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

На правах рукописи

Куц Даниил Олегович

Метод моделирования косвенной адресации в рамках динамической символьной интерпретации

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

Диссертация на соискание учёной степени
кандидата технических наук

Научный руководитель:
кандидат технических наук
Федотов Андрей Николаевич

Москва — 2023

Оглавление

Стр.

Стр.

Введение

В настоящее время программное обеспечение все глубже интегрируется во все сферы жизни общества, поэтому проблема безопасности программ становится как никогда актуальной. Выявление и устранение ошибок производится на протяжении всего жизненного цикла программного обеспечения, включая самые ранние этапы написания программного кода.

Из-за возросших размеров и сложности современного программного обеспечения для поиска дефектов применяются инструменты автоматического обнаружения ошибок в программах. Одним из основных методов, применяемых этими инструментами, является динамический анализ. При таком подходе, анализ программы производится на основе информации, собираемой во время ее исполнения. В настоящее время получил широкое распространение такой метод проведения динамического анализа, как символьная интерпретация.

Метод динамической символьной интерпретации позволяет подбирать новые входные данные, опираясь на логику и внутреннюю структуру программы. Суть данного метода заключается в построении математической модели исполнения программы. Для этого реальные входные данные заменяются на символьные переменные, которые могут принимать любые значения. В процессе исполнения программы ее инструкции последовательно интерпретируются, в результате чего над символьными переменными строятся ограничения в виде булевых формул в терминах SMT. Полученный набор ограничений используется, чтобы проверить, возможно ли привести программу в определенное состояние. Чтобы открыть новый путь исполнения программы достаточно изменить направление выполненных условных переходов. Для этого на основе математической модели исполнения программы составляется формула, где к выражению инструкции ветвления применена операция отрицания. С помощью решателя SMT-формул проверяется непротиворечивость полученной формулы, и подбираются значения символьных переменных, удовлетворяющих решению этой формулы. На основе этих значений конструируются новые входные данные, которые приведут программу к исполнению по новому пути.

Современные реалии таковы, что метод символьной интерпретации наиболее эффективен в гибридном подходе к фаззингу, который заключается в комбинированном применении методов фаззинга и символьной интерпретации для

исследования бинарных программ. Важным уточнением является применимость метода для анализа бинарных программ, поскольку исходный код исследуемых программ не всегда может быть доступен.

Существующие инструменты, реализующие метод символьной интерпретации, имеют ряд недостатков, которые снижают эффективность подхода. Так, обнаружение некоторых путей исполнения программы становится невозможным из-за неточности математической модели исполнения, учитывающей только явные зависимости по данным. Возникают сложности при моделировании передачи управления с косвенной адресацией (*jmp qword [rax]*). Один из вариантов использования косвенных переходов в бинарных программах это оптимизация компилятором оператора ветвления *switch* языка Си. В таком случае адрес перехода выбирается из расположенной в памяти таблицы переходов, получаемой на этапе компиляции программы. В таких косвенных переходах нет явной связи между условным выражением и направлением перехода, поэтому они не могут быть смоделированы классическим алгоритмом динамической символьной интерпретации.

По этой же причине данный метод не обрабатывает любые табличные преобразования, где доступ к таблице производится по символьному адресу. Табличные преобразования в программах позволяют выбрать некоторое константное значение в зависимости от других данных, не прибегая к многочисленным условным переходам. Такие конструкции нередко используются в программах, производящих сложную обработку данных, например, при вычислении контрольных сумм. Хотя напрямую табличные преобразования поток управления не изменяют, получаемые значения участвуют в дальнейших вычислениях и используются в условных переходах. Без поддержки таких зависимостей теряется связь между символьными переменными и данными, в результате чего в математической модели исполнения программы отстутствует часть ограничений.

Для корректной обработки косвенной адресации в рамках символьной интерпретации необходимо описать зависимость между символьным значением адреса и диапазоном значений в памяти, который он может адресовать. Здесь возникают проблемы определения границ участка памяти, к которому производится доступ, и способ построения SMT-выражений, описывающих доступ по символьному адресу. От точности определения участка памяти зависит корректность конструируемых ограничений. Способ построения выражений для описания кос-

венной адресации также важен, поскольку это влияет на производительность анализа в целом.

Различные решения вышеописанных проблем были предложены в работах ряда ученых (Д. Брамли, И. Юн, Л. Борзакелло, К. Кадар), но все они либо обладают низкой точностью, либо решают проблему лишь частично, либо реализованы в коммерческих закрытых инструментах (Mayhem). Таким образом, остается актуальной задача разработки метода, позволяющего учитывать косвенную адресацию в контексте динамической символьной интерпретации. Алгоритм, позволяющий обрабатывать косвенные переходы и символьную адресацию памяти сможет повысить точность динамического анализа, увеличить число обнаруживаемых путей исполнения и расширить спектр программ для анализа.

Целью данной работы является разработка метода моделирования косвенной адресации в рамках динамической символьной интерпретации. Разработанный метод должен быть применим к бинарным программам, работающим под управлением ОС Linux, и не требовать доступности исходного кода.

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

1. Разработать метод поиска и моделирования косвенных переходов.
2. Разработать алгоритм определения участка памяти, к которому может производиться доступ по символьно вычисляемому адресу.
3. Исследовать различные способы построения SMT-выражений для моделирования косвенной адресации с целью выбора оптимального подхода по точности и производительности.
4. Разработать метод моделирования чтений памяти по символьно вычисляемому адресу.
5. Реализовать разработанные методы в инструменте динамической символьной интерпретации. Оценить эффективность предложенных методов.

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

1. Разработан метод поиска и моделирования косвенных переходов. Метод позволяет обнаруживать такие конструкции в бинарном коде и определять целевые адреса переходов.

2. Разработан метод моделирования чтений памяти по символьно вычисляемому адресу. Разработанный метод выполняет построение SMT-выражений оптимальным способом, который был определен в проведенном исследовании.

Теоретическая и практическая значимость. Теоретическая значимость заключается в разработанных методах моделирования косвенных переходов и инструкций, осуществляющих чтение по символьному адресу. Кроме того, была проведена экспериментальная оценка эффективности различных подходов к построению символьных ограничений при их обработке в SMT-решателях.

Практическая значимость работы состоит в том, что предложенные методы моделирования косвенной адресации помогают улучшить результаты динамического анализа. Предложенные методы реализованы в инструменте динамической символьной интерпретации Sydr и применяются в Центре доверенного искусственного интеллекта ИСП РАН. Также разработанные методы внедрены и используются в процессах безопасной разработки ООО «Код Безопасности».

Методология и методы исследования. Результаты диссертационной работы получены с использованием методов динамической символьной интерпретации, анализа бинарного кода и динамической бинарной инструментации. Математическую основу исследования составляют теория алгоритмов, теория множеств и математическая логика.

Основные положения, выносимые на защиту:

1. Метод поиска и моделирования косвенных переходов. Производится обнаружение в бинарном коде косвенных условных переходов. Метод позволяет исследовать альтернативные пути исполнения в таких точках ветвления.
2. Метод моделирования чтений памяти по символьно вычисляемому адресу, позволяющий учитывать косвенную адресацию в символьной модели исполнения программы. Метод использует способ построения выражений, который позволяет учитывать символьные значения памяти.
3. Программный инструмент, реализующий методы поиска и моделирования косвенных переходов и чтений по символьно вычисляемому адресу.

Апробация работы. Основные результаты работы обсуждались на Открытой конференции ИСП РАН в 2017, 2019 и 2020 гг.; на международной конференции Ivannikov Memorial Workshop-2021, Нижний Новгород, Россия.

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

Публикации. По теме диссертации опубликовано 4 научные работы, 4 из которых [1—4] изданы в журналах, входящих в перечень рецензируемых научных изданий ВАК при Минобрнауки РФ. Работы [1—4] индексированы системами Web of Science и Scopus. Зарегистрированы 4 программы для ЭВМ [5—8]. В работе [1] личный вклад автора заключается в реализации подсистемы генерации данных в составе инструмента Anxiety. В статье [2] автором было выполнено описание применения SMT-решателей в динамическом анализе и получен набор данных для проведения экспериментов. В совместной работе [3] представлен разработанный автором метод поиска и моделирования косвенных переходов. В статье [4] представлен разработанный автором метод обработки символьных адресов в рамках динамической символьной интерпретации.

Объем и структура работы. Диссертация состоит из введения, 4 глав и заключения. Полный объём диссертации составляет 113 страниц, включая 5 рисунков и 7 таблиц. Список литературы содержит 75 наименований.

Глава 1. Обзор работ

1.1 Динамическая символьная интерпретация

Символьная интерпретация, как один из методов автоматического анализа программ, впервые был описан в исследовании Дж. Кинга [9] в 1976 году. Метод заключается в построении математической модели исполнения программы, которая описывает ход исполнения программы в зависимости от полученных ей на вход данных. На основе этой модели становится возможным проверка различных свойств тестируемой программы, таких как возможность изменения потока управления или возникновения ошибочных ситуаций [10]. Для построения такой модели входные данные программы заменяются на специальные символьные переменные. В качестве входных данных программы как правило выступают либо читаемый программой файл, либо стандартный поток ввода. Также это могут быть аргументы командной строки, сетевой интерфейс или переменные окружения. В ходе исполнения программы каждая операция над входными данными интерпретируется, в результате чего происходит построение специальных ограничений над соответствующими символьными переменными. Эти ограничения имеют вид булевых формул в теориях (SMT, Satisfiability Modulo Theories) [11] и семантически эквивалентны произведенным в программе вычислениям. Результат выполнения операции описывается формулой над символьными переменными, то есть его значением можно управлять, изменяя определенным образом входные данные. Таким образом в ходе анализа происходит распространение символьных переменных вдоль потока данных в программе. Данные, не зависящие от символьных переменных, называют конкретными. В символьной интерпретации и построении математической модели участвуют только те операции, которые работают с символьными данными.

Символьная интерпретация производится вместе с исполнением программы, причем это может быть как реальное исполнение программы в системе, так и трансляция и эмуляция его ее кода без непосредственного запуска. В случае, когда программа запускается в системе, то символьная интерпретация производится методом инструментации – встраивания анализирующего кода в код тестируемой программы. Дополнительный анализирующий код позволяет по-

лучать необходимую для анализа информацию о ходе выполнения программы – исполняемые инструкции, базовые блоки программы, состояние регистров и памяти в каждый момент времени. Важным свойством инструментации является минимизация влияния на объект анализа – поведение инструментированной программы не должно отличаться от ее обычного исполнения. Существует множество различных способов проведения инструментации программ [12], среди них можно выделить статические и динамические методы. Статическая инструментация производится без запуска программы либо на стадии компиляции из исходного кода, либо уже над бинарным исполняемым файлом (Static Binary Instrumentation, SBI). Динамическая инструментация осуществляется непосредственно во время исполнения тестируемой программы используя методы динамической компиляции (JIT компиляции) или эмуляции кода. Наиболее популярными являются системы динамической бинарной инструментации (Dynamic Binary Instrumentation, DBI), позволяющие инструментировать отдельные процессы пользовательского пространства и быть полностью прозрачными для самой анализируемой программы. Системы DBI используют подход динамической инструментации, при которой блоки инструкций программы перекомпилируются и инструментируются непосредственно перед выполнением. Самыми распространенными DBI-фреймворками являются Pin [13], DynamoRIO [14] и Valgrind [15]. Фреймворки Pin и DynamoRIO предоставляют пользователю возможность инструментировать программу как на уровне отдельных инструкций, так и событий (системные вызовы, вызовы функций, создание потоков). Также инструменты имеют удобный и функциональный интерфейс, позволяющий напрямую проводить множество низкоуровневых трансформаций бинарного кода, благодаря чему они зачастую используются в качестве основы для инструментов анализа кода [16—18]. Немного другого способа придерживается фреймворк Valgrind. Вместо перекомпиляции блоков инструкций, программа сначала динамически транслируется в промежуточное представление VEX IR, над которым затем и производится инструментация. Несмотря на то что из-за такого решения страдает производительность, промежуточное представление дает возможность пользователю создавать платформо-независимые решения для инструментации [19; 20]. Аналогичным образом происходит инструментация бинарных программ при помощи платформы QEMU [21]. Эта платформа полносистемной эмуляции для различных архитектур, которая также может быть использована для инструментации работающих в пользовательском режиме программ. Инструментация

бинарной программы может быть реализована на уровне промежуточного представления TCG IR (Tiny Code Generator), в который динамически транслируется эмулируемый код.

В процессе символьной интерпретации программы происходит построение предиката пути. Это набор ограничений над символьными переменными, который обусловлен исполнением программы по определенному пути. Путь исполнения программы – это последовательность выполненных инструкций, он формируется потоком управления в точках ветвления программы. С точки зрения динамического анализа, каждому пути исполнения соответствует свой набор входных данных, необходимый для воспроизведения программы по этому пути. Основной задачей динамической символьной интерпретации является генерация входных данных для альтернативных путей исполнения, которые затрагивают новые участки кода и увеличивают покрытие программы. Чтобы заставить программу выполниться по другому пути, достаточно изменить поток управления в одной из точек ветвления. Точками ветвления в программе, как правило, являются условные переходы, где в зависимости от определенного условия происходит передача управления на различный код в программе. Чтобы инвертировать определенный условный переход, необходимо специальным образом составить новые ограничения, которые будут соответствовать альтернативному пути. То есть, используя существующий предикат пути, нужно описать новый путь исполнения, который будет отличаться направлением перехода в одной из точек ветвления. Составляется специальная формула, которая состоит из всех ограничений пути до выбранного перехода и ограничения самого условного перехода, но с отрицанием условия. Затем необходимо подобрать значения символьных переменных, которые не противоречат сконструированной формуле при подстановке. Для этого используются специальные инструменты – решатели булевых формул в теориях, SMT-решатели. Они могут принимать формулу на вход как через программный интерфейс, так и в текстовом виде. Самым распространенным форматом текстовой записи формул является язык SMT-LIB [22]. Для кодирования ограничений пути могут использоваться различные теории языка. В бинарном анализе, как правило, используются теории битвекторов (QF-BV) и битвекторов с массивами (QF-ABV), поскольку битовые вектора удобны для низкоуровневого представления данных программы. Задача разрешения SMT-формулы сводится к обычной задаче разрешимости булевых ограничений SAT, которая является NP-полной [23]. Это означает, что не существует эффективного алгоритма ее решения. Чем сложнее формула и чем

больше ограничений она содержит, тем дольше выполняется поиск решения. Для этого используются специальные алгоритмы DPLL/CDCL [24], заключающиеся в рекурсивном переборе всех возможных значений переменных с некоторыми оптимизациями. Так, базовый алгоритм DPLL обходит дерево возможных значений переменных в определенном порядке, отдавая предпочтение одиночным переменным. Более продвинутый алгоритм CDCL анализирует неудачные решения и запоминает конфликтные переменные, чтобы избежать перебора части значений в заведомо неправильном решении.

Производительность SMT-решателей является одним из основных ограничивающих факторов динамической символьной интерпретации. Некоторые исследования [25] показывают, что для слишком сложных SMT-формул, соответствующих, например, операциям над числами с плавающей точкой, обычный фаззинг подбирает решения быстрее, чем SMT-решатель. Решением формулы является набор значений для символьных переменных, которые при подстановке в ограничения не вызывают противоречий. Часто случается, что формулы не имеют возможных решений из-за наличия взаимоисключающих ограничений. Как правило, такие ситуации возникают из-за того, что выбранный условный переход невозможно инвертировать в контексте текущего пути исполнения из-за наличия противоречивых условных переходов или избыточных ограничений пути. Также неразрешимые формулы могут быть следствием неточности символьной интерпретации или потери символьных зависимостей. Если же SMT-решателю удалось обнаружить решение, то на основе подобранных значений символьных переменных составляется новый набор входных данных. Если запустить анализируемую программу на этом наборе, то она должна исполниться по изначальному пути исполнения вплоть до инвертированного условного перехода, где поток управления изменится, и программа продолжит исполняться по новому пути. Иногда, вследствие неточно составленных ограничений пути, сгенерированный набор может не приводить к открытию нового путь исполнения. Причиной тому могут быть неучтенные символьные зависимости или зависимость исполнения программы от случайных неконтролируемых значений (например, временные метки).

1.2 Обзор инструментов динамической символьной интерпретации

Одним из самых распространенных подходов к реализации динамической символьной интерпретации является совмещение конкретного исполнения программы и ее символьной интерпретации. Такой подход получил название конкретно-символьное исполнение, или конколик (от англ. concolic = concrete + symbolic) и впервые был описан в инструменте DART [26]. Инструмент предназначен для увеличения тестового покрытия программы. Символьная интерпретация в нем происходит вдоль одного пути исполнения, определяемого конкретным исполнением программы на начальных входных данных. Каждая исполняемая инструкция программы символьно интерпретируется, часть значений для интерпретации достается из конкретного состояния программы. Реализуется это через инструментацию тестируемой программы, которую DART выполняет над ее исходным кодом с помощью CIL [27]. В инструменте CUTE [28], разработанного теми же авторами, используется такая же реализация конкретно-символьного исполнения. Инструмент реализует ставшие стандартом оптимизации для решателя булевых ограничений, в частности быстрая лексикографическая проверка на несовместность последнего ограничения, удаление общих ограничений и инкрементальное решение формул. В обоих инструментах при интерпретации доступа к памяти по символьному адресу происходит его конкретизация, то есть замена символьного выражение на значение из конкретного исполнения. Используемое в этих инструментах конкретно-символьное исполнение позволяет инвертировать условные переходы, лежащие только на одном конкретном пути исполнения. Для полноценного исследования путей исполнения программы производится ее последовательный перезапуск и символьная интерпретация на разных входных данных. Для этого в CUTE применяется специальная стратегия обхода путей в глубину, предполагающей выбор и инвертирование наиболее перспективных путей исполнения.

В отличие от описанных инструментов, SAGE [29] для проведения динамической символьной интерпретации достаточно только бинарной программы. Инструмент поддерживает анализ Windows-программ, которые читают входные данные из файла. Авторы называют подход SAGE фаззингом методом белого ящика, поскольку также как и обычный фаззер, инструмент генерирует новые входные данные для бинарной программы, но при этом основываясь на ее внут-

ренней логике. Во время конкретного исполнения тестируемой программы все инструкции бинарного кода записываются в виде специальной трассы исполнения. Эта трасса затем используется для воспроизведения работы программы и выполнения символьной интерпретации. В ходе анализа SAGE итеративно исследует все пути исполнения программы. Из очереди, изначально состоящей только из одного файла, выбираются входные данные для конкретного исполнения программы. Инструмент инвертирует все условные переходы на записанном пути исполнения и генерирует соответствующие входные данные, которые затем добавляются в очередь для последующих итераций анализа. Для каждого сгенерированного входного файла запоминается его «глубина» – порядковый номер инвертированного условного перехода. При символьной интерпретации программы, запущенной на сгенерированном файле, инвертируются только те условные переходы, которые имеют больший порядковый номер, чем «глубина» файла. Это позволяет инструменту избежать повторного инвертирования условных переходов, расположенных на совпадающих частях путей исполнения. Выбор входного файла на очередной итерации анализа SAGE производит в соответствии с метрикой, позволяющей инструменту как можно быстрее увеличивать покрытие кода. Приоритет отдается входному файлу, который привел к исполнению большего числа новых базовых блоков в программе, поскольку на соответствующем пути исполнения находится больше неисследованного кода и новые условные переходы для инвертирования. Сгенерированные входные данные не всегда могут приводить к исполнению программы по предсказанному пути – такие явление называется в SAGE дивергенцией. Такие файлы, как правило, приводят к инвертированиям одних и тех же условных переходов и заполнению очереди одинаковыми входными данными, что существенно замедляет исследование программы. Входные файлы, на которых проявилась дивергенция, имеют наименьший приоритет при выборе следующего пути для анализа. Для того чтобы уменьшить нагрузку на SMT-решатель, SAGE производит различные оптимизации над построенными ограничениями пути. Для уменьшения размеров формул SAGE удаляет независимые ограничения. Конкретизация некоторых символьных вычислений (операции умножения и битовые сдвиги) позволяет избавиться от слишком сложных для SMT-решателя формул. Наиболее интересной является оптимизация поглощающих ограничений, которая, по утверждению разработчиков, очень полезна при анализе программ, выполняющих разбор структурированных файлов. Данный подход позволяет удалить ограничения пути, которые уже под-

разумеваются другими ограничениями. Так, при интерпретации цикла, который на каждой итерации добавляет в предикат пути все более строгое ограничение, в результате оптимизации останется только ограничение последней итерации, уже включающей в себя ограничения всех предыдущих итераций.

Стратегия динамического анализа, при которой для символьной интерпретации программы вдоль альтернативного пути исполнения требуется ее перезапуск на новом наборе входных данных, называется анализом в офлайн режиме. Недостатком этого режима является необходимость повторно интерпретировать один и тот же код при каждом запуске программы. Альтернативной стратегией анализа является динамическая символьная интерпретация в режиме онлайн. Такой подход впервые был реализован в инструменте EXE [30]. Основная идея заключается в том, чтобы проводить символьную интерпретацию вдоль разных путей исполнения за один запуск программы. Для этого в каждой точке ветвления программы, зависящей от символьных данных, процесс анализа клонируется (fork) для исследования двух разных направлений перехода, соответствующие ограничения добавляются в предикат пути в каждом процессе. Каждый разветвленный процесс затем делает запрос к SMT-решателю, чтобы проверить совместность получившегося предиката пути. Отрицательный ответ SMT-решателя означает, что заданный путь исполнения программы невозможен, и соответствующий процесс анализа завершается. В противном случае символьная интерпретация программы продолжается во всех разветвленных процессах по своему пути. Анализ в режиме онлайн позволяет моментально интерпретировать только что открытый код и выполнять символьную интерпретацию одновременно вдоль многих путей исполнения. Однако из-за экспоненциального роста числа путей исполнения в реальных программах, такой подход потребляет много памяти и требует внушительных вычислительных ресурсов.

Прямым развитием EXE является инструмент KLEE [31]. Эти инструменты реализуют отличный от конкретно-символьного исполнения подход к динамическому анализу. Вместо реального запуска и исполнения тестируемой программы, в KLEE происходит полноценная интерпретация инструкций программы. Инструмент производит конкретную и символьную интерпретацию не над бинарным кодом программы, а над инструкциями промежуточного представления LLVM IR, в которое программа предварительно компилируется. Особенностью инструмента является целенаправленный поиск ошибок в коде. Помимо обычного инвертирования условных переходов, KLEE также пытает-

ся подобрать входные данные, которые приведут к аварийному завершению программы, для чего при интерпретации опасных инструкций производятся специальные запросы к SMT-решателю. В KLEE используется специально разработанный для применения в анализе программ SMT-решатель STP [32]. Чтобы снизить нагрузку на STP, инструмент реализует целый ряд оптимизаций символьных ограничений. Помимо простых упрощений и математических преобразований формул, инструмент использует удаление независимых ограничений из предиката пути и кэш контрпримеров. Ограничения пути, конструируемые в результате символьной интерпретации, обладают рядом свойств, позволяющих использовать решения предыдущих запросов. Так, если подмножество ограничений всей формулы не имеет решения, то и вся формула также не имеет решения. И наоборот, если вся формула разрешима, то и ее подмножество ограничений тоже. Благодаря удобству использования и применимости для анализа программ реального мира, KLEE стал широко распространенным инструментом динамической символьной интерпретации, который в настоящее время регулярно обновляется и развивается. На базе этого инструмента было проведено множество исследований и создано много других инструментов [33—35].

Поскольку KLEE производит символьную интерпретацию основываясь только на исходном коде программы, то у инструмента возникают сложности с исследованием окружения программы и ее зависимостей. Инструмент S2E [36] решает эту проблему путем полносистемной эмуляции с использованием QEMU. Инструмент эмулирует работу всей операционной системы целиком, что позволяет ему производить символьную интерпретацию не только самого приложения, но и его зависимостей, окружения и ядра операционной системы. В качестве реализации символьной интерпретации S2E использует инструмент KLEE. Таким образом, эмулируемый код сначала транслируется в промежуточное представление TCG IR, используемое в QEMU, а затем либо перекомпилируется обратно в машинный код для конкретного исполнения, либо конвертируется в промежуточное представление LLVM IR для символьной интерпретации в KLEE. Двойная трансляция кода в разные промежуточные представления сильно сказывается на производительности подхода [37]. Чтобы дать пользователю возможность исследовать только интересующие его компоненты в полносистемной эмуляции, в S2E реализована выборочная символьная интерпретация. Подход подразумевает чередование конкретного исполнения и символьной интерпретации, при котором точность исследования новых путей исполнения остается приемлемой.

Часть бинарного кода символьно интерпретируется в режиме онлайн с помощью KLEE: для каждого инвертируемого условного перехода происходит разветвление процесса анализа и исследование кода продолжается одновременно по множеству путей исполнения. Другая часть бинарного кода исполняется только конкретно, исследование данного кода ограничено в рамках текущего пути исполнения. Обычно смена режима анализа с символьной интерпретации на конкретное исполнение и обратно происходит в момент вызова функций. Так, при включении режима конкретного исполнения происходит конкретизация всех символьных аргументов вызываемой функции. После возврата из этой функции режим символьной интерпретации возобновляется. Из-за пропуска символьной интерпретации на части кода теряется полнота и точность анализа. Чтобы преодолеть эти проблемы, S2E конструирует специальные ограничения пути, которые определяют возвращаемое значение функции, побочные эффекты ее исполнения и каким образом конкретизировались ее аргументы. Эти ограничения S2E помечает как нестрогие: для инвертирования некоторых дальнейших условных переходов может потребоваться изменить ход исполнения конкретизированной функции. Если инвертирование перехода становится невозможным из-за таких нестрогих ограничений в предикате пути, то анализ откатывается назад к вызову соответствующей функции и пытается изменить конкретизацию ее аргументов на альтернативные значения. Чтобы иметь возможность перенаправить конкретное исполнение функции по другому пути, S2E дополнительно собирает ограничения в точках ветвления и при конкретном исполнении.

