в 1976 г. в работе Кинга [16]. Суть данного подхода заключается поиске новых путей выполнения, зависящих от внешних данных, посредством моделирования. Преобразования символьных данных описываются на языке логических формул в теориях (SMT, Satisfiability Modulo Theories) [17], которые обрабатываются предназначенными для этого SMT-решателями [18–22]. Применение данного подхода к исследованию программ, основанного на их непосредственном запуске или эмуляции получило название динамическая символьная интерпретация (DSE, Dynamic Symbolic Execution) [23, 24].

Динамическая символьная интерпретация строит математическую модель анализируемого программного обеспечения, основываясь на той же базовой идее моделирования операций над абстрактными внешними данными. Более близкий к исходно статическим версиям подхода вариант динамической символьной интерпретации выполняет эмуляцию последовательности инструкций анализируемой программы с некоторой стартовой точки. Эмуляция может производиться как для бинарного кода, так и на уровне инструкций промежуточного представления и требует инициализации начального состояния. Фактически реального запуска во время эмуляции не происходит. Поэтому при анализе точек ветвления потока управления программы моделирование операций с символьными данными возможно для всех доступных направлений дальнейшего выполнения. В такой ситуации либо осуществляется выбор одного из направлений (например, на основе предварительно записанной трассы или с помощью результатов для текущего состояния модели, полученных при запросе к SMT-решателю), либо

происходит разделение процесса символьного анализа («fork») для одновременной обработки соответствующих разным путям последовательностей эмулируемых инструкций. Динамическая символьная интерпретация, задействующая эмуляцию выполнения программы, реализована, например, в инструментах KLEE [25], angr [26], S2E [27], BAP[28]. К недостаткам данного подхода следует отнести сложности моделирования окружения и вызовов сторонних библиотек, а также падение производительности анализа при исследовании множества путей исполнения одновременно.

Другим распространенным вариантом динамической символьной интерпретации можно назвать “конкретно-символьное“ выполнение (или “конколическое“, concolic = concrete + symbolic). Как следует, из названия, данный подход сочетает в себе конкретное выполнение программы, то есть ее непосредственный запуск на целевой системе, и символьную интерпретацию. Ключевое отличие заключается в возможности подставить конкретное значение переменной в случае необходимости, тем самым кардинально снижая неопределённость и повышая точность анализа. Использование запуска для конкретных входных данных предоставляет информацию о реальном состоянии регистров и памяти во время выполнения, что позволяет исправлять недочеты моделирования и поддерживать символьный контекст в виде абстрактных регистров и памяти. В качестве популярных инструментов, реализующих конкретно-символьный подход можно назвать QSYM [29], Sydr [11], SymCC [30], SymQEMU [31], Fuzzolic [32], Angora [33]. Тем не менее, у данного подхода существуют собственные ограничения. Во-первых, для доступности сопоставления конкретных значений внешних данных и их производных с символьными данными, размер последних должен соответствовать исходному размеру входа. То есть длина строки или размер прочитанного файла задают ограничение сверху на размер соответствующей таким внешним данным области символьной памяти. Например, если во время запуска был прочитан

пароль из пяти символов, то с помощью конкретно-символьного анализа можно сгенерировать новый пароль, длина которого не превышает пять символов. Во-вторых, область обнаружения новых путей обуславливается возможными отклонениями от заданного конкретными данными пути. Это означает, что для возможности анализа ветвлений, которые не были встречены на пути из текущего конкретного запуска, требуется перезапуск с другим набором входных данных. То есть, один конкретный запуск – один анализируемый путь исполнения. Причины, в следствие которых подобное сужение пространства поиска выглядит рациональным, представляют собой классические трудности при проведении символьной интерпретации. Для конкретно-символьного подхода они по-прежнему остаются актуальными, но в меньшем масштабе, что позволяет обеспечить приемлемую производительность при исследовании глубоких путей выполнения в больших программах и при применении таких инструментов в гибридном фаззинг-тестировании.

Основная проблема состоит в неограниченности числа возможных путей выполнения, которое растет экспоненциально относительно количества встреченных логических ветвлений. Примером такой ситуации являются встречающиеся в программах бесконечные циклы. С появлением каждого зависящего от входных данных условия ветвления (т. е. ветвления, являющегося символьным), такое условие для принятия решения о передаче управления должно быть внесено в специальный набор SMT-формул, называемый *предикатом пути*. Входные данные, удовлетворяющие условиям предиката пути позволяют воспроизвести такой путь при запуске. В случае конкретно-символьного направления динамической символьной интерпретации предикат пути опирается на путь из конкретного запуска и имеет линейно вложенную структуру в виде упорядоченной последовательности условий, объединяемых через конъюнкцию. Иначе попытка исследования всех направлений символьного ветвления порождает как минимум раздвоение

предиката, придавая его структуре свойства дерева, глубина которого потенциально не ограничена. Это влечёт экспоненциальный рост как объема памяти для хранения символьных формул, так и количества запросов к SMT-решателю, требующих временных ресурсов.

1.1. Решатели булевых ограничений в теориях

В динамической символьной интерпретации решатели булевых ограничений в теориях, или SMT-решатели, являются одним из основных элементов анализа и отвечают за проверку математической модели исполнения программы и генерацию новых входных данных.

В конкретно-символьном исполнении запросы к SMT-решателю для инвертирования направления ветвления формируются следующим образом. Выбирается целевое ветвление, условие его выполнения меняется на противоположное и добавляется к части предиката пути, обеспечивающей достижение данного ветвления (то есть ко всем предшествующим условиям предиката пути). Полученное выражение в виде SMT-формулы отправляется в виде запроса в SMT-решатель. Решатель пытается подобрать множество конкретных значений символьных переменных таким образом, чтобы при их подстановке в формулу, она оказалась непротиворечива (SAT). Если SMT-решателю не удается подобрать такие значения, то делается вывод о неразрешимости данного запроса (UNSAT). Это означает что запрашиваемый выбор направления ветвления неосуществим в рамках условий данного предиката пути. Если же SMT-решатель может решить запрос, то он возвращает соответствующую *модель* – набор конкретных значений символьных переменных. Поскольку символьные переменные представляют собой отображение реальных входных данных, то по модели можно воссоздать новый набор входных данных, воспроизводящих отклонение от изначального пути в месте целевого ветвления.

В основе задачи проверки выполнимости SMT-формул лежит известная задача выполнимости (SAT), классическая постановка которой выглядит следующим образом. Дано булево выражение в конъюнктивной нормальной форме, для которого необходимо выяснить, существует ли подстановка значений переменных, делающая формулу истинной. Данная задача была первой признана NP-полной (NPC, теорема Кука–Левина, 1971 [34]) и используется для доказательства принадлежности других NP-задач к данному классу с помощью сводимости. Это значит, что задача, к решению которой сводится SAT, не менее трудна, чем любая задача из NPC множества. В худшем случае её решение занимает экспоненциальные вычислительные ресурсы относительно длины входа.

В практическом анализе программ формальный язык булевых переменных для задачи выполнимости не слишком удобен, так как формулы должны включать арифметику, битовые операции, массивы памяти, условия на строки и т. п. Поэтому в символьной интерпретации используется задача SMT (Satisfiability Modulo Theories) выполнимости формул в теориях, где множество допустимых операций и типов данных определяется SMT-теорией, или логикой. Таким образом, SMT-решатель работает не только с булевыми переменными, но и с выражениями в теориях. Решение задачи SMT сводится к решению SAT, что означает ее принадлежность к классу NP. При анализе реальных программ, размеры памяти для хранения символьных ограничений могут достигать значительного объема, а число различных запросов к решателю может исчисляться сотнями тысяч. Поэтому SMT-решатели становятся «бутылочным горлышком» всего анализа, и именно рост вычислительных мощностей и повышение эффективности алгоритмов, используемых в SMT-решателях, привели к расширению области практической применимости динамической символьной интерпретации.

Для анализа SMT-формул решатель опирается на алгоритмы решения задачи выполнимости, в основе которых лежит идея перебора всех возможных

значений, но с разными оптимизациями. Алгоритм DPLL [35] — классический подход к решению задачи SAT для формул в конъюнктивной нормальной форме. Пример такой формулы:

$$(x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z) \wedge (y)$$

Основные этапы работы с формулами, выполняемые данным алгоритмом, включают:

- Рекурсивное ветвление: выбирается литерал (переменная и её полярность в виде присутствия или отсутствия отрицания), присваивается значение «истина» или «ложь» и рекурсивно проверяется упрощённая формула.
- Распространение единичного литерала: если в формуле присутствует конъюнкт, состоящий из единственной переменной (литерала), то его значение должно быть истинным, иначе формула невыполнима. Соответственно, значение такого литерала детерминировано для истинной формулы, то оно подставляется во все остальные конъюнкты.
- Устранение чистых литералов: если переменная встречается только с одной поляризацией в формуле (нет применения отрицания или наоборот), её можно назначить таким образом, чтобы удовлетворить все конъюнкты, в которых она участвует, без ограничения общности.
- Хронологический откат: при возникновении конфликта (неразрешимости формул с текущими значениями переменных) алгоритм откатывается к предыдущему решению и меняет назначение на противоположное.

Прорыв в производительности и масштабируемости стал возможен с адаптацией решателей, реализующих парадигму Conflict-Driven Clause Learning (CDCL) [36, 37]. Алгоритмы CDCL, в отличие от своего предшественника, не просто осуществляют поиск с возвратом, а извлекают новые знания из возникающих конфликтов, что радикально снижает пространство перебора.

Наиболее распространенными теориями в динамической символьной интерпретации являются теория битовых векторов (QF_BV) и массивов битвекторов (QF_ABV). Сами SMT-формулы представляют собой операции над

символьными и конкретными (константными) переменными, семантически эквивалентные моделируемым вычислениям в программе, например, соответствующим преобразованиям над регистрами и байтами памяти. SMT-формулы можно представить в удобном для человека виде с помощью языка SMT-LIBv2 [34]. Пример такой формулы приведен на рисунке 1. При её обработке в виде запроса SMT-решателем, он определит что формула имеет решение и приведет его.

```
1   (set-logic QF_BV)
2
3   ; объявляем две переменные длиной 8 бит
4   (declare-fun x () (_ BitVec 8))
5   (declare-fun y () (_ BitVec 8))
6
7   ; Условие 1: x + y == 42
8   (assert (= (bvadd x y) (_ bv42 8)))
9
10  ; Условие 2: x < y (знаковое сравнение)
11  (assert (bvslt x y))
12
13  ; Проверить выполнимость
14  (check-sat)
15
16  ; Если SAT, напечатать модель
17  (get-model)
```

Рисунок 1. Решение SMT-формулы в терминах битвекторной логики.

1.2 Оптимизации конкретно-символьного выполнения

На обработку запросов SMT-решателем в процессе анализа, в зависимости от размеров и сложности формул, может понадобиться от доли секунды до нескольких часов. Поэтому время на решение одного запроса традиционно устанавливается при помощи фиксированного таймаута. Чтобы как можно больше запросов укладывались в отведенное ограничение по времени, в динамической символьной интерпретации много внимания

уделяется различным оптимизациям SMT-формул. Среди наиболее распространенных можно выделить следующие:

- упрощение формул;
- инкрементальные решения;
- кэширование запросов;
- конкретизация части данных;
- слайсинг предиката пути.

Упрощение формул включает в себя наиболее доступные варианты оптимизаций и заключается в синтаксических модификациях выражений: удалении тождеств и логических тавтологий, распространении констант, приведении формул к каноническому виду.

Инкрементальные решения позволяют объединять несколько различных SMT-запросов в единый контекст решателя, для возможности переиспользования части решений между запросами. Данная оптимизация показывает большую эффективность [18] в конкретно-символьном подходе к динамической символьной интерпретации. Поскольку анализ идет вдоль одного пути исполнения, то у последовательных SMT-запросов на инвертирование условных переходов всегда есть довольно большая общая часть выражений из предиката пути. Использование инкрементальных решений [35] с помощью команд (push/pop) позволяет выделять изменяющиеся части формулы (условия ветвления), а выражения для оставшейся чатси предиката пути с частичным решением всегда держать в контексте решателя.

Еще одной оптимизацией, связанной с обработкой SMT-запросов является их кэщирование [36]. Каждый запрос сохраняется (в каноническом или нормализованном виде) вместе с полученным результатом. Если позднее встречается эквивалентная формула, то ответ возвращается сразу из кэша. В том числе, можно переиспользовать формулы из UNSAT-запросов в качестве подвыражений, так как заранее известно, что формула, содержащая противоречивое выражение, тоже будет ложной.

На уровне символьной интерпретации важно оптимизировать символьную модель программы так, чтобы генерировались более простые SMT-запросы. Поэтому одной из успешно решаемых динамической символьной интерпретации задач становится устранение избыточной помеченности в символьных выражениях. Данное явление связано с вовлечением в процесс моделирования слишком большого количества элементов программы, для которых можно отслеживать зависимость от начальных внешних данных. В отличие от недостаточной помеченности, как правило, возникающей в следствие конкретизации символьных данных, которые представляли бы интерес для анализа, избыточная помеченность, наоборот, вызывает включение лишних условий в формулы. Например, добавление в предикат условий от проверки регистра символов внутри функций стандартной библиотеки *tolower/toupper* для всех дальнейших ветвлений создаёт ненужную нагрузку на SMT-решатель. Для устранения подобной избыточности в ситуации с функциями стандартной библиотеки (strlen, memcpy, tolower/toupper, strtoull е др.) вместо символьной интерпретации их как последовательности инструкций можно воспользоваться готовыми формулами для их семантики. Еще одним способом является конкретизация части символьных данных. При конкретизации части программы, не представляющие интереса для анализа (логгирование) или слишком сложные для интерпретации (операции с плавающей точкой), исключаются из процесса символьного моделирования. Очень эффективным подходом является также слайсинг предиката пути [11], позволяющий отсечь нерелевантные для решения SMT-запроса части формулы, не связанные с целевым условным переходом. Слайсинг позволяет значительно сократить размеры формул и ускорить работу SMT-решателей. Эвристический подход представляют собой так называемые «оптимистичные» запросы, когда в запрос на инвертирование включается единственное условие из целевого ветвления. При этом отбрасываются все более ранние составляющие предиката пути. Формально такой запрос обладает

низкой точностью с точки зрения анализа, но, например, при гибридном фаззинге, он может привести к увеличению покрытия. Тем не менее, экспериментально доказано, что дополнительное использование облегченных “оптимистичных“ запросов увеличивает количество успешно сгенерированных тестовых данных, корректно инвертирующих направление целевого перехода [41].

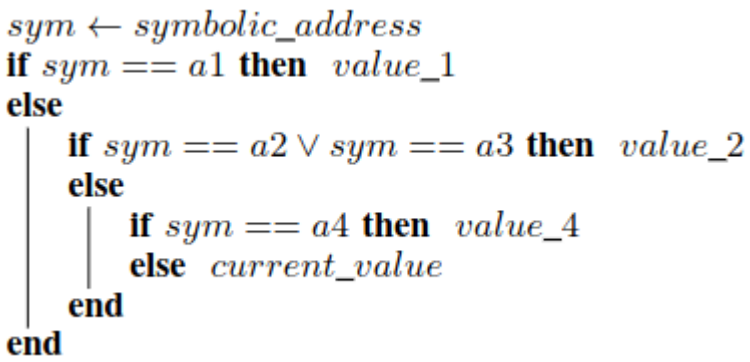


Рисунок 2. Структура SMT-формулы в виде вложенного if-then-else дерева для моделирования символьных адресов.

Отдельного внимания заслуживает возникновение недостаточной помеченности, когда зависимости интересной для анализа части программы больше не отслеживаются используемыми в модели символьными переменными. Предпосылки для недостаточной помеченности связаны с увеличением сложности процесса моделирования символьного контекста выполнения. В качестве одного из факторов возникновения конкретизации полезных символьных выражений можно назвать взаимодействие тестируемого приложения со средой исполнения. Частичная нейтрализация данного явления возможна с помощью отслеживания системных и библиотечных вызовов. Например, за счёт упомянутой ранее известной семантики функций стандартной библиотеки можно использовать комплексное символьное выражение, осуществляющее все необходимые преобразования контекста. Другая причина кроется в использовании символьных выражений для

вычисления адресов при операциях обращения к памяти [42]. Проблема моделирования символьных указателей является одной из фундаментальных в динамической символьной интерпретации бинарных программ. Даже при доступе к конкретному значению памяти по адресу, зависящему от символьных данных, её содержимое становится косвенно символьным. Если такие косвенные зависимости по данным игнорируются, возникает недостаточная помеченность, и, как следствие, потеря потенциальных путей исполнения в программе. Моделирование косвенной адресации позволяет решить эту проблему, однако соответствующая SMT-формула получается довольно громоздкой и рекурсивно вложенной. Пример структуры такой формулы представлен на рисунке 2. С одной стороны, подобная конструкция увеличивает вариативность пространства поиска, что может положительно сказываться на результатах исследования. С другой стороны, символьная адресация приводит к включению в символьные выражения объёмных участков памяти, содержимое которых также может оказаться символьным. В таких ситуациях, приходится искать баланс между производительностью и точностью, ограничивая потребление ресурсов памяти анализатора с помощью конкретизации символьных данных.

1.3. Способы обнаружения программных дефектов

Помимо обнаружения новых путей выполнения методы символьной интерпретации предлагают функционал для целенаправленного поиска программных дефектов и уязвимостей. Одним из первых промышленных символьных интерпретаторов, реализовавших направленные на поиск дефектов методы, стал инструмент Mayhem [43]. Так, в Mayhem были внедрены специальные стратегии обхода путей исполнения, сфокусированные не на увеличении покрытия, а на исследовании потенциально опасных путей (раскручивание итераций циклов). Дополнительно Mayhem отслеживал

зависимости от внешних данных определенных частей программы. При наличии символьного счетчика инструкций или аргументов форматной строки, инструмент генерировал приводящие к проявлению уязвимости входные данные для воспроизведения ошибок. Кроме того, авторы Mayhem первыми прибегнули к термину «предикат безопасности».