Другим инструментом динамического анализа, который выполняет только интерпретацию тестируемой программы без полноценного конкретного исполнения, является angr [38]. Это мощный кроссплатформенный фреймворк, реализующий множество разных методов бинарного анализа, включая дизассемблирование, статический анализ, фаззинг и символьную интерпретацию. Аналогично KLEE, динамическая символьная интерпретация в angr реализована в виде транслятора – без запуска программы на конкретное исполнение. Исследование путей исполнения программы инструмент производит в режиме онлайн, с ответвлением процесса анализа для каждого открытого пути. Для выполнения анализа инструменту достаточно только бинарного кода программы. Инструмент преобразует бинарный код в промежуточное представление VEX IR, инструкции которого он потом символьного интерпретирует. Использование промежуточного представления позволяет инструменту поддерживать символьную

интерпретацию для различных архитектур. Благодаря тому что фреймворк написан на Python, его удобно использовать в качестве ядра для написания своих инструментов анализа [39; 40]. Это же служит причиной невысокой производительности инструментов, учитывая требования, накладываемые анализом в режиме онлайн.

Гибридный подход, предложенный в Mayhem [41], представляет собой компромисс между онлайн и офлайн стратегиями, который призван использовать преимущества обоих подходов и смягчить их недостатки. Mayhem – это инструмент динамической символьной интерпретации бинарных программ, предназначенный для исследования новых путей исполнения, поиска программных ошибок и автоматического построения эксплойтов для них. Гибридный режим Mayhem предусматривает чередование двух стратегий анализа, начиная с онлайн режима анализа. Когда инструментом израсходовано отведенное количество свободной памяти или когда число анализируемых ответвлений достигает порогового значения, анализ в режиме онлайн приостанавливается. С этого момента исполнение уже ответвленных потоков программы продолжается в режиме офлайн без порождения новых ответвленных процессов. Все инвертируемые условные переходы в программе сохраняются в виде контрольных точек, которые содержат в себе текущее символьное состояние программы и информацию для конкретного воспроизведении программы с этой точки. Как только число активных интерпретируемых путей программы и количество потребляемой памяти опустится ниже определенного уровня, сохраненные контрольные точки будут восстанавливаться в памяти для продолжения анализа. Порядок контрольных точек для восстановления определяется эвристикой выбора наиболее перспективного пути с точки зрения наличия эксплуатируемых дефектов. Так, предпочтение отдается тем путям, которые открывают новый код, поскольку он с большей вероятностью будет содержать дефект, чем уже проанализированный код. Больший приоритет получают пути исполнения, на которых зафиксировано символьное обращение к памяти. Наивысший приоритет имеют пути, на которых обнаруживаются символьные указатели на инструкции. Конкретно-символьное исполнение в Mayhem разделено на два процесса – клиент конкретного исполнения и сервер символьной интерпретации. Такой подход позволяет запускать конкретное исполнение программы на любой платформе, в то время как символьная интерпретация всегда работает на целевой системе. Для проведения символьной интерпретации поток выполняемых инструкций переводится в промежуточное

представление BAP IR [42]. Наряду с исследованием всех путей исполнения, Mayhem имеет возможность ограничивать поиск путей в определенном направлении, для чего к ограничениям пути добавляются специальные предусловия пути. Символьная интерпретация с предусловиями была реализована в AEG [43] – предыдущем инструменте разработчиков. Предусловия позволяют сократить пространство входных данных и представляют собой дополнительную информацию о длине входных данных, известных префиксах и формате данных. Это позволяет пользователю целенаправленно исследовать только определенную часть программы либо вообще ограничить символьную интерпретацию одним путем исполнения. Последнее используется для автоматической генерации эксплойтов на имеющихся входных данных, которые приводят к аварийному завершению программы.

Рассмотренные инструменты разрабатывались для применения динамической символьной интерпретации в качестве отдельного самостоятельного метода анализа программ. В настоящее время все большую популярность приобретает подход, при котором символьная интерпретация используется совместно с инструментами фаззинга. Такой подход получил название гибридный фаззинг и заключается в том, чтобы использовать преимущества обоих методов и свести к минимуму их недостатки. Основой гибридного фаззинга является фаззер, который за счет быстрых и легковесных мутаций очень быстро увеличивает покрытие анализируемой программы и генерирует корпус входных данных. Символьный интерпретатор же выполняет роль «умного» мутатора, который используя медленный и тяжеловесный подход динамической символьной интерпретации позволяет открывать сложные пути, недоступные для фаззера, и целенаправленно искать программные ошибки. Инструмент Driller [40], созданный на основе фреймворка angr, производит запуск фаззера и символьного интерпретатора по очереди. Анализ начинается с фаззинга тестируемой программы, в качестве фаззера Driller использует AFL [44]. Когда AFL в течение нескольких итераций уже не может увеличить покрытие программы, анализ переключается в режим динамической символьной интерпретации. Driller пытается инвертировать условные переходы, которые еще не были пройдены в альтернативном направлении. Все сгенерированные входные данные попадают в корпус AFL, где они будут проанализированы фаззером для последующего использования в мутациях.

Другая концепция гибридного фаззинга была предложена в инструменте QSYM [45]. Она заключается в параллельном запуске фаззера и символьного

интерпретатора, где оба инструмента работают одновременно и помогают друг другу в режиме реального времени. Для этого необходимо чтобы инструмент динамической символьной интерпретации имел хорошую производительность и не занимал слишком много вычислительных ресурсов. В противном случае он будет не успевать открывать полезные пути и создавать помехи для процесса фаззинга. QSYM представляет собой инструмент, реализующий конкретно-символьное исполнение (конколик) в режиме офлайн. Основным преимуществом QSYM является его скорость анализа, которой авторам удалось добиться за счет эффективной архитектуры. Конкретное исполнение и символьная интерпретация в QSYM выполняются в рамках одного процесса. Инструментацию тестируемой программы QSYM выполняет непосредственно во время ее исполнения с помощью DBI фреймворка Pin. Поскольку инструментация производится над бинарной программой, а не над промежуточным представлением, такое решение обеспечивает высокую производительность, хотя и ограничено одной архитектурой. Для еще большего ускорения анализа, QSYM перестает символьно интерпретировать слишком часто исполняющиеся базовые блоки. Такая оптимизация позволяет ему без особого ущерба для точности анализа пропускать вычислительно емкие операции, ведущие к созданию слишком сложных ограничений пути. Высокая производительность символьной интерпретации позволяет QSYM эффективно проводить анализ в режиме офлайн. Чтобы при интерпретации новых путей исполнения избежать повторного инвертирования одних и тех же условных переходов, инструмент поддерживает специальный кэш переходов. Его особенностью является учет контекста условных переходов, а также то, что он все же позволяет инвертировать условные переходы по нескольку раз до определенного порога. Такой прием дает возможность SMT-решателю инвертировать один переход в разных контекстах пути, тем самым пытаясь обойти неточности символьной интерпретации. Для этих же целей QSYM поддерживает так называемые оптимистичные решения. Если SMT-решателю не удалось инвертировать условный переход в контексте текущего предиката пути, то QSYM производит попытку инвертирования этого перехода независимо от всего пути исполнения. Такой подход позволяет бороться с проблемой излишних ограничений (over-constraining), возникающей из-за наличия несовместных переходов на пути исполнения, которые на самом деле являются независимыми и могут не учитываться при инвертировании.

Инструмент SymCC [46] предлагает иной способ реализации конкретно-символьного исполнения, направленный на ускорение процесса инструментации и выполнения программы. Вместо используемого в QSYM метода динамической бинарной инструментации, SymCC внедряет код для проведения символьной интерпретации на этапе компиляции тестируемой программы. Такой подход похож на решения, применяемые в ранних символьных интерпретаторах DART и CUTE, но, в отличие от них, SymCC производит инструментацию не над исходным кодом программы, а над промежуточным представлением LLVM IR. Это обеспечивает инструменту простоту реализации, независимость от разных языков программирования и возможность использовать высокоуровневые знания о программе для символьной интерпретации. Таким образом, инструментация программы производится лишь один раз, что позволяет существенно снизить накладные расходы при выполнении программы. Реализацию бэкенда символьной интерпретации инструмент заимствует из QSYM, однако, благодаря применению статической инструментации, SymCC обладает гораздо более высокой производительностью анализа. Недостатком такого подхода является то, что для анализа программы необходимо наличие ее исходного кода.

Для решения этой проблемы авторы SymCC разработали новый инструмент SymQEMU [47], который использует платформу QEMU для инструментации программы. QEMU производит эмуляцию бинарного кода для различных архитектур, что позволяет SymQEMU не зависеть от наличия исходного кода анализируемых программ. Инструментация программы осуществляется на этапе трансляции бинарного кода в промежуточное представление TCG IR, используемое в QEMU для эмуляции. Использование промежуточного представления позволяет SymQEMU проводить символьную интерпретацию для различных архитектур. Чтобы компенсировать потери в производительности, связанные с трансляцией кода, SymQEMU компилирует инструментированный промежуточный код обратно в бинарный код, который затем исполняется непосредственно на процессоре. Так же как и в SymCC, в этом инструменте используется символьный бэкенд QSYM для проведения анализа. Использование особенностей QEMU позволило SymQEMU добиться лучшей чем в QSYM производительности анализа, несмотря на принципиальную схожесть двух способов инструментации кода.

Практически одновременно с SymQEMU был представлен еще один инструмент динамической символьной инструментации Fuzzolic [48], который также реализует конкретно-символьную модель исполнения в режиме офлайн

и заточен под использование в гибридном фаззинге. Авторы Fuzzolic независимо разработали аналогичный SymQEMU подход, основанный на использовании QEMU и модификации транслятора в промежуточное представление TCG для инструментации анализируемой программы. Основное отличие заключается в том, что в SymQEMU символьная интерпретация внедряется отдельно для каждой инструкции программы, в то время как Fuzzolic интерпретирует весь базовый блок. Fuzzolic поддерживает три режима символьной интерпретации. В начале анализа, до того как программа начинает работать с входными данными, никакой символьной интерпретации не производится, программа исполняется только конкретно. Другой режим используется для интерпретации кода, в котором открытие новых путей исполнения не представляет интерес для пользователя. В таком случае символьная интерпретация инструкций производится только для того, чтобы учесть влияние этого кода на символьное состояние программы. Инвертирование условных переходов в этом режиме не производится. Последний режим представляет собой стандартную символьную интерпретацию с инвертированием переходов и отправкой запросов SMT-решателю. Еще одним важным вкладом инструмента является попытка улучшить процесс решения ограничений пути. Если все предыдущие инструменты ограничивались только оптимизациями запросов SMT-решателю, то создатели Fuzzolic разработали свой инструмент Fuzzy-Sat [49]. Вместо точного, но дорогостоящего SMT-решателя, Fuzzolic использует быстрый и легковесный решатель Fuzzy-Sat, способный выдавать лишь приблизительные, но приемлемые для гибридного фаззинга решения. Основная идея Fuzzy-Sat заключается в том, что решение SMT-формулы для инвертирования одного перехода не должно сильно отличаться от конкретных значений символьных переменных текущего пути исполнения. Fuzzy-Sat находит решения, применяя алгоритмы мутации к конкретным значениям символьных переменных. Полезность приблизительных решений в символьной интерпретации программ была продемонстрирована инструментом QSYM и его оптимистичными решениями.

1.3 Моделирование косвенной адресации

Краеугольным камнем метода динамической символьной интерпретации является моделирование косвенной адресации. В классическом алгоритме анализа, который был описан в разделе 1.1, символьная интерпретация поддерживает только прямые зависимости по данным – когда в вычислении результата операций непосредственно участвуют символьные переменные. Так, операция чтения из памяти будет интерпретирована только в том случае, если значение, лежащее в ячейке памяти, будет символьным. Однако при анализе бинарного кода весьма распространены неявные зависимости по данным. В этом случае на поток управления программы влияют не сами значения данных, а другие значения из памяти, к которым происходит обращение в зависимости от этих данных. Такие зависимости также называют адресными, поскольку символьные данные участвуют в вычислении адресов, которые затем используются для доступа к нужным значениям в памяти программы. Эти значения могут не зависеть от символьных данных и быть константами, но прочитанное значение можно поменять, изменив вычисляемый адрес доступа. Такие косвенные зависимости от символьных данных интересны для анализа, если полученное значение из памяти влияет на ход исполнения программы, то есть используется либо в выражении условного перехода, либо напрямую для передачи управления. Далее в тексте работы при описании косвенной адресации будет также использоваться термин ”адресные зависимости”.

Адресные зависимости в бинарном коде на высоком уровне абстракции, как правило, соответствуют табличным преобразованиям данных в программе, например, при вычислении контрольных сумм или при трансформации формата данных. Адресные зависимости появляются и при широком использовании логики указателей в программе, когда часть указателей становятся символьными. Отдельно среди адресных зависимостей можно выделить косвенные переходы. В отличие от обычных условных переходов, они имеют более двух направлений и организованы через доступ к специальной таблице, в которой находятся целевые адреса для перехода. На основе условного выражения вычисляется адрес доступа к таблице переходов, затем по этому адресу из таблицы достается нужный целевой адрес, куда и происходит безусловный переход. Поскольку условный переход, организованный с помощью косвенной адресации, заканчивается инструкцией

безусловного перехода, то такие конструкции не будут инвертироваться в стандартном алгоритме символьной интерпретации. Для успешного инвертирования таких переходов необходимо моделировать обращение к таблице переходов. Поскольку понятие косвенных переходов относится к бинарному коду, то для описания условных переходов, соответствующих им в программном коде, в данной работе используется выражение табличные условные переходы, либо табличные переходы. Хотя табличные переходы — это лишь частный случай адресных зависимостей в целом, в некоторых инструментах динамической символьной интерпретации они обрабатываются отдельно. Это объясняется тем, что такие конструкции соответствуют в программах обычным операторам ветвления с множественным выбором направления, и от способности инструментов их инвертировать зависит качество проводимого ими анализа. Если обычные табличные преобразования влияют на поток управления лишь через последующие условные переходы, то в данном случае результат обращения к таблице переходов не участвует ни в каких условных переходах, а используется для перехода напрямую. Следовательно, их инвертирование не может быть произведено стандартным алгоритмом и требует дополнительных действий.

1.3.1 Анализ косвенных переходов в бинарном коде

Для моделирования косвенных переходов необходимо их правильно распознавать и обнаруживать все возможные направления перехода. Как правило, табличные условные переходы организованы в несколько этапов – преобразование условия перехода в индекс, вычисление адреса нужной записи в таблице, использование прочитанного значения из таблицы для передачи управления. При этом преобразование условия в индекс происходит в отдельном базовом блоке, а доступ к таблице и переход могут выполняться как за одну, так и за несколько инструкций. Все направления перехода содержатся в специальной таблице, генерируемой во время компиляции и расположенной в секции данных программы. В самой таблице могут храниться как непосредственно целевые адреса для передачи управления, так и определенные значения, которые затем используются для вычисления целевых адресов. Динамическая символьная интерпретация выполняется последовательно над инструкциями бинарного кода, поэтому от-

личить табличный условный переход от обычных табличных преобразований и корректно определить все направления перехода является нетривиальной задачей. Для реализации этого алгоритма в инструменте динамической символьной интерпретации можно изучить применяемые подходы в других инструментах анализа бинарного кода, которые сталкиваются с похожей задачей.

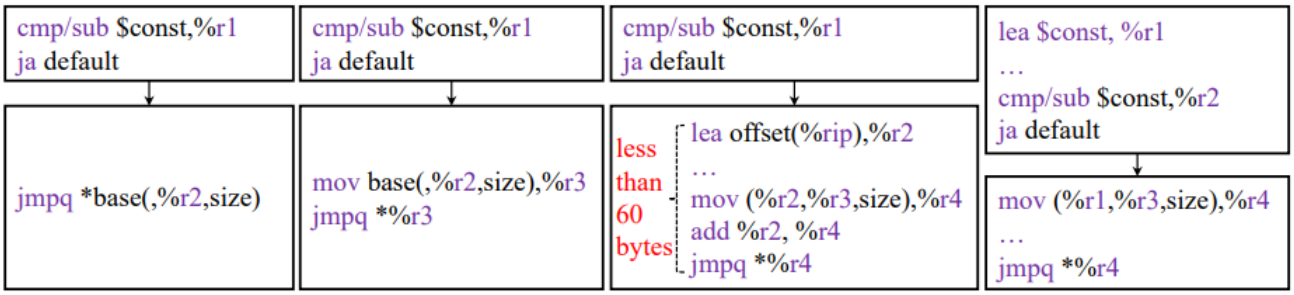


Рисунок 1.1 — Примеры компиляции табличных условных переходов

Инструмент radare2 [50] – это отладчик командной строки, который предоставляет широкие возможности для дизассемблирования, отладки, обратного инжиниринга. В этом инструменте разрешение табличных условных переходов производится простым поиском по шаблону. Всего radare2 поддерживает обнаружение четырех различных типов организации табличных переходов, которые представлены на рисунке 1.1 [51]. Сначала базовый блок проверяется на соответствие с каждым из известных шаблонов. Это может быть как одна или две инструкции в конце базового блока, которые читают целевой адрес из таблицы и передают на него управление, так и несколько расположенных по всему базовому блоку (а иногда и в предыдущем) инструкций, выполняющих сложное вычисление целевого адреса со смещениями. Каждый шаблон предполагает обязательное наличие в предыдущем базовом блоке операции вычисления и валидации индекса в таблице переходов. Для этого выполняется поиск инструкций cmp/sub, в которых производится сравнение индекса с максимальным значением, которое соответствует размеру таблицы переходов. В каждом шаблоне определена инструкция, в которой происходит вычисление адреса доступа к таблице в определенном формате. Формат адресного операнда позволяет распознать значение базового адреса, по которому расположена таблица переходов в памяти, и индексного регистра, который отвечает за адресацию к нужной строке таблицы. Используя полученные значения, инструмент точно определяет таблицу переходов в памяти программы и распознает все направления переходов. Однако существенным недостатком подхода является то, что он распознает табличные пе-

реходы только в строго определенном формате и, следовательно, сильно зависит от компилятора и его настроек.

Другой подход к разрешению табличных переходов использует Dyninst [52] – фреймворк для динамической инструментации программ, предоставляющий обширный API для инструментации, патчинга и анализа бинарных программ во время исполнения. На базе Dyninst разработаны различные инструменты динамического анализа программ [53—55], сам фреймворк уже включает в себя некоторые базовые алгоритмы анализа. Так, для поиска табличных переходов Dyninst выполняет обратный слайсинг по операнду безусловного перехода в конце базового блока. Среди отобранных инструкций, если первое чтение из памяти имеет определенный формат вычисления адреса, такая конструкция распознается как табличный условный переход, и из формата адреса извлекается значение базового адреса (расположение таблицы переходов) и индексный регистр. Затем производится еще один обратный слайсинг по индексному регистру – на 50 операций вверх или до начала текущей функции. На отобранных инструкциях с помощью упрощенного анализа присвоений значений (Value Set Analysis, VSA) [56] инструмент определяет диапазон возможных значений индексного регистра, что соответствует размерам таблицы переходов. Данный метод обладает большей точностью, так как он не зависит от конкретных реализаций, зависимых от компилятора, и выполняет поиск основываясь на общем свойстве табличных переходов. Применение процедуры VSA позволяет точнее определять размеры таблицы переходов, чем просто поиск первой операции сравнения. Ключевым недостатком этого подхода является то, что адрес размещения таблицы в памяти все еще определяется в соответствии со строго заданным шаблоном операнда, что существенно снижает потенциал алгоритма.

Похожий метод для обработки табличных условных переходов применяется в angr. При обнаружении инструкции безусловного перехода на регистр в конце базового блока, angr выполняет для него обратный слайсинг в переделах трех базовых блоков назад. В отличие от предыдущих подходов, angr не пытается по отдельности распознать и оценивать инструкции доступа к таблице и формат вычисления адреса. Инструмент применяет полноценный алгоритм VSA сразу для итогового выражения, по которому совершается переход в конце базового блока. Таким образом, он избегает зависимости от конкретных реализаций табличных переходов разными компиляторами и определяет диапазон возможных направлений переходов напрямую. В динамическом анализе, при наличии информации о

текущем переходе, это позволит обнаружить все другие направления. Метод обладает высокой точностью, но реализация полноценного алгоритма VSA достаточно сложна и его проведение может сказаться на производительности анализа.

1.3.2 Частичная поддержка косвенной адресации

Большинство инструментов динамической символьной интерпретации либо не поддерживают обработку косвенной адресации вообще, либо делают это лишь частично. Отсутствие поддержки таких зависимостей заключается в конкретизации символьных адресов в процессе анализа. Это означает, что символьное выражение адреса не будет участвовать в интерпретации доступа к памяти. Вместо него будет использовано текущее конкретное значение адреса на данном пути исполнения – в качестве целевой ячейки памяти для записи или чтения. В результате, символьность операции зависит только от содержимого этой памяти в случае чтения или записываемого значения в случае доступа на запись [57].

Среди современных инструментов динамической символьной интерпретации распространена частичная поддержка косвенной адресации. Реализованные в этих инструментах подходы не учитывают адресные зависимости в модели символьной интерпретации, но способны обрабатывать некоторые частные случаи. Достоинством таких решений является то, что они сравнительно легко реализуемы и не создают слишком большой нагрузки на анализ. Эффективность разрабатываемых решений с точки зрения производительности чрезвычайно важна, поскольку основным практическим применением инструментов динамической символьной интерпретации в реальном мире является гибридный фаззинг, где любое излишнее замедление символьных интерпретаторов может серьезно снизить эффективность анализа. В данном случае частичная поддержка адресных зависимостей может привести к открытию хотя бы части зависимых от них путей исполнения, а работа по исследованию оставшихся путей будет переложена на процесс фаззинга. Рассмотрим подробнее реализацию таких подходов в различных инструментах.

В инструменте QSYM для обработки адресных зависимостей и инвертирования табличных условных переходов используется один и тот же алгоритм. Каждый раз, когда во время интерпретации встречается доступ к памяти по

символьному адресу, инструмент пытается перебрать возможные значения этого адреса с помощью SMT-решателя. На каждое новое значение адреса генерируется отдельный набор входных данных. Все сгенерированные таким образом входные данные соответствуют одному и тому же пути исполнения, однако за счет различающегося адреса велика вероятность того, что косвенно будет инвертирован переход, зависящий от этого символьного доступа к памяти. Именно таким способом работает инвертирование табличных условных переходов в QSYM: при обработке символьного доступа к таблице переходов будут сгенерированы входные данные, соответствующие доступу к различным записям в таблице, что, в свою очередь, приведет к переходу в различных направлениях. При этом, в данном методе не требуется выполнять никаких дополнительных действий по обнаружению местоположения таблицы переходов и ее размеров. Поскольку число различных значений символьного адреса может быть очень велико, инструмент перебирает не все возможные варианты, а придерживается определенной стратегии, которая заключается в последовательном определении граничных значений адреса. На основе предиката пути составляется формула, в которой выражение символьного адреса сравнивается с его конкретным значением на текущем пути исполнения. При определении верхней границы задается такое условие, что символьный адрес должен быть строго больше его конкретного значения. Затем выполняются итеративные запросы к SMT-решателю, в процессе которых производится перебор значений символьного адреса в большую сторону. Для этого перед каждой итерацией к формуле добавляется новое ограничение, которое устанавливает, что символьный адрес должен быть больше полученного на предыдущей итерации значения. Процесс идет до тех пор, пока не определится максимально возможное в рамках заданного пути значение символьного адреса, то есть пока SMT-решатель не перестанет находить решения. Аналогичным образом выполняется поиск нижней границы символьного адреса, только на этот раз в формулу добавляются сравнения в меньшую сторону. В результате этих двух переборов генерируются входные данные, соответствующие не только максимальному и минимальному значению символьного адреса, но и некоторым промежуточным значениям. Описанный алгоритм применяется для всех инструкций, которые выполняют чтение или запись по символьному адресу. Немного другой подход используется для интерпретации инструкций безусловного перехода на символьный адрес. В данной ситуации любое изменение символьного адреса однозначно соответствует альтернативному пути исполнения, поэтому це-

лесообразно перебрать все возможные значения. Для этого применяется такой же алгоритм итеративных запросов к SMT-решателю, только на каждой итерации к формуле добавляется ограничение на неравенство символьного адреса и нового подобранного значения. Таким образом, запросы производятся до полного перебора всех возможных решений. Метод обработки адресных зависимостей, применяемый в QSYM, достаточно легко реализуем и не требует дополнительного анализа инструкций и памяти. К недостаткам можно отнести то, что при интерпретации каждого доступа по символьному адресу генерируется большое число входных данных, которые изменяют адрес доступа, но не меняют путь исполнения программы. Специальная стратегия перебора адресов смягчает ситуацию, однако это приводит к другой проблеме: не полное покрытие направлений в табличных условных переходах. Единственным исключением являются ситуации, когда табличный переход организован с помощью одной инструкции безусловного перехода – в этом случае инструмент подбирает все возможные значения. QSYM полностью перекладывает всю работу на SMT-решатель, хотя запросы к нему могут быть очень дорогими с точки зрения производительности, что приводит к снижению эффективности всего анализа.