Предикаты безопасности бывают различных видов, в зависимости от типа проявляемого ими программного дефекта [12]. С точки зрения символьных выражений они представляют собой набор дополнительных символьных ограничений, содержащих условия возникновения ошибки. Простейшим примером является инструкция деления, для которой можно подобрать входные данные, обращающие операнд-делитель в нуль. При формулировании предикатов безопасности сначала определяются операции с символьными значениями, которые могут потенциально удовлетворять условиям возникновения ошибки. Такие условия добавляются к предикату пути и отправляются в качестве запроса к решателю. Если полученная система символьных уравнений оказывается совместной, SMT-решатель сгенерирует соответствующие тестовые данные. Модели и характеристики основных возможных уязвимостей описаны с помощью специальной классификации CWE (Common Weakness Enumeration) [44]. Некоторые из них, такие как разыменование нулевого указателя или выход за границу массива, могут приводить к повреждениям памяти. Поэтому при верификации наличия ошибки для сгенерированных данных удобно пользоваться специальными обработчиками-санитайзерами. Изначально они были разработаны для исследования программных продуктов корпорации Google [45] в рамках фаззинг-тестирования, так как не всегда наличие ошибки приводит к аварийному завершению выполнения программы. Санитайзеры встраивают набор проверок, который при выявлении присутствия ошибки останавливает запуск вместе с выводом отчета о находке. Для детекции различных классов ошибок существует несколько санитайзеров. AddressSanitizer [46], LeakSanitizer

и MemorySanitizer [47] обрабатывают ситуации, связанные с некорректной работой при обращении с памятью. UnderfinedBehaviorSanitizer отслеживает неопределенное поведение, как в случаях деления на нуль или целочисленных переполнений. Для контроля корректности процесса взаимодействия между потоками применяется ThreadSanitizer [48]. Весомое отличие санитайзеров от предикатов безопасности заключается в том, что они лишь подтверждают наличие программного дефекта, проявленного во время обработки входных данных, не давая пропустить его при анализе. В то время как задача предиката безопасности представляет собой целенаправленное конструирование таких данных. Стоит также отметить, что встраивание инструментации для проверок санитайзерами, как правило, производится на этапе компиляции, что означает необходимость в исходном коде.

Инструмент SAVIOR [49] применяет символьные предикаты безопасности на входных данных, получаемых от фаззера. При этом он исследует не все пути, а только те, которые потенциально могут привести к ошибке. SAVIOR использует статический анализ с целью получения разметки потенциально опасных мест в программе. Разметку инструмент производит с помощью UndefinedBehaviour Sanitizer. Поскольку в реализации SAVIOR используется LLVM/KLEE в качестве основы для символьной интерпретации, то теоретически данный анализатор применим для различных архитектур, но по умолчанию разработчиками SAVIOR заявляется поддержка только для архитектуры x86/x86_64.

В инструменте IntScope [50] предлагался подход конструирования символьных предикатов безопасности для поиска ошибок целочисленных переполнений в бинарном коде в процессе символьной интерпретации инструкций с арифметическими операциями. При этом проверка на наличие ошибки выполнялась при наличии «стока», то есть места в программе, где потенциально некорректный результат операции был бы использован. Такими местами являются, например, условный переход, выделение памяти, вызов

функций и т. п. Инструмент работает только с бинарным кодом x86/x86_64 программ ОС Windows и Linux. Поддержка других архитектур, таких как ARM и RISC-V потребует трансляции кода в промежуточное представление и адаптацию используемых алгоритмов.

1.5. Обзор инструментов динамической символьной интерпретации

В качестве первопроходцев, предложивших concolic-направление динамической символьной интерпретации, можно назвать работы 2005 года с описанием инструмента DART [51] и его улучшенной версией CUTE [52], разработанные одним и тем же коллективом с целью автоматизации создания тестов для программ на языке Си. Первоначальным этапом анализа идёт статический парсинг для создания перечня интерфейсов программы, получающих входные данные извне. Поскольку задумка авторов была в полной автоматизации, для первичной инициализации конкретных значений случайными данными перед запуском инструменту необходимо знать тип переменных. Данная информация извлекается из исходного кода с применением инструментатора CIL [53]. Далее символьный интерпретатор производит исследование ветвлений вдоль полученного пути и приступает к "направленному поиску" с помощью сгенерированных тестовых данных. При этом условия ветвлений предыдущего запуска сохраняются, чтобы не повторять анализ уже изученных путей и иметь возможность открывать новые. В CUTE авторы продвинулись в части обработки символьных выражений, внедрив набор оптимизаций в виде объединения общих выражений, инкрементального решения вложенных предикатов предыдущих ветвлений и проверки отсутствия взаимоисключений из-за отрицания целевого условия ветвления среди ранее добавленных в предикат пути условий. Также простые уравнения для ограничений равенства символьных указателей между собой и проверка их неравенства нулю. На данный момент, подобная предобработка

символьных выражений перед отправкой запроса решателю является стандартной практикой.

Спустя три года при участии разработчика DART, П. Годефройда, была выпущена статья про инструмент SAGE [9], применявшийся для промышленного тестирования продукции компании Microsoft. Хотя среди его принципов работы проглядывает наследие идей вышеописанных интерпретаторов для языка Си, новшество состоит в том, что данный анализатор предназначен для работы непосредственно с бинарным кодом кодом процессорной архитектуры x86. При перечислении причин, которыми руководствовались при этом создатели SAGE, прослеживаются трудности внедрения единообразного процесса тестирования внутри корпорации. В следствие использования различными командами отличающихся языков программирования и иногда несовместимых процессов сборки, бинарный код стал наиболее удобным вариантом для этапа инструментации. Она в данном анализаторе проводится в *оффлайн*-режиме посредством трассирования фреймворком iDNA [54] инструкций предварительно запущенной программы. Уже после запуска для записывания трассы начинается её воспроизведение в процессе символьной интерпретации в соответствии с семантикой инструкций. При добавлении ограничений в предикат пути применяются пометки символьных данных на уровне гранулярности отдельного байта и конкретизация при разыменовании символьных указателей. В силу индустриальных требований перед разработчиками SAGE стояла задача спроектировать механизмы работы для исследования глубоких путей в масштабных приложениях, получающих на вход большие объемы данных, что в контексте символьной интерпретации довольно трудновыполнимо. Решение было организовано посредством приоритизации сгенерированных входных данных в виде алгоритма "генерационного поиска", работающего в циклическом режиме с помощью повторения четырех шагов. Первым делом производится тестовый запуск программы для выявления потенциальных помех

при анализе в виде ошибок или значительного расхода ресурсов памяти. Далее следует повторный запуск для записи трассы. Следующим шагом при просмотре трассы осуществляется оценка входа на основе покрытия новых базовых блоков. Поскольку целью создания входа было отклонение от предыдущего пути, отсутствие новых областей кода свидетельствует о том, что инвертировать ветвление не удалось или новый путь был проложен в ранее изученную часть приложения. Такие входные данные отсекаются назначением низкого приоритета ради повышения эффективности распределения ресурсов. Более того, каждому новому входному файлу сопоставляется значение параметра глубины, после которой можно начинать инвертирование условных переходов. Глубина при этом соответствует целевому ветвлению, для которого был сгенерирован текущий входной файл, ведь все ветвления, встреченные ранее на пути, уже были обработаны в одном из предыдущих запусков. Наиболее приоритетные файлы добираются до финального этапа алгоритма, на котором происходит символьная интерпретация и генерация новых входных данных. В результате представленной работы авторы впервые провели аналогию между конкретно-символьным выполнением и фаззинг-тестированием, обозначая свой подход как "метод белого ящика", а их инструмент SAGE успешно анализировал и находил порядка трети ошибок, обнаруженных методами фаззинга, в многочисленных Windows-приложениях.

Другое направление символьной интерпретации, осуществляемое линейкой связанных между собой инструментов, представляет *онлайн*-выполнение, начавшее свое развитие благодаря усилиям разработчиков из Стэнфорда. Вместо итеративных попыток, каждая из которых ограничивается единственным путем выполнения, в данном подходе предлагается исследование нескольких путей внутри одного и того же запуска. Первым примером реализации такого подхода стал в 2005 г. прототип EGT [55] будущего интерпретатора EXE [56], выпущенного годом позднее. Как и для аналогов того времени DART и CUTE, инструментация проводится с помощью

CIL над исходным кодом, написанном на языке программирования Си. При появлении символьного условного перехода процесс клонируется с помощью системного вызова fork(), чтобы иметь возможность проанализировать оба направления в зависимости от выполнения условия. Так как подобная процедура ведет к экспоненциальному росту числа процессов и потребляемой ими памяти, при клонировании один процесс переходит в режим ожидания, пока другой продолжает выполнение, что по сути воплощает алгоритм поиска в глубину. На случай если активный процесс локализует свое существование в бесконечном цикле в системе предусмотрен лимит времени, после которого он будет завершен, а соответствующий спящий процесс продолжит изучение альтернативного направления ветвления. По причине того, что решатель, использованный в прототипе, не предоставлял возможности для работы с символьными указателями, авторы собственноручно написали SMT-решатель STP, внедрив его в EXE. Этот инструмент поддерживает символьную инструментацию на уровне отдельного бита и способен анализировать обращения по символьным указателям и преобразования типов для символьных данных, а также находить арифметические переполнения и ошибки выхода за границу массива. В общих чертах, процесс инструментирования и символьный анализ в EXE концептуально повторяет принципы работы прототипа. Во время компиляции для каждого выражения, операции присваивания значения переменной и условного ветвления добавляются проверки на символьность операндов. Операнды считаются конкретными, только если каждый бит операнда имеет конкретное значение. Помимо поиска обходом в глубину, в EXE предлагается поисковая эвристика для выбора приоритетных путей среди ожидающих исследования. Вместе с клонированием процесса в символьном условном переходе текущий контекст выполнения отправляется поисковому серверу, после чего процесс засыпает в ожидании ответа. Сервер при помощи контекста выбирает из очереди ветвления с наименьшим числом запусков, после чего командует соответствующему процессу продолжать символьное

выполнение. На некоторое время такой процесс возвращается к методу обхода в глубину, после чего процедура повторяется. После генерации новых входных данных они верифицируются повторным запуском для кода без инструментации, чтобы избежать ложно-положительных результатов. Более того, благодаря подробному погружению в устройство механизмов для решения SMT-запросов данный коллектив сумел разработать качественный набор оптимизаций выражений на основании тождественных преобразований, который вместе с решателем STP [19] был интегрирован в виртуальную машину символьного исполнения KLEE [25]. Преобразования для оптимизации запросов содержат:

- распространение констант для арифметических операций и коэффициентов линейных уравнений;
- замену операциями битового сдвига умножения числа на двойку, возведенную в степень;
- уточнение до наиболее строгих ограничений (например, более слабое условие {x < 10} можно отбросить, если есть условие {x = 5});
- выделение подмножеств ограничений, независимых по переменным.

Более того, накопление вложенных ограничений во время символьной интерпретации позволяет делать обобщенные выводы об отсутствии необходимости в части запросов к решателю, что является весьма предпочтительным. Например, если успешно удалось подобрать решение, удаление части условий не приведет к появлению противоречий среди оставшихся. И наоборот, не требуется проверять совместность предиката, если ранее было выделено его несовместное подмножество. Для этого кэшируется соответствующая информация о ранее произведенных запросах. Нововведением KLEE в части инструментации стала трансляция исходного кода в промежуточное представление LLVM IR [57], что позволило расширить набор поддерживаемых языков программирования. Виртуальный набор инструкций представления LLVM интерпретатор преобразует в символьные

выражения. В организации символьных процессов для различных выборов условных переходов KLEE выполняет обязанности распределения ресурсов и переключения контекста, что отчасти похоже на работу операционной системы. Чтобы избежать путаницы авторы заменяют процесс термином "состояние", обозначающим контекст из набора древовидных символьных выражений для регистров, стека и кучи. Заранее перед клонированием состояния в месте символьного ветвления инструмент проверяет разрешимость обоих направлений с помощью решателя. Если клонирование необходимо, используется компактное представление контекста состояния программы за счет разделяемой памяти и механизма copy-on-write. Среди прочего, в KLEE добавились модели проверок гипотетически опасных ситуаций наподобие неправильных обращений к памяти и деления на нуль. Несмотря на солидный возраст виртуальная машина KLEE до сих пор поддерживается и продолжает свое развитие. Во многом это связано с значительным количеством сторонних разработчиков, переиспользующих KLEE. На данный момент, список таких работ насчитывает более трехсот публикаций. Несмотря на то, что базовый принцип работы KLEE сглаживает архитектурные зависимости среди набора процессоров для его регулярного тестирования с 2024 г. появился Apple Silicon архитектуры ARM.

Одной из ранних разработок, основанных на предыдущем инструменте, является фреймворк анализа программного обеспечения S2E [27], предназначенный для автоматизированного исследования путей, сопровождаемого применением разнообразных модульных анализаторов. Задачи таких инструментов могут включать поиск программных дефектов, сбор отладочной информации и статистики или верификацию заявленных свойств приложения. Помимо предоставляемых по умолчанию модулей, пользователь может самостоятельно сконструировать подходящий под его цели инструмент. S2E, будучи ориентированным 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. Модели целостности определяют желаемую степень строгости для условия достижимости моделируемого пути. Снижение строгости данного условия может быть использовано в целях увеличения производительности, если пользователь согласен на уступку в виде ложно-положительных срабатываний. Наиболее строгая модель предполагает анализ на основе полностью конкретных данных. Выборочная символьная интерпретация призвана сосредоточить фокус анализатора на ветвлениях в целевой области исполняемого кода, в которой могут применяться символьные данные. К примеру, можно установить границу при переходе в код системного или библиотечного вызова. Тогда для условных переходов внутри внешних библиотек не будут создаваться запросы на инвертирование, но при возвращении в целевой код этот процесс возобновится. Сама виртуальная машина S2E работает с бинарным кодом программы посредством свободной многофункциональной платформы для эмуляции QEMU [58]. QEMU официально поддерживает несколько архитектур (включая ARM и RISC-V), для которых машинный код может быть преобразован независимое внутреннее представление TCG. В S2E после создания такого представления происходит его трансляция в LLVM IR, из которого далее извлекает символьный формулы инструмент KLEE. Хотя подобная конструкция обладает существенным преимуществом универсальности, это достигается в ущерб скорости работы анализатора [59] из-за нескольких этапов трансляции между представлениями.

Похожие идеи были воплощены в 2016 г. в фреймворке angr [26], создание которого позиционировалось как подытоживание и структуризация накопленных знаний в области автоматизированного анализа программного обеспечения. Фреймворк предназначен для удобства создания собственных инструментов исследования бинарного кода. Для этого в angr реализован интерфейс доступа к модулям чтения разнообразных форматов исполняемого кода, их дизассемблирования и трансляции в абстрактное представление VEX

IR [60], для инструкций которого доступны различные преобразования и, в том числе, техники символьной интерпретации. С точки зрения кроссплатформенности, на данный момент, модули фреймворка полноценно поддерживают ряд архитектур, среди которых есть ARM, а также возможна интерпретация отдельных инструкций RISC-V, для которой пока не реализованы методы для отслеживания неявных переходов и соглашения о вызовах. Недостатком, как и в случае S2E, является замедление, которое связано с тем, что angr написан на языке программирования Python. Примечателен данный фреймворк тем, что для демонстрации его возможностей авторы впервые реализовали концепцию гибридного фаззинга в инструменте Driller [61]. Ключевое наблюдение состоит в разделении всех возможных входных данных на обширный класс, легко обнаруживаемый мутационным фаззингом, и узкие специфические случаи, доступные символьному интерпретатору благодаря дорогостоящим вычислениям. Поэтому Driller комбинирует эти подходы, избирательно применяя символьные техники фреймворка angr, когда работа открытого инструмента мутационного фаззинга AFL [62] перестает приносить прирост покрытия. При запуске символьного интерпретатора, тот получает входные данные и записывает соответствующие им трассы выполнения, с помощью которых требуется выявить ранее не инвертированные условия переходов, которые оказались слишком сложными для фаззера. Вычислив новые входные данные, потенциально открывающие пути в неизведанные области кода, инструмент отправляет их фаззеру и прекращает работу, пока ситуация с покрытием не повторится.

Следующим значимым этапом стало появление динамического символьного интерпретатора QSym [29] в 2018 г. Данный инструмент вслед за Driller развивал направление гибридного фаззинга. Фундаментальной идеей и отличием, стоящими за этим анализатором, оказалось качественное ускорение производительности символьного интерпретатора, достаточное для совместного применения с фаззером в параллельном режиме. Сделав скорость

работы приоритетом, авторы QSym систематически избавлялись от времязатратных составляющих. Первой среди них оказалась трансляция в упрощенное промежуточное представление. Вместе с экономией времени, этот шаг принес возможность динамической инструментации машинных инструкций, выполняемой с помощью фреймворка Pin [63]. Инструментация производится только в случае присутствия символьных операндов. Как отмечают авторы, таких инструкций в рамках помеченных базовых блоков оказывается, как правило, меньше пятидесяти процентов. Далее авторы воспользовались вариантом символьного оффлайн-выполнения, избавившись от необходимости выделения дополнительных ресурсов памяти на сохранение переиспользуемого контекста при выборе направления условного перехода. Это связано с тем, что более вероятное направление, скорее всего, удастся ранее изучить фаззингом, а значит символьному интерпретатору неинтересно повторять уже проделанную работу и проводить поиск для обоих вариантов ветвления. Тем не менее, если это потребуется, за счет высокой производительности можно использовать перезапуск. Более того, в QSym применяется контекстное кэширование символьных базовых блоков, позволяющее, с одной стороны, различать источники перехода в базовый блок, отвечающие за разные пути, а с другой, постепенно ограничивать ресурсы на инвертирование ветвлений в многократно повторяющемся контексте. Авторы также отказались от анализа системного окружения, конкретизируя поток символьных данных, передаваемых внешним вызовам. Ещё одно нововведение представляют собой оптимистичные решения. Если не удается получить решение запроса для предиката пути, убирается фиксация предыдущих условий переходов и оставляется единственное условие для инвертирования целевого перехода, которое, как показывает практика, часто приводит к достижению поставленной задачи. 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 гибридного фаззинг Angora [33]. В отличие от классический символьных интерпретаторов вроде QSYM, он использует не SMT-решатели, а комбинацию анализа помеченных данных (taint tracking) и градиентных методов для решения символьных ограничений. Это позволяет существенно ускорить работу и обойти «бутылочное горлышко» SMT-решателей. Angora компилирует программу в LLVM IR, помечает байты входных данных специальными метками и отслеживает, какие ветвления зависят от этих данных. Это даёт точное понимание того, какие именно байты внешних данных влияют на условие перехода. Вместо SMT-запросов Angora строит модель ветвления и использует локальный поиск (градиентоподобный алгоритм), чтобы подобрать значения входных байтов, удовлетворяющих условию. Это позволяет быстро генерировать новые входные данные для расширения тестового покрытия. Поскольку Angora опирается не на бинарный код, а на инструментацию LLVM, то теоретически поддерживаются различные архитектуры с LLVM-бэкендом. Однако на практике анализируются только x86_64 бинарные программы для Linux, вследствие чего поддержка других архитектур остается под вопросом.

Инструмент динамического символьного исполнения Manticore [64] был представлен в 2019 году и задумывался как универсальная платформа для анализа бинарных программ, смарт-контрактов и даже исходного кода на некоторых языках. В отличие от многих более узконаправленных решений, Manticore стремится к многоцелевому применению и гибкости. Как и angr, Manticore написан на Python и предоставляет удобный API для исследователей безопасности и разработчиков анализаторов. Но вместе с тем использование интерпретируемого языка ограничивает его применимость в рамках гибридного фаззинга из-за падения производительности. Manticore поддерживает символьную интерпретацию на уровне байткода и бинарного кода. Он умеет

работать с бинарными программами Linux в формате ELF, а также со смарт-контрактами Ethereum (EVM) и байт-кодом WASM. Последнее отличает его от многих других инструментов, которые ориентированы исключительно на классический нативный бинарный код. В Manticore реализован анализ для машинного кода платформ x86/x86_64, AArch64 и ARMv7. Однако разработчики прекратили поддержку данного инструмента, и его развитие за исключением предложений изменений (PR) от внешних пользователей не планируется.

Следующая пара инструментов SymCC [30] и SymQEMU [31] были представлены командой Себастьяна Поплау в 2020 и 2021 годах, соответственно. В SymCC в качестве очередного способа повышения производительности предлагалось отказаться от динамической инструментации, повторяющейся при каждом запуске тестируемого приложения, заменив её на статическую инструментацию на этапе компиляции. Такой подход применяется для добавлении проверок санитайзеров при создании исполняемых файлов. Также подобный вариант можно увидеть в вышеописанных ранних работах по динамическому символьному выполнению – DART, CUTE и EXE, где функциональность для создания символьных выражений внедрялась непосредственно в исходный код на языке программирования Си. К преимуществам данного метода можно отнести единоразовые затраты времени на дополнительную работу компилятора при инструментировании кода, после чего конструирование символьных формул будет производиться нативно вместе с выполнением программы для любого последующего запуска. Чтобы миновать специфические особенности компиляции каждого отдельного языка программирования, авторы вернулись к применению внутреннего представления в виде биткода LLVM, на уровне которого вставляются символьные проверки и преобразования. Для включения в область исследования внешнего окружения подключаемые библиотеки тоже должны быть скомпилированы посредством SymCC. Можно использовать

сторонний код и без инструментации, что будет равносильно конкретизации внешних вызовов. Подобно S2E, использующему KLEE при работе с формулами, для решения символьных выражений и взаимодействия с SMT-решателем в SymCC предоставляется возможность задействовать соответствующие оптимизированные компоненты из QSym, либо более простую авторскую реализацию для работы с решателем Z3 [20].

Так как требование наличия исходного кода является существенным ограничением, что отмечали ещё разработчики SAGE десятилетием ранее, подход встраиваемой символьной инструментации для бинарного кода был реализован в инструменте SymQEMU. Вместо представления LLVM IR в нем модификации применялись к промежуточному представлению TCG (Tiny Code Generator) посредством дополнительного набора инструкций для данного представления, которое затем преобразуется обратно в машинный код заданной архитектуры. В данном случае, TCG ops применяется исключительно в качестве этапа кроссплатформенной инструментации. Так что вместо эмуляции с помощью QEMU полученный исполняемый файл в дальнейшем запускается на процессоре с целью сохранения высокого уровня производительности. Аналогично предыдущему инструменту в SymQEMU символьные формулы обрабатываются с помощью частичного заимствования функционала Qsym.

Конкретно-символьный исполнитель SymFit [65] основан на инструменте SymQEMU, но с рядом важных улучшений производительности. SymFit 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 счёт разграничения кэшей обработанного SymFit кода в конкретном и

символьном режимах, снижаются накладные расходы и ускоряется выполнение программы. Такое динамическое переключение позволяет SymFit достигать более высокой производительности по сравнению с SymQEMU, особенно на реальных приложениях. Как и SymQEMU, за счет использования промежуточного представления QEMU TCG, SymFit может быть легко портирован для анализа бинарного кода архитектур RISC-V и ARM.

Инструмент SymSan [66] по своему подходу больше напоминает SymCC, поскольку также использует инструментацию LLVM на этапе компиляции анализируемой программы. 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 как и Angora, теоретически может поддерживать различные архитектуры, но пока по умолчанию он предназначен только для x86_64 программ.

Еще один известный ориентированный на применение в гибридном фаззинге инструмент FUZZOLIC [32] был представлен в 2021 году. В отличие от классических решений, таких как QSYM или SymQEMU, он делает акцент на более тесной интеграции с инфраструктурой QEMU и внедрении новых методов оптимизации работы с символическими выражениями. Одним из ключевых новшеств FUZZOLIC является использование блоков TCG для построения символьных выражений. Если в QSYM или SymQEMU инструментирование осуществлялось на уровне отдельных инструкций, что влекло дополнительные накладные расходы, то FUZZOLIC обрабатывает предварительно оптимизированные базовые блоки после трансляции QEMU. Такой подход позволяет значительно снизить издержки на трассировку и повысить производительность символьного анализа. Вторым важным отличием является архитектура, в которой трассировщик и решатель вынесены в разные процессы и взаимодействуют через разделяемую память. Это снимает

зависимость скорости анализа от работы SMT-решателя и обеспечивает более плавное масштабирование при увеличении числа запросов: в предшествующих системах решение формул часто блокировало процесс исполнения, превращаясь в узкое место. Кроме того, FUZZOLIC поддерживает символическую обработку 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. Одним из достижений коллектива, создавшего FUZZOLIC, является разработка нового решателя Fuzzy-Sat. В отличие от традиционных SMT-решателей, таких как Z3, Fuzzy-Sat ориентирован на приблизительные решения: вместо строгой корректности он предлагает достаточно точный и быстрый результат, который позволяет исследовать новые пути исполнения, в том числе, при гибридном фаззинге. Такой компромисс между точностью и скоростью открывает новые возможности для повышения покрытия кода без существенного роста вычислительных затрат. Несмотря на то что FUZZOLIC построен на базе QEMU, он не поддерживает архитектуры ARM и RISC-V.

Фреймворк конколической интерпретации Triton [67] предоставляет примитивы для проведения динамического анализа исполняемого кода, основываясь на инструментации с помощью Pin [63]. Компоненты Triton включают в себя методы для отслеживания помеченных данных, создания символьных выражений и модификации ограничений предиката пути, сохранения контекста символьных состояний, интерфейс для передачи запросов SMT-решателю а также конструкторы 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 возможных инструментов, созданных на основе Triton, могут

выступать средства воспроизведения трасс выполнения, сбора покрытия, символьные интерпретаторы, отладчики и др. Из существующих символьных интерпретаторов можно назвать описанные ниже TritonDSE и Sydr [11], которые входят в комплексы оркестрации гибридного фаззинга PASTIS [67] и Sydr-Fuzz [13], соответственно. Символьные формулы, представимые в виде абстрактного синтаксического дерева, имеют вложенную структуру. В Triton с целью сокращения ресурсов для их обработки применяются уникальные идентификаторы, с помощью которых при конструировании новых символьных выражений можно ссылаться на предыдущие, используя их в качестве подвыражений. При этом можно проводить манипуляции на уровне отдельного бита, но при выполнении, например, сложения или других операций с битовыми векторами их размеры должны соответствовать. Triton поддерживает решение запросов при помощи SMT-решателей Z3 [20] и Bitwuzla [21], хотя пользователь может добавить другой решатель самостоятельно.

Коллектив, создавший Triton, также разработал написанный на Python символьный интерпретатор TritonDSE, который относится к инструментам динамического анализа. Конкретный запуск в нем применяется для предварительной записи трассы выполненных инструкций посредством динамического бинарного инструментатора QBDI, который также используется для сбора покрытия. 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. Поскольку полноценного моделирования взаимодействий с операционной системой в данном анализаторе не происходит, часть результатов может оказаться ложно-положительными.

Символьный интерпретатор Sydr использует динамическую бинарную инструментацию с помощью фреймворка DynamoRIO. Архитектура Sydr построена на параллельной работе двух процессов – конкретном и символьном вычислителях (Рисунок 3). Впервые подобная концепция была описана в 2012 г. в статье [43], представившей закрытый инструмент Mayhem, в которой также впервые был предложен термин "предикат безопасности". Работа символьного исполнителя в Sydr основывается на применении интерфейсов Triton для конструирования символьных выражений. Базовый сценарий работы инструмента предполагает перенаправление потока инструкций от динамического инструментатора в символьный исполнитель. Однако символьный процесс может также использоваться в режиме эмуляции, который организуется с помощью написания дополнительного конфигурационного файла. В данном символьном интерпретаторе 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.

Символьный интерпретатор Sydr входит в комплекс динамического анализа Sydr-Fuzz, с помощью которого можно проводить гибридное фаззинг-тестирование с использованием Sydr и таких инструментов фаззинга с обратной связью по покрытию как libFuzzer [6], AFL++[7] и honggfuzz [8]. 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, а также создавать отчеты о найденных аварийных завершениях.

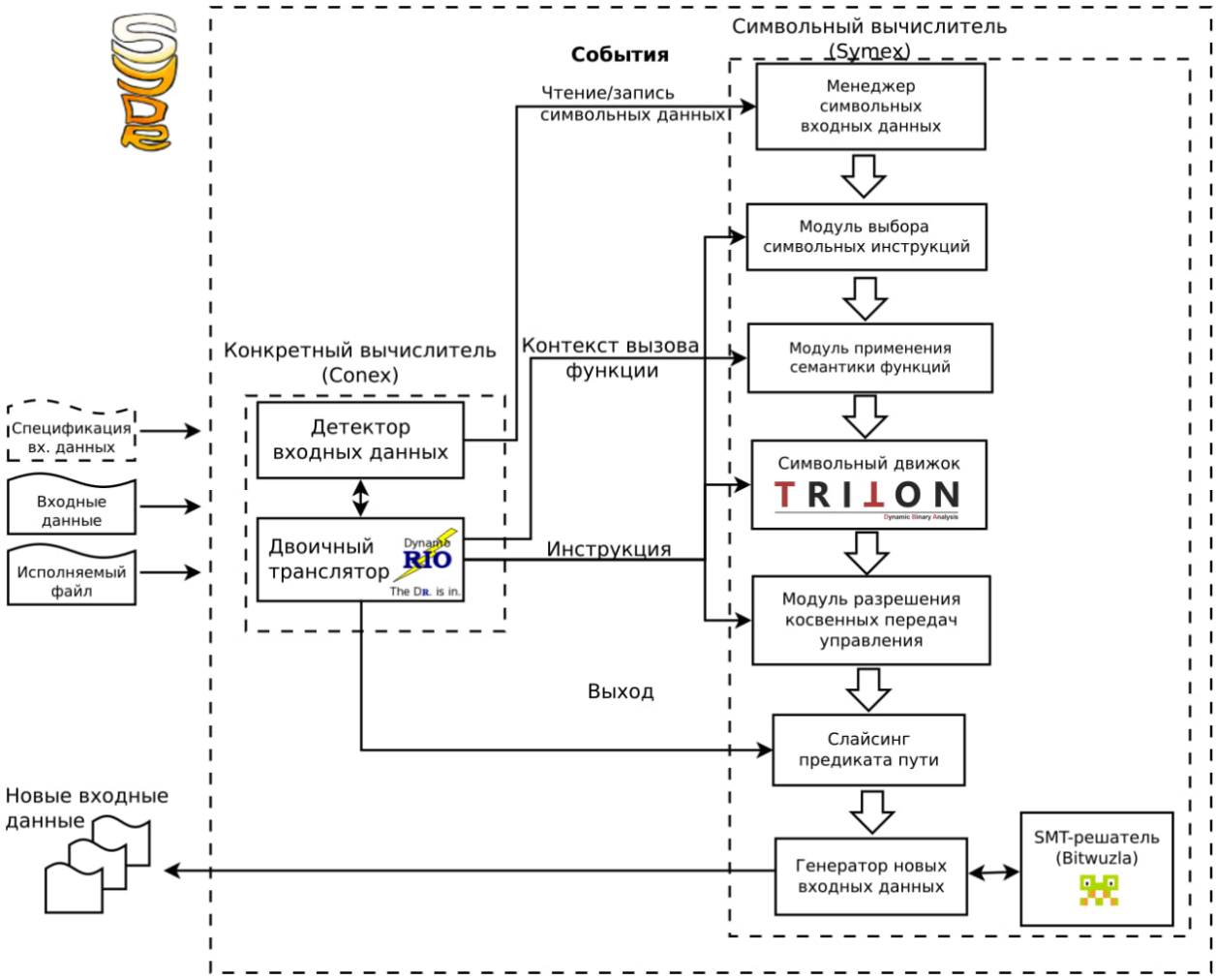


Рисунок 3. Архитектура инструмента динамической символьной интерпретации Sydr.

1.5. Символьная интерпретация бинарного кода для архитектур Байкал-М (AArch64) и RISC-V

Разработанный компанией «Байкал Электроникс» российский процессор Байкал-М основан на архитектуре ARMv8-A [69]. Производство данного чипа, ориентированного на использование в настольных компьютерах и тонких клиентах, началось в 2020 году. Байкал-М использует ядра Cortex-A57, то есть готовую лицензионную микроархитектуру ARM. Это значит, что он реализует

ISA ARMv8-A (AArch64) полностью, а также поддерживает исполнение 32-битного кода в режиме AArch32.

RISC-V [70] – это открытая и свободная архитектура набора команд (ISA), представленная в 2010 году командой исследователей Калифорнийского университета в Беркли. В отличие от ARM или x86, RISC-V предоставляет свободные от лицензионных отчислений спецификации для создания собственных решений, что сделало архитектуру особенно популярной в академическом сообществе и производстве встраиваемых систем. Лаконичность, модульность и расширяемость RISC-V позволяют использовать её как в микроконтроллерах и IoT-устройствах, так и в серверных процессорах и ускорителях. Сегодня RISC-V активно развивается международным консорциумом RISC-V International и рассматривается как стратегическая альтернатива проприетарным архитектурам. В рамках поддержки данного стандарта российскими компаниями в 2022 году был создан Альянс RISC-V.

При анализе бинарного кода последовательность преобразований символьных данных описывается на основе трассы выполнения, то есть последовательности инструкций выполняющейся программы. Для каждой процессорной архитектуры существует набор инструкций, описание которого закреплено в виде спецификаций. Вместе со спецификациями инструкций архитектурный стандарт, как правило, включает набор регистров, которые могут быть использованы в качестве операндов инструкции или иметь специальное назначение. Кроме того, в стандарте описываются механизмы адресации при обращениях к памяти, а также механизмы передачи управления, соглашение о вызовах, порядок организации хранения данных на стеке и пр. Отличительная особенность RISC-архитектур состоит в том, что инструкции имеют фиксированный одинаковый размер, что облегчает их декодирование.

Концепция архитектур Байкал-М (AArch64) и RISC-V предполагает разграничение арифметических операций и операций обращений к памяти. Стоит упомянуть, что упрощение функционала отдельной инструкции, к

которому тяготеют вышеуказанные архитектуры, приводит к увеличению среднестатического размера базовых блоков (линейных последовательностей инструкций, заканчивающихся инструкцией передачи управления). Также при этом снижается сложность моделирования операционной семантики инструкции при проведении динамической символьной интерпретации, что является одним из фактором стремления разработчиков анализаторов для x86//x86_64 использовать упрощенное промежуточное представление. В отличие от легковесного декодирования при анализе бинарного кода описанных RISC-архитектур, x86//x86_64 [71] как классическая CISC-архитектура с большим и сложным набором инструкций требует значительно больше усилий на данном этапе. Помимо того, что размер инструкции в x86//x86_64 может варьироваться от 1 до 15 байт, некоторые из них реализуют достаточно высокоуровневые операции, например обработку строк. При этом размер инструкции в случае Байкал-М (AArch64) составляет фиксированное значение 4 байта, а для RISC-V помимо 4-байтовых стандартных инструкций присутствует набор сокращенных (или сжатых) инструкций размером 2 байта. Стоит отметить, что для архитектуры Байкал-М (AArch64) различные режимы выполнения Thumb [72] неактуальны, поскольку для 64-битного ARMv8-A существует только один фиксированный режим выполнения.

Также необходимо упомянуть, что обе архитектуры являются модульными, то есть существует некоторый базовый набор инструкций, который может быть дополнен опциональными расширениями. Так, для RISC-V существует множество стандартных расширений [73]: M – целочисленное умножение и деление, A – атомарные инструкции, F – работа с 32-битными регистрами с плавающей точкой, D – расширение для 64-битных чисел с плавающей точкой, Q – 128-битные числа с плавающей точкой, C – сжатые 2-байтовые инструкции. Для поддержки определенного расширения с точки зрения символьной интерпретации необходимо поддержать в зависимости от используемого метода инструментации либо символьную семантику данных

инструкций, либо обеспечить их трансляцию в промежуточное представление. В данном случае ситуация аналогична анализу x86/x86_64 архитектуры, для которой также существуют свои расширения (SSE, AVX, AVX-512).

Как видно из обзора методов анализа и инструментов, большинство существующих реализаций направлено на анализ программ архитектуры x86/x86_64. Это обосновывается широким распространением и популярностью данной архитектуры. Более того, основная часть современных инструментов являются результатом академических исследований, поэтому поддерживается и тестируется преимущественно одна целевая x86-архитектура, даже если сам подход зависит от промежуточного представления и абстрагирован напрямую от архитектурно-зависимых деталей. Те инструменты, которые поддерживают анализ бинарного кода для архитектур Байкал-М (AArch64), RISC-V являются либо прототипами, либо поддерживают какую-то одну отдельную архитектуру, либо имеют очень ограниченную функциональность в контексте применения для анализа реальных программ. Например, поддержка обеих архитектур имеется в фреймворке angr. Однако при попытке организовать динамический анализ больших программ методом конкретно-символьного исполнения, возникают проблемы, связанные как с производительностью самого инструмента, так и с отсутствием реализации важных средств анализа. Для RISC-V в angr не поддерживается функционал диспетчера косвенных переходов и соглашения о вызовах, что делает символьную интерпретацию таких программ очень трудозатратной для специалиста по безопасности. Такие инструменты как BinSym [74] и SymEx-VP [75] реализуют поддержку только 32-битной версии архитектуры RISC-V. BinSym реализует символьную интерпретацию посредством эмуляции бинарного кода, производительность которой не подходит для гибридного фаззинга. Инструмент SymEx-VP специально предназначен для анализа узкого сегмента ПО в виде прошивок отдельных аппаратных устройств интернета вещей (IoT, Internet of Things). Платформа для анализа BINSEC [76] предоставляет возможности для анализа

бинарного кода архитектур x86, ARM и RISC-V, в том числе реализуется символьная интерпретация. Однако, так же как и в BinSym, она реализована в виде эмуляции бинарного кода, что сокращает возможности практического применения на реальных программах и в гибридном фаззинге.