Такой же тактики при обработке адресных зависимостей придерживается инструмент SymQEMU. В нем доступ к памяти на чтение или запись по символьному адресу обрабатывается как «фиктивный» условный переход, где в качестве ограничения для предиката пути используется не условие перехода, а равенство символьного адреса и его конкретного значения. В результате такой доступ к памяти по символьному адресу будет инвертирован наравне с остальными условными переходами в пути, то есть SMT-решатель подберет одно альтернативное значение адреса доступа с соответствующим набором входных данных. Так же как и в QSYM, здесь нет отдельной поддержки для табличных условных переходов, они обрабатываются при доступе к таблице переходов. Такой метод позволяет сильно сократить число запросов к SMT-решателю и количество генерируемых входных данных, явно не открывающих новые пути исполнения. С другой стороны, для табличных условных переходов большинство направлений перехода остается не исследованным.

В SymCC, в отличие от вышеописанных инструментов, табличные условные переходы обрабатываются отдельно от адресных зависимостей. Табличных переходов нет в исходном коде программ, они появляются вследствие оптимизаций компилятора. Так как в SymCC инструментация производится не над

бинарным кодом, а на уровне промежуточного представления LLVM IR, инструмент при выполнении символьной интерпретации работает напрямую с операторами ветвления. При обработке оператора ветвления `switch`, который и является источником табличных переходов, инструмент располагает всей необходимой информацией для его инвертирования. В представлении LLVM имеется информация об условном выражении, обо всех направлениях перехода и соответствующих им значениях условия, а также о направлении перехода для `else`-ветви. Таким образом, благодаря своей архитектуре SymCC может инвертировать табличные переходы так же, как и обычные условные переходы, без выполнения каких-либо дополнительных действий. Для обработки адресных зависимостей здесь используется тот же подход, что и в SymQEMU. При интерпретации операций, в которых для доступа к памяти используется символьный адрес, инструмент выполняет запрос к SMT-решателю с целью подобрать одно альтернативное значение адреса. Соответствующий набор входных данных может косвенно привести к открытию нового пути исполнения.

Метод обработки адресных зависимостей, реализованный в инструменте Fuzzolic, также заключается в простом переборе значений символьного адреса. Основным отличием Fuzzolic является то, что для поиска новых значений адреса он не использует решатель ограничений. При обнаружении доступа к памяти по символьному адресу, в том числе в контексте табличного перехода, инструмент производит так называемый SMT-фаззинг выражения символьного адреса. Для этого к каждому символьному байту в адресе последовательно применяется ряд мутаций, заключающийся в переборе определенных граничных значений байта. При мутации определенного символьного байта остальные конкретизуются, в результате получается перебор различных возможных значений всего символьного адреса. Каждое такое значение адреса проверяется решателем на совместимость с предикатом пути, при положительном решении конструируется новый набор входных данных. Фаззинг символьного адреса не ставит перед собой цели перебрать все допустимые значения адреса, поэтому процесс перебора байт прекращается по достижении определенного порога. Это либо общее число сгенерированных входных данных для каждого символьного адреса, либо число неудачных попыток мутировать байт, когда решатель признал предложенное значение несовместным с предикатом пути. Отдельно стоит отметить, что в данном случае проверка значения адреса в решателе является не полноценным запросом для поиска решения, а легковесной проверкой на совместимость уже предложен-

ного решения. Таким образом, Fuzzolic, так же как и остальные рассмотренные инструменты, производит перебор возможных значений символьного адреса, однако вместо долгих и сложных алгоритмов, используемых в решателях, Fuzzolic использует обычные быстрые мутации фаззинга, которые дополнительно обеспечивают некоторую случайность решений. Такой подход позволяет быстрее генерировать новые входные данные, однако сохраняются все остальные недостатки подхода, связанные с неточностью решений и неполным исследованием путей.

1.3.3 Полноценное моделирование косвенной адресации

Для более точной символьной интерпретации необходима полноценная поддержка косвенной адресации на уровне математической модели исполнения программы [58]. Такой подход позволяет исследовать больше разных состояний программы и целенаправленно генерировать только нужные входные данные, в отличие от методов частичной поддержки, где для каждого символьного адреса создается множество входных данных, большая часть из которых не приносит пользы.

В инструменте KLEE символьная память представляется в виде набора независимых объектов в памяти программы. Каждый объект имеет свой базовый адрес и размер, который при создании объекта конкретизуется если он символьный. Каждый объект моделируется с помощью массива битвекторов. KLEE интерпретирует как чтения, так и записи по символьному адресу, что позволяет инструменту моделировать все аспекты поведения программы. Для обработки таких операций, KLEE прибегает к помощи SMT-решателя, чтобы определить, какие из объектов в памяти программы могут быть затронуты текущим доступом по символьному адресу. Решатель подбирает возможное значение адреса, а KLEE находит объект в памяти, которому соответствует это значение, и разветвляет процесс анализа для интерпретации нового символьного состояния программы. Каждый следующий запрос к SMT-решателю ищет возможное значение адреса за пределами уже подобранных на предыдущих запросах объектов в памяти. В предикате пути ответвленных символьных состояний выражение символьного адреса ограничено в пределах соответствующего объекта памяти. Моделирование сим-

вольного доступа на разные адреса внутри выбранного объекта производится с помощью теории битовых векторов языка SMT. KLEE также пытается определить, возможен ли некорректный доступ к памяти, когда символьный адрес может принимать значения за пределами всех объектов в памяти программы. Если решателю удается подобрать такое значение, то процесс анализа разветвляется и для этого случая. Рассмотренный подход похож на методы частичной поддержки адресных зависимостей, так как здесь тоже создается новое состояние для каждого альтернативного значения символьного адреса. Однако, благодаря поддержке KLEE структурированной модели памяти, число таких состояний сильно меньше, чем при полном переборе значений адреса. К тому же, каждое символьное состояние учитывает ограничения символьного адреса при дальнейшей символьной интерпретации программы.

Для улучшения масштабируемости анализа можно пожертвовать полнотой моделирования и интерпретировать только часть адресных зависимостей. Такой подход используется в инструментах angr и Mayhem. В angr применяется плоская модель памяти, при которой не производится отслеживание различных объектов в памяти. Инструмент рассматривает каждый байт как отдельный элемент, для которого независимо отслеживается символьное состояние. Операции с памятью, затрагивающие сразу несколько последовательных байт, интерпретируются как одна операция, для которой символьные состояние ячеек памяти объединяется в одну формулу. В целях обеспечения масштабируемости анализа, angr обрабатывает только те адресные зависимости, которые с большей вероятностью будут полезны для открытия новых путей и которые не создадут слишком большой нагрузки на анализ. Для всех остальных адресных зависимостей выполняется конкретизация символьного адреса. При интерпретации записи по символьному адресу всегда происходит конкретизация адреса к его максимально возможному значению, которое определяется с помощью SMT-решателя. Это делается с целью максимизировать вероятность появления ошибки исполнения программы, поскольку еще одной целью инструмента является поиск аварийных завершений. Операции чтения по символьному адресу интерпретируются по-другому. С помощью SMT-решателя инструмент определяет диапазон возможных значений символьного адреса, и, если он слишком велик (стандартный порог 1024 байта), происходит конкретизация символьного адреса к любому из значений. В противном случае, angr запрашивает у SMT-решателя список всех возможных значений символьного адреса и составляет специальную формулу, устанавлива-

ющую взаимосвязь между разным значениями адреса и соответствующих ячеек памяти. Эта формула представляет собой вложенные `if-then-else`(ITE) выражения, которые позволяют выбирать результат в зависимости от условия равенства символьного адреса тому или иному значению. Чем больше различных значений адреса обнаружил SMT-решатель, тем больший размер и вложенность имеет итоговая формула. Максимальный диапазон символьных адресов позволяет контролировать размеры конструируемых формул для ограничения пути, взамен часть зависимостей теряется. Инструмент позволяет настраивать пороговый размер чтений, тем самым давая возможность пользователю самому определять баланс между производительностью и полнотой анализа. По этой же причине, angr опционально позволяет пользователю включать интерпретацию записей по символьному адресу. В таком случае моделирование так же происходит только если диапазон значений символьного адреса не больше заданного порога. Для каждого возможного значения адреса составляется ITE формула, которая выбирает одно из двух значений для записи: то, которое будет записано при соответствующем значении символьного адреса, либо значение, которое было в памяти до этого. Символьное состояние для каждого обработанного адреса затем обновляется построенной формулой.

Обработка только части адресных зависимостей для обеспечения приемлемой производительности анализа впервые была предложена в инструменте Mayhem. Основная идея подхода заключается в том, чтобы моделировать только операции чтения и только если возможная область чтения достаточно мала (не больше 1024 байт). Все остальные операции с обращением к памяти по символьному адресу должны конкретизироваться. Mayhem представляет всю память программы в плоском виде, но при моделировании чтения использует понятие объекта в памяти – снимок части всей глобальной памяти, к которой производится доступ. Границы объекта Mayhem определяет с помощью итеративных запросов к SMT-решателю. Зная физические предельные значения символьного адреса (в худшем случае, когда все байты адреса символьные – все адресное пространство), инструмент методом бинарного поиска по очереди определяет максимальное и минимальное возможное значение адреса. Так как бинарный поиск требует большого числа запросов к SMT-решателю, а каждый запрос является достаточно трудоемкой операцией, то Mayhem применяет целый ряд оптимизаций, чтобы улучшить эффективность подхода. Перед запросами к SMT-решателю инструмент выполняет процедуру VSA для определения более узких предварительных

границ символьного адреса. Чем меньше предполагаемый диапазон значений, тем меньше итераций бинарного поиска предстоит выполнить. Исследуя преобразования над символьным адресом в коде программы, метод VSA позволяет получить уточненные предельные значения адреса в виде интервала с определенным шагом. Дело в том, что символьный адрес не обязательно должен принимать все значения в найденном диапазоне. Так, если размер доступа к памяти составляет 4 байта, то список возможных адресов из диапазона будет вычисляться с шагом в 4. Чтобы при доступе к одному и тому же объекту в памяти не выполнять повторный поиск его границ, Mayhem использует кэш уточнений. Для каждого шагового интервала, получаемого из процедуры VSA, в кэше сохраняются найденные с помощью SMT-решателя и бинарного поиска точные границы объекта. Если решение для заданного интервала удается обнаружить в этом кэше, то вместо выполнения бинарного поиска SMT-решатель используется для проверки соответствия данного решения для обрабатываемого символьного адреса. Чтобы максимально оптимизировать обращения к SMT-решателю, Mayhem использует кэширование самих запросов. Каждый обработанный запрос приводится к каноничному виду и сохраняется вместе с решением в специальном кэше лемм. Это позволяет не перерешивать структурно эквивалентные формулы, а моментально получать ответ из кэша еще до отправки запроса к решателю. После определения точных границ объекта символьного чтения, инструмент моделирует доступ по символьному адресу внутри этого объекта. Вместо перечисления всех возможных значений адреса подряд и составления вложенной ITE формулы как в angr, Mayhem конструирует формулу в виде бинарного дерева поиска над диапазоном адресов. Такая формула также составляется через `if-then-else` выражения, но имеет гораздо меньший уровень вложенности, что обеспечивает быстроту обработки в SMT-решателе. Однако общий размер формулы таким подходом снизить нельзя, поскольку общее число листьев дерева остается таким же – это число всех возможных значений символьного адреса. Поэтому в Mayhem применяется еще одна оптимизация – уменьшение числа листьев дерева путем их объединения в линейное выражение. Инструмент представляет возможные варианты символьного чтения в виде точек на плоскости адрес-значение и объединяет несколько последовательных точек в одну линию. Уравнение линии зависит от символьного адреса, и при подстановке его возможных значений позволяет вычислить соответствующее значение памяти. В результате процесса линеаризации память представляется в виде множества линий и изолированных точек.

Финальная формула для моделирования доступа к памяти конструируется в виде бинарного дерева поиска над этим множеством. Такое дерево имеет гораздо меньше листьев, чем обычное дерево бинарного поиска, что обеспечивает меньшие размеры ограничений и более высокую производительность анализа.

Таким образом, все рассмотренные методы моделирования адресных зависимостей можно разделить на два класса, каждый из которых имеет свои преимущества и недостатки. Методы ограниченной поддержки адресных зависимостей имеют высокую производительность анализа и хороши для гибридного фаззинга, однако они не позволяют строить точную модель исполнения программы и поэтому пропускают потенциальные пути исполнения. С другой стороны, методы полноценной поддержки адресных зависимостей позволяют моделировать все аспекты поведения программ и открывать множество новых путей исполнения. Однако такие методы создают большую дополнительную нагрузку на анализ, вследствие чего имеют меньшую производительность. Лишь небольшое число инструментов реализует полноценную обработку адресных зависимостей, но они либо имеют проблемы с производительностью на реальных программах из-за жестких ограничений режима анализа онлайн (angr, KLEE), либо являются закрытыми коммерческими разработками (Mayhem).

1.3.4 Символьная интерпретация в инструменте Sydr

Предложенные в рамках данной работы методы внедрялись в состав инструмента динамической символьной интерпретации Sydr, разрабатываемого в ИСП РАН. Анализ в Sydr представляет собой конкретно-символьное выполнение, для реализации которого используется библиотека Triton [59]. Эта библиотека отвечает за моделирование инструкций программы, а также за отслеживание символьного состояния памяти и регистров при исполнении программы. Символьная интерпретация в Triton реализована таким образом, что при моделировании инструкций отслеживаются только прямые зависимости по данным. Результатом моделирования инструкции является выражение, которое вычисляет новые значения для операндов-приемников в зависимости от операндов-источников этой инструкции. В случае, когда операнд-источник является символьным (т.е. зависит от входных данных программы), в выражении используются соответствующие

символьные переменные. В противном случае используется конкретное значение операнда в текущей точке исполнения программы.

Листинг 1.1 Табличный условный переход

```
1  add    rax, QWORD PTR [r14 + 8]
```

Проверка символьности происходит в зависимости от типа операнда: в символьной модели программы проверяется либо состояние регистра, либо состояние ячеек памяти. Таким образом, в библиотеке Triton операнд доступа к памяти эквивалентен ячейкам памяти, на которые этот операнд указывает. Операнд доступа к памяти представляет собой выражение, вычисляющее адрес памяти, который необходимо разыменовать. Иногда регистры, участвующие в вычислении адреса памяти, сами зависят от входных данных. Такие конструкции, когда символьным является само значение адреса разыменования, называют косвенной адресацией. На листинге 1.1 приведен пример инструкции, в которой одним из операндов-источников является доступ к памяти. Результатом моделирования этой инструкции в Triton будет записанное в операнд-приемник `rax` выражение вида $Value_{rax} + Memory[Addr]$, где $Value_{rax}$ - это значение (символьное или конкретное) регистра `rax`, $Memory[Addr]$ - значение (символьное или конкретное) памяти по адресу $Addr = Value_{r14} + 8$. При вычислении адреса $Addr$ используется конкретное значение регистра `r14`, а символьные выражения для `r14`, если они есть, будут проигнорированы. В результате, значение операнда-приемника `rax` является независимым от символьных выражений, соответствующих регистру `r14`, поскольку Triton всегда конкретизирует значения регистров при вычислении адреса доступа к памяти. Таким образом, инструмент Sydr не поддерживает моделирование косвенной адресации, что ведет к неточности символьной модели программы и потере потенциально новых путей исполнения.

Глава 2. Поиск и моделирование косвенных переходов

2.1 Косвенные переходы

Косвенные переходы в бинарном коде представляют собой инструкции безусловной передачи управления. Для получения адреса назначения перехода в них используются табличные преобразования. В исходном коде программ на языках `C/C++` таких переходов, как правило, нет – они появляются в результате оптимизации компилятором условных операторов. Чаще всего табличные переходы образуются при компиляции оператора ветвления `switch`. Линейная последовательность операторов `if`, при соответствии определенным критериям, также может быть преобразована в табличный переход.

Оператор ветвления `switch` имеет несколько направлений перехода, которые выбираются в зависимости от условного выражения, которое обязательно должно сводиться к целочисленному типу. Также оператор может иметь направление перехода для случаев, когда ни одно из перечисленных условий не выполняется – ветвь `default`, соответствующая `else`-ветви обычного условного перехода. Оператор `switch` является удобной заменой нескольких обычных операторов `if`, последовательно выполняющих сравнения. Оператор `switch` может быть скомпилирован в бинарный код несколькими способами [60]:

– линейная последовательность условных переходов;
– условные переходы в виде дерева;
– табличный переход.

Самым простым способом является компиляция в линейную последовательность условных переходов, когда на каждую ветвь создается отдельный условный переход. При выключенных оптимизациях компилятора будет использоваться именно этот способ. Такой метод не эффективен, поскольку размер генерируемого бинарного кода прямо пропорционален числу ветвей в операторе `switch`. К тому же, чем ниже находится нужный переход, тем больше сравнений приходится выполнять программе, что приводит к уменьшению скорости работы программы. Как следствие, такой способ применяется только для в тех случаях, когда число ветвей `switch` не велико.

Другим способом является формирование из условных переходов дерева поиска. Выбирается отрезок, границами которого являются наименьшее и наибольшее значения условий для ветвей перехода, и над ним с помощью условных переходов строится бинарное дерево поиска. Листьями этого дерева являются направления переходов оператора `switch`. Такой метод не дает больших преимуществ по размеру бинарного кода, однако при частых переходах по различным ветвям требуется производить меньше сравнений.

Самым оптимальным и распространенным способом компиляции условного оператора `switch` является организация табличного перехода (`jump table`, `branch table`). Табличный переход преобразует условное выражение в индекс, который затем используется для получения нужного адреса перехода из заранее составленной таблицы. При этом создание такого перехода возможно только при соблюдении определенных условий:

– Достаточное количество различных направлений перехода. В противном случае организация табличного перехода нецелесообразна, так как это не принесет заметного увеличения скорости. Как правило, табличные переходы создаются при пяти и более различных ветвей [61].
– Условие перехода должно быть представлено целочисленным значением, поскольку оно преобразуется в индекс для доступа к таблице переходов.
– Пространство значений, образуемое условиями всех ветвей, должно быть достаточно плотным. В противном случае таблица переходов будет занимать очень много места, так как ее размеры совпадают со значениями индекса, вычисляемого из условного выражения. Если значения ветвей перехода слишком разрежены, но поддаются кластеризации, то может быть применен комбинированный подход [62]. В таком случае каждый кластер преобразуется в отдельный табличный переход, которые затем объединяются через дерево условных переходов.

При организации табличного перехода сначала создается сама таблица переходов. Каждой ветви условного перехода, за исключением `default`-ветви, соответствует свое целочисленное значение. Все используемые значения упорядочиваются по возрастанию и преобразуются к индексу. Для этого из значения каждой ветви вычитается наименьшее из всех значений, таким образом полученные индексы всегда начинаются с нуля. Затем все целевые адреса переходов объединяются в таблицу таким образом, что позиция адреса в этой таблице соответствует индексу нужной ветви. Для составления таблицы все индексы должны

формировать непрерывную численную последовательность, поэтому всем пропущенным значениям индексов ставится в соответствие `default`-ветвь. Таким образом, чем сильнее разрежены значения ветвей, тем больше одинаковых адресов `default`-ветви будет содержать таблица, вследствие чего она занимает больше места в памяти программы. Таблица с целевыми адресами переходов обычно располагается в секциях `.rodata` или `.data`. Доступ к этой таблице производится на основе условного выражения – некоторого операнда, который содержит конкретное значение условия в текущий момент исполнения. Операнд также приводится к индексу путем вычитания из него наименьшего значения из всех ветвей. Получившийся индекс I_{cond} дополнительно проверяется на выход за границу таблицы – в этом случае управление сразу передается на `default`-ветвь. Используя базовый адрес размещения таблицы переходов в памяти A_{base}, вычисляется адрес нужной строки в таблице:

$$A_{entry} = A_{base} + I_{cond} \times size, \tag{2.1}$$

где $size$ - это размер одной записи таблицы в байтах. По вычисленному адресу A_{entry} из таблицы читается целевой адрес и на него совершается переход. Ассемблерный код такого табличного перехода приведен в листинге 2.1.

Листинг 2.1 Табличный условный переход

```
1  sub    rax, 0x61
2  cmp    al, 0x8
3  ja     0x4005dc  ; default branch
4  mov    rax, [rax * 8 + 0x400688] ;Memory access to the address table
5  jmp   rax
```

Существует также альтернативный вариант создания таблиц переходов, при котором вместо абсолютных значений целевых адресов в таблице находятся специальные значения, используемые в дальнейшем для вычисления нужного адреса перехода. В этом случае все направления переходов имеют общий базовый адрес, к которому для получения абсолютного адреса нужно добавить определенное смещение. Значения этих смещений и лежат в таблице, а такие таблицы обычно называют таблицами смещений (`offset table`).

Вызов функции по указателю из массива (листинг 2.2) – единственный случай, в котором табличный переход образуется не за счет компиляторных оптимизаций. Несмотря на то, что такой код не является условным переходом, он так

же представляет собой точку ветвления в программе и имеет аналогичную табличным переходам структуру бинарного кода. Единственным отличием является инструкция передачи управления в конце базового блока: используется инструкция вызова функции `call`, а не `jmp`.

Листинг 2.2 Код с использованием таблицы указателей на функции

```
1  bool op1(int n);
2  bool op2(int n);
3  bool op3(int n);
4  bool op4(int n);
5
6  bool foo(int data, int type)
7  {
8    // Definition of function pointers array.
9      bool (*operations[4])(int n) = {op1, op2, op3, op4};
10
11   // Call function depending on array index.
12     return operations[type](data);
13 }
```

Табличные переходы являются частным случаем адресных зависимостей, поэтому могут быть обработаны с помощью одного и того же алгоритма. Все операции доступа по символьному адресу в таком алгоритме интерпретируются одинаково и либо тут же инвертируются, либо добавляются в символьную модель программы. В последнем случае инвертирование табличных переходов будет происходить при интерпретации символьных инструкций перехода `jmp/call`. Многие инструменты динамической символьной интерпретации используют такой подход [45—48]. Но существует и альтернативный способ, который был реализован в рамках данной работы – инвертирование табличных переходов отдельно от остальных адресных зависимостей. Такой подход хотя и является более сложным в реализации, однако обладает рядом преимуществ. Основное отличие табличных переходов заключается в том, что получаемое по символьному адресу значение сразу же используется для совершения перехода. Следовательно, не возникает необходимости в конструировании сложных ограничений пути, которые увеличивают потребление памяти и нагрузку на SMT-решатель. Табличные переходы, так же как и обычные условные переходы, должны добавляться в предикат пути и учитываться в кэше переходов. Добавление в предикат пути поз-

воляет увеличить точность анализа и улучшить результаты различных эвристик по учету пути исполнения при инвертировании переходов [63; 64]. Кэширование условных переходов во избежание повторного инвертирования поддерживается практически всеми гибридными фаззерами. Наиболее оптимальные алгоритмы кэширования помимо самого условного перехода учитывают также его контекст – другие условные переходы, расположенные на пути исполнения. Добавление табличных переходов в кэш позволяет избежать повторного инвертирования самого перехода и дополняет контекст, делая весь алгоритм точнее.

2.2 Поиск косвенных переходов в бинарном коде

Алгоритм поиска косвенных переходов в бинарном коде является частью разработанного метода моделирования косвенных переходов. Метод реализован в качестве подсистемы инструмента динамической символьной интерпретации Sydr [3]. Инструмент работает с бинарным кодом анализируемой программы и реализует конкретно-символьный подход к символьной интерпретации. Конкретное исполнение и символьная интерпретация разделены на два параллельно работающих процесса. Конкретный исполнитель реализован в виде клиента DynamoRIO – платформы динамической бинарной инструментации, используемой для анализа конкретного состояния программы. Символьная интерпретация программы производится на уровне отдельных инструкций бинарного кода. Для этого конкретный исполнитель инструментирует базовые блоки программы таким образом, что перед исполнением каждой инструкции в процесс символьного интерпретатора отправляется специальное сообщение, содержащее все необходимые для ее интерпретации данные.

Табличные переходы должны интерпретироваться отдельно от остальных адресных зависимостей, поэтому первой задачей при инвертировании таких переходов является их обнаружение. Эта задача может быть решена с помощью поиска переходов по известным шаблонам. Табличные переходы могут иметь разную структуру в зависимости от архитектуры, разрядности платформы, компилятора и его настроек. На листингах 2.3, 2.4 и 2.5 приведены примеры различной организации табличных переходов. Поиск по шаблонам не является оптимальным способом поиска, поскольку требует жесткого соответствия структуры перехода

(которая может измениться с версией компилятора) и перебора большого числа вариантов.

Листинг 2.3 Условный переход с использованием таблицы адресов

```
1  mov    rax, [rax * 8 + 0x400688] ;Memory access to the address table
2  jmp    rax
```

Листинг 2.4 Условный переход с использованием таблицы смещений

```
1 mov    eax, [rdx + rax]   ;Memory access to the offset table
2 movsxd rdx, eax
3 lea    rax, [rip + 0x110]
4 add    rax, rdx
5 jmp    rax
```

Листинг 2.5 Условный переход для вызова функции из массива указателей

```
1  lea    rdx, [rax * 8]
2  lea    rax, [rip + 0x200872]
3  mov    rax, [rdx + rax]   ;Memory access to the address table
4  call   rax
```

Разработанный метод поиска и моделирования косвенных переходов использует другой подход для обнаружения таких переходов в бинарном коде, который заключается в определении основного алгоритма работы табличного перехода, а не поиске его частных реализаций. Все табличные переходы имеют общее свойство, которое не зависит от вида перехода и типа таблицы (адреса или смещения). Сначала происходит чтение из таблицы, затем это значение косвенно или непосредственно становится целью перехода на последней инструкции текущего базового блока. Алгоритм поиска табличных переходов реализован на уровне анализа базовых блоков с помощью соответствующей инструментации DynamoRIO. Определение наличия такого перехода в базовом блоке происходит непосредственно перед его исполнением. Благодаря кэшированию DynamoRIO, для каждого базового блока этот поиск выполняется только один раз, независимо от того, сколько раз он исполнялся в программе. Алгоритм состоит из двух этапов. Во-первых, производится быстрая и легковесная проверка чтобы определить,

может ли базовый блок потенциально содержать табличный переход. Рассматриваются только те базовые блоки, которые заканчиваются инструкциями вызова `call` и безусловного перехода `jmp`, у которых в качестве цели перехода выступает либо регистр, либо операнд доступа к памяти. В остальных случаях происходит передача управление на жестко заданный адрес, что не может представлять условный переход. Во-вторых, в отобранных базовых блоках производится поиск чтения из памяти, который должен удовлетворять двум условиям:

1. Доступ к памяти должен осуществляться по вычисляемому адресу.
2. Адрес перехода на последней инструкции базового блока должен явно зависеть от прочитанного значения.