1.6. Выводы

На основании проведенного обзора подходов динамической символьной интерпретации, можно заключить, что для архитектур Байкал-М (AArch64) и RISC-V среди всех инструментов, полноценную поддержку анализа бинарного кода могут обеспечивать только инструменты SymQEMU и SymFit (расширение SymQEMU). За счет использования промежуточного представления QEMU TCG, поддержка символьной интерпретации бинарного кода для Байкал-М (AArch64) / RISC-V обеспечивается довольно просто. Данные инструменты, в отличие от многих других, обладают достаточной производительностью для гибридного фаззинг-тестирования, т. е. имеют широкое практическое применение. К недостаткам данных инструментов можно отнести скудный функционал символьной интерпретации, основанной еще на кодовой базе QSym. Так, в этих инструментах отсутствует разрешение символьных указателей, возможность распараллеливания запросов к SMT-решателям, нет поддержки символьных предикатов безопасности для целенаправленного поиска ошибок. Ввиду всего вышеперечисленного, задача разработки метода гибридного фаззинга для бинарных программ архитектуры Байкал-М (AArch64) и RISC-V является актуальной.

Глава 2. Метод динамической символьной интерпретации бинарного кода архитектуры Байкал-М (AArch64)

2.1. Особенности моделирования символьного контекста для AArch64

Моделирование символьного контекста выполнения для бинарного кода AArch64 включает набор архитектурно-зависимых компонентов для работы с низкоуровневым представлением программы. В силу ограничений, накладываемых на решение запросов SMT-решателем, в процессе интерпретации при составлении символьных выражений учитываются преобразования, соответствующие операционной семантике целочисленных инструкций. Это связано с тем, что вычисления для значений с плавающей точкой существенно замедляют работу решателя и, как следствие, решить запрос в установленное время не удается. 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. В предлагаемом методе динамической символьной интерпретации используется контролируемое динамическим бинарным инструментатором конкретное выполнение программы. Инструментатор

перенаправляет информацию о текущей исполняющейся инструкции процессу обработки символьного представления вместе с необходимыми конкретными значениями. Символьный исполнитель дизассемблирует операционный код инструкции, который в случае AArch64 всегда составляет 32 бита (т. е. 4 байта). Также производится набор проверок на наличие символьных операндов и необходимость обновления символьного стека вызовов. Для архитектуры AArch64 инструкциями, осуществляющими вызов функций, являются *BL* и *BLR*, а для возврата из функции используется инструкция *RET*, которая по сути передает управление на инструкцию, адрес которой хранится в регистре x30 (Procedure Link Register), и является сокращенной версией (alias) записи инструкции *BR x30*. Отслеживание данных инструкций необходимо для сохранения консистентности 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.

Одной из особенностей архитектуры ARM/AArch64 можно назвать выделение в отдельную группу инструкций обращения к памяти (load/store), что в целом свойственно семейству RISC-архитектур. При этом возможные размеры области памяти, к которой может обращаться инструкция, определяются степенями двойки и составляют 1, 2, 4 и 8 байтов. Само значение размера задается непосредственно кодом инструкции и может не совпадать с размером используемого для его хранения регистра. При присвоении прочитанного значения в регистр большего размера происходит знаковое или беззнаковое расширение, что также зависит от семантики выбранной инструкции. В Triton для работы с битовыми векторами, отображающими значения области памяти, к которым обращается инструкция, применяется абстрактный класс *MemoryAccess*. Данный класс хранит информацию об идентификаторах регистровых операндов, с помощью которых вычисляется

целевой адрес, а также об операнде для хранения самого значения, читаемого или записываемого в память. При реализации предлагаемого метода в данном интерфейсе Triton 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.

Среди прочего, в механизмах адресации с индексированием для архитектуры ARM/AArch64 присутствует несколько необычный формат применения коэффициента масштабирования к индексному регистру. Обычно данный коэффициент представляет собой дополнительный константный числовой множитель. Например, в выражении «Addr = Reg1 + Reg2 * 8», в качестве такого значения используется число 8, которое умножается на индексный регистр Reg2, после чего происходит сложение с базовым регистром Reg1, в результате чего получается итоговое значение адреса. В AArch64 вместо готового множителя внутри инструкции обращения к памяти предусмотрена возможность применения операции битового сдвига влево, которая может быть совмещена с операцией расширения индексного регистра. Данная операция эквивалентна по сути умножению на степени двойки от 0 до 3, что позволяет адресовать блоки памяти размером от 1 до 8 байтов.

Примеры таких инструкций загрузки значений из памяти с расшифровкой производимых ими операций продемонстрированы в первых трёх строках таблицы на Рисунке 4. Первая из приведенных инструкций использует сдвиг влево (обозначенный как *lsl* в соответствии с названием выполняющей аналогичную операцию инструкции), в то время как две остальные совмещают битовый сдвиг с операцией расширения индексного регистра w2. Поскольку третья инструкция читает для вычисленного адреса всего один байт, прочитанное значение также подвергается операции расширения до размера целевого регистра w0.

```
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.

На уровне отдельной инструкции при вычислении символьного адреса с использованием вышеописанного механизма встроенного сдвига не возникает существенных сложностей. Однако 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 4 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.

Подробнее механизм обнаружения и обработки табличных переходов будет рассмотрен в следующем подразделе.

Изменение флаговых значений в AArch64 происходит с помощью специальных инструкций *CMP* и *TST*, а также предназначенных для этого арифметических инструкций, обычно имеющих в названии постфикс «S». Например, инструкция сложения *ADD* не затрагивает флаги, а для инструкции *ADDS* они будут выставлены. Также в отдельную семантическую группу выделяются инструкции, в вычислениях которых дополнительно используется флаг переполнения (*carry*-флаг).

При конструировании символьных выражений для предикатов безопасности, проверяющих возможность ошибки целочисленного переполнения, учитываются как знаковые, так и беззнаковые переполнения для инструкций сложения, вычитания, умножения, а также операций битовых сдвигов. Для этого с помощью символьного исполнителя производится развертывание абстрактного синтаксического дерева, сгенерированного Triton для операционной семантики выбранной арифметической инструкции. Данная операция выполняется посредством предназначенного для этого интерфейса Triton, позволяющим получать дочерние узлы вершины абстрактного дерева. Во время разбора из него требуется извлечь подвыражения, соответствующие исходным операндам инструкции. С их помощью воспроизводится арифметическая операция с добавлением условия присутствия переполнения, тем самым формируя предикат безопасности. Для корректного разбора синтаксического дерева и воссоздания нужной операции следует принимать во внимание способ составления семантики на языке битовых векторов. В качестве примера, на Рисунке 4 приведен фрагмент функционального метода Triton для конструирования операционной семантики инструкции SBC, реализующей вычитание с учетом флага переполнения (*cf*).

```
/* Получение символьных операндов */
auto op1 = this->symbolicEngine->getOperandAst(inst, src1);
auto op2 = this->symbolicEngine->getOperandAst(inst, src2);
auto op3 = this->symbolicEngine->getOperandAst(inst, cf);

/* Конструирование выражения для операционной семантики */
auto node = this->astCtxt->bvadd(
    this->astCtxt->bvadd(
        op1,
        this->astCtxt->bvnot(op2)),
    this->astCtxt->zx(dst.getBitSize()-1, op3)
);
```

Рисунок 4. Конструирование символьного выражения операционной семантики инструкции вычитания с битом переноса в терминах битовых векторов.

Итоговое древовидное выражение семантики будет основано на двух операциях сложения битовых векторов (*bvadd*), одно из которых вложено в другое. Вычитание здесь превращается в сложение, так как значение исходного операнда-вычитаемого будет взято с отрицанием (на рис. 4 обозначено *bvnot(op2)*) в виде дополнительного узла в дереве. Поскольку подобная инструкция фактически включает две арифметические операции сложения, допускающие возможность переполнения, будет сконструировано два предиката безопасности. При развертывании итогового выражения первым будет получен узел типа triton::ast::BVADD_NODE, содержащий сложение подвыражения для беззнаково расширенного (*zx* на рис. 4) до размера регистра флага переполнения с результатом сложения непосредственно операндов инструкции, которое будет представлено вложенным подвыражением типа triton::ast::BVADD_NODE. Из него также извлекаются операнды для составления предиката безопасности.

2.2. Обнаружение косвенных переходов в бинарном коде

Ветвления с множественным выбором (в виде конструкции switch) предполагают наличие нескольких вариантов выбора следующей инструкции при передаче 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 начальных инструкций таких базовых блоков последовательно хранятся в памяти. Выбор нужного участка кода происходит на основе вычисления адреса нужной ячейки памяти. Данный адрес формируется посредством добавления смещения к базовому адресу участка памяти, где располагается начало таблицы указателей на исполняемый код. Таким образом, возникает две таблицы: одна, состоящая из базовых блоков, в исходном коде, а вторая располагается в памяти в виде набора указателей на базовые блоки из первой таблицы.

На Рисунке 5 выделен приведен пример ассемблерного кода для архитектуры AArch64, содержащий табличный переход (в Приложении 2 можно также ознакомиться с исходным кодом данной программы на языке программирования Си). Светлой областью выделен базовый блок, в котором происходит чтение адреса целевого перехода из памяти и передача управления. В данном случае она представлена вызовом функции и осуществляется с помощью инструкции BLR x8. Как можно заметить, для вычисления базового адреса таблицы указателей 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,

```
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
```

Рисунок 5. Пример организации таблицы переходов в ассемблерном коде.

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*.

Детектирование таких таблиц 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*.

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 также организуется в виде хранящейся в памяти таблицы смещений, для которой осуществляется операция чтения с помощью load-инструкции тоже на основе индексируемого базового адреса. Размеры блоков памяти для хранения отступов составляют от 1 до 8 байтов. Поскольку ресурс значения смещения для кодирование номера инструкции или записи, соответствующей некоторому базовому блоку, ограничен, при добавлении значения смещения в адресное выражение, оно умножается на соответствующий размер. Как было упомянуто в предыдущем разделе, в случае архитектуры AArch64 для подобных вычислений используется встроенный механизм арифметических инструкций в виде операции битового сдвига на константное значение.

На Рисунке 6 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. 5) по второй инструкции загрузки адресного значения. В данном примере снова источником выбора значения отступа становится регистр x8, полученный из предыдущего базового блока (как по управлению, так и по расположению в коде программы). После чтения значения одного байта из памяти, оно умножается на 4 и добавляется к базовому адресу начала таблицы. Для корректной обработки метод предусматривает аналогичный множитель в модели предиката для косвенных ветвлений для таблиц смещений.

```
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>
```

Рисунок 6. Базовый блок косвенной передачи управления на основе таблицы смещений.

Если условие switch-выражения оказывается символьным, с высокой вероятностью символьными окажутся вычисления с выбором отступа. 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. 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.

2.3. Выводы

В данной главе был представлен разработанный метод динамической символьной интерпретации бинарного кода программ архитектуры Байкал-М (AArch64). Метод способен распознавать косвенные табличные переходы в машинном коде и точно определять их границы для исследования всех направлений ветвлений на основе анализа инструкции сравнения из предыдущего базового блока. Также при разработке метода были выявлены и исправлены несколько ошибок в интерфейсах открытой символьной библиотеки Triton.

Глава 3. Метод символьной интерпретации целочисленных инструкций архитектуры RISC-V

3.1. Общая характеристика метода

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. Исключением являются принадлежащих M-расширению инструкции, связанные с операциями умножения и деления. Кроме того, в базовом расширении 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. Метод способен конструировать символьные AST-выражения в терминах битовых векторов, отображающие операционную семантику инструкций, для которых осуществляется символьная интерпретация. Соответствующие функциональные методы и генерируемые ими символьные выражения в данной работе также называются "символьной семантикой". 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. 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. Листинг набора инструкций, реализованных в рамках предлагаемого метода, приведен в Приложении 3.

3.2. Реализация предлагаемого метода

Предлагаемый метод символьной интерпретации набора целочисленных инструкций был интегрирован в модуль разрешения архитектурных зависимостей символьной библиотеки Triton, схематично представленной на Рисунке 7. 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 [77], 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. Несмотря на то что конструирование семантики производится только для вышеперечисленных в подразделе 3.1 модулей расширений, для проведения этапа дизассемблирования используются также идентификаторы инструкций из стандартных расширений *AFD* и регистров значений с плавающей точкой. В соответствии с соглашениями о вызовах помимо линейной нумерации регистров общего назначения применяются дополнительные имена для выделенных подмножеств. Например, группе регистров x10-x17, используемых для передачи аргументов функции, соответствуют имена a0-a7. Поэтому спецификация 64-битного регистра x10 (a0) выглядит следующим образом:

REG_SPEC(X10, X10, x10, a0, triton::bitsize::qword-1, 0, TT_MUTABLE_REG).

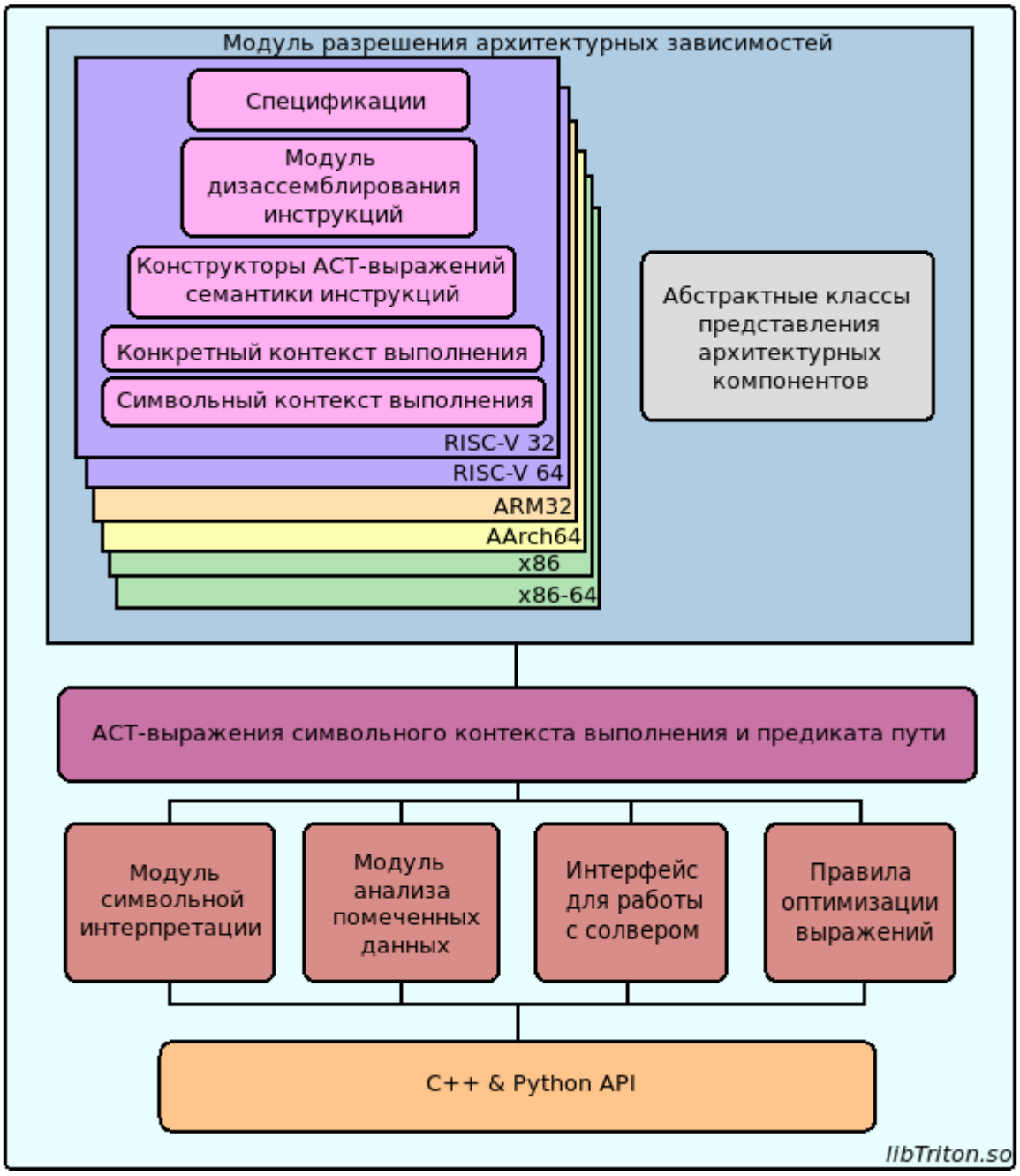


Рисунок 7. Символьная библиотека Triton.

Первые четыре поля содержат имена регистра для идентификаторов Capstone, Triton, название в нижнем регистре для создания переменных и название из соглашения о вызовах. Далее идут два значения, определяющие битовый размер регистра. Они задают номера первого и последнего битов, то есть 0 и 63, соответственно. Последним следует флаг, от которого зависит возможность изменения значения регистра. У архитектуры RISC-V есть один неизменяемый

регистр общего назначения x0, имеющий фиксированное нулевое значение. Помимо 32 целочисленных регистров общего назначения и аналогичного набора регистров с плавающей точкой в спецификацию входит регистр *pc* указателя текущей инструкции. Флаговые регистры в данной архитектуре не используются. 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 соответствующий абстрактный класс Triton *triton::arch::Instruction*. С помощью Capstone заполняются поля данного класса, включающие в себя название, адрес и операционный код инструкции, сведения о количестве, типах и значениях операндов, а также флаг принадлежности к группе инструкций, влияющих на поток управления.

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. Поэтому на рисунке 7 оба варианта архитектуры обозначены одинаковым цветом, в отличие от ARM32 и AArch64, семантика инструкций которых содержит существенные отличия и реализована отдельно друг от друга. Для RISC-V различие итоговых выражений заключается в размерах используемых битовых векторов в соответствии с размерами регистровых операндов инструкций.

Часть функциональных методов-конструкторов для 16 стандартных и 8 сокращенных инструкций при этом относится только к 64-битной версии архитектуры, в то время как отличительной для RISC-V 32 является единственная инструкция C.JAL из сокращенного набора.

Рассмотрим подробнее работу функциональных методов отдельных инструкций на примере конструирования символьного выражения для операции знакового деления. Соответствующий код на языке программирования C++ из библиотеки Triton приведен на Рисунке 8. При вызове метода-конструктора, 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. Для инструкции деления DIV в качестве делимого и делителя используются операнды *src1* и *src2*, символьные выражения которых извлекаются с помощью метода *getOperandAst*. Далее для полученных выражений в соответствии с операционной семантикой инструкции строится новое выражение, содержащее их в виде поддеревьев. На Рисунке 8 это строки с 11 по 16. В данном случае, семантика операции деления включает в себя логическую проверку делителя на равенство нулю с помощью логического битвекторного выражения if-then-else (обозначено как *ite* в строке 11), для чего в строке 13 создается константное нулевое значение битового вектора с помощью метода *bv*. При выполнении указанного условия в качестве содержащего значение выражения для целевого регистра *dst* будет выбран константный битовый вектор для значения -1 соответствующего размера, возвращаемого методом *dst.getBitSize* в строке 14. Для не равного нулю делителя символьное выражение выставит значение из поддерева, содержащего необходимую битвекторную операцию знакового деления *bvsdiv*.