На этом этапе анализ производится с помощью обратного слайсинга по инструкциям. Обособленным случаем, не требующим проведения обратного слайсинга, является ситуация, когда инструкции перехода и чтения из памяти совпадают. Такая конструкция сразу же распознается как прямой табличный переход:

```
1 jmp    [rax * 8 + 0x400688]
```

Алгоритм обратного слайсинга обходит инструкции с конца базового блока и отслеживает распространение данных между ними. При обходе инструкций составляется список отслеживаемых регистров $\langle RegList \rangle$, зависимости по памяти в данном алгоритме не учитываются. На каждой инструкции определяются списки ее операндов источников $\langle OpSrc \rangle$ и операндов приемников $\langle OpDst \rangle$. Затем определяется влияет ли эта инструкция на отслеживаемые в текущий момент регистры. Если какой-либо регистр reg из $\langle OpDst \rangle$ находится в $\langle RegList \rangle$, то reg удаляется из $\langle RegList \rangle$, и вместо него туда добавляются все регистры из $\langle OpSrc \rangle$. Если $\langle OpSrc \rangle$ состоит из одного операнда доступа к памяти M, то выполняется проверка на его соответствие определенным критериям. Операнд доступа к памяти M должен быть вычисляемым адресом, но он не должен вычисляться относительно указателя на инструкцию программы (программного счетчика, `program counter`). Таблицы переходов никогда не располагаются в секции исполняемого кода программы `.text`, поэтому адрес, вычисленный относительно программного счетчика (регистр `rip/eip`), не может указывать на таблицы. Если доступ к памяти M соответствует этим критериям, то такая инструкция распознается как доступ к таблице переходов, алгоритм обратного слайсинга завершается, а табличный переход считается обнаруженным. В противном случае слайсинг продолжается до тех пор, пока не будет достигнуто начало базового блока.

Результатом поиска табличного перехода в базовом блоке является адрес инструкции, осуществляющей доступ к таблице переходов. Информация о том, что эта инструкция является частью табличного перехода, передается символьный интерпретатор для дальнейшей обработки. Данные о других инструкциях, в которых происходит вычисление адреса и передача управления, на данном этапе анализе не важны. Главная задача этого алгоритма – отделить инструкцию обращения к таблице переходов по вычисляемому адресу от остальных адресных зависимостей.

2.3 Моделирование косвенных переходов

2.3.1 Определение направлений перехода

Разработанный метод моделирования косвенных переходов заключается в построении таких символьных ограничений, которые позволят обнаружить входные данные для исполнения программы по альтернативным направлениям этих переходов. Работа метода осуществляется в несколько этапов:

- распознавание границ таблицы переходов в памяти программы;
- конструирование символьных ограничений для каждой ветви перехода из таблицы;
- добавление ограничений в предикат пути программы.

Табличный переход должен интерпретироваться таким же образом, как и обычный условный переход, с добавлением в предикат пути соответствующих текущему исполнению ограничений и созданием запросов SMT-решателю для инвертирования перехода. Отличие заключается в том, что для табличного перехода инвертирование подразумевает открытие не одного альтернативного направления, а сразу нескольких. Для того чтобы составить SMT-запрос для инвертирования обычного условного перехода достаточно взять отрицание от ограничения пути, накладываемого условием перехода. В случае табличного перехода необходимо точно знать все доступные направления перехода и какие ограничения пути для этих переходов нужно составить. Определение возможных направлений табличного перехода является нетривиальной задачей при анализе

бинарного кода. Информация о количестве переходов теряется при компиляции, а точное расположение таблицы в памяти закодировано в инструкциях лишь частично и сильно зависит от способа организации и вида табличного перехода. Решению данной проблемы не может помочь даже наличие отладочной информации в бинарном файле программы – она не содержит данных о табличных переходах. Для того чтобы определить направления табличного перехода необходимо как можно более точно распознать таблицу переходов.

Так же как и обнаружение табличных переходов, данный этап анализа выполняется в конкретном исполнителе, поскольку включает в себя исследование адресного пространства исполняемой программы. Перед непосредственным выполнением каждой инструкции программы конкретный исполнитель отправляет в символьный интерпретатор специальное сообщение `InstructionEvent`, которое содержит информацию об этой инструкции и необходимые для ее интерпретации конкретные данные. Если на этапе анализа текущего базового блока был найден табличный переход, то при обработке инструкции доступа к таблице переходов дополнительно передается специальная метка. Поскольку поиск границ таблицы переходов и пересылка соответствующего участка памяти является трудоемким процессом, этот этап анализа производится только для тех табличных переходов, которые зависят от входных данных. Проверка на символьность производится в символьном интерпретаторе, поэтому распознавание таблицы переходов производится только после того, как будет получено ответное сообщение, которое содержит результат проверки. Если адрес, по которому производится доступ к таблице, не зависит от символьных переменных, то дальнейшего анализа и интерпретации табличного перехода не выполняется. В случае если адрес символьный, то производится определение границ таблицы переходов и ее пересылка обратно в символьный интерпретатор для инвертирования табличного перехода.

Рисунок 2.1 — Передача данных о табличном переходе между процессами анализа

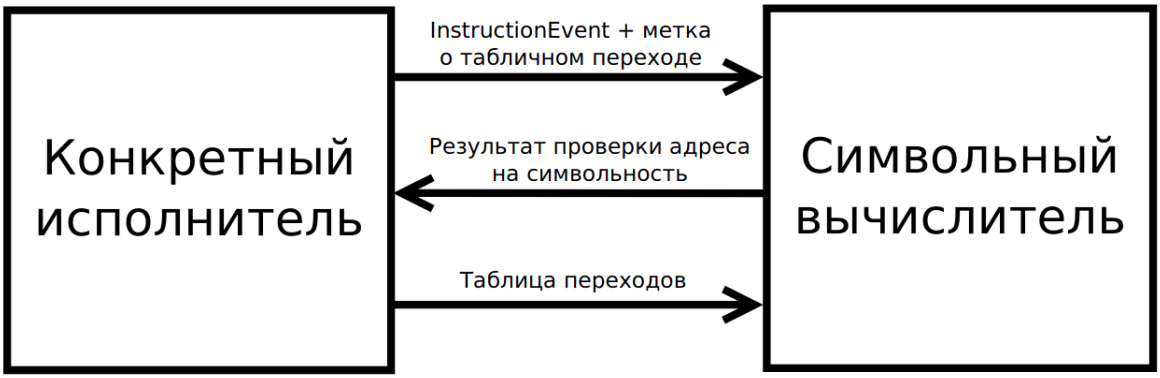


Таблица переходов представляет собой набор адресов или смещений, записанный в памяти программы. Эти данные называют таблицей, так как схематически их можно представить в виде таблицы с двумя столбцами, где первый столбец – это адрес в памяти программы, по которому располагается значение, а второй – это само значение. Конкретный исполнитель может получить любую информацию о конкретном состоянии программы в данный момент времени. Так, для инструкции доступа к таблице определяется конкретный адрес A_{concr}, по которому производится чтение в рамках текущего пути исполнения. По этому адресу осуществляется доступ к одной из строк таблицы, расположение этой строки относительно всей таблицы неизвестно. Также известно содержимое самой памяти по адресу A_{concr}, по которому можно определить тип таблицы переходов – таблица с адресами или смещениями. Используя информацию о том, какого формата значения должны лежать в таблице, алгоритм определяет границы этой таблицы в памяти.

От точки текущего доступа производится обход памяти в сторону больших и меньших адресов с шагом, равным размеру доступа к памяти. Обход продолжается до тех пор, пока содержимое памяти соответствует заданному формату таблицы:

– *Абсолютный адрес*. В остальных ячейках памяти ожидается адрес такой же разрядности. Дополнительно проверяется, что эти адреса соответствуют сегменту кода .text программы, то есть являются корректными указателями на исполняемый код.
– *Смещение*. Значения смещений в табличных переходах в архитектуре x86 являются отрицательной величиной. Соответственно в остальных ячейках памяти также ожидается некоторое отрицательное значение, которое имеет такой же размер (как правило 4 байта).

Поиск левой и правой границ таблицы останавливается, когда значение перестает соответствовать формату, либо когда достигается пороговое значение для размера таблицы. Иногда в памяти могут располагаться сразу несколько подряд идущих таблиц, образуя непрерывное пространство значений одного формата. В таком случае алгоритм выйдет за границу текущей таблицы и начнет анализировать смежную таблицу. Сами таблицы переходов в программах чаще всего имеют сравнительно небольшой размер, поэтому чтобы ограничить число ошибочно распознанных направлений перехода и предотвратить избыточный обход памяти, максимальный размер распознаваемой таблицы ограничен. При поиске одной

границы обход производится не более, чем на 100 строк таблицы. Ошибочно определенные направления перехода (принадлежащие соседним таблицам) ведут к излишней нагрузке на SMT-решатель при инвертировании табличного перехода. Для инвертирования перехода по таким направлениям будет осуществляться отдельный запрос, однако в силу невозможности символьного адреса получить доступ вне текущей таблицы SMT-решатель не сможет найти для них решение. Поскольку табличные переходы создаются компилятором при наличии нескольких направлений перехода (пяти и более), то случай, когда распознанная таблица имеет меньше трех корректных строк, считается ошибочно обнаруженным табличным переходом, дальнейшая обработка такой таблицы прекращается.

В результате работы алгоритма определяются два адреса – левая и правая граница таблицы переходов. Адреса границ, а также содержимое соответствующего участка памяти пересылается в процесс символьной интерпретации.

2.3.2 Инвертирование перехода

Символьный интерпретатор отслеживает символьное состояние программы, производит интерпретацию символьных инструкций и построение предиката пути. Для интерпретации инструкций процесс получает от конкретного исполнителя все необходимые для этого данные из конкретного состояния программы. Для интерпретации инструкции доступа к таблице переходов дополнительно принимается сама таблица переходов – набор адресов доступа и соответствующих им значений памяти. Эти данные используются для построения ограничений пути, необходимых для корректного инвертирования табличного перехода.

Точкой ветвления в программе является инструкция передачи управления в конце базового блока, однако определение цели перехода происходит уже при выполнении инструкции доступа к таблице. В зависимости от того, по какому индексу произойдет обращение к таблице, будет получено соответствующее значение адреса (или смещения), которое и определит фактическое направление перехода на инструкции `jmp/call`. Поэтому ограничения для предиката пути, описывающие ветвление пути исполнения, создаются при интерпретации инструкции доступа к таблице, но не добавляются в символьную модель программы. Сама инструкция чтения из памяти формально не является ветвью, поэтому для

корректного отображения ветвей при записи трассы исполнения эти ограничения добавляются в предикат пути при интерпретации ближайшей инструкции `jmp/call`. Трасса исполнения программы представляет собой набор совершенных условных переходов, соответствующих пути исполнения программы. Каждая ветвь определяется адресом точки ветвления (`source`) и адресом направления перехода (`destination`). Для табличных переходов, аналогично обычным условным переходам, `source`-адресом является инструкция передачи управления. В качестве `destination`-адреса для описания ветви используются адреса из таблицы переходов. Если в таблице переходов находятся смещения, то производится дополнительное вычисление адресов перехода для всех ветвей. Зная текущий адрес перехода $Dest_{current}$ и текущее значение смещения $Offset_{current}$, можно вычислить некое базовое значение $Base = Dest_{current} - Offset_{current}$, относительно которого вычисляются все целевые направления перехода. Тогда, имея в распоряжении всю таблицу переходов со смещениями $Offset_i$, можно вычислить все соответствующие адреса перехода $Dest_i = Base + Offset_i$.

Для составления ограничений обычного условного перехода используется символьное выражение условия этого перехода – операции `cmp` и значений флагов. В табличном переходе условием, от которого зависит направление перехода, является значение символьного адреса доступа к таблице. Одним из способов инвертирования табличного перехода является перебор всех возможных значений, которые может принимать символьный адрес. Обычно перебор производится с помощью итеративных запросов к SMT-решателю. Такой подход не является оптимальным, поскольку в таблице переходов часть ячеек соответствует одинаковым направлениям перехода – `default`-ветвь, заполняющая пробелы между индексами.

В реализованном методе для инвертирования табличного перехода составляется отдельное ограничение пути для каждого обнаруженного на предыдущем этапе направления перехода. Символьные ограничения составляются только для уникальных переходов. Таблица, полученная от конкретного исполнителя, содержит повторяющиеся значения, которые соответствуют `default`-ветви или если разные ветви оператора `switch` ведут к исполнению одного и того же кода. Сначала таблица переориентируется относительно направлений перехода: каждому уникальному адресу перехода ставится в соответствие список соответствующих строк в таблице. Для исполнения программы по альтернативному пути на табличном переходе, необходимо чтобы символьный адрес доступа к таблице стал

указывать на запись в таблице, которая содержит другое значение перехода. Поэтому для каждого такого направления создается ограничение в виде SMT-формулы, в которой символьный адрес приравнивается к конкретному адресу строки таблицы. Если таких строк несколько, то создается несколько равенств, которые затем объединяются через дизъюнкцию:

```
Branch condition: (sym_addr == address_1) OR (sym_addr == address_4)
```

Полученные ограничения добавляются в предикат пути как условный переход с несколькими направлениями перехода. При инвертировании перехода запросы к SMT-решателю будут создаваться для каждого направления, кроме соответствующего текущему пути исполнения.

Таким образом, инвертирование табличного перехода происходит практически идентично обычным условным переходам. В трассе исполнения программы табличные переходы присутствуют и имеют корректные значения для ветвей, что делает возможным проводить последующую валидацию инвертирования. Табличные переходы также добавляются в кэш ветвей, поэтому их инвертирование производится в том же порядке, что и для остальных ветвей.

2.3.3 Экспериментальная оценка метода

Для оценки разработанного метода моделирования косвенных переходов инструмент Sydr был протестирован на наборе из 14 программ с открытым исходным кодом [65]. Все приложения являются однопоточными и получают входные данные из файла. Программы, выбранные для проведения эксперимента выбраны таким образом, что покрывают большинство типов работы, включая работу с изображениями, архивами, аудиофайлами, исполняемыми файлами, структурированными документами разных форматов. Для тестирования использовался сервер со следующей спецификацией: процессор AMD EPYC 7702 (128 ядер), 256 Гб оперативной памяти.

Анализ в Sydr проводится в два этапа:

1. *Построение предиката пути*. На этом этапе происходит конкретно-символьное исполнение программы и построение символьных ограничений пути. Как только анализируемая программа завершается или

срабатывают ограничения по таймеру/потребляемой памяти, то построение предиката прекращается и анализ переходит на вторую стадию.

2. *Инвертирование условных переходов*. На основе построенного предиката пути для каждого условного перехода составляются запросы к SMT-решателю, которые проверяют возможность инвертирования этих переходов. При положительном ответе SMT-решателя генерируется соответствующий набор входных данных. Поскольку инвертирование различных условных переходов происходит независимо друг от друга, то данный этап может выполняться в многопоточном режиме.

Общее время анализа каждой программы при проведении экспериментов было ограничено двумя часами, при этом на стадию построения предиката пути отводилось не более двадцати минут. Если по достижении двадцати минут конкретно-символьное выполнение программы все еще продолжалось, то программе посылался сигнал завершения работы и запускалась стадия инвертирования условных переходов на том предикате пути, который успел построиться за отведенное время. Инвертирование условных переходов происходило в один поток и в том порядке, в котором они расположены на пути исполнения (прямой порядок инвертирования). Под анализом программы в данном эксперименте подразумевается динамическая символьная интерпретация вдоль одного пути исполнения. Стартовый набор входных данных для каждой программы был подобран вручную таким образом, чтобы соответствовать корректному сценарию работы программы, затрагивая при этом как можно больше кода программы.

Для оценки метода моделирования косвенных переходов запуск анализа проводился дважды: стандартный режим работы инструмента Sydr и редим с включенным методом моделирования косвенных переходов. В таблице 1 приведены результаты эксперимента. В столбце *Application* перечислены приложения, на которых проводился эксперимент. В списке 14 программ, программа `histmap` запускалась дважды с двумя различными входными данными, которые затрагивают разные части программы. Столбцы *default* и *jump_table* содержат значения для стандартного запуска Sydr и для Sydr с инвертированием табличных переходов соответственно. В столбце *Time* приведено общее время анализа приложения. На некоторых приложениях Sydr отработал полностью, в остальных случаях анализ был завершен по таймауту в два часа. Столбец *Branches* показывает число условных переходов в построенном предикате пути программы. В столбце *Queries*

приведено количество запросов к SMT-решателю, которые он успел обработать за отведенное время.

Таблица 1 — Результаты анализа Sydr

Application	Branches		Queries		Time	
	default	jump_table	default	jump_table	default	jump_table
bzip2recover	5131	5131	5131	5131	51m8s	51m3s
cjpeg	197	197	7986	8010	120m	120m
faad	653	652	422272	458145	120m	120m
foo2lava	6119	6127	910725	910725	120m	120m
hdp	3851	3828	67383	67476	120m	120m
histmap_pgm	25410	25446	967187	967187	120m	120m
histmap_ppm	8058	8247	8058	8121	27m	28m52s
jasper	17710	18207	837670	837669	120m	120m
libxml2	16232	17532	53548	53699	120m	120m
minigzip	8977	8977	8977	8977	31m34s	29m42s
muraster	4998	4726	6018	7102	120m	120m
pk2bm	3673	3673	3673	3673	21m30s	21m39s
readelf	5815	6141	64093	64196	120m	120m
yices-smt2	9183	9647	19386	19543	120m	120m
yodl	4795	5201	4795	4831	33m47s	34m59s

Точность проведенного анализа приведена в таблице 2. В столбце *SAT* указано количество запросов SMT-решателю, для которых удалось найти решение. Ответ SMT-решателя на запрос может быть SAT (решение формулы найдено), UNSAT (решения не существует) и UNKNOWN (решение неизвестно). Также обработка запроса может быть прервана по таймауту, который для эксперимента был установлен в 30 секунд. Значение *SAT* показывает сколько условных переходов было инвертировано в процессе анализа. Столбец *Correct* показывает число корректно инвертированных условных переходов.

Таблица 2 — Точность инвертирования переходов

Application	Correct		SAT	
	default	jump_table	default	jump_table
bzip2recover	2101	2101	2101	2101
cjpeg	50	50	50	50
faad	427	426	431	430
foo2lava	27	27	31	31
hdp	815	809	1050	1037
histmap_pgm	17062	17088	17063	17089
histmap_ppm	106	106	107	107
jasper	6572	6766	6604	6798
libxml2	545	545	1085	1069
minigzip	3896	3896	7569	7569
muraster	2652	3227	3861	3228
pk2bm	181	182	183	183
readelf	629	639	727	739
yices-smt2	2056	2114	2596	2699
yodl	159	180	275	313

Считается, что условный переход был инвертирован корректно в том случае, если соответствующий набор входных данных привел программу к исполнению по такому пути, на котором переход был совершен в требуемом направлении. Для проверки корректности сгенерированных входных данных был написан вспомогательный инструмент Tracer, выполненный в виде плагина DynamoRIO. Инструмент Tracer запускает программу на входных данных и записывает трассу исполнения программы T_1 в виде списка условных переходов, которые представлены тройкой $\langle module, source\_offset, destination\_offset \rangle$. Значения $source\_offset$ и $destination\_offset$ являются смещениями соответствующих инструкций (начало и цель условного перехода) от начала исполняего модуля. Для каждого сгенерированного набора входных данных также сохраняется трасса

условных переходов T_2, которая строится на основе предиката пути программы. Последний переход в такой трассе записывается с инвертированным направлением. Трасса переходов T_2 является подмножеством трассы T_1 по нескольким причинам:

– Трасса T_2 не является полной: она обрывается в определенной точке пути исполнения и не отображает трассу после инвертированного перехода, поскольку этот путь исполнения неизвестен на данный момент.
– Трасса T_2 строится на основе предиката пути, который содержит только условные переходы, зависящие от символьных данных. Инструмент Tracer не учитывает символьные данные, вследствие чего в трассу T_1 записываются все условные переходы в программе.

В случае корректно инвертированного условного перехода, трасса T_1 должна полностью содержать T_2. Основным условием корректности является наличие в трассе T_1 последнего перехода из T_2. При этом часть трассы T_2 может не входить в T_1 – такое расхождение путей исполнения допускается, поскольку при конкретном перезапуске программы может различаться запуск различных системных функций, которые не учитываются в символьной интерпретации и в целом не влияют на основной путь исполнения программы. Если проверяемый инвертированный переход не был найден в трассе T_1, либо был найден, но в неправильном направлении, то инвертирование перехода считается некорректным. Корректность инвертирования переходов является одним из важнейших показателей анализа, поскольку отражает конечную цель анализа – открытие новых путей исполнения. На корректность влияет множество факторов, в том числе полнота символьной модели программы, ошибки символьной интерпретации, правильность составления запросов из предиката пути и т.д.

Сравнивая характеристики анализа из двух таблиц, можно оценить пользу от реализации инвертирования табличных переходов в инструменте Sydr. В большинстве приложений благодаря реализованному методу удалось обнаружить новые условные переходы, об этом свидетельствует увеличение значений в столбце *Branches*. Те примеры, на которых табличных переходов не было обнаружено, можно использовать, для оценки дополнительной нагрузки на анализ от реализованного метода и как это отображается на общей эффективности анализа. Время анализа программ, которые уложились в двухчасовой таймаут (`bzip2recover`, `minigzip`, `pk2bm`, `histmap_ppm`, `yodl`) либо незначительно увеличилось, либо не изменилось вовсе. Для остальных программ нагрузку на анализ можно оценить

по числу обработанных запросов к SMT-решателю. Значения столбца *Queries* показывают, что производительность анализа незначительно снизилась только на одном примере (`hdp`). Это является следствием того, что новые условные переходы оказались сложными для SMT-решателя и он потратил на их обработку больше времени, из-за чего в итоге решилось меньше условных переходов за то же время. На всех остальных число обработанных запросов либо увеличилось, либо осталось неизменным.

По числу инвертированных условных переходов и корректности инвертирования можно сделать вывод об общем эффекте от поддержки табличных переходов. В целом, заметный прирост корректно инвертированных переходов наблюдается на шести примерах (`jasper`, `muraster`, `histmap_pgm`, `readelf`, `yices-smt2`, `yodl`). На всех остальных (кроме `hdp`) реализованный метод не повлиял. В приложении `cjpeg` удалось обнаружить новые условные переходы, но ввиду слишком сложных запросов к SMT-решателю на конкретно этом примере, за два часа удается обработать лишь первые 197 запросов. Таким образом, обнаруженные табличные переходы не были обработаны в рамках эксперимента и не повлияли на результат анализа этого примера. В программе `libxml2`, несмотря на новые условные переходы и большее число запросов, новые пути исполнения открыть не удалось. Уменьшение числа SAT свидетельствует о том что, для новых запросов SMT-решатель не смог найти решение. Такое происходит из-за слишком сильных ограничений пути.

Также стоит отметить, что с поддержкой табличных переходов в результирующей таблице число условных переходов перестало быть равным числу обработанных запросов на примерах, которые успели полностью отработать. Это связано с тем, что табличный условный переход считается одним переходом в столбце *Branches*, но для его инвертирования требуется несколько запросов к SMT-решателю (на каждое направление перехода), вследствие чего значения в столбце *Queries* отличаются в большую сторону (`histmap_ppm`, `yodl`). Еще одним интересным параметром анализа может быть соотношение между числом инвертированных и правильно инвертированных условных переходов (столбцы *SAT* и *Correct* соответственно). Так, для программы `muraster` корректность инвертирования переходов увеличилась с 68% до практически 100%.

Таким образом, результаты эксперимента показывают, что реализованный метод моделирования косвенных переходов позволяет находить новые условные переходы в программах и в целом открывать больше новых путей исполнения.

При этом реализованный метод не приводит к заметному снижению производительности.

Глава 3. Моделирование чтений памяти по символьному адресу

3.1 Обработка доступа к памяти

В процессе символьной интерпретации программы достаточно часто возникают ситуации, когда обращение к памяти происходит по адресу, вычисляемому с использованием символьных переменных. Само значение, читаемое или записываемое по такому адресу, зачастую не является символьным, однако тем не менее становится определенным образом зависимым от входных данных. В данной работе такие зависимости называются адресными, базовый алгоритм динамической символьной интерпретации их не поддерживает. Это означает, что при интерпретации доступа к памяти символьный адрес конкретизируется к значению, которое он принимает на текущем пути исполнения программы. Некоторые инструменты поддерживают адресные зависимости лишь частично – путем подбора возможных значений символьного адреса с помощью SMT-решателя. Такой подход приводит к созданию большого числа новых входных данных, большая часть из которых не приводят к открытию новых путей исполнения. Кроме того, адресные зависимости никак не учитываются в символьной модели программы, что ведет к пропуску символьных точек ветвления и потенциальным ошибкам при инвертировании условных переходов ввиду отсутствия части ограничений пути. Конкретизация символьных адресов подразумевает добавление в предикат пути жестких ограничений выражения адреса на текущее значение. В противном случае это может привести к тому, что ограничения для инвертирования последующих условных переходов будут слишком слабыми. В результате SMT-решатель либо подберет решение для невозможного в реальности пути исполнения, либо найденного решение будет неточным, и сгенерированный набор входных данных не будет приводить к инвертированию соответствующего условного перехода в реальном исполнении программы.