```
1  void riscvSemantics::div_s(triton::arch::Instruction& inst) {
2      auto& dst  = inst.operands[0];
3      auto& src1 = inst.operands[1];
4      auto& src2 = inst.operands[2];
5
6      /* Create symbolic operands */
7      auto op1 = this->symbolicEngine->getOperandAst(inst, src1);
8      auto op2 = this->symbolicEngine->getOperandAst(inst, src2);
9
10     /* Create the semantics */
11     auto node = this->astCtxt->ite(
12         this->astCtxt->equal(op2,
13                              this->astCtxt->bv(0, op2->getBitvectorSize())),
14         this->astCtxt->bv(-1, dst.getBitSize()),
15         this->astCtxt->bvsdiv(op1, op2)
16     );
17
18     /* Create symbolic expression */
19     auto expr = this->symbolicEngine->createSymbolicExpression(
20         inst, node, dst, "DIV operation"
21     );
22
23     /* Spread taint */
24     expr->isTainted =
25         this->taintEngine->setTaint(dst,
26             this->taintEngine->isTainted(src1) |
27             this->taintEngine->isTainted(src2)
28         );
29
30     /* Update the symbolic control flow */
31     this->controlFlow_s(inst);
32 }
```

Рисунок 8. Функциональный метод конструирования символьного выражения семантики инструкции деления с учетом знаковости DIV.

Сконструированное выражение ассоциируются с целевым операндом, а также производится распространение пометки. Данные операции на Рисунке 8 выполняются в строках 19-20 и 24-28, соответственно. Итоговое выражение для инструкции при этом получает идентификатор и становится доступным для применения в рамках ссылочной (reference) организации хранения выражений

Triton. После этого увеличивается на размер инструкции символьное значение счётчика инструкций (строка 31), если иное не предусмотрено семантикой инструкции. 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) [78].

3.3. Псевдоинструкции и сокращенные инструкции

Псевдоинструкции являются частью спецификации ISA архитектуры RISC-V и представляют собой альтернативное прочтение стандартных инструкций при использовании предназначенных для этого специфических комбинаций операндов. Чаще всего в качестве такого операнда выступает регистр x0, всегда сохраняющий нулевое значение. Иногда также используется регистр x1 или константные значения. 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*. Фактически нулевые операнды по-прежнему остаются прописаны в операционном коде инструкции, однако в следствие особенностей инструмента дизассемблирования Capstone (который для RISC-V пока продолжает находиться в стадии рефакторинга разработчиками) при заполнении класса инструкции количество операндов для вариаций-псевдоинструкций сокращается. При этом идентификаторы для них также не предусмотрены, хотя, к примеру, аналогичный alias-механизм для архитектуры AArch64 их поддерживает. В результате, абстрактный класс, отображающий, например, 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. При этом в качестве промежуточного звена может быть использован вспомогательный метод для выбора метода-конструктора нужной символьной семантики. Например, на Рисунке 9 представлен вспомогательный функциональный метод *addi_s* для распознавания псевдоинструкций инструкции ADDI, реализующей сложение регистрового операнда с константным значением.

```
1  void riscvSemantics::addi_s(triton::arch::Instruction& inst) {
2      /*
3         mv rd, rs -- [addi rd, rs, 0] -- Copy register
4         nop       -- [addi x0, x0, 0] -- No operation
5       */
6      switch (inst.operands.size()) {
7        case 0: this->controlFlow_s(inst); return;   // nop
8        case 2: addi_mv_s(inst);  return;            // mv
9        default: add_s(inst);  return;               // addi
10     }
11 }
```

Рисунок 9. Промежуточный метод распознавания псевдоинструкций инструкции ADDI.

Данная инструкция имеет две вариации, получаемые посредством зануления значений операндов (строки 3 и 4). Соответственно, первая псевдоинструкция MV предназначена для копирования значения регистра, а псевдоинструкция NOP не вносит содержательных изменений и имеет служебное назначение. Поскольку для нулевых операндов псевдоинструкций соответствующие поля абстрактного класса *triton::arch::Instruction* не заполняются, в данном случае для распознавания вариаций достаточно узнать

количество операндов дизассемблированной инструкции. В зависимости от него (строка 6 на рисунке 9) для NOP достаточно обновить значение счетчика инструкций (строка 7), для MV получить готовое выражение регистрового значения и проассоциировать его с целевым регистром (что выполняется вызываемым в строке 8 методом *addi_mv_s*), а для стандартной инструкции ADDI можно воспользоваться конструктором инструкции сложения ADD, так как получение ACT-выражения для регистрового и константного видов операндов выполняется универсальным методом *getOperandAst*.

Сокращенные инструкции из C-расширения образуют набор базовых операций в виде передачи управления, обращений к памяти и простых арифметических операций. Они схожи с псевдоинструкциями наличием заранее предопределенных операндов. Однако основное их отличие заключается в том, что на их кодирование отводится всего 16 бит вместо 32, поскольку предопределенные семантикой операнды не прописаны в виде выделенных битов операционного кода, как в ситуации с представляющими для семантики лишь частные случаи псевдоинструкциями. Соответственно, размер сокращенных инструкций следует учитывать при обновлении символьного значения счетчика инструкций. Кроме того, по причине 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.

3.4. Апробация метода

Представленный метод символьной интерпретации целочисленных инструкций для архитектуры RISC-V был протестирован для обширного набора синтетических тестов, включающих символьную эмуляцию с последующей верификацией сгенерированных входных данных с помощью конкретного

запуска. В качестве примера на Рисунке 10 приведен набор синтетических тестов для интерпретации семейства инструкций деления с остатком REM/REMW/REMU/REMUW. Данные инструкции последовательно символьно интерпретируются. Для каждой из них производится сравнение полученного после интерпретации инструкции контекста регистров и памяти с аналогичными преобразованиями, осуществляемыми основанным на QEMU эмулятором CPU Unicorn [79]. С целью тестирования граничных случаев используются вспомогательные инструкции. Например, для загрузки нулевого значения в регистр *x17* применяется инструкция загрузки константного значения LUI *x17 #0*, после чего регистр *x17* выступает к качестве нулевого операнда-делителя для следующих 4 инструкций.

```
# REM/U/W
(b"\xb3\x65\xa6\x02", "rem   x11, x12, x10"),
(b"\xb3\x75\xa6\x02", "remu  x11, x12, x10"),
(b"\xbb\x75\xa7\x02", "remuw x11, x14, x10"),
(b"\xbb\x65\xa7\x02", "remw  x11, x14, x10"),
# div-by-zero corner case
(b"\xb7\x08\x00\x00", "lui   x17, #0"),
(b"\xb3\x65\x16\x03", "rem   x11, x12, x17"),
(b"\xbb\x75\x17\x03", "remuw x11, x14, x17"),
(b"\xb3\x75\x16\x03", "remu  x11, x12, x17"),
(b"\xbb\x65\x17\x03", "remw  x11, x14, x17"),
# signed overflow
(b"\xb7\xf8\xff\xff", "lui   x17, #0xfffff"),
(b"\x93\x98\x38\x01", "slli  x17, x17, #19"),
(b"\x13\x05\xf0\xff", "addi  x10, x0,  #-1"),
(b"\xbb\xe5\xa8\x02", "remw  x11, x17, x10"),
(b"\x93\x05\x16\x00", "addi  x11, x12, #1"),
(b"\x93\x98\x08\x02", "slli  x17, x17, #32"),
(b"\xb3\xe5\xa8\x02", "rem   x11, x17, x10"),
```

Рисунок 10. Синтетический тестовый набор Triton для инструкций операции деления с остатком архитектуры RISC-V.

Для демонстрации возможностей интерпретации бинарного кода архитектуры RISC-V с помощью Triton в рамках данной работы были добавлены тестовые примеры из категории задач «захвата флага» (capture the flag/CTF). Пример для архитектуры RISC-V 64 [80] включает файл *crackme_hash.c*, содержащий исходный код с символьным ветвлением, в котором печатается одна из строк «Win» либо «fail». Его бинарная версия находится в файле *crackme_hash.* Для запуска эмуляции можно использовать два Python-скрипта (*solve.py* и *solve-with-abv-logic.py*), в начале каждого из которых приведен ожидаемый результирующий вывод. Данные скрипты отличаются друг от друга набором используемых оптимизаций, из-за чего они генерируют различные итоговые последовательности входных данных («mnadg» и «qcgln»). Тем не менее, оба варианта при запуске самого бинарного файла с таким входом приводят к печати строки «Win».

Присутствие заранее собранного бинарного файла в тестовом примере обусловлено не столько удобством пользователя, сколько требованиями работоспособности предоставляемых скриптов. Как можно заметить на Рисунке 11, скрипт для эмуляции бинарного кода опирается на заранее известные адреса инструкций.

```
# 5de:   00000517              auipc   a0,0x0
# 5e2:   17a50513              addi    a0,a0,378 # 758 <_IO_stdin_used+0x8>
# 5e6:   fbbff0ef              jal     ra,5a0 <puts@plt>
#        ...
# 758:   006e6957              .4byte  0x6e6957 # "Win"

if pc == 0x5e6:
    # We validated the crackme
    VALID = True
```

Рисунок 11. Проверка достижения целевого ветвления в скрипте эмуляции.

При самостоятельной или автоматической сборке тестового кода с использованием другой версии или набора оптимизаций компилятора, велика

вероятность изменения адресов инструкций в исполняемом файле. Из чего следует, что для корректной работы эмуляции пользователю потребовалось бы заново проинициализировать скрипт с помощью правильных адресов. В свете чего наглядно видно преимущество применения конкретно-символьного подхода, при котором соответствующая инструментация выполняется в автоматическом режиме.

Соответственно, на основе представленного в данной главе метода был реализован метод динамической символьной интерпретации бинарного кода RISC-V 64, который обсуждается в главе 4 настоящей работы.

3.5. Выводы

В данной главе был представлен разработанный метод символьной интерпретации набора целочисленных инструкций архитектуры RISC-V, включающего псевдоинструкции и сокращенные инструкции. Метод способен конструировать символьные выражения для операционной семантики инструкций из расширений *RV64IMC* и *RV32IMC*. При апробации метода, реализованного в открытой символьной библиотеке Triton, использовались как синтетические тесты, так и тестовые примеры для проведения эмуляции, что демонстрирует применимость метода в динамической символьной интерпретации. На основе представленного в данной главе метода был реализован использующий конкретно-символьный подход метод динамической символьной интерпретации бинарного кода RISC-V 64, который обсуждается в главе 4 настоящей работы.

Глава 4. Метод динамической символьной интерпретации бинарного кода архитектуры RISC-V 64

4.1. Особенности моделирования символьного контекста для RISC-V 64

Разработанный метод динамической символьной интерпретации опирается на метод конструирования символьной семантики инструкций RISC-V 64, описанный в главе 3. 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.

4.2. Перехват вызовов функций DynamoRIO для RISC-V

При реализации поддержки анализа бинарных программ RISC-V 64 в Sydr возникла необходимость доработки библиотеки DynamoRIO [81], используемой в Sydr в качестве инструментатора для получения информации о ходе конкретного исполнения программы. DynamoRIO представляет собой кроссплатформенный фреймворк для динамической бинарной инструментации (Dynamic Binary Instrumentation, DBI). Его основная задача – предоставить разработчикам возможность анализировать и модифицировать поведение произвольных приложений на уровне машинного кода во время их выполнения, без необходимости доступа к исходному коду или перекомпиляции. DynamoRIO работает на различных платформах и поддерживает анализ программ, в том числе, для процессорных архитектур x86/x86_64, ARM (AArch64) и RISC-V. Однако функционал для поддержки RISC-V в DynamoRIO продолжает находиться в стадии разработки, работая в экспериментальном режиме, с множеством не реализованных функций. Чтобы обеспечить работоспособность метода динамической символьной интерпретации программ архитектуры RISC-V 64, реализуемой инструментом Sydr, была произведена доработка модуля перехвата вызовов библиотеки DynamoRIO.

Принцип работы DynamoRIO можно описать как интерактивный отладчик, который перехватывает выполнение приложения и незаметно для него вставляет собственный код инструментации в поток исполняемых процессором инструкций. Основные модули данного фреймворка и решаемые ими задачи можно представить следующим образом:

- Диспетчер выполнения DynamoRIO обеспечивает перехват контроля над исполняемым процессом. При запуске приложения, инструментируемого DynamoRIO, диспетчер загружается и настраивает окружение.

- Диспетчер базовых блоков анализирует машинный код приложения и разбивает его на базовые блоки, которые затем декодируются в внутреннее представление на основе собственной структуры данных DynamoRIO и подготавливаются для дальнейшей инструментации.
- За инструментацию и анализ программы отвечает клиентский модуль, который реализуется пользователем с использованием API DynamoRIO. При написании данного модуля пользователь настраивает необходимые ему преобразования и инструментацию кода посредством реализации набора функций с обратным вызовом (callback). Впоследствии эти функции вызываются фреймворком при наступлении представляющих интерес для анализа стадий выполнения исследуемой программы: загрузка разделяемых библиотек, системные вызовы, вызов функций, исполнение базового блока, исполнение отдельной инструкции.
- Диспетчер кэша кода отвечает за сохранение ранее проинструментированных базовых блоков программы, чтобы избежать повторного применения преобразований к одному и тому же коду.

Одним из основных механизмов, используемых при анализе, является перехват вызовов функций и возвратов из них. В библиотеке DynamoRIO данная функциональность реализуется с помощью модуля *drwrap*, который позволяет проинструментировать вызов 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

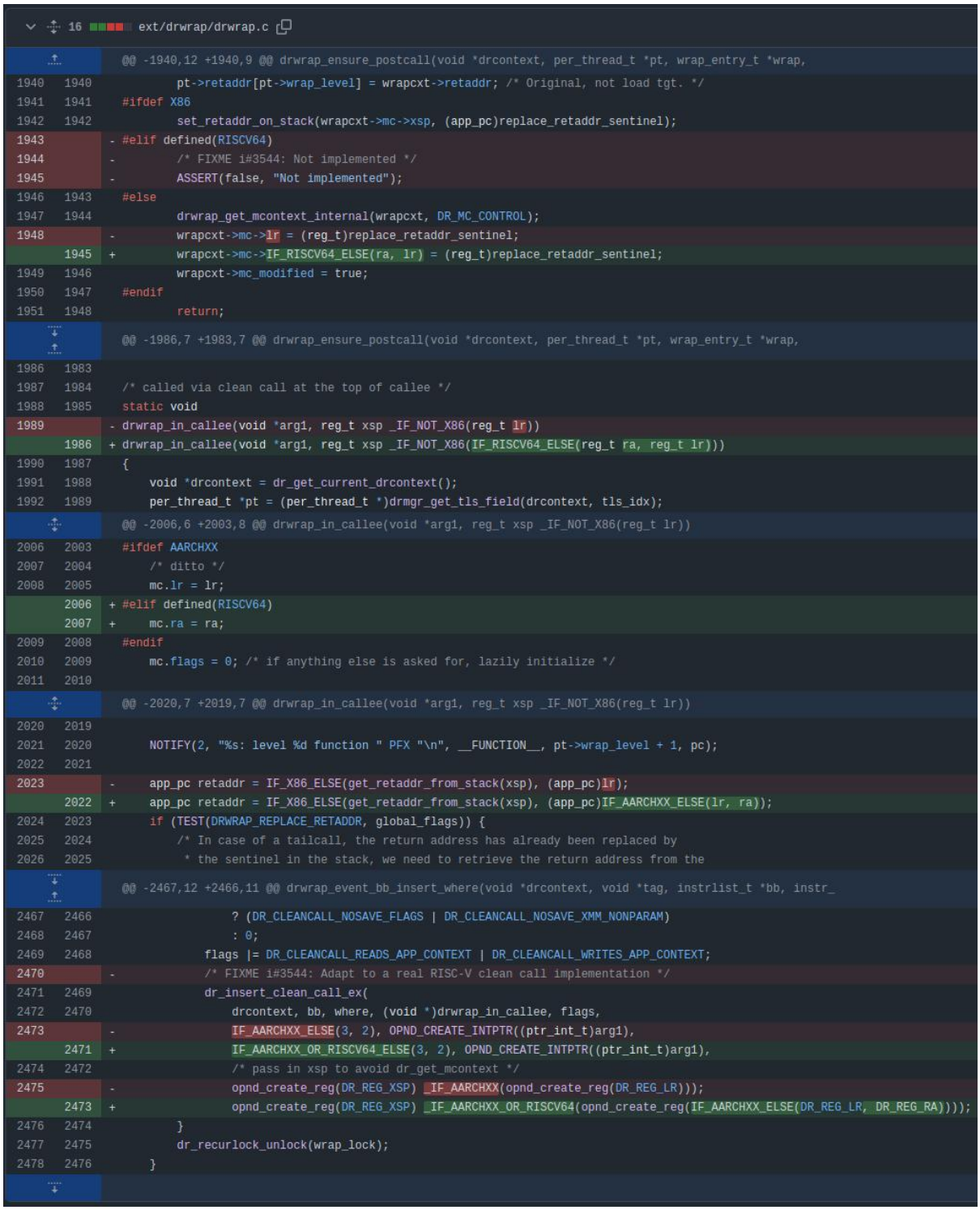
пользовательскими данными между *pre* и *post* обертками.

Механизм перехвата вызовов функций широко используется в инструменте динамической символьной интерпретации Sydr. Помимо работы модуля моделирования символьной семантики для ряда функций стандартной библиотеки (например, strcpy, memchr, tolower, strtoull и др.), при помощи механизма перехвата также поддерживается отслеживание выделения и освобождения памяти для теневой кучи (shadow heap) функциями malloc, realloc, calloc, free. Перехват и анализ функций форматной печати (printf и др.) и вызова команд (execv и др.), необходим для работы соответствующих предикатов безопасности. Кроме того, для части функций вывода производится конкретизация с целью преодоления избыточной помеченности. Функционирование данного модуля DynamoRIO критически важно для символьной интерпретации в Sydr, поскольку без возможности анализа на уровне отдельных функций значимая часть функционала становится недоступна.

Поддержка перехвата вызовов функций для платформы RISC-V в DynamoRIO была реализована лишь частично. На момент разработки метода интерпретации в Sydr поддерживалась вставка и вызов только pre-оберток. При попытке вызова post-обертки в библиотеке DynamoRIO возникала ошибка сегментации, что приводило к аварийному завершению тестируемой программы и, как следствие, всего процесса символьной интерпретации. Принципиальная проблема заключается в способе, который DynamoRIO использует для вызова post-обертки. При обычном исполнении какой-либо функции в программе, перед ее вызовом в стек помещается адрес возврата (следующая инструкция после call). По завершению выполнения функции, на адрес возврата из стека передается управление. Чтобы внедрить вызов функции-обработчика в эту систему, DynamoRIO использует метод «украденного возврата», который состоит в том чтобы перед вызовом целевой функции сохранить реальный адрес возврата в безопасном месте, и подменить

его на адрес специального кода-трамплина. Код-трамплин сохраняет все регистры и флаги программы, загружает контекст для выполнения функции post, вызывает ее. После выполнения post-обертки код-трамплин очищаетс стек, восстанавливает флаги и регистры и выполняет переход на реальный сохраненный адрес возврата.

Код, для реализации данной функциональности находится в файле DynamoRIO *ext/drwrap/drwrap.c* (и соответствующем заголовочном файле *drwrap.h*). Адрес возврата хранится в определенном для каждой из архитектур регистре. Состояние программы (регистры и флаги) описывается в виде специальной структуры машинного контекста DynamoRIO *dr_mcontext_t*, через которую и осуществляется доступ к различным регистрам. Механизм подмены адреса возврата был реализован только для архитектур семейства x86 и ARM. Поэтому в рамках данной работы в DynamoRIO была поддержана работа с регистрами, соответствующими машинному контексту RISC-V 64, в частности регистром *ra*, который хранит адрес возврата. Для этого были изменены методы *drwrap_ensure_postcall()*, *drwrap_in_callee()* и *drwrap_event_bb_insert_where()*. Детализация изменений приведена в листинге кода на Рисунке 12. Дополнительно также потребовалось реализовать инициализацию соглашения о вызовах для платформы RISC-V в DynamoRIO (*DRWRAP_CALLCONV_RISCV_LP64*). Данная реализация позволила поддержать полноценный перехват вызовов функций в DynamoRIO для RISC-V, что обеспечило работоспособность поддерживаемых инструментом Sydr современных техник анализа в виде моделирования символьной семантики функций и предикатов безопасности для бинарных программ данной архитектуры.



```
16 ext/drwrap/drwrap.c

@@ -1940,12 +1940,9 @@ drwrap_ensure_postcall(void *drcontext, per_thread_t *pt, wrap_entry_t *wrap,
1940 1940          pt->retaddr[pt->wrap_level] = wrapcxt->retaddr; /* Original, not load tgt. */
1941 1941  #ifdef X86
1942 1942          set_retaddr_on_stack(wrapcxt->mc->xsp, (app_pc)replace_retaddr_sentinel);
1943      - #elif defined(RISCV64)
1944      -         /* FIXME i#3544: Not implemented */
1945      -         ASSERT(false, "Not implemented");
1946 1943  #else
1947 1944          drwrap_get_mcontext_internal(wrapcxt, DR_MC_CONTROL);
1948      -         wrapcxt->mc->lr = (reg_t)replace_retaddr_sentinel;
     1945 +         wrapcxt->mc->IF_RISCV64_ELSE(ra, lr) = (reg_t)replace_retaddr_sentinel;
1949 1946          wrapcxt->mc_modified = true;
1950 1947  #endif
1951 1948          return;

@@ -1986,7 +1983,7 @@ drwrap_ensure_postcall(void *drcontext, per_thread_t *pt, wrap_entry_t *wrap,
1986 1983
1987 1984  /* called via clean call at the top of callee */
1988 1985  static void
1989      - drwrap_in_callee(void *arg1, reg_t xsp _IF_NOT_X86(reg_t lr))
     1986 + drwrap_in_callee(void *arg1, reg_t xsp _IF_NOT_X86(IF_RISCV64_ELSE(reg_t ra, reg_t lr)))
1990 1987  {
1991 1988      void *drcontext = dr_get_current_drcontext();
1992 1989      per_thread_t *pt = (per_thread_t *)drmgr_get_tls_field(drcontext, tls_idx);

@@ -2006,6 +2003,8 @@ drwrap_in_callee(void *arg1, reg_t xsp _IF_NOT_X86(reg_t lr))
2006 2003  #ifdef AARCHXX
2007 2004      /* ditto */
2008 2005      mc.lr = lr;
     2006 + #elif defined(RISCV64)
     2007 +     mc.ra = ra;
2009 2008  #endif
2010 2009      mc.flags = 0; /* if anything else is asked for, lazily initialize */
2011 2010

@@ -2020,7 +2019,7 @@ drwrap_in_callee(void *arg1, reg_t xsp _IF_NOT_X86(reg_t lr))
2020 2019
2021 2020      NOTIFY(2, "%s: level %d function " PFX "\n", __FUNCTION__, pt->wrap_level + 1, pc);
2022 2021
2023      -     app_pc retaddr = IF_X86_ELSE(get_retaddr_from_stack(xsp), (app_pc)lr);
     2022 +     app_pc retaddr = IF_X86_ELSE(get_retaddr_from_stack(xsp), (app_pc)IF_AARCHXX_ELSE(lr, ra));
2024 2023      if (TEST(DRWRAP_REPLACE_RETADDR, global_flags)) {
2025 2024          /* In case of a tailcall, the return address has already been replaced by
2026 2025           * the sentinel in the stack, we need to retrieve the return address from the

@@ -2467,12 +2466,11 @@ drwrap_event_bb_insert_where(void *drcontext, void *tag, instrlist_t *bb, instr_
2467 2466                  ? (DR_CLEANCALL_NOSAVE_FLAGS | DR_CLEANCALL_NOSAVE_XMM_NONPARAM)
2468 2467                  : 0;
2469 2468              flags |= DR_CLEANCALL_READS_APP_CONTEXT | DR_CLEANCALL_WRITES_APP_CONTEXT;
2470      -             /* FIXME i#3544: Adapt to a real RISC-V clean call implementation */
2471 2469              dr_insert_clean_call_ex(
2472 2470                  drcontext, bb, where, (void *)drwrap_in_callee, flags,
2473      -                 IF_AARCHXX_ELSE(3, 2), OPND_CREATE_INTPTR((ptr_int_t)arg1),
     2471 +                 IF_AARCHXX_OR_RISCV64_ELSE(3, 2), OPND_CREATE_INTPTR((ptr_int_t)arg1),
2474 2472                  /* pass in xsp to avoid dr_get_mcontext */
2475      -                 opnd_create_reg(DR_REG_XSP) _IF_AARCHXX(opnd_create_reg(DR_REG_LR)));
     2473 +                 opnd_create_reg(DR_REG_XSP) _IF_AARCHXX_OR_RISCV64(opnd_create_reg(IF_AARCHXX_ELSE(DR_REG_LR, DR_REG_RA))));
2476 2474          }
2477 2475          dr_recurlock_unlock(wrap_lock);
2478 2476      }
```

Рисунок 12. Поддержка архитектуры RISC-V в модуле перехвата вызовов фреймворка DynamoRIO.

4.3. Анализ косвенных табличных переходов для RISC-V 64

В результате присутствия для архитектуры RISC-V технических ограничений в работе используемого инструментатора DynamoRIO наиболее значимое отличие наблюдается для косвенных ветвлений. Помимо вышеописанного механизма перехвата вызовов функций, к возможностям данного инструмента, используемым в Sydr, 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, 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.

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 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.

Подробное объяснение механизма работы таблиц переходов с чтением из памяти как целевых адресов в исходном коде, так и смещений относительно базового адреса приведено в посвященном архитектуре ARM/AArch64 разделе 2.2 настоящей работы.

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 Рисунках 13а-в 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, 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*.

```
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
```

Рисунок 13а. 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
```

Рисунок 13б. 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
```

Рисунок 13в. Базовый блок с разбиением, отделяющим блок с паттерном и нецелевой load-инструкцией от блока с целевой load-инструкцией и инструкцией передачи управления.

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
```

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 на рис. 13а и 13в. 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.

4.4. Выводы

В данной главе был представлен разработанный метод динамической символьной интерпретации бинарного кода программ архитектуры RISC-V 64. Метод способен распознавать и анализировать косвенные ветвления даже в условиях ограниченной информации о списке инструкций базового блока.

Глава 5. Экспериментальная оценка разработанных методов динамической символьной интерпретации

5.1. Экспериментальная оценка для архитектуры Байкал-М (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. Среди открытых современных аналогов, поддерживающих архитектуру ARM/AArch64, наиболее близким к нему по принципу работы и набору поддерживаемых символьных техник анализа, можно назвать инструмент SymQEMU [40], который так же как и Sydr реализует конкретно-символьное исполнение бинарной программы. С целью оценки разработанного метода было проведено экспериментальное сравнение двух данных динамических анализаторов. Сравнение проводилось с помощью набора открытых приложений, собранных на процессоре Байкал-М. Эксперимент проводился на ЭВМ с 8 ядрами Cortex-A57 ARM и 32Гб RAM, под управлением ОС ALT Workstation 10.1. Каждое приложение запускалось на анализ с одним набором конкретных входных данных, который представлял собой корректный с точки зрения программы вход и соответствовал типичному сценарию работы. Задачей инструмента было обнаружение альтернативных направлений ветвления для заданного пути выполнения с генерацией соответствующих входных данных. Для набора всех сгенерированных входных данных было измерено покрытие строк кода, чтобы оценить качество анализа сравниваемых инструментов. Результаты сравнения представлены в Таблице 1.

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, для 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. В качестве ограничения времени для каждого приложения был использован лимит в 20 минут.

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.

Как видно из результатов эксперимента, практически для всех приложений предлагаемый метод динамической символьной интерпретации продемонстрировал лучшие результаты с точки зрения достигнутого покрытия кода. На проектах *jsoncpp* и *woff2* доля относительно общего обнаруженного покрытия Sydr достигает 100%. Это означает, что по сравнению с SymQEMU инструмент позволил увеличить покрытие программы без потери каких-либо путей исполнения. Из восьми проанализированных приложений полностью в отведенный лимит времени уложились только четыре проекта, на трех из них

Sydr показал лучшую производительность. Например, анализ приложения *lcms* с помощью Sydr занял всего 9.7 секунды, в то время как для SymQEMU он занимает больше 16 минут. И хотя за это время SymQEMU сгенерировал в 4 раза больше новых входных данных, это не привело к увеличению процента покрытия кода, значение которого для Sydr больше. Обратная ситуация наблюдается для проекта *jsoncpp*, где Sydr для анализа потребовалось больше времени (11.4 секунд против 4.2 для SymQEMU) на генерацию входных файлов (907 против 193), что позволило получить новое уникальное покрытие. Единственным приложением, для которого Sydr показал результат хуже, является *libxml2*, что связано с особенностью данной программы в виде обработки большого массива входных данных в цикле. Соответственно, Sydr тратит много времени на анализ каждой итерации и генерацию новых входных данных, в то время как в SymQEMU присутствует оптимизация специально для таких случаев. Можно заключить, что эффективность работы Sydr при генерации новых входных данных и инвертировании условных переходов в среднем выше, за счет чего данный инструмент способен достигать большего покрытия даже при меньшем числе сгенерированных тестов.

Разработанный метод динамической символьной интерпретации бинарных программ для Байкал-М (AArch64) был протестирован при применении в контексте гибридного фаззинга. Эксперименты по гибридному фаззингу выполнялись на ЭВМ с 96 ядрами HiSilicon Kunpeng-920 и 128Гб RAM под управлением ОС CentOS Stream 8. При проведении эксперимента использовался гибридный фаззер Sydr-Fuzz, сконфигурированный на запуск и синхронизацию работы динамического символьного интерпретатора Sydr и инструмента фаззинга с обратной связью по покрытию. В качестве инструмента фаззинга использовались libFuzzer и AFL++. Работа гибридного фаззинга сравнивалась с обычным запущенным в виде двух параллельных процессов фаззинг-тестированием. При осуществлении экспериментального сравнения для гибридного фаззинга применялся разработанный Google фреймворк для

оценки фаззинг-инструментов FuzzBench [82]. Помимо фиксированного набора приложений для исследования (бенчмарков), данный фреймворк предоставляет изолированную среду исполнения на основе Docker-контейнеров в качестве стабильного окружения для запуска инструментов. Во время тестирования производится регулярное обновление текущей информации о достигнутом покрытии для автоматической визуализации в виде графиков, отражающих изменение данной метрики в ходе эксперимента. По умолчанию метрика включает покрытие для базовых блоков, но также доступны измерения для покрытия строк кода. Желаемые параметры фаззинг-сессии указываются с помощью конфигурационного файла. Ограничение времени, отведенного на фаззинг-тестирование в данном эксперименте, составляет 23 часа. В ходе эксперимента запускалось по 5 экземпляров фаззинга (trials) для усреднения результатов.

На Рисунке 14 приведены результаты сравнения гибридного фаззинга Sydr+libFuzzer и обычного фаззинга с двумя процесссами libFuzzer. Показатель «*avg. score*» обозначает средний процент достигнутого покрытия на всех проектах-бенчмарках (чем ближе к 100%, тем результаты лучше). Показатель «avg. rank» обозначает средний рейтинг фаззера на всех бенчмарках (чем ближе к 1, тем лучше).

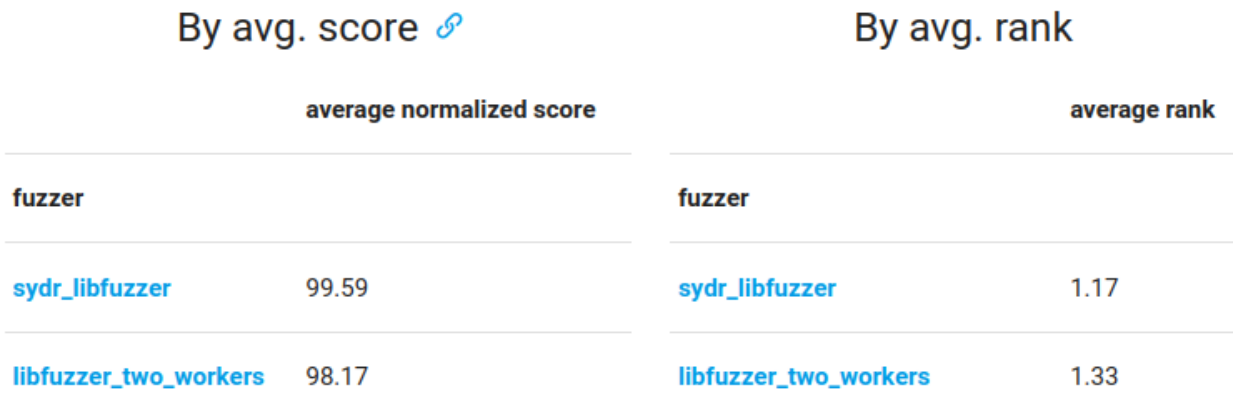


Рисунок 14. Результаты тестирования на FuzzBench Sydr+libFuzzer и 2xlibFuzzer.

Из результатов видно, что применение гибридного фаззинга с Sydr при анализе тестовых приложений, собранных для AArch64, показывает лучшие результаты чем применение классического фаззинга. Это свидетельствует о работоспособности предложенного подхода и его практической значимости. Пример результата FuzzBench для бенчмарка *freetype2* приведен на Рисунке 15.

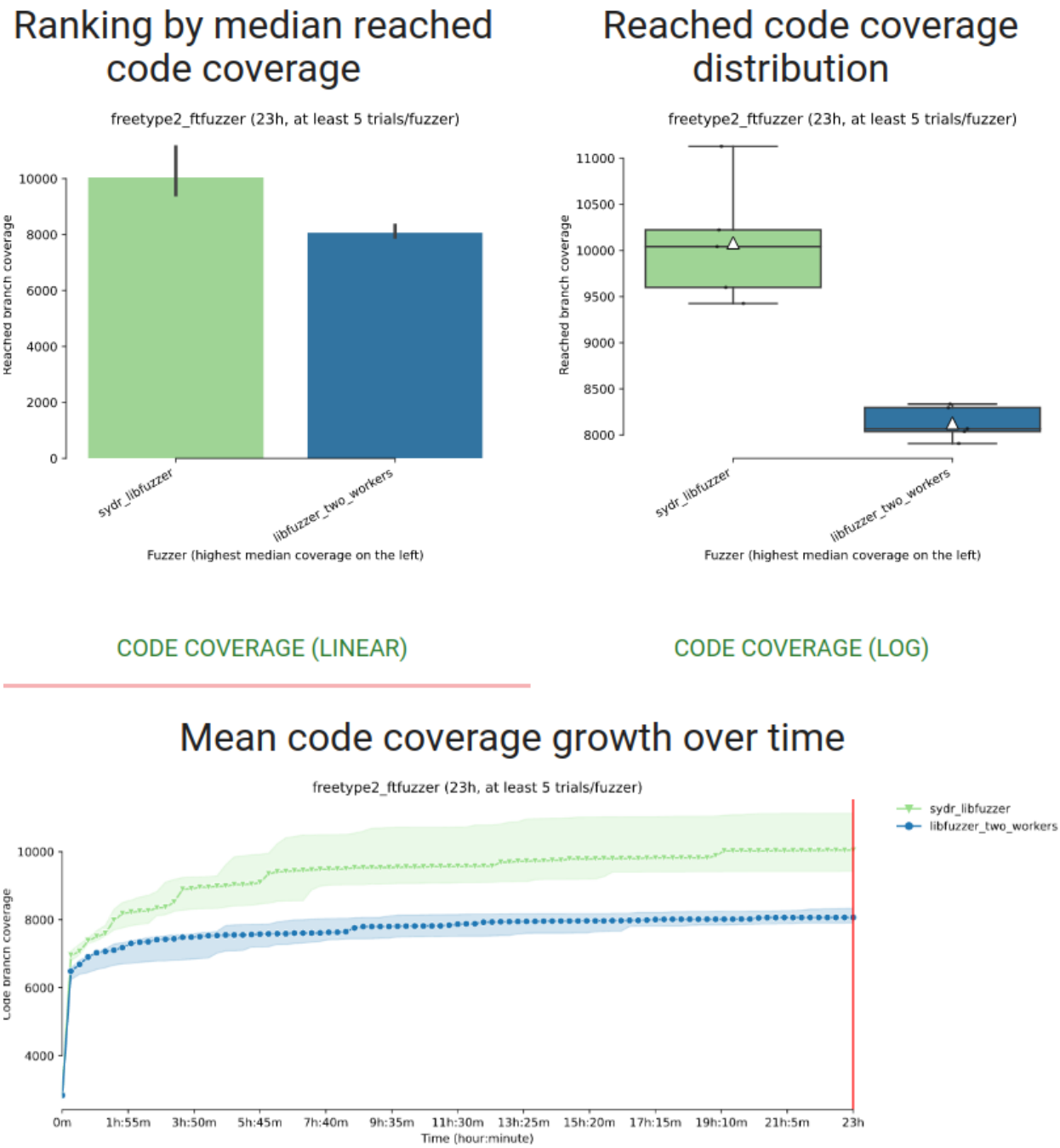


Рисунок 15. Результаты FuzzBench для бенчмарка *freetype2* при сравнении Sydr+libFuzzer и 2xlibFuzzer.