Символьные доступы к памяти в бинарном коде как правило соответствуют работе с массивами данных на высокоуровневом представлении программы. Распространенным случаем является доступ к массиву данных по индексу, который вычисляется на основе внешних данных программы. Такой прием называется табличным преобразованием, когда данные трансформируются при помощи

заранее подготовленной таблицы с константными значениями. Табличные преобразования широко используются в функциях конвертации данных, например для изменении регистра символов `tolower()`/`toupper()` или для перевода символов из одной кодировки в другую. Также табличные преобразования используются при вычислении контрольных сумм и в хэш-функциях. В листинге 3.1 приведен пример программы с табличным преобразованием, в котором входные программы функции используются для выбора значения из статического массива `table`. Выбранное значение затем используется в условном переходе, и для того чтобы его инвертировать это значение должно быть представлено символьным выражением, описывающим зависимость между входным параметром a и значениями из массива `table`.

Листинг 3.1 Пример кода с табличным преобразованием

```
1 int table[6] = {3, 7, 14, 0, 5, 11};
2
3 int foo(int a) {
4     int res = table[a];
5     if (res == 5) {
6         abort();
7     }
8     return res;
9 }
```

Для поддержки адресных зависимостей необходимо отдельно реализовать интерпретацию инструкций, осуществляющих доступ к памяти по символьному адресу. При этом инструкции могут выполнять как чтение, так и запись по изменяемому адресу. Разработанный метод моделирования символьных указателей обрабатывает их по-разному:

– Для операций записи символьное выражение адреса всегда конкретизируется. Интерпретация инструкции происходит стандартным способом.
– Для операций чтения строится специальное выражение, устанавливающее взаимосвязь между прочитанным из памяти значением и символьным адресом. Это выражение используется при интерпретации инструкции вместо соответствующего операнда доступа к памяти.

Данный метод не осуществляет поддержку адресных зависимостей для операций записи с целью оптимизации производительности всего анализа. Символьные чтения имеют более высокий приоритет для анализа, поскольку польза

от их обработки проявляется моментально. Значение, прочитанное по символьному адресу, сразу же участвует в дальнейших вычислениях программы, которые в свою очередь также становятся символьными и начинают участвовать в интерпретации. Соответственно, происходит обнаружение и инвертирование новых условных переходов, которые без поддержки адресной зависимости не зависели бы от символьных переменных. Табличные преобразования данных представляют собой именно операции чтения по символьному адресу. При обработке символьных записей складывается иная ситуация. Определенное значение может быть записано в зависимости от символьного адреса в различные ячейки памяти. Интерпретация такой операции может привести к открытию новых переходов только в том случае, если в дальнейшем из этой памяти будет прочитано записанное значение. Интерпретация символьной записи требует создания отдельных ограничений пути для каждого байта всей памяти, к которой возможен доступ по символьному адресу, что может соответствовать довольно большой области памяти. Таким образом, обработка символьной записи может привести к резкому увеличению символьных ограничений, при этом открытие и интерпретация нового кода программы возможна только при определенных условиях.

Листинг 3.2 Табличное преобразование в бинарном коде

```
1    lea    rax, [rip + 0x200899]          ;Base address of table
2    movsxd rdi, edi                       ;User-controlled index
3    mov    eax, DWORD PTR [rax + rdi * 4] ;Table transformation
4    cmp    eax, 0x5
5    je     794 <foo + 0x14>
6    repz ret
```

Косвенные переходы при обращении к таблице переходов также производят чтение по символьному адресу. Если символьное чтение происходит в контексте табличного перехода, то оно отдельно обрабатывается методом, описанным в главе 2. В табличном переходе, полученное значение используется для совершения перехода, тогда как в общем случае это значение используется в дальнейшей работе программы произвольным способом. В примере бинарного кода, приведенного в листинге 3.2, прочитанное значение из памяти используется в операции сравнения `cmp` обычного условного перехода. Для удобства изложения, операции чтения по символьному адресу будут называться символьными указателями, либо просто символьными адресами.

3.2 Определение границ участка памяти

Разработанный метод моделирования чтений по символьно вычисляемому адресу состоит из двух последовательных этапов:

– определение области памяти, из которой производится чтение по символьно вычисляемому адресу;
– символьная интерпретация операции чтения, которая заключается в построении соответствующего предиката пути.

Чтобы составить ограничение пути, описывающее символьное чтение, необходимо определить область памяти, в пределах которой может изменяться символьный адрес. Аналогичная задача возникает при инвертировании табличных условных переходов. Однако в отличие от поиска таблиц переходов, когда в памяти лежат значения строго определенного вида, в данном случае память содержит произвольные и не связанные между собой значения. По этой причине определение границ памяти производится на основе анализа символьного выражения адреса, по которому осуществляется доступ. В рамках метода моделирования символьных чтений было реализовано несколько подходов к определению границ доступа к памяти:

– бинарный поиск границ с использованием SMT-решателя;
– синтаксический анализ символьного выражения адреса;
– выбор фиксированного участка памяти.

Схема взаимодействия между двумя процессами анализа Sydr, так же как и в случае табличных переходов, организована в три этапа (рисунок 2.1). Разница заключается в том, что в данном случае определение границ происходит на стороне символьного интерпретатора. Для каждой выполняемой инструкции конкретный исполнитель отправляет сообщение с данными для ее интерпретации. Если операндом-источником инструкции выступает обращение к памяти, то символьный интерпретатор проверяет, что выражение адреса зависит от символьных переменных, и определяет приблизительную область доступа к памяти. Если выражение адреса не является символьным, то интерпретация инструкции выполняется стандартным способом. Граничные адреса памяти отправляются в конкретный исполнитель, который проверяет, что выбранная область памяти доступна для чтения, и выравнивает границы по размеру символьного чтения.

Соответствующие значения в памяти пересылаются обратно в символьный интерпретатор для дальнейшей обработки.

Определение области памяти при моделировании символьного доступа является важным этапом алгоритма, поскольку корректность и размер выбранных границ доступа сильно влияют на производительность анализа. При моделировании символьных чтений происходит построение соответствующих символьных ограничений предиката пути, размер которых напрямую зависит от выбранной области памяти. Слишком большие размеры формул ведут к увеличению времени обработки в SMT-решателе, а в случае некорректно определенных границ памяти часть ограничений будет соответствовать несуществующим путям исполнения программы. Для того чтобы не создавать чрезмерную нагрузку на SMT-решатель, алгоритм поддерживает моделирование доступа к области памяти, не превышающей определенной величины. Размер области памяти измеряется в элементах доступа, каждый из которых соответствует размерности символьного чтения и может изменяться в пределах от 1 до 64 байт (при этом величина должна являться степенью двойки). Измерение памяти в элементах доступа, а не в абсолютных значениях, позволяет масштабировать максимальный размер области памяти относительно текущего доступа.

По умолчанию область памяти для моделирования ограничена сверху в 200 элементов доступа, однако может изменяться пользователем как в большую, так и в меньшую сторону. Такое ограничение было выбрано, основываясь на применении подхода для анализа прикладных программ реального мира. Чем меньше область памяти выбирается для построения ограничений, тем меньше памяти потребляется при анализа и тем быстрее происходит обработка символьных ограничений. Однако при выборе слишком маленькой области памяти пропускается большинство зависимостей и эффект от поддержки символьных указателей теряется. Выбор 200 элементов памяти для моделирования символьного доступа представляет собой компромисс между производительностью и точностью анализа.

Простейшим способом определения области памяти для интерпретации символьных чтений является выбор участка памяти фиксированного размера. Левая и правая границы участка располагаются на одинаковом удалении от текущего адреса доступа таким образом, чтобы весь участок имел максимально поддерживаемую длину (по умолчанию 200 элементов доступа). Такой способ является самым быстрым и легковесным из всех, но при этом имеет наименьшую точность.

Подход применяется в том случае, когда невозможно применить какой-либо из других способов.

3.2.1 Бинарный поиск с использованием SMT-решателя

Данный способ заключается в определении максимального и минимального значения, которое может принимать символьное выражение адреса в контексте текущего пути исполнения. Соответственно диапазон памяти, чтение из которого необходимо смоделировать, заключается между этими двумя значениями. Граничные значения адреса определяются методом бинарного поиска с помощью запросов к SMT-решателю. Поиск максимального (минимального) значения также может быть произведен с помощью последовательных запросов к SMT-решателю, которые подбирают значение символьного адреса большее (меньшее), чем полученное на предыдущей итерации. Такой подход используется в QSYM [45] для перебора значений символьного адреса. Однако он не эффективен для определения граничных значений, поскольку требует большего количества запросов к SMT-решателю, что в рамках динамического анализа является дорогостоящей операцией. Определение границ выполняется в два этапа – поиск максимального и минимального значений адреса выполняется по отдельности.

Для поиска точного значения методом бинарного поиска необходимо определить общий диапазон возможных значений, принимаемых символьным адресом. Изначально известно только конкретное значение символьного адреса, соответствующее текущему пути исполнения программы. В общем случае, символьный адрес может принимать значения от конкретного значения до границы адресного пространства. Однако имеющееся ограничение максимально поддерживаемого размера памяти позволяет значительно сократить диапазон поиска и уменьшить число итераций бинарного поиска. Поскольку общий размер памяти ограничен в 200 ячеек доступа, то для удобства считается, что в каждую сторону от текущего адреса доступ может быть осуществлен не далее чем на 100 ячеек. Бинарный поиск точной левой и правой границы памяти осуществляется над диапазоном памяти в 100 ячеек доступа в соответствующую сторону от текущего адреса доступа. Точка разбиения диапазона памяти на каждой итерации вычисляется таким образом, чтобы быть кратной размеру доступа к памяти. В ре-

зультате финальная искомая граница памяти получается выровненной по размеру символьного чтения. Алгоритм бинарного поиска в худшем случае требует $log_2 N$ итераций, а поскольку в данном подходе диапазон памяти равен 100 ячейкам доступа (независимо от их размера), то к SMT-решателю требуется произвести не более семи запросов для определения каждой границы.

Запросы к SMT-решателю формируются из специально составляемых уравнений с выражением символьного адреса и всего предиката пути, построенного на текущий момент исполнения программы. Ограничения предиката пути позволяют максимально точно определить значения символьного адреса, поскольку в таком случае учитывается контекст пути исполнения программы. Однако в некоторых случаях такой подход ведет к чрезмерным ограничениям (overconstrained) символьного выражения адреса. В результате символьный адрес ограничен настолько, что может принимать только одно единственное значение – свое конкретное значение текущего пути исполнения. SMT-решатель в данном случае не сможет подобрать альтернативные значения адреса, и в результате бинарного поиска границы памяти сойдутся к одной точке. Поэтому, чтобы избежать выполнения множества бесполезных запросов к SMT-решателю, вначале производится проверка на ограниченность символьного адреса, которая имеет следующий логический вид:

$$PathPredicate \wedge (sym\_addr \neq concrete\_addr), \tag{3.1}$$

где $PathPredicate$ – ограничения предиката пути, sym_addr – символьное выражение адреса, $concrete\_addr$ – конкретное значение адреса доступа. Если решатель посчитал такую формулу несовместной (`UNSAT`), то символьный адрес не может принимать другие значения, и определение границ с помощью SMT-решателя в данном случае невозможно. Применение оптимистичных решений, при которых предикат пути не участвует в запросе, может помочь преодолеть проблему чрезмерных ограничений. Однако в этом случае границы области доступа определяются неточно, поскольку единственным ограничением адреса становится само его символьное выражение. Поэтому при чрезмерных ограничениях символьного адреса применяется другой способ поиска границ доступа. Если такой проблемы нет, и решатель посчитал формулу разрешимой (`SAT`), то граничные значения по очереди определяются с помощью бинарного поиска.

На каждой итерации бинарного поиска к SMT-решателю составляются запросы, имеющие следующий вид:

$$PathPredicate \wedge (sym\_addr > pivot\_addr) \tag{3.2}$$

для поиска правой границы доступа к памяти и

$$PathPredicate \wedge (sym\_addr < pivot\_addr) \tag{3.3}$$

для поиска левой границы. Значение $pivot\_addr$ является абсолютным адресом в памяти, определяющим диапазон поиска, и пересчитывается на каждой итерации. Для удобства все следующие формулы приведены для поиска правой границы, вычисление левой границы производится аналогично. Поиск значений производится на отрезке $[concrete\_addr, concrete\_addr + 100 \times size]$, где $size$ – размер символьного чтения. На первой итерации $pivot\_addr$ равняется максимальному значению отрезка. Если SMT-решатель смог найти решение для этой формулы, то правая граница находится дальше, чем выбранное ограничение. В таком случае границей выбирается конец рассматриваемого отрезка. В противном случае рассматриваемый отрезок разбивается на две части и бинарный поиск продолжатся. На каждой следующей итерации значение $pivot\_addr$ выбирается таким образом, чтобы разделять рассматриваемый диапазон на две равные части. Поскольку граница памяти должна быть выровнена по размеру символьного чтения $size$, то разбиение отрезка происходит в соответствие с ячейками доступа. Бинарный поиск продолжается до тех пор, пока отрезок не сойдется в одну ячейку памяти. Таким образом будет определена точная граница доступа к памяти.

Данный подход позволяет достаточно точно определить область памяти для моделирования символьного чтения, однако он обладает рядом недостатков. Применение SMT-решателя невозможно в том случае, когда построенный предикат пути накладывает слишком сильные ограничения на выражение символьного адреса. В таком случае символьное чтение может быть либо конкретизировано, либо интерпретировано с границами, найденными другим методом, и быть полезным для оптимистичных решений в дальнейшем. Также метод бинарного поиска требует множества дополнительных запросов к SMT-решателю во время анализа. Принимая во внимание большое число символьных чтений при интерпретации программы и размеры предиката пути, использование этого подхода оборачивается значительным падением производительности анализа, практически сводя на нет все преимущества точного определения границ. Данный подход все же может

быть применен при анализе небольших программ с малым числом символьных чтений, но при анализе которых необходима высокая точность определения границ. В таком случае и размер максимально поддерживаемой области памяти при обработке символьных указателей может быть скорректирован соответствующим образом.

3.2.2 Синтаксический анализ символьного выражения

Альтернативным подходом определения границ памяти является анализ символьного выражения адреса. Данный подход основывается на предположении о том, что адрес доступа лишь частично является символьным. Как правило, от символьных переменных зависит только небольшая часть адреса, которая определяет изменяемое смещение относительно определенного базового адреса. Базовый адрес указывает на начало определенной сущности в памяти программы (таблицы переходов, массива и т.д.), к которой выполняется символьный доступ. Соответственно и изменение символьного адреса, определяемое зависимым от входных данных смещением, должно находиться в пределах этой сущности. В большинстве случаев базовый адрес представляется константой и является либо минимально возможным значением символьного адреса, либо достаточно близок к нему. Определив константную часть символьного выражения адреса, алгоритм принимает ее как левую границу области памяти, к которой производится доступ. Правая граница при этом определяется исходя из ограничений на максимально поддерживаемый размер области памяти – то есть в 200 ячейках доступа от левой границы.

Для определения константной части адреса алгоритм выполняет рекурсивный обход SMT-формулы, представляющей символьный адрес. Во время этого обхода собираются все константные составляющие, удовлетворяющие определенным условиям. Сложив все обнаруженные таким образом значения, можно получить предполагаемую левую границу доступа. Алгоритм начинает работу с анализа общей формулы символьного адреса и рекурсивно обходит ее AST (Abstract Syntax Tree, абстрактное синтаксическое дерево). Сам алгоритм состоит из следующих основных шагов:

– Если текущее выражение является символьной переменной, то оно разыменовывается и алгоритм рекурсивно применяется к тому выражению, на которое эта переменная ссылается.
– Если рассматриваемое выражение не зависит от символьных переменных, оно конкретизируется, а ее константное значение запоминается.
– Если выражение является операцией сложения или вычитания (операторы `bvadd`/`bvsubb`), то их операнды также рекурсивно анализируются алгоритмом. Информация об операции сохраняется в виде знака, с которым будут складываться обнаруженные константные значения.
– Отдельно обрабатываются операции сложения/вычитания со стековыми регистрами (`ebp`, `esp`) и константными смещениями. Если выражение вычисляет адрес относительно стековых регистров и имеет вид, например, `[epb – 0x8]` или `[0x4 + esp]`, то такое выражение конкретизируется и запоминается как константа.
– Если выражение является символьным, но при этом представляет собой формулу вычисления программного счетчика на символьной ветви (`ite`-оператор), то такое выражение конкретизируется, полученное значение адреса запоминается как константа.
– Во всех остальных случаях выражения определяются как символьные и игнорируются.

По завершению обхода символьной формулы все обнаруженные константные значения суммируются. При этом учитывается знак каждой константы, отражающий соответствующую операцию (сложение или вычитание).

Также возможен случай, когда одним из подвыражений символьного адреса является формула, полученная в результате интерпретации другого символьного чтения. Такая формула имеет вид вложенных `ite`-выражений и имеет специальную метку. Вместо конкретизации этой формулы для текущего пути исполнения, алгоритм совершает обход всех `ite`-выражений и собирает все возможные константные значения, которые могут быть результатом вычисления данной формулы. При суммировании всех констант во время вычисления левой границы из всех этих значений выбирается такое, чтобы левая граница имела наименьшее значение.

Рассмотрим работу алгоритма на примере символьного адреса на листинге 3.3. На первой строке листинга приведен пример инструкции, осуществляющей чтение по символьному адресу. Символьное выражение этого адреса приведено

Листинг 3.3 Анализ символьного выражения адреса

```
1 1733f movsxd rax, dword ptr [{rdx} + rax*4]
2 AST = (bvadd ref!1519 (bvmul ref!1506 (_ bv4 64)))
3     ref!1519 -> (bvadd (_ bv895833 64) (bvadd ref!1518 (_ bv7 64)))
4     ref!1518 -> (ite (= ref!1516 (_ bv1 1))
5                      (_ bv140737283085843 64)
6                      (_ bv140737283085112 64))
7 base address = 895833 + 140737283085843 + 7 = 0x7ffff3d18173
```

на строке 2. На первой итерации алгоритм анализирует все выражение адреса, которое состоит из операции сложения и двух операндов, которые добавляются в очередь на рекурсивный обход. Один из операндов представляет символьное выражение, состоящее из операции умножения (`bvmul`). В соответствии с алгоритмом, такое выражение целиком воспринимается как символьная часть адреса, поэтому в дальнейшем разборе AST'а не участвует. Другим операндом является символьная переменная `ref!1519`, которая ссылается на выражение на строке 3. Оно, в свою очередь, также представляет собой операцию сложения двух операндов, один из которых является константным значением, а другой – очередным символьным выражением. Константное значение запоминается для последующего вычисления, а выражение анализируется на новой итерации алгоритма. Оно также представляет собой сложение символьной переменной `ref!1518` и константы `0x7`. Символьная переменная разыменовывается на строке 4: она ссылается на `ite`-выражение с вычислением программного счетчика. Это выражение является символьным, поскольку условие `ite`-оператора зависит от символьной переменной, однако оно вычисляется в один из двух константных адресов, лежащих в ветвях `then`/`else` выражения. Алгоритм вычисляет данное выражение, чтобы получить нужное константное выражение в рамках текущего пути исполнения. На этом обход AST'а формулы завершается. Все найденные константы (выделены на листинге цветом) складываются и образуют предполагаемую левую границу объекта в памяти, к которому производится доступ по символьному адресу.

Данный подход позволяет достаточно эффективно определять приблизительные границы области доступа к памяти. Анализ символьного выражения адреса производится быстро и не создает большой нагрузки на анализ. Левая граница объекта в памяти определяется с высокой точностью: в случае ошибки погрешность может составлять всего несколько ячеек доступа. Основным

недостатком подхода является отсутствие каких-либо предположений о правой границе участка памяти. Метод анализа синтаксического дерева обладает гораздо большей производительностью, чем метод бинарного поиска с помощью SMT-решателей. В то же время он обеспечивает более высокую точность, по сравнению с обычным выбором участка памяти фиксированного размера. Исходя из этого, при интерпретации символьных указателей в ходе анализа применяется именно этот подход.

3.3 Моделирование символьного чтения

Следующим шагом после определения участка памяти, из которого производится чтение, является конструирование соответствующего ограничения пути. Оно должно описывать результат чтения из этой памяти в зависимости от символьного выражения адреса. В рамках работы было реализовано три способа построения таких ограничений:

– вложенные `if-then-else` деревья (ITE-деревья);
– двоичные деревья поиска (BST-деревья);
– линеаризованные двоичные деревья поиска.

Полученная формула используется при интерпретации самой инструкции, осуществляющей символьное чтение, в качестве выражения для операнда доступа к памяти. В результате факт доступа к памяти по адресу, зависящему от входных данных, учитывается в символьной модели исполнения программы, даже если в текущей точке это не влияет на путь исполнения программы.

3.3.1 Вложенные ITE-деревья

Построение вложенных ITE-деревьев является самым простым способом конструирования ограничений при интерпретации символьных указателей. Вся область памяти, к которой производится доступ по символьному адресу, представляется в виде пар $\langle aN; value\_N \rangle$, где aN это адрес, а $value\_N$ – значение,

расположенное по этому адресу в памяти. Адреса при этом идут подряд с определенным шагом, равным размеру символьного доступа.

Листинг 3.4 Вложенное ITE-дерево

```
if (sym == a1) {
    value_1
} else {
    if (sym == a2 || sym == a3) {
        value_2
    } else {
        if (sym == a4) {
            value_4
        } else {
            current_value
        }
    }
}
```

Используя эти значения, составляется специальная формула, которая описывает результат чтения в зависимости от символьного адреса. Логическое представление такой формулы изображено на листинге 3.4. С помощью оператора языка SMT-LIBv2 `if-then-else` (`ite`-оператор) производится перебор всех возможных значений из заданной области памяти. Оператор `if-then-else` имеет вид $(ite(cond)(expr1)(expr2))$, на листинге 3.4 для большей наглядности он представлен как $if(condition)then(expr1)else(expr2)$. Такое выражение позволяет формуле принимать одно из двух значений в зависимости от выполнения условия $conditon$. При построении формулы последовательно обходятся все пары адрес-значение, для каждой такой пары составляется свое `ite`-выражение. Условием в таких выражениях выступает равенство символьного адреса конкретному значению aN, в `then`-ветви лежит соответствующее значение памяти $value\_N$. Для `else`-ветви строится `ite`-выражение для следующей пары адрес-значение. Таким образом, через ветвь else конструируется вложенное дерево из `ite`-выражений. Последняя ветвь else в таком дереве соответствует случаю, когда символьный адрес принимает значение за пределами выбранной области памяти. Если границы доступа к памяти были выбраны корректно, то SMT-решатель не сможет подобрать такое значение символьного адреса, чтобы выражение в последней `else` -ветви стало результатом чтения. Однако границы могут быть выбраны не совсем

точно, и символьный адрес действительно может принимать значения за пределами поддерживаемой области памяти. Поэтому в последней `else`-ветви лежит значение, расположенное по текущему адресу доступа (`current_value`).

Символьные ограничения в виде вложенного ITE-дерева довольно просты в составлении, однако такие формулы имеют большой размер, прямо пропорциональный размеру выбранной области памяти. Объединение нескольких `ite`-выражений в одно помогает сократить итоговый размер формулы. Если несколько ячеек памяти содержат одинаковые значения, то соответствующие пары адрес-значение могут быть описаны одним `ite`-выражением. Для этого перед построением всей формулы производится переупорядочивание пар $\langle aN; value\_N \rangle$ таким образом, что каждому уникальному значению $value\_N$ ставится в соответствие список адресов, по которым это значение расположено в памяти. В листинге 3.4 такому случаю соответствует значение $value\_2$, которое расположено в двух ячейках памяти $a2$ и $a3$. Условным выражением в данной ситуации является равенство символьного адреса этим ячейкам памяти, объединенным через дизъюнкцию.

3.3.2 Двоичные деревья поиска

Другим способом интерпретации символьных чтений является построение двоичного дерева поиска, или BST-дерева (Binary Search Tree). В данном случае вся область памяти также представляется в виде множества пар адрес-значение, упорядоченных по значению адреса. Затем с помощью `ite`-выражений строится дерево, осуществляющее двоичный поиск по памяти. Пример организации итоговой формулы представлен на листинге 3.5.

При построении дерева на каждой итерации все пространство адресов делится на два отрезка. Операторы `ite` используются для сравнения символьного адреса с серединой текущего отрезка из пространства адресов. В обеих ветвях `ite`-оператора располагаются вложенные `ite`-выражения для меньших отрезков. Когда размеры отрезков сходятся в один адрес, то then/else ветви соответствуют листьям двоичного дерева и содержат значения памяти. На листинге 3.5 выражение символьного адреса sym последовательно сопоставляется с различными точками на отрезке адресов 0x100-0x400 (значения выбраны для удобства). Ана-

Листинг 3.5 Двоичное дерево поиска

```
if (sym < 0x300) {
    if (sym < 0x100) {
        current_value
    }
    else {
        if (sym == 0x100) {
            value_1
        }
        else {
            value_2
        }
    }
} else {
    if (sym < 0x400) {
        value_3
    }
    else {
        if (sym == 0x400) {
            value_4
        }
        else {
            current_value
        }
    }
}
```

логично ITE-дереву, для случаев, когда символьный адрес выходит за границы указанного диапазона, листьями BST-дерева является значение памяти, соответствующее прочитанному значению на текущем пути исполнения.

BST-дерево имеет меньшую вложенность, по сравнению с ITE-деревом, что положительно сказывается на потребляемой памяти и скорости обработки в SMT-решателе.