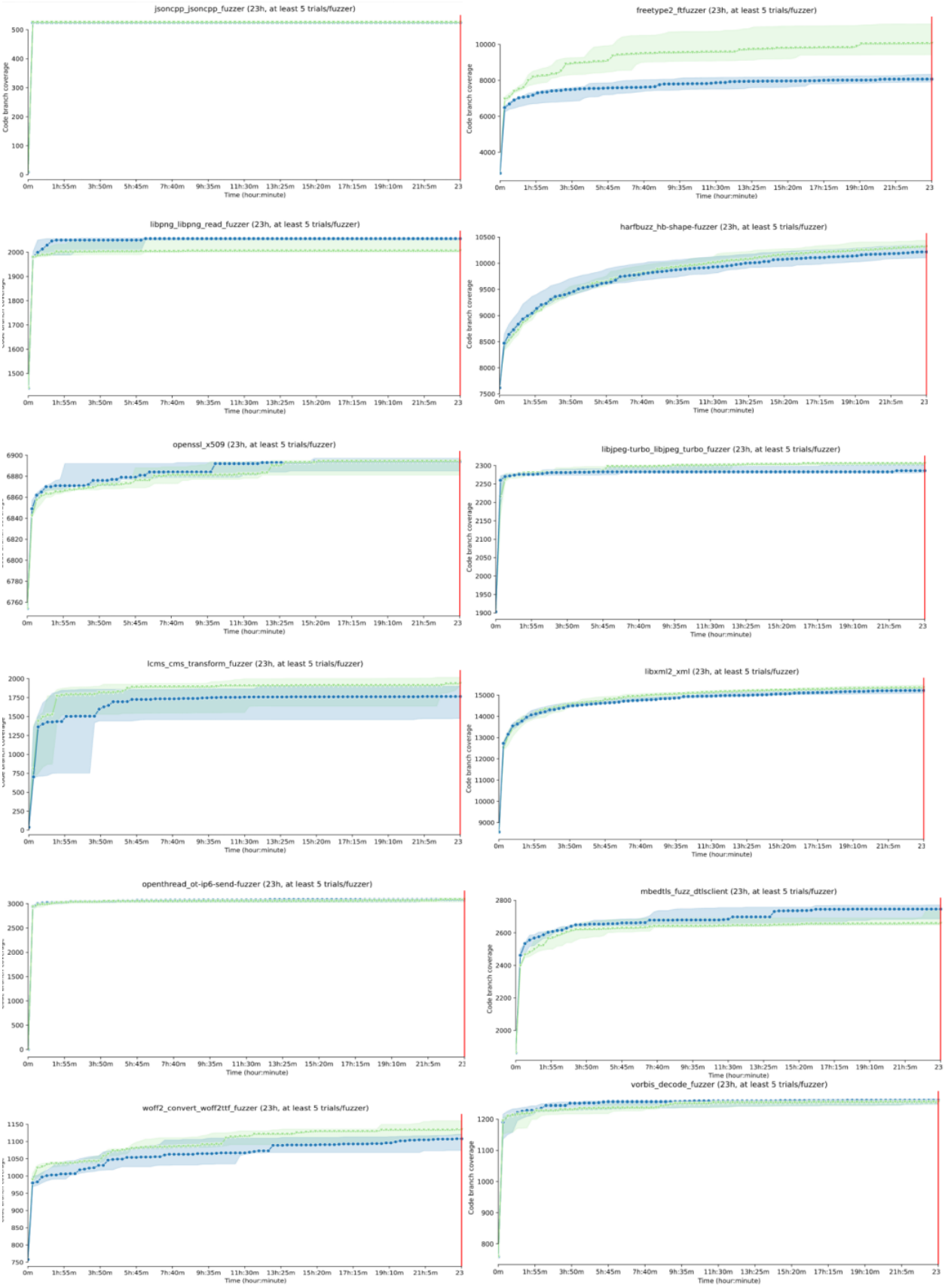


Рисунок 16. Графики роста покрытия FuzzBench для эксперимента Sydr+libFuzzer и 2xlibFuzzer.

Результаты FuzzBench для каждого бенчмарк-проекта включают график роста покрытия во время анализа и две диаграммы с отображением среднего достигнутого покрытия и распределения достигнутого покрытия для всех запущенных экземпляров (trials). Из результатов на рисунке 13 видно, что гибридный фаззинг с Sydr (отмечен зеленым) достигает большего покрытия с самого начала эксперимента. Причем это наблюдается для всех 5 запущенных экземпляров (trials) без исключений, поскольку диаграммы распределения покрытия не пересекаются.

На рисунке 16 представлены графики роста покрытия для всех 12 запущенных бенчмарк-проектов FuzzBench. График для Sydr+libFuzzer обозначен зеленым цветом. На 10 проектах из 12 гибридный фаззинг с Sydr смог достичь большего покрытия по итогам 23 часового эксперимента. Для оставшихся двух примеров незначительное преимущество в результатах для среднего покрытия продемонстрировал обычный фаззинг (2xlibFuzzer).

На рисунке 17 приведены результаты сравнения гибридного фаззинга Sydr+AFL++ и обычного фаззинга с двумя параллельно запущенными процессами AFL++. Параметры данного эксперимента аналогичны предыдущему. Из полученных результатов также видно, что гибридный фаззинг Sydr и AFL++ сумел достичь гораздо большего среднего покрытия на большинстве бенчмарков. Графики роста покрытия для эксперимента с AFL++ представлены на Рисунке 18.

By avg. score

fuzzer	average normalized score
sydr_aflplusplus	99.97
aflplusplus_two_instances	91.16

By avg. rank

fuzzer	average rank
sydr_aflplusplus	1.17
aflplusplus_two_instances	1.83

Рисунок 17. Результаты тестирования на FuzzBench Sydr+AFL++ и 2xAFL++.

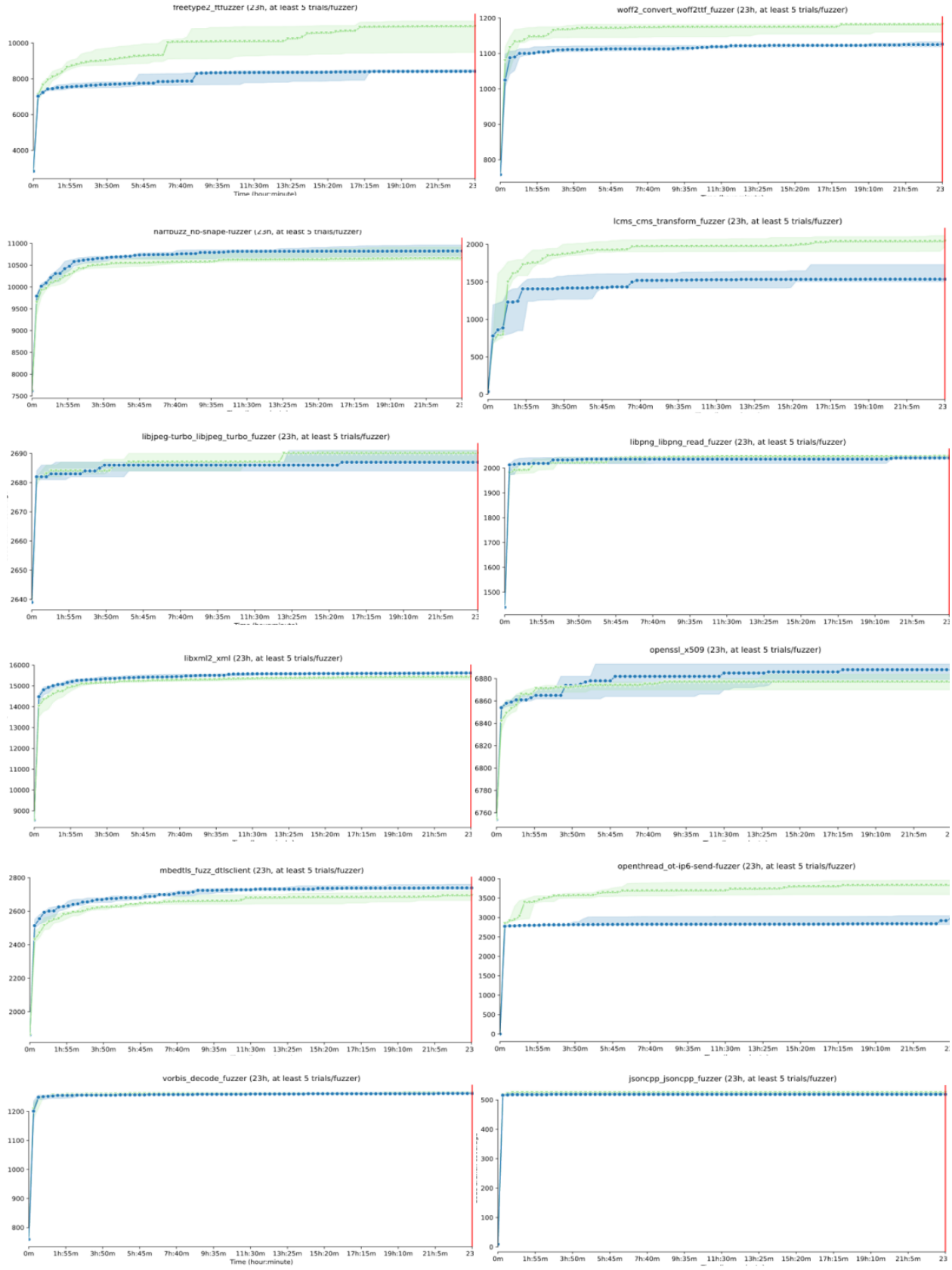


Рисунок 18. Графики роста покрытия FuzzBench для эксперимента Sydr+AFL++ и 2xAFL++.

График для Sydr+AFL++ на рисунке 18 обозначен зеленым цветом. Из всех 12 запущенных бенчмарков на 8 гибридный фаззинг с Sydr смог достичь большего покрытия по итогам 23 часового эксперимента. Результаты несколько отличаются от экспериментов для libFuzzer. Расхождение результирующих метрик покрытия между гибридным и классическим фаззингом оказывается больше, причем как в случае преимущества, достигнутого за счет символьной интерпретации Sydr, так и при лидерстве двухпоточного AFL++. Это обуславливается характеристиками самого фаззера AFL++ и механизмом для его синхронизация с символьной интерпретацией. В совокупности, данные эксперименты показали, что предложенные подходы динамической символьной интерпретации работоспособны, приводят к увеличению покрытия и обладают практической применимостью для анализа бинарных программ на платформах Байкал-М (AArch64).

5.2. Экспериментальная оценка для архитектуры RISC-V 64

Для оценки разработанного метода динамической символьной интерпретации для бинарных программ RISC-V 64, 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, в который базовая поддержка архитектуры RISC-V 64 была добавлена при участии автора настоящей работы. Способ проведения и оценки эксперимента аналогичен сравнению с SymQEMU, описанному в разделе 5.1 данной работы. Для каждого из тестового набора реальных приложений, скомпилированых под архитектуру RISC-V 64, оба инструмента были запущены на анализ вдоль одного конкретного пути. Ограничение времени для данного эксперимента было увеличено до 60 минут, так как эксперимент

проводился на виртуальной машине QEMU с эмуляцией CPU RISC-V VirtIO board (спецификации rv64imafdcsu) с 8 ядрами и 32Гб RAM под управлением ОС Ubuntu20.04.

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.

Результаты сравнения представлены в Таблице 2. 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 разделе 5.1. Из Таблицы 2 можно видеть, что для двух третей тестируемых приложений Sydr демонстрирует лучшие показатели достигнутого покрытия, а в остальных случаях получает равные результаты для данной метрики. Идентичные показатели покрытия представлены бенчмарк-проектами *ffmpeg*, *lcms* и *re2*. За исключением *re2* длительность анализа этих проектов посредством Sydr существенно меньше, как и количество сгенерированных им входных файлов. В случае *re2* анализ программы с помощью Sydr занял полторы минуты, в то время как SymQEMU справился за 5 секунд. Данный результат обуславливается особенностью обработки больших циклов с символьными данными в инструменте SymQEMU, которая позволяет ему пропускать уже исследованные итерации. Также стоит отметить, что на части программ Sydr показал заметно лучшую производительность анализа. Это особенно проявляется на *lcms* (1

минута 20 секунд для Sydr против 22 минут для SymQEMU) и *openjpeg* (1 минута 28 секунд против 49 минут, соответственно).

Для архитектуры RISC-V 64 сравнительный запуск гибридного фаззинга посредством инфраструктуры FuzzBench не проводился по причине ограниченных вычислительных ресурсов. Для данной архитектуры было проведено сравнение в режиме ручного тестирования между гибридным фаззингом с помощью фреймворка Sydr-Fuzz, сконфигурированного на запуск Sydr и libFuzzer, и 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.

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

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

Данные результаты свидетельствуют о том, что разработанные методы символьной интерпретации набора инструкций RISC-V и динамической символьной интерпретации RISC-V 64 позволяют достигать большего покрытия, чем имеющиеся аналоги, и имеют хорошую производительность. Результаты гибридного фаззинга говорят о практической применимости предложенных методов для анализа реальных программ. Также стоит упомянуть, что инструмент Sydr с реализованными методами был успешно

запущен и использован для анализа на одноплатном миникомпьютере от SiFive HiFive Unleashed [83], имеющим RISC-V 64 архитектуру.

Помимо этого в рамках работы Центра доверенного искусственного интеллекта ИСП РАН с помощью разработанных методов было проведено тестирование двух используемых для обработки изображений фреймворков машинного обучения ncnn [84] и librealsense [85], скомпилированных под платформу RISC-V. Проект ncnn – это открытый высокопроизводительный фреймворк от компании Tencent для инференса нейронных сетей, оптимизированный для мобильных платформ. Проект librealsense - это кроссплатформенная библиотека для работы с камерами глубины Intel RealSense. По итогам тестирования, в ncnn было обнаружено 6 новых программных ошибок [86-91], включающих аварийные завершения с ошибкой сегментации, утечку памяти, разыменование нулевого указателя, а также преобразования целочисленного значения между знаковыми и беззнаковыми типами, приводящие к ошибкам выделения памяти.

5.3. Выводы

В данной главе была представлена оценка эффективности реализацции разработанных методов динамической символьной интерпретации бинарного кода программ архитектуры Байкал-М (AArch64) и RISC-V 64 в инструменте Sydr, входящем в фреймворк гибридного фаззинга Sydr-Fuzz. На основе полученных результатов, можно заключить, 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.

Заключение

В процессе выполнения представленной научно-квалификационной работы были разработаны методы динамической символьной интерпретации бинарного кода Байкал-М (AArch64) и RISC-V 64, обеспечивающие возможность анализа косвенной адресации и применимые в контексте гибридного фаззинга. Данные методы были реализованы в инструменте Sydr, входящем в фреймворк динамического анализа Sydr-Fuzz. Методы позволяют обнаруживать косвенные табличные переходы и определять соответствующие целевые адреса исполняемого кода для указанных архитектур.

Помимо этого, был разработан метод символьной интерпретации набора целочисленных инструкций архитектуры RISC-V, включающего сокращенные инструкции и псевдоинструкции. Метод основан на конструировании выражений для операционной семантики машинных инструкций в виде абстрактных синтаксических деревьев. Набор поддерживаемых методом инструкций включает инструкции для модульных расширений IMC архитектуры RISC-V как для 64-битных, так и для 32-битных программ. Представленный метод был реализован в открытом фреймворке символьной интерпретации Triton и в дальнейшем может быть использован сообществом разработчиков для исследования программ и создания собственных инструментов динамического анализа.

Предложенные методы были апробированы на наборах синтетических и реальных тестовых приложений и продемонстрировали свою работоспособность. Для оценки реализованных в инструменте Sydr методов динамической символьной интерпретации было проведено экспериментальное сравнение с открытым аналогом SymQEMU на наборах бинарных программ архитектур Байкал-М (AArch64) и RISC-V 64. Сравнение проводилось на основе метрики достигнутого покрытия, для которой Sydr показал лучшие результаты. Кроме того, было проведено сравнение с инструментами фаззинг-

тестирования для оценки эффективности применения представленных методов в контексте гибридного фаззинга. Для обеих архитектур гибридный фаззинг также продемонстрировал лучшие показатели метрики достигнутого покрытия.

Таким образом, можно заключить, что были выполнены все задачи, поставленные в рамках представленной работы, а именно:

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 литературы

1. *Westland, J. C.* The cost of errors in software development: evidence from industry [Текст] / J. Christopher Westland // Journal of Systems and Software. — 2002. — Т. 62, № 1. — 2002. — С. 1–9.

2. *Howard, M.* The trustworthy computing security development lifecycle [Текст] / M. Howard, S. Lipner. // 20th Annual Computer Security Applications Conference, Tucson, AZ, USA. — 2004. — С. 2—13.

3. ГОСТ Р 56939-2016: Защита информации. Разработка безопасного программного обеспечения. Общие требования. — Национальный стандарт РФ, 2016.

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

5. *Miller, B.* An Empirical Study of the Reliability of UNIX Utilities [Текст] / Barton Miller, Lars Fredriksen, Bryan So // Communications of the ACM. Т. 33. — 1990. — С. 32–44.

6. *Serebryany, K.* Continuous Fuzzing with libFuzzer and AddressSanitizer [Текст] / Kosta Serebryany // 2016 IEEE Cybersecurity Development (SecDev) / IEEE. — 2016. — С. 157.

7. *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.

8. honggfuzz. Security oriented software fuzzer. [Electronic Resource]. — URL: https://github.com/google/honggfuzz (visited 27.05.2025).

9. *Molnar, D.* Automated whitebox fuzz testing [Текст] / D. Molnar, P. Godefroid, M. Levin // Network and Distributed System Security Symposium, NDSS. — 2008. — С. 416—426.

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

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

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. *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.

14. 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.

15. 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.

16. *King, J. C.* Symbolic execution and program testing [Текст] / J. C. King // Communications of the ACM. — 1976. — Т. 19, № 7. — С. 385—394.

17. *De Moura, L.* Satisfiability modulo theories: introduction and applications [Текст] / L. De Moura, N. Bjørner // Communications of the ACM. — 2011. —Т. 54, № 9. — С. 69—77.

18. SMT Solvers in Application to Static and Dynamic Symbolic Execution: A Case Study [Текст] / N. Malyshev [и др.] // 2019 Ivannikov Ispras Open Conference (ISPRAS). — IEEE. 2019. — С. 9—15.

19. Xu, Lin, and Berthe Y. Choueiry. "A new efficient algorithm for solving the simple temporal problem." (2003).

20. De Moura, Leonardo, and Nikolaj Bjørner. "Z3: An efficient SMT solver." International conference on Tools and Algorithms for the Construction and Analysis of Systems. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

21. Niemetz, Aina, and Mathias Preiner. "Bitwuzla." International Conference on Computer Aided Verification. Cham: Springer Nature Switzerland, 2023.

22. Liang, Tianyi, et al. "An efficient SMT solver for string constraints." Formal Methods in System Design 48.3 (2016): 206-234.