3.3.3 Линеаризованные BST-деревья

Альтернативным способом построения ограничений, описывающих чтение по символьному адресу, является двоичное дерево поиска, построенное над линейными выражениями. Основная идея заключается в том, чтобы оптимизировать стандартное BST-дерево, уменьшив число его листьев. Символьные чтения зачастую соответствуют табличным преобразованиям в программе, когда данные трансформируются в соответствии с определенной константной таблицей в памяти. В работе [41] отмечается, что во многих случаях содержимое таких таблиц имеет линейную зависимость от позиции в таблице. Это означает, что взаимосвязь между содержимым ячейки памяти и ее индексом – смещением относительно начала таблицы – можно описать линейным уравнением. Таким образом, доступ к различным ячейкам памяти может быть смоделирован всего одним `ite`-выражением, что значительно сокращает общее число листьев и уменьшает размер двоичного дерева. Полученная в итоге такой оптимизации формула получила название линеаризованное BST-дерево, ее логическая организация представлена на листинге 3.6.

Листинг 3.6 Линеаризованное BST-дерево

```
if (sym < 24) {
    if (sym < 16) {
        2 * sym + 1
    }
    else {
        -3 * sym + 61
    }
}
else {
    if (sym < 36) {
        17
    }
    else {
        current_value
    }
}
```

Алгоритм построения линеаризованного BST-дерева состоит из следующих шагов:

1. Выражение символьного адреса преобразуется в символьный индекс.
2. Все адреса в рассматриваемой области памяти пересчитываются в индексы. Соответственно область памяти представляется в виде набора точек на плоскости (индекс $\times$ значение памяти).
3. Последовательно расположенные точки объединяются с помощью линий.
4. Над полученным набором линий и точек строится двоичное дерево поиска.

Поскольку целью линеаризации является построение линейных уравнений, то для удобства все адреса преобразуются в индексы. В реализованном в рамках данной работы методе индекс ячейки доступа – это смещение в байтах от начала объекта в памяти. Для этого адреса всех пар адрес-значение из рассматриваемой области памяти $\langle aN; value\_N \rangle$ пересчитываются относительно левой границы: $idx\_N = aN - a0$. Таким образом, индексы соседних элементов отличаются на размер символьного чтения $size$: $idx\_N = N \times size$. Для символьного адреса также составляется выражение преобразование в символьный индекс.

Полученный набор пар индекс-значение представляется в виде точек на плоскости. Каждая такая точка – это отдельный лист в итоговом BST-дереве. Чтобы сократить число листьев, необходимо объединить точки одной прямой линией – тогда несколько значений памяти могут быть описаны одной линейной формулой. Однако в данном случае нельзя применить подход, при котором все точки объединялись бы наименьшим числом прямых. Поскольку конечной целью является построение двоичного дерева поиска, то все прямые должны быть упорядочены относительно индексов. Другими словами, любые две линии должны проходить через точки так, чтобы соответствующие им индексы образовывали непересекающиеся отрезки. Например, нельзя проводить две линии, объединяющие индексы $(0, 4, 16)$ и $(8, 12)$ соответственно, поскольку над такими линиями нельзя построить BST-дерево с поиском по символьному индексу. Еще одним требованием к линиям является то, что все коэффициенты в уравнениях прямых должны быть целочисленными. Данное ограничение обусловлено языком построения формул SMT-LIBv2 и применяемой в динамическом анализе теорией битовых векторов.

Поскольку не каждые две соседние точки могут быть объединены одной прямой линией, удовлетворяющей заданным ограничениям, то в результате получается набор из линий и отдельных точек. Этот набор упорядочивается в соответствии с индексами, к которым эти линии и точки относятся. Затем, так же как в предыдущем методе, происходит построение BST-дерева. Отличие заключается в том, что вместо пространства адресов используются индексы, причем лини представлены только одним индексом из нескольких. Если листом в таком дереве является линия, то соответствующая ветвь в `ite`-выражении представлена линейной формулой, зависящей от символьного индекса. Аналогично другим методам, для индексов, выходящих за рамки выбранной области памяти, используется значение памяти, прочитанное на текущем пути исполнения.

Получившаяся в итоге формула, так же как и стандартное BST-дерево, имеет небольшую вложенность, но при этом гораздо меньшие размеры. В рамках исследования в инструменте Sydr были реализованы все три описанных метода конструирования ограничений.

3.4 Комбинированный метод моделирования символьных чтений

Построение линеаризованых BST-деревьев является наиболее перспективным способом моделирования доступов к памяти по символьному адресу, поскольку позволяет создавать небольшие по размерам ограничения пути, которые не будут перегружать SMT-решатель. Однако при реализации построения линеаризованных BST-деревьев выявились следующие недостатки такого подхода:

– Если область памяти, к которой производится доступ по символьному адресу, содержит символьные значения, то они не будут учитываться в итоговом выражении.
– Большие размеры доступа к памяти обуславливают слишком большие абсолютные значения коэффициентов в линейных уравнениях, что приводит к увеличенному потреблению памяти при конструировании предиката пути.

Для устранения перечисленных недостатков был разработан ряд улучшений конструирования ограничений пути при обработке символьных чтений.

Улучшенный метод получил название комбинированный метод моделирования символьных чтений. Помимо устранения вышеперечисленных недостатков, разработанный подход также включает в себя дополнительные улучшения, направленные на оптимизацию и повышение эффективности алгоритма при его применении в динамическом анализе.

3.4.1 Поддержка символьной памяти

Одним из недостатков оригинального подхода построения линеаризованных BST-деревьев является отсутствие поддержки символьных переменных в памяти, к которой моделируется символьный доступ. Для объединения различных ячеек памяти с помощью линейных выражений могут быть использованы только конкретные значения памяти. Чаще всего символьные чтения в программах производятся из константного статического массива данных, вследствие чего содержимое такой памяти в символьной модели программы всегда является конкретным. Но иногда возникают ситуации, когда доступ по символьному адресу производится к массиву или буферу данных, который сам зависит от входных данных. В случае, когда содержимое части ячеек памяти зависит от символьных переменных, построение дерева стандартным способом может быть реализовано двумя способами:

– Конкретизация символьной памяти при моделировании символьного чтения. На этапе объединения различных точек (индекс; значение памяти) для всех символьных ячеек памяти берутся конкретные значения из памяти для текущего пути исполнения. В самой символьной модели программ эти значения остаются символьными, таким образом это не оказывает влияния на символьную интерпретацию остальных инструкций программы. Построение линеаризованного BST-дерева выполняется обычным способом, однако такое моделирование символьного чтения не учитывает зависимость части прочитанных значений от входных данных. В некоторых случаях это может приводить к генерации некорректных входных данных.
– Поддержка символьных ячеек памяти в виде отдельных листьев линеаризованного BST-дерева. В оригинальном алгоритме те точки (индекс;

значение памяти), которые не удалось объединить с помощью линейных выражений, в конечном BST-дереве остаются в качестве отдельных листьев. Таким образом, для индексов, которым соответствует символьное значение памяти, можно не выполнять объединение с другими точками, а оставить их в качестве независимых точек. Итоговое дерево двоичного поиска в таком случае будет моделировать доступ к памяти полностью – с учетом и символьного адреса доступа, и символьного содержимого памяти. Однако из-за наличия множества отдельных листьев такое символьное выражение будет иметь большие размеры, что сводит на нет пользу от линеаризации.

В ходе исследования был разработан еще один способ моделирования таких доступов к памяти, который позволяет учитывать символьные ячейки памяти и при этом сохранять все преимущества линеаризованных BST-деревьев. Суть этого метода заключается в раздельном моделировании чтения из конкретных и символьных участков памяти. Для символьных ячеек памяти производится построение вложенного ITE-дерева. Для оставшихся конкретных ячеек памяти происходит построение линеаризованного BST-дерева обычным способом. Оба выражения объединяются в одну формулу через последнюю `else`-ветвь ITE-дерева. Логическая схема итогового выражения представлена на листинге 3.7.

Главным преимуществом такого подхода является то, что линеаризация выполняется над ячейками памяти, которые не разделены отдельными символьными значениями. Как следствие, больше значений могут быть объединены с помощью линейных выражений. Индексы, соответствующие символьным значениям памяти, не учитываются при построении двоичного дерева поиска, поскольку оно является последней ветвью основного вложенного ITE-дерева, которое уже описывает доступ к соответствующим ячейкам памяти. Итоговое выражение имеет сравнительно небольшие размеры и достаточно эффективно позволяет моделировать символьное чтение из символьной памяти.

Еще одним ограничением при построении формулы является размер доступа к памяти. В x86-64 инструкциях размеры чтения варьируются от 1 до 64 байт. Пропорционально размеру ячеек памяти увеличиваются и конкретные значения памяти. Это приводит к тому, что на этапе линеаризации вычисляются очень большие коэффициенты линейных выражений, которые увеличивают потребление памяти при конструировании предиката пути. Само символьное выражение также сильно увеличивается в размерах, поскольку требует дополнительного при-

Листинг 3.7 Комбинированное линеаризованное BST-дерево с поддержкой символьной памяти

```
if (sym == a2) {
    SymExpr_2
} else {
    if (sym == a4) {
        SymExpr_4
    } else {
        if (sym < a6) {
            if (sym < a5) {
                -5 * sym + 16
            }
            else {
                value_5
            }
        }
        else {
            if (sym < a8) {
                8 * sym + 2
            }
            else {
                current_value
            }
        }
    }
}
```

ведения к одному размеру битовых векторов, соответствующих значениям памяти и индекса. В результате процесса линеаризации большая часть листьев BST-дерева представляется в виде $M = \alpha \times idx + \beta$, где α и β – коэффициенты линейного уравнения, idx – символьное выражение индекса, M – конкретное значение в ячейке памяти, вычисляемое в зависимости от ее индекса. Все описанные переменные в этом выражении являются битовыми векторами, имеющими определенную размерность. Размерность битового вектора значения памяти M может принимать любое значение по степени двойки от 1 до 64 байт включительно. Размерность символьного индекса idx совпадает с размерностью символьного выражения адреса и может равняться либо 4 байтам (для 32-битных программ), либо 8 байтам (для 64-битных программ). Размерность битовых векторов A и B по правилам построения SMT-формул должна равняться размерности idx, поскольку

они вместе участвуют в арифметических операциях. Если размеры доступа к памяти совпадают с размерностью адреса, то в SMT-формуле линейного выражения все переменные имеют одинаковую размерность. В программах в большинстве случаев символьный доступ осуществляется к ячейкам памяти небольшого размера от 1 до 8 байт. В этом случае вычисления происходят в размерности индекса idx, а результат всего линейного выражения обрезается до размерности памяти с помощью операции `extract`.

Иначе обрабатываются ситуации, когда размер доступа к памяти больше, чем размерность адреса. Чтобы с помощью линейного выражения можно было вычислить значение памяти M, битовые вектора α, β и idx должны иметь соответствующую размерность, поэтому idx в SMT-формуле расширяется до нужного размера с помощью операции `extend`. Эта дополнительная операция применяется во всех составляемых линейных выражениях, в результате чего итоговое BST-дерево усложняется. Чтобы избежать лишнего усложнения и увеличения размеров SMT-формулы, для моделирования символьных чтений длиной больше 8 байт используется обычное вложенное ITE-дерево. Таким образом, итоговая формула не будет содержать лишних преобразований и битовых векторов с очень большими абсолютными значениями.

3.4.2 Другие оптимизации метода

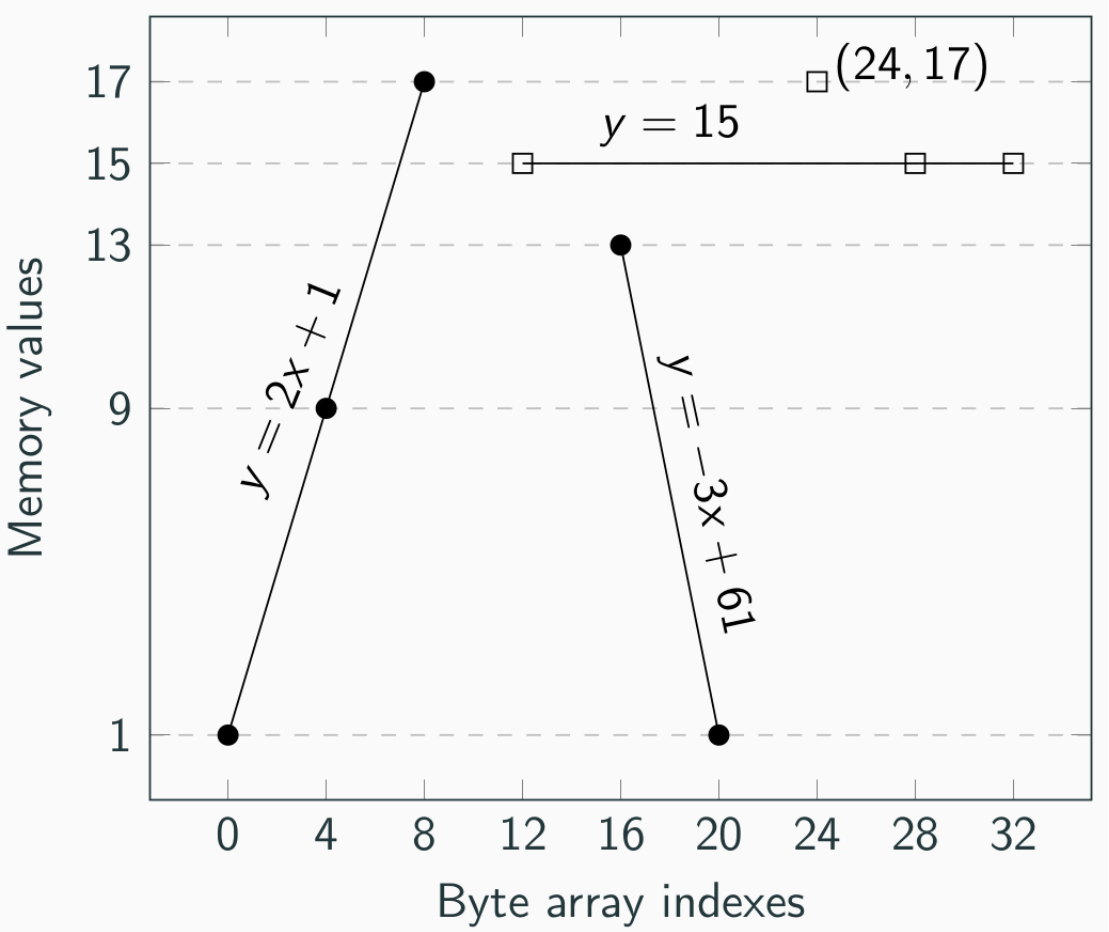
Помимо вышеописанных изменений моделирования символьных чтений, также были разработаны дополнительные улучшения подхода, направленные на оптимизацию итогового выражения и метода построения формул. Самой простой оптимизацией является обработка символьных чтений из небольшого участка памяти. Иногда границы области памяти определяются так, что в ней содержится небольшое число элементов доступа. В таком случае проводить сложный процесс линеаризации и построения двоичного дерева поиска не оправдывается с точки зрения эффективности итогового символьного выражения. Получившееся линеаризованное BST-дерево не будет давать заметного улучшения производительности, поэтому для символьных чтений из области памяти, содержащей менее чем 10 элементов, выполняется построение обычного вложенного ITE-дерева.

Другим улучшением является поиск горизонтальных линий на этапе линеаризации. Линейное выражение для горизонтальных линий представляет собой простое константное значение, поэтому чем больше таких линий обнаружится в процессе линеаризации, тем проще получится итоговое дерево. Горизонтальные линии можно провести, когда несколько ячеек памяти содержат одинаковое значение. При анализе реальных программ такое может встречаться – в том числе и из-за неточного определения границ доступа, когда память за пределами реального массива данных заполнена нулями или другими одинаковыми значениями.

Перед началом построения линий над множеством точек в памяти, производится анализ значений памяти с целью определить число ячеек памяти, которые содержат одинаковые значения. Если число таких ячеек больше трех, то они объединяются в одну горизонтальную линию, независимо от их взаимного расположения. То есть такие ячейки памяти необязательно должны быть расположены рядом друг с другом, они могут находиться в разных частях области памяти. Поскольку основным требованием линеаризации является смежность всех объединяемых в одну линию точек (для возможности построения двоичного дерева поиска в дальнейшем), то такие горизонтальные линии не могут быть использованы в самом BST-дереве. Поэтому в данном случае для них конструируется вложенное ITE-дерево, которое затем объединяется вместе с линеаризованным BST-деревом, построенном над оставшимися точками. Таким образом, здесь применяется подход, аналогичный тому, с помощью которого обрабатываются символьные ячейки памяти.

Стоит отметить, что при добавлении горизонтальной линии в виде ITE-дерева в начале общей SMT-формулы требует явного перечисления соответствующих символьных индексов. В то время как при линеаризации такие горизонтальные линии будут просто представлены одним битовым вектором в соответствующем листе BST-дерева. Поэтому при поиске горизонтальных линий для добавления в начало выражения выбираются только те линии, которые состоят из трех и более точек. Если горизонтальная линия состоит из двух точек, то с большой вероятностью она будет обработана в процессе линеаризации и учтена в итоговой SMT-формуле более эффективным способом.

Рисунок 3.1 — Представление участка памяти в виде точек на плоскости при линеаризации



Алгоритм 1: Алгоритм линеаризации

Входные данные: $Memory$ — память в виде набора точек (индекс, значение), упорядоченного по индексу.

Результат: $points$ — список обособленных точек, $lines$ — упорядоченный список линий, группирующих две или более стоящих подряд точек.

```
points ← ∅
lines ← ∅
p ← Memory.begin()
while p ≠ Memory.end() do
    cur_point ← p
    next_point ← p.next()
    if next_point = Memory.end() then
        /* Текущая точка - последняя в списке.                    */
        points.insert(cur_point)
        break
    end
    /* Построение линии между двумя точками. Функция
       возвращает None, если корректной линии не существует.
       */
    line ← BuildLine(cur_point, next_point)
    if line = None then
        points.insert(cur_point)
    else
        lines.insert(line)
        while p ≠ Memory.end() do
            /* Пропуск всех последующих точек, которые лежат на
               данной линии.                                      */
            if p ∉ line then
                break
            end
            p.next()
        end
    end
end
```

Общий алгоритм построения линеаризованного BST-дерева выглядит следующим образом:

1. Из всего участка памяти отбираются символьные ячейки, для которых конструируется вложенное ITE-дерево. Далее работа происходит только с оставшимися конкретными ячейками памяти.
2. Затем выполняется поиск горизонтальных линий и построение для них ITE-дерева. Это дерево добавляется в качестве последней `else`-ветви предыдущего выражения, если оно есть.
3. Производится процесс линеаризации над оставшимися ячейками памяти, которые представляются в виде точек $\langle M; idx \rangle$ на плоскости, где ось ординат представляет значение памяти, а ось абсцисс – индекс ячеек памяти (рисунок 3.1). Находящиеся рядом точки объединяются через линейные выражения в соответствии с условиями, описанными в главе 3.3.3.
4. Над полученным множеством линий и независимых точек строится двоичное дерево поиска, которое добавляется в последнюю `else`-ветвь предыдущего ITE-дерева, если оно есть.

Сам процесс линеаризации представлен в алгоритме 1. Производится обход всех ячеек памяти, которые упорядочены по возрастанию индекса. Для каждой точки проверяется возможность провести через нее и одну соседнюю точку корректную прямую линию, то есть линию, имеющую целочисленные коэффициенты. Если такой линии не существует, то текущая точка остается независимой, и алгоритм переходит к следующей. Если прямую провести можно, то составляется соответствующее линейное выражение, и для следующих по порядку ячеек памяти проверяется их принадлежность к данной линии. Все ячейки памяти, которые лежат на этой прямой, объединяются с помощью соотвтетсвующего линейного выражения, и алгоритм переходит к следующей точке, которая н лежит на построенной прямой.

Таким образом, комбинированный метод интерпретации символьных указателей использует вложенные ITE-деревья и линеаризованые двоичные деревья поиска для моделирования символьных чтений. Символьные ограничения, конструируемые разработанным методом, имееют сравнительно небольшую вложеность и обладают возможностью моделировать доступ к символьной памяти. Также был реализован ряд улучшений, направленный на уменьшение потреб-

ления памяти при построении предиката пути и повышение эффективности обработки формул в SMT-решателе.

3.5 Экспериментальная оценка метода

3.5.1 Исследование эффективности обработки разных типов ограничений в SMT-решателях

Для выбора оптимального способа построения символьных ограничений при интерпретации символьных указателей, была проведена оценка производительности SMT-решателей при обработке формул, генерируемых в ходе анализа инструментом Sydr. Для этого в инструменте Sydr были реализованы три способа построения ограничений пути: вложенные ITE-деревья, двоичные деревья поиска (BST-деревья) и линеаризованные BST-деревья. Также процесс символьной интерпретации в Sydr был изменен таким образом, чтобы в ходе анализа при обработке чтений по символьному адресу ограничения пути одновременно создавались все три типа ограничений. Запросы к SMT-решателю, генерируемые при инвертировании условных переходов, были сохранены на диск в формате SMT-LIB v2 в виде текстовых файлов. Такой подход к сбору тестового корпуса для эксперимента обеспечивает абсолютную идентичность трех типов SMT-запросов, тем самым позволяя достоверно сравнить производительность решателей при их обработке.

Для сбора тестовых корпусов инструмент Sydr был запущен на 16 приложениях, покрывающих различные области применения. Каждое приложение было проанализировано на нескольких заранее выбранных входных данных, соответствующих различным путям исполнения. Из всех сгенерированных в ходе анализа запросов для каждого приложения были отобраны от десятков до нескольких сотен запросов, которые содержали в себе формулы для символьных указателей и различались между собой по размеру и составу предиката пути. В итоге, для каждой программы были получены три набора SMT-запросов, отличающиеся между собой только методом построения формул. Чтобы исключить особенности реализации конкретного SMT-решателя из результатов, при проведении эксперимента

были задействованы три разных инструмента: Z3 [66], Yices2 [67] и Bitwuzla [68]. В ходе эксперимента, результаты которого приведены в таблице 3, каждый из решателей был по очереди протестирован на всех собранных наборах запросов. В ходе тестирования SMT-решателя измерялось его время решения всего набора запросов. Текстовые файлы с SMT-запросами передавались на вход решателю через интерфейс командной строки, суммарное время обработки всего набора усреднялось за несколько итераций. Результаты измерения для решателей приведены в столбцах *Z3*, *Yices2* и *Bitwuzla*. Время обработки решателем разных наборов запросов приведено в столбцах *LIN* (линеаризованные деревья поиска), *ITE* (вложенные ITE-деревья) и *BST* (деревья бинарного поиска). Поскольку количество отобранных репрезентативных SMT-запросов для каждого приложения отличается, то разница во временах обработки между разными программами в данном эксперименте не отражает влияние конкретной программ на сложность формул. Однако на корректность сравнения между разными типами формул в рамках одной программы это не влияет. Для лучшей визуализации результатов, лучшее из трех времен для каждого SMT-решателя в таблице 3 выделено шрифтом. Несколько выделенных значений для одного SMT-решателя означает, что результаты равны и отличаются в пределах незначительной погрешности. Значение `T/O` (`timeout`) свидетельствует о том, что данный корпус запросов не смог обработаться SMT-решателем за отведенные 2 часа.

Самую лучшую эффективность метод с LIN-запросами показал в эксперименте с Z3. Запросы с LIN-формулами имеют лучшее время решения на всех программах, за исключением одного примера (`tiff2pdf`). Запросы с ITE-формулами имеют лучшее время только на 4 программах, причем на 3 из них (`libxml2`, `pk2bm`, `suricata`) результаты совпадают с методом линеаризованных деревьев. Запросы с BST-формулами в эксперименте с Z3 показали наихудший результат, имея сопоставимое время решения только на одном примере `suricata`. В эксперименте с Yices2 метод с LIN-формулами также проявил себя лучше остальных. Такие запросы имеют наименьшее время решения на 12 программах, в то время как ITE-запросы показали лучшие результаты всего на 6, причем на 3 из них (`jasper`, `readelf`, `suricata`) времена этих методов совпадают. Опять хуже всего при обработке в Yices2 себя показал BST-метод, имея только 2 лучших результата (эквивалентный остальным методам на `jasper` и лучший на `tiff2pdf`). При тестировании Bitwuzla метод LIN-запросов оказался наиболее эффективен на 10 приложениях из 16. Причем на всех 10 приложениях остальные методы

Таблица 3 — Анализ производительности SMT-решателей

Application	Z3			Yices2			Bitwuzla		
	LIN	ITE	BST	LIN	ITE	BST	LIN	ITE	BST
cjpeg	**1m7s**	1m16s	1m14s	**1.2s**	2s	2.6s	30.8s	**7s**	20.6s
eperl	**11.4s**	12.4s	20.3s	4.8s	**2.4s**	2.9s	17s	19.5s	**13.5s**
hdp	**18s**	27.5s	19.9s	**1.7s**	2.1s	2.5s	**8.2s**	17s	13s
jasper	**8.2**s	8.8s	9.8s	**1.4s**	**1.5s**	**1.5s**	**5.1s**	5.5s	5.7s
libcbor	**11.5s**	T/O	17.4s	**0.8s**	1s	1.3s	**2.8s**	6.4s	4.3s
libxml2	**4.2s**	**4.4s**	5.1s	**0.9s**	1.3s	1.5s	**3s**	8.8s	7.3s
minigzip	**0.9s**	4.4s	8.6s	**0.2s**	1.4s	1.9s	**0.5s**	10.1s	9s
muraster	**4.2s**	4.6s	5.8s	1.2s	**1s**	1.2s	**4.7s**	5.3s	5s
openssl	**2.4**s	6s	7.7s	**3s**	4.2s	5s	24s	**22s**	31s
pk2bm	**5.2s**	**5.2s**	7s	**1s**	1.2s	1.5s	**2.6s**	6.8s	6.1s
re2	**6.4s**	7s	7.2s	2.8s	**0.9s**	1.3s	**2.8s**	3.6s	5s
readelf	**2.3s**	3.2s	3.6s	**1s**	**1.1s**	1.2s	13.4s	**11.5s**	**11.6s**
sqlite3	**41s**	50s	47.6s	**3.7s**	5.5s	5.8s	19s	37s	**14.5s**
suricata	**3.2s**	**3.2s**	**3.5s**	**0.6s**	**0.6s**	0.8s	**3.7s**	4s	4.4s
tiff2pdf	1m33s	**39s**	1m5s	8.6s	7.2s	**6.9s**	37.5s	**18s**	20s
yodl	**4.7s**	6.9s	10s	**1.4s**	2.2s	2.4s	**3.9s**	9.5s	9.5s

не показали сопоставимого результата и были заметно хуже. Запросы с ITE-формулами показали лучшее время на 4 программах, а с BST-формулами – на 3. Запросы с BST-формулами хотя и имеют меньшее число лучших результатов, однако в целом имеют лучшее время решения, по сравнению с ITE-подходом. Для остальных решателей BST-запросы оказались наименее эффективным способом представления формул. Также стоит отметить, что результаты обработки трех типов формул для Yices2 имеют более близкие значения, чем для других протестированных SMT-решателей. Решатель Yices2 имеет немного лучшее время работы, чем Bitwuzla. Оба инструмента способны обрабатывать SMT-запросы за сильно меньшее время, по сравнению с Z3. Для проведения дальнейших экспериментов был выбран SMT-решатель Bitwuzla, для чего отдельно была реализована его поддержка в инструменте анализа Sydr. Такой выбор обусловлен тем, что несмотря на чуть более высокую скорость работы Yices2, Bitwuzla имеет более высокую точность решений [69].

Результаты проведенного эксперимента показывают, что использование для символьных указателей формул в виде линеаризованных деревьев поиска приводит к большей эффективности SMT-решателя при обработке запросов. В целом, для всех решателей метод LIN-формул показал либо самое лучшее время решения, либо не хуже, чем у методов ITE и BST-формул. Также, формулы с линеаризованными деревьями имеют гораздо меньшие размеры в сравнении с остальными подходами, что положительно сказывается и на скорости их обработки, и на потреблении памяти во время анализа.

3.5.2 Оценка производительности комбинированного метода обработки символьных указателей

В данном разделе приведены экспериментальные результаты применения разработанного метода интерпретации символьных указателей в составе инструмента динамической символьной интерпретации Sydr. Обработка символьных указателей в Sydr настроена следующим образом. Обнаружение границ области памяти, к которой производится доступ по символьному адресу, производится методом анализа символьного выражения этого адреса. Правая граница при этом определяется так, что вся область памяти имеет размер в 200 элементов доступа.

В качестве метода построения символьных ограничений для интерпретации символьного чтения используется описанный в этой главе комбинированный подход. Для решения символьных ограничений при инвертировании условных переходов используется SMT-решатель Bitwuzla. Для тестирования использовался сервер со следующей спецификацией: процессор AMD EPYC 7702 (128 ядер), 256 Гб оперативной памяти.

Целью одного из проведенных экспериментов является сравнение производительности инструмента анализа при работе в стандартном режиме и с поддержкой символьных указателей. Результаты такого сравнения приведены в таблице 4. Столбцы *default* соответствуют запускам Sydr в режиме стандартной символьной интерпретации, а столбцы *symaddr* – запускам Sydr с включенной обработкой символьных указателей. Все остальные настройки Sydr в обоих режимах работы полностью совпадают. Для проведения эксперимента были выбраны 11 программ с открытым исходным кодом, в которых при анализе было обнаружено большое число чтений по символьному адресу. Каждая программа анализировалась инструментом Sydr вдоль одного пути исполнения на подобранном вручную входном файле, обеспечивающим корректный сценарий работы программы и хорошее начальное покрытие кода. Общее время анализа каждой программы было ограничено одним часом. Инвертирование условных переходов в данном эксперименте выполнялось в отдельном потоке одновременно с построением предиката пути. Условные переходы инвертировались в один поток и в прямом порядке. Лимит времени на обработку одного запроса в SMT-решателе составлял 30 секунд. Время, отводимое на построение предиката пути, отдельно ограничено не было.

В столбце *Path predicate time* приведено время, затрачиваемое инструментом на интерпретацию всего пути исполнения программы и построение предиката пути. Включение режима анализа символьных указателей ожидаемо приводит к увеличению этого времени, поскольку с поддержкой дополнительных зависимостей в программе возрастает само число символьных инструкций, которые необходимо интерпретировать. Особенно сильно время построения предиката пути увеличилось для приложений `cjpeg`, `minigzip` и `yices-smt2`. В столбце *Total time* приведено общее время анализа программы, включая этап построения предиката пути. За отведенный час Sydr успел инвертировать все условные переходы на шести программах (`libcbor`, `libxml2`, `minigzip`, `sqlite3`, `yices-smt2` и `yodl`) в обычном режиме анализа и только на четырех (`libcbor`, `libxml2`, `sqlite3` и `yodl`) в режиме интерпретации символьных чтений. Общее время анализа

Таблица 4 — Сравнение производительности Sydr при работе в двух режимах

Application	Path predicate time		Total time		Queries/min	
	default	symaddr	default	symaddr	default	symaddr
cjpeg	47s	1m9s	60m	60m	136.3	66.1
hdp	18s	26s	60m	60m	219.5	55.5
jasper	9m37s	12m45s	60m	60m	840.2	1008.0
libcbor	3.9s	4.2s	4.0s	13.6s	6105.0	2545.6
libxml2	11s	14s	2m6s	6m42s	4385.7	1560.4
minigzip	54s	3m26s	9m3s	60m	991.9	20.3
openssl	2m12s	2m17s	60m	60m	46.0	45.4
readelf	9s	12s	60m	60m	73.3	25.3
sqlite3	13s	16s	2m30s	5m49s	31028.0	13713.3
yices-smt2	11s	25s	5m32s	60m	1982.1	281.8
yodl	7s	8s	14s	52s	27317.1	7716.9

увеличилось для всех программ, которые успевали отработать до остановки по таймауту. Чтобы оценить производительность анализа в случаях, когда программа была остановлена в обоих режимах анализа, была рассчитана скорость обработки запросов SMT-решателем. Эти данные приведены в столбце *Queries/min* – это число запросов, которое SMT-решатель в среднем успевал обработать за одну минуту. Результаты показывают, что скорость решения обычных запросов в разы выше тех, которые содержат ограничения для символьных чтений. Однако есть и исключения, например для программы `openssl` скорость решения практически не изменилась, а для `jasper` запросы с символьными чтениями обрабатывались быстрее обычных.

Результаты оценки точности инвертирования переходов для этого же эксперимента приведены в таблице 5. Столбец *SAT* показывает число запросов, для которых SMT-решатель смог подобрать решение (результат Satisfiable, SAT). Это соответствует количеству сгенерированных входных данных для инвертирования условных переходов. Во втором столбце *Accuracy* представлена точность инвертирования – соотношение между числом SAT и теми переходами, которые были

Таблица 5 — Точность инвертирования переходов

Application	SAT		Accuracy	
	default	symaddr	default	symaddr
cjpeg	1267	714	83.9%	79.4%
hdp	6117	1551	68.2%	64.9%
jasper	19494	37481	98.2%	99.2%
libcbor	312	131	100%	100%
libxml2	1247	1298	83.0%	91.2%
minigzip	7569	962	51.2%	60.0%
openssl	947	194	84.8%	77.2%
readelf	1360	139	47.7%	91.4%
sqlite3	8414	10340	99.9%	99.9%
yices-smt2	5540	2649	79.5%	82.2%
yodl	1150	1424	98.2%	98.9%

инвертированы корректно, то есть действительно приводили к исполнению программы в альтернативном направлении соответствующего условного перехода. Корректность инвертирования для каждого сгенерированного набора входных данных была доказана с помощью инструмента Tracer, описание которого приведено в разделе 2.3.3. В проведенном эксперименте Sydr успел в обоих запусках полностью проанализировать только четыре программы за отведенное время: `libcbor`, `libxml2`, `sqlite3` и `yodl`. Число инвертированных условных переходов на этих программах возросло, что свидетельствует о генерации большего числа новых входных данных при включении интерпретации символьных указателей. На остальных примерах число сгенерированных входных данных заметно снизилось. Однако это обусловлено тем, что за отведенный час успело инвертироваться меньше условных переходов из-за более низкой производительности SMT-решателя для запросов, содержащих формулы интерпретации символьных чтений. Точность инвертировния условных переходов в целом возросла. Хотя для некоторых программ она незначительно снизилась (`cjpeg`, `hdp`, `openssl`), для

большинства программ поддержка адресных зависимостей привела к увеличению точности (для `readelf` точность возросла до приемлемого уровня).

Результаты проведенного эксперимента показывают, что включение символьной интерпретации для указателей сильно сказывается на производительности анализа, в частности на скорости обработки запросов SMT-решателем. Из-за поддержки адресных зависимостей происходит увеличение символьной части исполнения программы, что приводит также к увеличению времени построения предиката пути. Хотя время, затрачиваемое на построение предиката пути, для некоторых программ возросло в разы, оно составляет сравнительно небольшую часть от общего времени анализа. Наиболее критичной частью с точки зрения производительности анализа является эффективность обработки SMT-запросов в решателе.

3.5.3 Влияние обработки символьных указателей на прирост покрытия

Для того чтобы оценить, насколько интерпретация символьных указателей увеличивает символьную часть исполнения программы и как это отражается на конструируемом предикате пути, инструмент Sydr запускался в двух режимах работы на анализ 11 программах до полного построения предиката пути. По окончании интерпретации программы, Sydr генерирует трассу символьных точек ветвления, которая является основой предиката пути. Каждая ветвь в трассе представлена тройкой $\langle module\_name, src\_offset, dst\_offset \rangle$, что позволяет провести корректное сравнение сконструированных во время двух запусков предикатов пути. Результаты такого сравнения представлены в таблице 6.

Столбцы *default* и *symaddr* представляют запуски инструмента Sydr в обычном режиме и в режиме интерпретации символьных указателей соответственно. Столбец *Total* представляет общее число символьных условных переходов в предикате пути. Эти данные показывают, что символьная модель программы сильно возрастает при включении анализа символьных указателей. Число обнаруженных инструментом Sydr символьных ветвей увеличилось на всех программах, для некоторых даже больше, чем в два раза (`cjpeg`, `minigzip`, `yices-smt2`). В столбце *Unique* посчитано число уникальных символьных переходов в рамках соответствующего режима анализа. Для этого из общего списка переходов для каждого

Таблица 6 — Число исследованных ветвей

Application	Total		Unique		New unique	
	default	symaddr	default	symaddr	default	symaddr
cjpeg	6012	28834	122	197	0	75
hdp	25947	28481	345	371	1	27
jasper	771800	1093891	91	101	0	10
libcbor	122	158	31	34	0	3
libxml2	8934	10203	393	438	0	45
minigzip	8977	52888	23	68	0	45
muraster	7102	7109	75	82	0	7
openssl	7545	7788	195	215	0	20
readelf	11773	14995	793	802	0	9
sqlite3	6970	8992	53	65	0	12
yices-smt2	10092	22737	82	532	0	450
yodl	4613	4927	40	53	0	13

режима были убраны полностью повторяющиеся условные переходы. Число уникальных символьных ветвей на порядки ниже их общего числа ввиду наличия в программах циклов и повторных вызовов функций. Увеличение числа уникальных ветвей показывает реальное количество нового кода в программе, который удается проанализировать благодаря интерпретации символьных указателей. Последний столбец *New and unique* показывает число уникальных символьных ветвей относительно альтернативного режима анализа. В соответствующих колонках указано число таких символьных условных переходов, которые не были обнаружены в другом режиме. Данные в колонке *symaddr* показывают, что интерпретация символьных чтений приводит к открытию множества новых условных переходов, которые ранее при анализе не встречались. На некоторых программах (`minigzip`, `yices-smt2`) число новых ветвей даже больше, чем число уникальных условных переходов в предикате пути при стандартном режиме анализа. В свою очередь, колонка *default* практически для всех программ содержит ноль. Это

означает, что все условные переходы, которые были обнаружены при обычной символьной интерпретации, были также обнаружены и при анализе с поддержкой символьных зависимостей. Только на единственной программе `hdp` произошла потеря одной символьной ветви. Это объясняется тем, что при обработке адресных зависимостей в программе начинает интерпретироваться множество новых инструкций, которые не были символьными до этого. В результате символьные данные начинают распространяться по-другому, некоторые данные становятся символьными, часть наоборот – конкретизируется. В результате конкретизации части данных некоторые условные переходы, которые были символьными при обычной символьной интерпретации, перестают зависеть от входных данных. Такие условные переходы добавляются в предикат пути только за счет того, что при анализе учитывались не все зависимости в программе. При попытке их инвертирования составляются неточные символьные ограничения, что в результате приводит либо к неразрешимым SMT-запросам, либо к ошибочным входным данным.

Таким образом, интерпретация символьных чтений приводит к построению более точного и полного предиката пути. Поддержка адресных зависимостей позволяет открывать множество новых условных переходов, потенциально ведущих к увеличению покрытия, которое невозможно достичь при стандартной символьной интерпретации.

Главной целью интерпретации адресных зависимостей является увеличение тестового покрытия программы. Для оценки прироста покрытия был проведен следующий эксперимент. Sydr был запущен на анализ 11 программ в двух разных режимах и работал без ограничения по времени до тех пор, пока все условные переходы не были инвертированы. Все условные переходы инвертировались в прямом порядке в несколько потоков для ускорения эксперимента. При работе инструмента использовалось кэширование инвертированных условных переходов: если определенный переход был успешно инвертирован, то он исключался из последующих запросов к SMT-решателю, чтобы избежать повторного инвертирования. Условные переходы различались по имени модуля, адресу и направлению перехода, контекст исполнения для перехода не учитывался. Результатом запуска Sydr являлся корпус новых входных данных, полученных при инвертировании всех условных переходов в предикате пути. На основе этих корпусов было рассчитано достигнутое в ходе анализа покрытие по базовым блокам программы.

Таблица 7 — Оценка покрытия

Application	**Code coverage (%)**		**Coverage diff (%)**	
	default	**symaddr**	**default/symaddr**	**symaddr/default**
cjpeg	22.75	23.02	0	0.27
hdp	10.57	10.86	0	0.28
jasper	10.03	10.17	0	0.13
libcbor	79.55	80.37	0.41	1.23
libxml2	7.85	8.86	0	1.01
minigzip	30.85	30.66	0.61	0.43
openssl	5.32	5.31	0.02	0.01
readelf	15.40	15.12	1.08	0.81
sqlite3	5.50	5.60	0	0.1
yices-smt2	3.48	3.51	0	0.03
yodl	26.97	27.72	0	0.75

Подсчет покрытия осуществлялся с помощью клиента DynamorRIO drcov [70]. Тестируемая программа запускалась через drcov-клиент на соответствующем корпусе входных данных, в результате генерировался корпус специальных файлов, содержащих информацию о покрытии. Эти файлы затем использовались при расчете итогового покрытия с помощью инструмента IDA Pro [71] и его плагина Lighthouse [72]. Использование этих инструментов позволяет получить величину покрытия по базовым блокам в процентном соотношении от кода всей программы. Также плагин Lighthouse позволяет вычислять разницу между покрытиями, собранными на разных корпусах входных данных. Полученные в ходе эксперимента результаты представлены в таблице 7. В столбце *Code coverage* приведено общее покрытие, достигнутое инструментом Sydr при анализе в обычном режиме и с поддержкой символьных адресов. Значение покрытия представлено в процентах от общего кода программы. Столбец *Coverage diff* показывает разницу между этими двумя покрытиями. В колонке *symaddr/default* приведено уникальное покрытие, достигнутое при анализе с включенной интерпретацией символьных чтений и не открытое при стандартном запуске Sydr.

Обратная ситуация приведена в колонке *default/symaddr*. Числа в ней представляют процент покрытия от общего кода программы, который был найден только при обычном запуске Sydr.

Результаты эксперимента показывают, что разработанный метод поддержки символьных указателей позволяет открывать новый код, недостижимый при обычном анализе, на всех программах. Для большей части программ такой анализ приводит обнаружению нового кода без каких-либо потерь покрытия по сравнению с обычным анализом. Поскольку в ходе эксперимента проводилось инвертирование условных переходов только вдоль одного пути исполнения программы, то общие цифры прироста покрытия сравнительно небольшие. Тем не менее, на двух программах (`libcbor` и `libxml2`) применение разработанного метода прирост нового покрытия составил больше процента. В целом, хорошего прироста покрытия удалось достичь на 7 программах из 11. Однако на нескольких программах (`libcbor`, `minigzip`, `openssl` и `readelf`) уникальное покрытие также обнаружилост и при стандартном запуске инструмента Sydr. На программе `openssl` оба запуска позволили достигнуть практически одинакового покрытия, доля нового уникального кода составляет в обоих случаях довольно мала (0.02% и 0.01% соответственно). В случае `minigzip` и `readelf` стандартный анализ имеет немного большее итоговое покрытие. Причиной этому являются слишком сложные ограничения, получаемые в результате интерпретации символьных чтений, которые приводят к тому, что для части запросов к SMT-решателю решение либо не находится вообще, либо занимает слишком много времени и завершается по ограничению времени.

Проведенные эксперименты показывают, что интерпретация адресных зависимостей позволяет проводить более полный анализ программы, обнаруживать много новых условных переходов и в целом приводить к увеличению достигнутого покрытия. С другой стороны анализ адресных зависимостей создает большую нагрузку на SMT-решатель, что отрицательно сказывается на производительности всего анализа. В ходе проведенных экспериментов было показано, что поддержка символьных чтений всегда приводит к обнаружению нового кода и открытию нового уникального покрытия. В то же время, для некоторых программ возможно небольшое ухудшение результатов, вследствие конструирования слишком сложных ограничений пути. Исходя из этого, оптимальной стратегией применения данного режима при анализе реальных программ является чередование со стандартным режимом символьной интерпретации в ходе исследования

путей исполнения программы. Также снизить нагрузку поможет кэширование инвертированных условных переходов, которое позволяет избежать повторного инвертирования переходов. Тем самым анализ с поддержкой символьных указателей будет сосредоточен на инвертировании в основном новых условных переходов, которые не были обнаружены ранее. В свою очередь обычные условные переходы будут легко и быстро инвертироваться при стандартном запуске инструмента, что позволит избежать потери новых путей исполнения из-за слишком сложных SMT-запросов.

Глава 4. Программная реализация предложенных методов

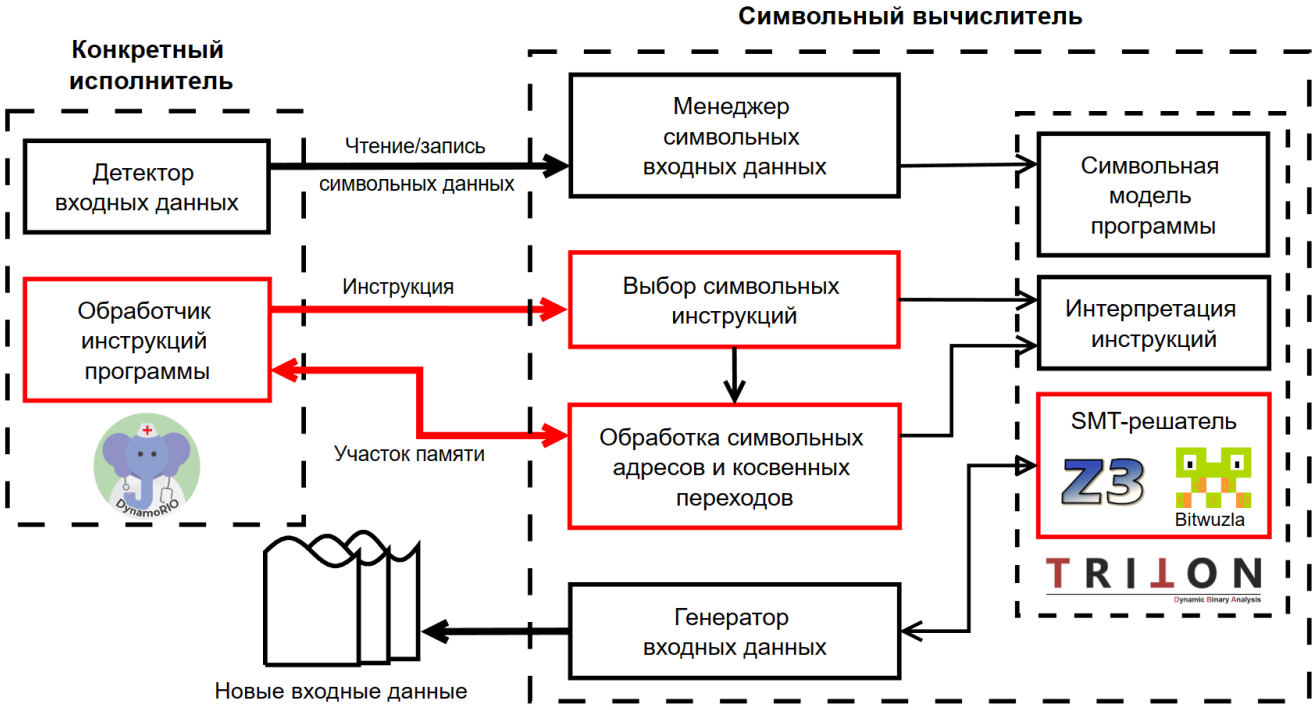
4.1 Инструмент динамической символьной интерпретации Sydr

Все разработанные методы были реализованы в инструменте динамической символьной интерпретации Sydr, разрабатываемом в ИСП РАН. Sydr производит анализ бинарных x86/x86-64 программ для ОС Linux и не требует наличия исходных кодов. Инструмент реализует конкретно-символьное выполнение и построен с использованием DBI-фреймворка DynamoRIO и библиотеки символьной интерпретации Triton [59]. Анализ в Sydr разделен на два процесса, работающих параллельно: конкретный исполнитель и символьный вычислитель. Конкретный исполнитель представляет собой DynamoRIO-клиент, в контексте которого запускается исследуемая программа. Клиент производит инструментирование кода программы на уровне системных сигналов, вызовов функций, базовых блоков и отдельных инструкций. Символьный вычислитель отвечает за символьную интерпретацию программы, в качестве реализации символьного движка используется библиотека Triton. В данном процессе производится отслеживание символьного состояния программы, интерпретация отдельных инструкций программы, построение предиката пути и генерация новых входных данных.

Символьный вычислитель производит символьную интерпретацию программы, основываясь на информации о реальном исполнении программы, получаемой от конкретного исполнителя. Передача информации между двумя процессами организована через разделяемую память. При возникновении определенных событий во время исполнения программы конкретный вычислитель записывает всю необходимую для обработки этого события информацию в разделяемую память. Символьный вычислитель всегда находится в ожидании очередного события, и при получении информации из разделяемой памяти он выполняет обработку события в символьной модели программы. Такими событиями являются обращение программы ко входным данным, вызовы некоторых функций, нуждающихся в дополнительной интерпретации со стороны символьного вычислителя, а также начало исполнения каждой инструкции в программе. На рисунке 4.1 приведена схема работы Sydr. Конкретный исполнитель инициирует отправку информации о событиях в символьный вычислитель. Так, при

обнаружении чтения из источника входных данных эта информация отправляется в менеджер символьных входных данных, который производит создание символьных переменных для соответствующих регистров и ячеек памяти. Для каждой инструкции передается вся нужная для ее символьной интерпретации информация: ее операционный код, регистр флагов, значения регистров и памяти, соответствующие ее операндам. Символьный вычислитель проверяет, что данная инструкция работает с символьными данными и выполняет ее интерпретацию. С помощью Triton выполняется построение символьных ограничений пути, которые соответствуют семантике интерпретируемой инструкции.

Рисунок 4.1 — Схема инструмента Sydr



При обработке символьных инструкций ветвления происходит обновление предиката пути программы – ограничений, соответствующих всем символьным переходам на текущем пути исполнения программы. Все добавляемые в предикат пути переходы инвертируются генератором входных данных. Генератор работает асинхронно по отношению к обработке событий от конкретного исполнителя. Для каждого условного перехода с помощью Triton конструируется формула для инвертирования его направления и выполняется соответствующий запрос к SMT-решателю. В качестве SMT-решателя Triton поддерживает Z3, также в рамках данной работы была добавлена поддержка Bitwuzla. На основе моделей, получаемых от SMT-решателя происходит генерация новых входных данных, что и является основным результатом работы инструмента.

4.2 Реализация методов моделирования косвенной адресации

Для обработки косвенных переходов и для моделирования символьных чтений требуются значения ячеек памяти из всей области, к которой может происходить доступ по символьному адресу. Такой участок памяти может иметь довольно большие размеры, напрямую зависящие от размера доступа к памяти, и его пересылка между процессами анализа является накладной операцией. Большинство инструкций программы не являются символьными, для них символьной интерпретации не производится, поэтому и память пересылать в таком случае нет необходимости. При реализации методов в схему работы инструмента были внесены такие изменения, что для обработки инструкций с символьными указателями и табличными переходами выполнялся двойной обмен сообщениями между процессами конкретного исполнителя и символьного вычислителя. Такое взаимодействие между процессами анализа отображено на рисунке 4.1.Реализованные в рамках работы модули инструмента выделены на рисунке красным цветом. Обработчик инструкций в конкретном исполнителе был модифицирован таким образом, чтобы распознавать факт наличия косвенных переходов и определять границы таблицы переходов в памяти программы. В символьном вычислителе блок выбора символьных инструкций был улучшен для поддержки инструкций, работающих с вычисляемыми символьными адресами в качестве операндов, был реализован модуль обработки символьных адресов и косвенных переходов. Также в открытой библиотеке символьной интерпретации Triton была добавлена поддержка более производительного SMT-решателя Bitwuzla.