23. Chen, Ting, et al. "State of the art: Dynamic symbolic execution for automated test generation." Future Generation Computer Systems 29.7 (2013): 1758-1773.

24. Baldoni, Roberto, et al. "A survey of symbolic execution techniques." ACM Computing Surveys (CSUR) 51.3 (2018): 1-39.

25. *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. T. 8. — 2008. — С. 209—224.

26. *Shoshitaishvili Y.* Sok:(state of) the art of war: Offensive techniques in binary analysis [Текст] Y. Shoshitaishvili [и др.] // 2016 IEEE Symposium on Security and Privacy (SP). — IEEE. 2016. — С. 138—157.

27. *Chipounov, V.* S2E: A platform for in-vivo multi-path analysis of software systems [Текст] / V. Chipounov, V. Kuznetsov, G. Candea // Acm Sigplan Notices. — 2011. — Т. 46, № 3. — С. 265—278.

28. Brumley, David, et al. "BAP: A binary analysis platform." International Conference on Computer Aided Verification. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

29. *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.

30. *Poeplau, S.* Symbolic execution with SymCC: Don't interpret, compile! [Текст] / S. Poeplau, A. Francillon // 29th USENIX Security Symposium (USENIX Security 20). — 2020. — С. 181—198.

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

32. Borzacchiello, Luca, Emilio Coppa, and Camil Demetrescu. "FUZZOLIC: Mixing fuzzing and concolic execution." Computers & Security 108 (2021): 102368.

33. Chen, Peng, and Hao Chen. "Angora: Efficient fuzzing by principled search." 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018

34. Cook, S. A. The complexity of theorem proving procedures. In Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), 1971. (и/или Levin, L. A. Universal sequential search problems. Problems of Information Transmission, 9(3):265–266, 1973.)

35. Davis, M., Logemann, G., & Loveland, D. (1962). A Machine Program for Theorem-Proving. Communications of the ACM, 5(7), 394–397.

36. Marques-Silva, J. P., & Sakallah, K. A. (1999). GRASP: A Search Algorithm for Propositional Satisfiability. IEEE Transactions on Computers, 48(5), 506–521.

37. Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., & Malik, S. (2001). Chaff: Engineering an Efficient SAT Solver. In Proceedings of the 38th Design Automation Conference (DAC), 530–535.

38. The smt-lib standard: Version 2.0 [Текст] / C. Barrett, A. Stump, C. Tinelli[и др.] // Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK). Т. 13. — 2010. — С. 14.

39.. Liu, Tianhai, et al. "A comparative study of incremental constraint solving approaches in symbolic execution." Haifa Verification Conference. Cham: Springer International Publishing, 2014.

40. Visser, Willem, Jaco Geldenhuys, and Matthew B. Dwyer. "Green: reducing, reusing and recycling constraints in program analysis." Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering. 2012.

41. *Parygina D*. Strong Optimistic Solving for Dynamic Symbolic Execution. [Текст] / D. Parygina, A. Vishnyakov, A. Fedotov // 2022 Ivannikov Memorial Workshop (IVMEM). — IEEE, 2022. — С. 43—53.

42. *Kutz D.*Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution [Текст] / D. Kuts // 2021 Ivannikov Memorial Workshop (IVMEM). — IEEE. 2021. — С. 42—49.

43. Cha S. K. Unleashing mayhem on binary code [Текст] / S. K. Cha [и др.] // 2012 IEEE Symposium on Security and Privacy. — IEEE. 2012. — С. 380—394.

44. CWE - Common Weakness Enumeration [Electronic Resource]. — URL: https://cwe.mitre.org/ (visited 28.05.2025).

45. Google Sanitizers [Electronic Resource]. — URL: https://github.com/google/sanitizers. (visited 28.05.2025).

46. *Serebryany , K.* AddressSanitizer: A Fast Address Sanity Checker [Текст] / K. Serebryany, D. Bruening, A. Potapenko, D. Vyukov // 2012 USENIX Annual Technical Conference (USENIX ATC 12). — 2012. — С. 309—318.

47. *Stepanov, E*. MemorySanitizer: Fast detector of uninitialized memory use in C++ [Текст] / E. Stepanov, K. Serebryany // 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). — 2015. — P. 46—55.

48. *Serebryany, K.* ThreadSanitizer: Data Race Detection in Practice [Текст] / K. Serebryany, T. Iskhodzhanov // Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA '09). — 2009. — С. 62—71.

49. Chen, Yaohui. SAVIOR: Towards Bug-Driven Hybrid Testing / Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, Long Lu // 2020 IEEE Symposium on Security and Privacy (SP). — 2020. — Pp. 1580–1596.

50. Wang, Tielei. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution / Tielei Wang, Tao Wei, Zhiqiang Lin, Wei Zou // NDSS. — 2009.

51. *Godefroid, P.* DART: Directed automated random testing [Текст] / P. Godefroid, N. Klarlund, K. Sen // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. — 2005. — С. 213—223.

52. *Sen, K.* CUTE: A concolic unit testing engine for C [Текст] / K. Sen, D. Marinov, G. Agha // ACM SIGSOFT Software Engineering Notes. — 2005. — Т. 30, № 5. —С. 263—272.

53. *Necula G. C.* CIL: Intermediate language and tools for analysis and transformation of C programs [Текст] / G. C. Necula [и др.] // International Conference on Compiler Construction. — Springer. 2002. — С. 213—228.

54. *Bhansali S.* Framework for instruction-level tracing and analysis of programs [Текст] / S. Bhansali, W. Chen, S. De Jong, A. Edwards, and M. Drinic. // In Second International Conference on Virtual Execution Environments VEE. — 2006. — С. 154—163.

55. *Cadar C.* Execution generated test cases: How to make systems code crash itself [Текст] / C. Cadar, D. Engler // In Proceedings of the 12th International SPIN Workshop on Model Checking of Software. — 2005. — С. 2— 23.

56. *Cadar C.* EXE: Automatically generating inputs of death [Текст] / C. Cadar [и др.] // ACM Transactions on Information and System Security (TISSEC). — 2008. — Т. 12, № 2. — С. 1—38.

57. *Lattner, C.* LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation [Текст] / C. Lattner, V. Adve // Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Т. 4. — 2004. — С. 75.

58. *Bellard, F.* QEMU, a fast and portable dynamic translator. [Текст] / F. Bellard // USENIX annual technical conference, FREENIX Track. Т. 41. — Califor-nia, USA. 2005. — С. 10—5555.

59. *Poeplau, S.* Systematic comparison of symbolic execution systems: intermediate representation and its generation [Текст] / S. Poeplau, A. Francillon // Proceedings of the 35th Annual Computer Security Applications Conference. — 2019. — С. 163—176.

60. *Nethercote, N.* Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation / N. Nethercote, J. Seward // SIGPLAN Not. — 2007. — Т. 42, № 6. — С. 89—100.

61. *Stephens N.* Driller: Augmenting fuzzing through selective symbolic execution. [Текст] / N. Stephens [и др.] // NDSS. Т. 16. — 2016. — С. 1—16.

62. *Zalewski, M.* AFL: American Fuzzy Lop technical "whitepaper" [Electronic Resource] / M. Zalewski. — URL: https://lcamtuf.coredump.cx/afl/technical_details.txt (visited 05.06.2025).

63. *Luk C.-K.* Pin: building customized program analysis tools with dynamic instrumentation [Текст] / C.-K. Luk [и др.] // Acm sigplan notices. — 2005. — Т. 40, № 6. —С. 190—200.

64. Mossberg, Mark, et al. "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts." 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2019.

65. Qi, Zhenxiao, et al. "{SymFit}: Making the Common (Concrete) Case Fast for {Binary-Code} Concolic Execution." 33rd USENIX Security Symposium (USENIX Security 24). 2024.

66. Chen, Ju, et al. "{SYMSAN}: Time and space efficient concolic execution via dynamic data-flow analysis." 31st USENIX Security Symposium (USENIX Security 22). 2022.

67. *Saudel, F*. Triton: A Dynamic Symbolic Execution Framework [Текст] / Florent Saudel, Jonathan Salwan // Symposium sur la s´ecurit´e des technologies de l'information et des communications. SSTIC. — 2015. — С. 31–54.

68. 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.

69. ARM Limited, ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile, 2017, v8.2 Beta.

70. RISC-V Foundation, "ISA Formal Spec Public Review." — URL: https://github.com/riscv/ISA Formal Spec Public Review (visited 05.06.2025).

71. Intel 64 and IA-32 Architectures Software Developer Manuals. — URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited 05.06.2025)

72. ARM7TDMI Technical Reference Manual, The Thumb instruction set [Electronic Resource]. — URL: https://developer.arm.com/documentation/ddi0210/c/ (visited 05.06.2025).

73. E. Cui, T. Li, and Q. Wei, “RISC-V instruction set architecture extensions: A survey,” IEEE Access, vol. 11, 2023.

74. Tempel, Sören, et al. "Accurate and Extensible Symbolic Execution of Binary Code based on Formal ISA Semantics." 2025 Design, Automation & Test in Europe Conference (DATE). IEEE, 2025.

75. Tempel, Sören, Vladimir Herdt, and Rolf Drechsler. "SymEx-VP: An open source virtual prototype for OS-agnostic concolic testing of IoT firmware." Journal of Systems Architecture 126 (2022): 102456.

76. David, Robin, et al. "BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis." 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Vol. 1. IEEE, 2016.

77. Capstone Engine Disassembly Framework [Electronic Resource]. — URL: https://github.com/capstone-engine/capstone (visited 18.09.2025).

78. RISC-V Basic Support Github Pull Request [Electronic Resource]. — URL: https://github.com/JonathanSalwan/Triton/pull/1318 (visited 09.09.2025).

79. Nguyen, Anh Quynh and DANG Hoang Vu. Unicorn: Next generation cpu emulator framework. BlackHat USA 476. 2015

80. Triton RISC-V 64 capture the flag emulation example [Electronic Resource]. — URL: https://github.com/JonathanSalwan/Triton/tree/master/src/examples/python/ctf-writeups/custom-crackmes/riscv64-hash (visited 12.09.2025).

81. Bruening, Derek. Efficient, Transparent, and Comprehensive Runtime Code Manipulation: Ph.D. thesis / Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science. # 2004.

82. *Metzman, J.* FuzzBench: an open fuzzer benchmarking platform and service [Текст] / J. Metzman [и др.] // Proceedings of the 29th ACM Joint Meeting on

European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — ACM, 2021. — С. 1393—1403.

83. HiFive™ Unleashed [Electronic Resource]. — URL: https://www.sifive.com/boards/hifive-unleashed (visited 25.09.2025).

84. ncnn framework [Electronic Resource]. — URL: https://github.com/Tencent/ncnn (visited 28.09.2025).

85. Intel® RealSense™ SDK 2.0 cross-platform library[Electronic Resource]. — URL: https://github.com/IntelRealSense/librealsense (visited 28.09.2025).

86. ncnn Github Issue 6213[Electronic Resource]. — URL: https://github.com/Tencent/ncnn/issues/6213 (visited 28.09.2025).

87. ncnn Github Issue 6215. URL[Electronic Resource]. — https://github.com/Tencent/ncnn/issues/6215 (visited 28.09.2025).

88. ncnn Github Issue 6216 [Electronic Resource]. — URL: https://github.com/Tencent/ncnn/issues/6216 (visited 28.09.2025).

89. ncnn Github Issue 6214 [Electronic Resource]. — URL: https://github.com/Tencent/ncnn/issues/6214 (visited 28.09.2025).

90. ncnn Github Issue 6212 [Electronic Resource]. — URL: https://github.com/Tencent/ncnn/issues/6212 (visited 28.09.2025).

91. ncnn Github Issue 6211 [Electronic Resource]. — URL: https://github.com/Tencent/ncnn/issues/6211 (visited 28.09.2025).

Список рисунков

Список таблиц

Список алгоритмов

Список приложений

Приложение 1. Акты внедрения предлагаемых методов.

АКТ

о внедрении результатов диссертационной работы

Логуновой Влады Игоревны

на тему "Разработка методов гибридного фаззинга для приложений процессорных архитектур Байкал-М и RICSV64", представленной на соискание ученой степени кандидата технических наук по специальности 2.3.5 — "Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей"

Практические результаты диссертационной работы Логуновой Влады Игоревны на тему "Разработка методов гибридного фаззинга для приложений процессорных архитектур Байкал-М и RICSV64":

1) Метод динамической символьной интерпретации бинарного кода архитектуры Байкал-М

2) Метод символьной интерпретации набора целочисленных инструкций архитектуры RISC-V, включающего сокращенные инструкции и псевдоинструкции, были использованы ООО "Лаборатория безопасности" при организации гибридного фаззинга для исследования ПО. Реализованные в инструменте Sydr методы позволили расширить спектр исследуемых приложений, предоставив возможность анализа программ, разработанных для аппаратных архитектур ARM (Aarch64, Байкал-М) и RISCV64, исследуемых в интересах ООО «Базальт СПО».

Генеральный директор

ООО "Лаборатория безопасности"

29.09.2025 Д. В. Пономарев

Акт

о внедрении результатов кандидатской диссертационной работы

Логуновой Влады Игоревны

Результаты диссертационного исследования Логуновой Влады Игоревны на тему "Разработка методов гибридного фаззинга для приложений процессорных архитектур Байкал-М и RICSV64", реализованные в инструменте диниамической символьной интерпретации Sydr, успешно внедрены ООО НТЦ "Фобос-НТ" и используются в работе испытательной лаборатории для организации гибридного фаззинга. Использование данного инструмента позволило выполнить анализ программ, разработанных для аппаратных архитектур ARM (Aarch64, Байкал-М) и RISCV64, исследуемых в интересах ООО «Базальт СПО». ARM64 (Aarch64), а так же увеличить покрытие исследуемого ПО.

Руководитель центра
сертификационных испытаний
ООО НТЦ "Фобос-НТ"

29.09.2025 

П.В. Мельников

Приложение 2. Исходный код для примера косвенной адресации.

Исходный код с косвенными табличными переходами, ассемблерная версия которого для AArch64 приведена на рис. 5 главы 2.

```
#include <stdlib.h>
#include <stdio.h>

typedef void (*Handler)(void);

void func5() { printf( "Function 5\n" ); }
void func4() { printf( "Function 4\n" ); }
void func3() { printf( "Function 3\n" ); }
void func2() { printf( "Function 2\n" ); }
void func1() { printf( "Function 1\n" ); }
void func0() { printf( "Function 0\n" ); }

Handler jump_table[6] = {func0, func1, func2, func3, func4, func5};

int main (int argc, char **argv)
{
    if (argc != 2) {
        printf("Usage: <prog> <N>\n");
        return 0;
    }

    int value = argv[1][0] - '0';

    if ((value < 0) || (value > 5)) {
        printf("Argv should be 0-5\n");
        return 0;
    }
    jump_table[value]();

    return 0;
}
```

Приложение 3. Листинг набора поддерживаемых методом целочисленных инструкций процессорной архитектуры RISC-V.

Листинг содержит инструкции, для которых в предлагаемом главой 3 методе была реализована символьная интерпретация.

Mnemonic	Description
ADD	Add (register)
ADDI (pseudo: MV, NOP)	Add (immediate)
ADDIW (pseudo: SEXT.W)	Add word (immediate)/
ADDW	Add word (register)
AND	And (register)
ANDI	And (immediate)
AUIPC	Add upper intermediate to pc
BEQ (pseudo: BEQZ)	Branch if equal
BGE (pseudo: BLEZ, BGEZ)	Branch if greater or equal
BGEU	Branch if greater or equal (unsigned)
BLT (pseudo: BLTZ, BGTZ)	Branch if less
BLTU	Branch if less (unsigned)
BNE (pseudo: BNEZ)	Branch if not equal
DIV	Signed integer division
DIVU	Unsigned integer division
DIVUW	Word unsigned integer division
DIVW	Word signed integer division
JAL (pseudo: J)	Jump and link
JALR (pseudo: JR, RET)	Jump and link register
LB	Load register signed byte
LBU	Load register unsigned byte
LD	Load register double word
LH	Load register signed halfword
LHU	Load register unsigned halfword
LUI	Load upper intermediate

LW	Load register signed word
LWU	Load register unsigned word
MUL	Signed multiply
MULH	Signed multiply high
MULHSU	Signed–unsigned multiply high
MULHU	Unsigned multiply high
MULW	Word multiply
OR	Or (register)
ORI	Or (immediate)
REM	Signed integer reminder
REMU	Unsigned integer reminder
REMUW	Word unsigned integer reminder
REMW	Word signed integer reminder
SB	Store register byte
SD	Store register double word
SH	Store register halfword
SLL	Logical left shift (register)
SLLI	Logical left shift (immediate)
SLLIW	Word logical left shift (immediate)
SLLW	Word logical left shift (register)
SLT (pseudo: SLTZ, SGTZ)	Set if less than register
SLTI	Set if less than immediate
SLTIU (pseudo: SEQZ)	Set if less than immediate (unsigned)
SLTU (pseudo: SNEZ)	Set if less than register (unsigned)
SRA	Arithmetic right shift (register)
SRAI	Arithmetic right shift (immediate)
SRAIW	Word arithmetic right shift (register)
SRAW	Word arithmetic right shift (immediate)
SRL	Logical right shift (register)
SRLI	Logical right shift (immediate)
SRLIW	Word logical right shift (register)
SRLW	Word logical right shift (immediate)
SUB (pseudo: NEG)	Subtract
SUBW (pseudo: NEGW)	Subtract word

SW	Store register word
XOR	Exclusive or (register)
XORI (pseudo: NOT)	Exclusive or (immediate)

RISC–V compressed instruction extension.

Mnemonic (compressed)	Description
C.ADD	Add (register)
C.ADDI	Add (immediate)
C.ADDI16SP	Add to SP (immediate, multiplied by 16)
C.ADDI4SPN	Add to SP (immediate, multiplied by 4)
C.ADDIW	Add word (immediate)
C.ADDW	Add word (register)
C.AND	And (register)
C.ANDI	And (immediate)
C.BEQZ	Branch if equal to zero
C.BNEZ	Branch if not equal to zero
C.J	Jump
C.JAL	Jump and link
C.JALR	Jump and link register
C.JR	Jump register
C.LD	Load register double word
C.LDSP	Load register double word from SP
C.LI	Load immediate
C.LUI	Load upper intermediate
C.LW	Load register signed word
C.LWSP	Load register word from SP
C.MV	Move register
C.NOP	No operation
C.OR	Or (register)
C.SD	Store register double word
C.SDSP	Store register double word to SP
C.SLLI	Logical left shift (immediate)

C.SRAI	Arithmetic right shift (immediate)
C.SRLI	Logical right shift (immediate)
C.SUB	Subtract
C.SUBW	Subtract word
C.SW	Store register word
C.SWSP	Store register word to SP
C.XOR	Exclusive or (register)