В целом процесс обработки таких инструкций реализован следующим образом:

1. При отправке инструкции конкретный исполнитель проверяет, что инструкция либо обращается к таблице переходов, либо имеет чтение из памяти в качестве одного из операндов-источников, и в случае необходимости ставит соответствующую пометку для символьного вычислителя.
2. Символьный вычислитель проверяет что адрес, по которому производится доступ к памяти/таблице переходов является символьным. Если это так, то для символьных указателей определяются приблизительные границы доступа к памяти с использованием эвристического анализа

выражения символьного адреса. Также опционально возможно использование SMT-решателя.

3. Конкретный исполнитель в это время ожидает ответа для таких инструкций. Если адрес не является символьным то никаких дополнительных действий не производится и анализ программы продолжается дальше. В противном случае с помощью специальных вызовов DynamoRIO происходит чтение памяти в определенных границах и отправка этого массива значений обратно в символьный вычислитель для интерпретации. Для табличных условных переходов предварительно следует определить границы памяти. Делается это методом, описанным в главе 2.3.1. – от текущего адреса доступа в обоих направлениях выполняется обход памяти, пока не лимит элементов доступа не будет исчерпан, либо пока условия таблицы не нарушатся.
4. Символьный вычислитель получает весь участок памяти в виде массива данных, а также информацию о начальном адресе памяти и количестве элементов. По этим данным происходит приведение участка памяти в вид ⟨адрес, значение⟩. Далее, с использованием полученных значений, выполняется символьная интерпретация инструкций с учетом используемых ею символьных адресов.

Обработка табличного условного перехода происходит непосредственно перед символьной интерпретацией текущей инструкции в Triton путем построения символьных ограничений для каждого направления перехода. Данные ограничения описывают условный переход с несколькими направлениями и должны быть добавлены в предикат пути программы. Однако построение этих ограничений производится на инструкции доступа к таблице переходов, а сама передача управления происходит, как правило, через несколько инструкций. Для корректного отображения символьной ветви в Triton и для совпадения адресов источника и направления перехода, ограничения пути должны быть добавлены в предикат непосредственно на инструкции передачи управления. В таком случае обработка табличных переходов будет полностью идентична обычным условным переходам, с разницей лишь в числе направлений и ограничений. Более того, поскольку в данном случае передача управления моделируется разработанным методом отдельно, то перед интерпретацией соответствующей инструкции необходимо удалить все символьные пометки с ее операндов. В тривиальном случае доступ к таблице переходов и сама передача управления выполняется одной ко-

мандой. В таком случае после построения символьных ограничений они сразу же добавляются в предикат пути, а интерпретация инструкции в Triton пропускается полностью. Во всех остальных случаях метод запоминает построенные ограничения и интерпретирует все инструкции в обычном режиме до тех пор, пока обработка не перейдет к первой инструкции передачи управления с момента доступа к таблице переходов.

Обработка остальных символьных указателей реализована в Sydr по-другому. В данном случае необходимо провести моделирование чтения по символьному адресу в рамках отдельного операнда инструкции и встроить результат моделирования в процесс символьной интерпретации инструкции в Triton. Само построение символьных ограничений происходит в отдельной функции-обработчике для каждой инструкции. Из символьной модели достаются выражения для всех операндов-источников инструкции, затем они используются при составлении ограничений для моделирования семантики инструкции. Результат записывается в операнд-приемник, соответствующие изменения вносятся в символьную модель программы. Чтобы не вносить изменения в сам фреймворк Triton, не изменять обработчики всех инструкций, работающих с памятью и не усложнять интеграцию с инструментом Sydr, поддержка интерпретации символьных адресов была реализована следующим образом:

1. Перед тем как выполнить символьную интерпретацию инструкции в Triton, Sydr отдельно производит моделирование символьного чтения. В результате конструируется символьное выражение, которое описывает результат чтения из памяти.
2. Это символьное выражение добавляется в символьную модель программы в качестве операнда-источника текущей инструкции, который представляет собой символьный доступ к памяти. Если данному операнду уже соответствует какое-либо символьное выражение, то оно сохраняется отдельно и вместо него перезаписывается построенное методом выражение.
3. В Triton производится символьная интерпретация текущей инструкции, в процессе которой для операнда, выполняющего символьное чтение из памяти, будет автоматически использовано построенное методом ограничение.

4. По завершении интерпретации этой инструкции в Triton, исходное состояние операнда-источника восстанавливается в символьной модели программы.

Таким образом, разработанные методы моделирования косвенной адресации были реализованы и интегрированы в инструмент Sydr с минимальными изменениями архитектуры инструмента и не требуют изменений в сторонних используемых библиотеках.

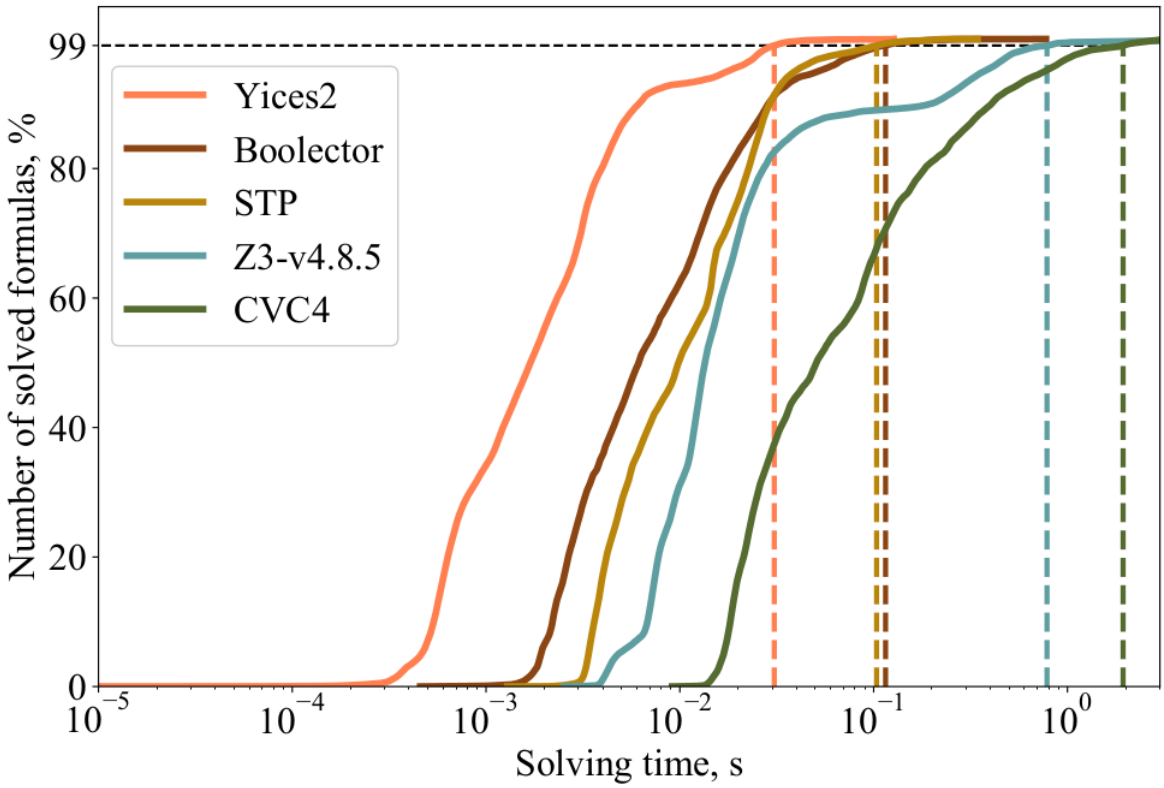
4.3 Решатели булевых формул в теориях

4.3.1 Исследование производительности SMT-решателей

Производительность SMT-решателя при обработке запросов является одним из ограничивающих факторов в динамической символьной интерпретации. Сложность конструируемых ограничений, а также выбор конкретного инструмента может сильно повлиять на качество и производительность всего анализа в целом, как было показано в таблицах 3 и 4.

В работе [2] приводится исследование о применении различных SMT-решателей для задач статической и динамической символьной интерпретации. В частности, для оценки SMT-решателей в динамической символьной интерпретации был собран корпус запросов, полученных в результате анализа ряда проектов с открытым исходным кодом с помощью Anxiety [1] - инструмента динамической символьной интерпретации, опыт разработки котрого был использован при создании инструмента Sydr. Полученные запросы представляют собой текстовый файл с SMT-формулой, записанной в терминах логики QF_BV языка SMT-LIBv2. В исследовании принимали участие SMT-решатели Yices2 [67], Boolector [73], STP [32], Z3 [66] и CVC4 [74]. На рисунке 4.2 изображено время решения набора запросов для каждого инструмента. Из результатов измерения видно, что наименьшее время решения имеет Yices2, а наибольшее - CVC4. Результаты для Boolector, STP и Z3 довольно близки при обработке 80% корпуса, однако к концу эксперимента время Z3 заметно ухудшается. По итогам измерений лучшие результаты с точки зрения скорости решения формул показывают инструменты Yices2

Рисунок 4.2 — Время обработки запросов в различных SMT-решателях



и Boolector. Эти данные подтверждаются и проведенными в этой работе экспериментами (таблица 3), где вместо Boolector был выбран более современный аналог от тех же разработчиков - SMT-решатель Bitwuzla [68]. Основываясь только на скорости обработки запросов, наилучшим SMT-решателем для использования в инструменте динамической символьной интерпретации является Yices2. Однако недостатком данного инструмента является наличие неточных и ошибочных решений запросов [69], что имеет критическое значение для символьной интерпретации. Исходя из этих соображений, для использования в инструменте был выбран SMT-решатель Bitwuzla.

4.3.2 Поддержка Bitwuzla

В инструменте Sydr взаимодействие с SMT-решателями организуется через библиотеку Triton, которая по умолчанию поддерживает только решатель Z3. В ходе работы, в целях улучшения производительности методов моделирования адресных зависимостей, в основной репозиторий проекта Triton была добавлена поддержка более быстрого SMT-решателя Bitwuzla [75]. Triton использует SMT-

решатели не как отдельные исполняемые программы, а в качестве связываемых библиотек, используя их C/C++ интерфейс для отправки запросов и получения результата. Процесс взаимодействия с решателями в Triton построен следующим образом. Пользователь формирует запрос - определенное символьное выражение, которое объединяется с другими выражениями из предиката пути. Запрос представляет собой одну большую вложенную формулу в виде TritonAst - внутреннем представлении Triton. Чтобы отправить этот запрос в SMT-решатель, необходимо провести конвертацию TritonAst в выражения для выбранного решателя, используя его API. Для каждого запроса объект решателя создается заново, затем в него добавляются выражения, полученные в результате конвертирования запроса. После этого вызывается проверка совместимости введенных формул, и если решатель вернул положительный результат SAT, то Triton дополнительно запрашивает модель решения. Сгенерированная решателем модель также анализируется в Triton и передается пользователю уже в обработанном виде, который связывает решение с использованными в запросе символьными переменными $\langle SymbolicVariable, Value \rangle$.

Для поддержки Bitwuzla в составе Triton был реализован преобразователь выражений из TritonAst в BitwuzlaAst. Конвертация формул между форматами является трудоемким процессом, поскольку рекурсивно обходит все узлы и листья абстрактного синтаксического дерева формулы. Время конвертации напрямую зависит от размера запроса, и в некоторых случаях может происходить дольше, чем само решение в SMT-решателе. Поэтому при реализации преобразователя особое внимание было уделено производительности. Чтобы не создавать лишних объектов, размерности битовых векторов добавлялись в специальный кэш для повторного переиспользования. Целочисленные параметры формулы в TritonAst представлены специальными узлами дерева, в то время как в Bitwuzla такие параметры используются нативно. Поэтому при обходе дерева TritonAst спуск в целочисленные узлы дерева не производился, а соответствующие значения использовались при построении BitwuzlaAst с более высокого уровня дерева.

Также Triton поддерживает введение дополнительных ограничений на время работы и потребление памяти SMT-решателем. Ограничение времени решения одного запроса в решателе может предотвратить зависание на неопределенное время и расходование вычислительных ресурсов на слишком сложные запросы. Вместо долгой обработки одного запроса, результатом которой может стать

отсутствие решения, решение нескольких более простых запросов является более выгодной стратегией с точки зрения динамического анализа. Расходование памяти SMT-решателем также может значительно увеличиться в процессе обработки запроса, что негативно повлияет на процесс анализа в целом. В Bitwuzla по умолчанию отсутствует поддержка таких ограничений. Соответствующая функциональность была реализована с помощью специальных `callback`-функций, которые вызываются SMT-решателем самостоятельно в определенные моменты работы.

Заключение

Основные результаты работы заключаются в следующем.

1. Разработан метод поиска и моделирования косвенных переходов. Поиск косвенных переходов выполняется с помощью обратного слайсинга в пределах базового блока. Для определения границ таблицы переходов используется разработанный эвристический подход, основанный на анализе содержимого таблицы.
2. Разработан метод моделирования чтений памяти по символьно вычисляемому адресу. Метод позволяет моделировать символьные чтения с помощью выражения, которое представляет собой двоичное дерево поиска. Данное выражение оптимизируется с помощью линеаризации и построения горизонтальных линий. Используется комбинированный подход построения выражения для учета символьного содержимого памяти.
3. Предложенные методы реализованы в инструменте динамической символьной интерпретации Sydr, разрабатываемого в ИСП РАН. Проведена оценка методов на наборе прикладных программ реального мира, которая показала, что предложенные методы позволяют увеличить число инвертируемых символьных условных переходов и достичь большего покрытия на анализируемых программах. Проведено исследование производительности SMT-решателей при обработке выражений, моделирующих символьные чтения, по итогам которой определены оптимальный вид выражений и предпочтительный SMT-решатель.

Для развития предложенных методов может рассматриваться использование отладочной информации во время поиска границ доступа к памяти при моделировании символьных чтений.

Список литературы

1. Anxiety: a dynamic symbolic execution framework [Text] / A. Gerasimov [et al.] // 2017 Ivannikov ISPRAS Open Conference (ISPRAS). — IEEE, 2017. — P. 16—21.
2. SMT Solvers in Application to Static and Dynamic Symbolic Execution: A Case Study [Text] / N. Malyshev [et al.] // 2019 Ivannikov Ispras Open Conference (ISPRAS). — IEEE, 2019. — P. 9—15.
3. Sydr: Cutting edge dynamic symbolic execution [Text] / A. Vishnyakov [et al.] // 2020 Ivannikov ISPRAS Open Conference (ISPRAS). — IEEE, 2020. — P. 46—54.
4. *Kuts, D.* Towards symbolic pointers reasoning in dynamic symbolic execution [Text] / D. Kuts // 2021 Ivannikov Memorial Workshop (IVMEM). — IEEE, 2021. — P. 42—49.
5. *Свидетельство о гос. регистрации программы для ЭВМ*. Инструмент динамической символьной интерпретации «Sydr» [Текст] / А. В. Вишняков [и др.] ; Ф. государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук. — № 2020662214 ; заявл. 30.09.2020 ; опубл. 09.10.2020, 2020662214 (Рос. Федерация).
6. *Свидетельство о гос. регистрации программы для ЭВМ*. Sydr-fuzz [Текст] / А. Н. Федотов, А. В. Вишняков, Д. О. Куц ; Ф. государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук. — № 2021665874 ; заявл. 24.09.2021 ; опубл. 04.10.2021, 2021665874 (Рос. Федерация).
7. *Свидетельство о гос. регистрации программы для ЭВМ*. Anxiety: модульный инструмент итеративного динамического символьного исполнения [Текст] / С. П. Вартанов [и др.] ; Ф. государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук. — № 2017660037 ; заявл. 13.09.2017 ; опубл. 13.09.2021, 2017660037 (Рос. Федерация).

8. *Свидетельство о гос. регистрации программы для ЭВМ*. Anxiety: модуль параллельных вычислений для инструмента итеративного динамического символьного исполнения [Текст] / С. П. Вартанов [и др.] ; Ф. государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук. — № 2017660154 ; заявл. 18.09.2017 ; опубл. 18.09.2017, 2017660154 (Рос. Федерация).
9. *King, J. C.* Symbolic execution and program testing [Текст] / J. C. King // Communications of the ACM. — 1976. — Т. 19, № 7. — С. 385—394.
10. Symbolic Security Predicates: Hunt Program Weaknesses [Текст] / A. Vishnyakov [и др.] // 2021 Ivannikov Ispras Open Conference (ISPRAS). — IEEE. 2021. — С. 76—85.
11. *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.
12. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed) [Текст] / D. C. D'Elia [и др.] // Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security. — 2019. — С. 15—27.
13. 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.
14. *Bruening, D.* Transparent dynamic instrumentation [Текст] / D. Bruening, Q. Zhao, S. Amarasinghe // Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments. — 2012. — С. 133—144.
15. *Nethercote, N.* Valgrind: a framework for heavyweight dynamic binary instrumentation [Текст] / N. Nethercote, J. Seward // ACM Sigplan notices. — 2007. — Т. 42, № 6. — С. 89—100.
16. *Guizani, W.* Server-side dynamic code analysis [Текст] / W. Guizani, J.-Y. Marion, D. Reynaud-Plantey // 2009 4th International Conference on Malicious and Unwanted Software (MALWARE). — IEEE. 2009. — С. 55—62.
17. *Bruening, D.* Practical memory checking with Dr. Memory [Текст] / D. Bruening, Q. Zhao // International Symposium on Code Generation and Optimization (CGO 2011). — IEEE. 2011. — С. 213—223.

18. *Fratric, I.* WinAFL [Electronic Resource] / I. Fratric. — URL: https://github.com/googleprojectzero/winafl (visited on 09/08/2022).

19. *Drewry, W.* Flayer: Exposing application internals [Текст] / W. Drewry, T. Ormandy. — 2007.

20. *Newsome, J.* Dynamic Taint Analysis for Automatic Detection, Analysis, and SignatureGeneration of Exploits on Commodity Software. [Текст] / J. Newsome, D. X. Song // NDSS. T. 5. — Citeseer. 2005. — С. 3—4.

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

22. 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). T. 13. — 2010. — С. 14.

23. *Cook, S. A.* The complexity of theorem-proving procedures [Текст] / S. A. Cook // Proceedings of the third annual ACM symposium on Theory of computing. — 1971. — С. 151—158.

24. *Davis, M.* A machine program for theorem-proving [Текст] / M. Davis, G. Logemann, D. Loveland // Communications of the ACM. — 1962. — Т. 5, № 7. — С. 394—397.

25. Just fuzz it: solving floating-point constraints using coverage-guided fuzzing [Текст] / D. Liew [и др.] // Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — 2019. — С. 521—532.

26. *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.

27. 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.

28. *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.

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

30. 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.

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

32. *Ganesh, V.* A decision procedure for bit-vectors and arrays [Текст] / V. Ganesh, D. L. Dill // International conference on computer aided verification. — Springer. 2007. — С. 519—531.

33. SCSE: Boosting Symbolic Execution via State Concretization [Текст] / H. Wang [и др.] // IEICE TRANSACTIONS on Information and Systems. — 2019. — Т. 102, № 8. — С. 1506—1516.

34. Chopped symbolic execution [Текст] / D. Trabish [и др.] // Proceedings of the 40th International Conference on Software Engineering. — 2018. — С. 350—360.

35. *Busse, F.* Running symbolic execution forever [Текст] / F. Busse, M. Nowack, C. Cadar // Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. — 2020. — С. 63—74.

36. *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.

37. *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.

38. 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.

39. Assisting malware analysis with symbolic execution: A case study [Текст] / R. Baldoni [и др.] // International conference on cyber security cryptography and machine learning. — Springer. 2017. — С. 171—188.

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

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

42. BAP: A binary analysis platform [Текст] / D. Brumley [и др.] // International Conference on Computer Aided Verification. — Springer. 2011. — С. 463—469.

43. Automatic exploit generation [Текст] / T. Avgerinos [и др.] // Communications of the ACM. — 2014. — Т. 57, № 2. — С. 74—84.

44. *Zalewski, M.* American Fuzzy Lop [Electronic Resource] / M. Zalewski. — URL: http://lcamtuf.coredump.cx/afl/ (visited on 09/08/2022).

45. 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.

46. *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.

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

48. *Borzacchiello, L.* FUZZOLIC: Mixing fuzzing and concolic execution [Текст] / L. Borzacchiello, E. Coppa, C. Demetrescu // Computers & Security. — 2021. — Т. 108. — С. 102368.

49. *Borzacchiello, L.* Fuzzing symbolic expressions [Текст] / L. Borzacchiello, E. Coppa, C. Demetrescu // 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). — IEEE. 2021. — С. 711—722.

50. radare2 [Electronic Resource]. — URL: https://github.com/radareorg/radare2 (visited on 09/08/2022).

51. SoK: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask [Текст] / C. Pang [и др.] // 2021 IEEE Symposium on Security and Privacy (SP). — IEEE. 2021. — С. 833—851.

52. *Buck, B.* An API for runtime code patching [Текст] / B. Buck, J. K. Hollingsworth // The International Journal of High Performance Computing Applications. — 2000. — Т. 14, № 4. — С. 317—329.

53. What you see is not what you get! thwarting just-in-time rop with chameleon [Текст] / P. Chen [и др.] // 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). — IEEE. 2017. — С. 451—462.

54. Playing inside the black box: Using dynamic instrumentation to create security holes [Текст] / B. P. Miller [и др.] // Parallel Processing Letters. — 2001. — Т. 11, 02n03. — С. 267—280.

55. *Mußler, J.* Reducing the overhead of direct application instrumentation using prior static analysis [Текст] / J. Mußler, D. Lorenz, F. Wolf // European Conference on Parallel Processing. — Springer. 2011. — С. 65—76.

56. *Balakrishnan, G.* Analyzing memory accesses in x86 executables [Текст] / G. Balakrishnan, T. Reps // International conference on compiler construction. — Springer. 2004. — С. 5—23.

57. A survey of symbolic execution techniques [Текст] / R. Baldoni [и др.] // ACM Computing Surveys (CSUR). — 2018. — Т. 51, № 3. — С. 1—39.

58. Memory models in symbolic execution: key ideas and new thoughts [Текст] / L. Borzacchiello [и др.] // Software Testing, Verification and Reliability. — 2019. — Т. 29, № 8. — e1722.

59. *Saudel, F.* Triton: Concolic execution framework [Текст] / F. Saudel, J. Salwan // Symposium sur la sécurité des technologies de l'information et des communications (SSTIC). — 2015.

60. *Cifuentes, C.* Recovery of jump table case statements from binary code [Текст] / C. Cifuentes, M. Van Emmerik // Science of Computer Programming. — 2001. — Т. 40, № 2/3. — С. 171—188.

61. *Sale, A.* The implementation of case statements in Pascal [Текст] / A. Sale // Software: Practice and Experience. — 1981. — Т. 11, № 9. — С. 929—942.

62. *Hennessy, J. L.* Compilation of the Pascal case statement [Текст] / J. L. Hennessy, N. Mendelsohn // Software: Practice and Experience. — 1982. — Т. 12, № 9. — С. 879—882.

63. *Parygina, D.* Strong Optimistic Solving for Dynamic Symbolic Execution [Текст] / D. Parygina, A. Vishnyakov, A. Fedotov // arXiv preprint arXiv:2209.03710. — 2022.

64. *Chen, P.* Matryoshka: fuzzing deeply nested branches [Текст] / P. Chen, J. Liu, H. Chen // Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. — 2019. — С. 499—513.

65. Sydr benchmark [Electronic Resource]. — URL: https://github.com/ispras/sydr_benchmark (visited on 09/08/2022).

66. *Moura, L. d.* Z3: An efficient SMT solver [Текст] / L. d. Moura, N. Bjørner // International conference on Tools and Algorithms for the Construction and Analysis of Systems. — Springer. 2008. — С. 337—340.

67. *Dutertre, B.* Yices 2.2 [Текст] / B. Dutertre // International Conference on Computer Aided Verification. — Springer. 2014. — С. 737—744.

68. *Niemetz, A.* Bitwuzla at the SMT-COMP 2020 [Текст] / A. Niemetz, M. Preiner // arXiv preprint arXiv:2006.01621. — 2020.

69. SMT-COMP 2020: QFBV (Single Query Track) [Electronic Resource]. — URL: https://smt-comp.github.io/2020/results/qf-bv-single-query (visited on 12/15/2022).

70. Code Coverage Tool [Electronic Resource]. — URL: https://dynamorio.org/page_drcov.html (visited on 09/08/2022).

71. IDA Pro [Electronic Resource]. — URL: https://hex-rays.com/ida-pro (visited on 09/08/2022).

72. Lighthouse: A Coverage Explorer for Reverse Engineers [Electronic Resource]. — URL: https://github.com/gaasedelen/lighthouse (visited on 09/08/2022).

73. *Niemetz, A.* Boolector 2.0 [Текст] / A. Niemetz, M. Preiner, A. Biere // Journal on Satisfiability, Boolean Modeling and Computation. — 2014. — Т. 9, № 1. — С. 53—58.

74. Cvc4 [Текст] / C. Barrett [и др.] // International Conference on Computer Aided Verification. — Springer. 2011. — С. 171—177.

75. Triton PR: Add Bitwuzla solver [Electronic Resource]. — URL: https://github.com/JonathanSalwan/Triton/pull/1062 (visited on 12/15/2022).

Список рисунков

Список таблиц