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»

Автореферат
диссертации на соискание учёной степени
кандидата технических наук

Москва — 2023

Работа выполнена в Федеральном государственном бюджетном учрежде нии науки Институт системного программирования им. В.П. Иванникова Российской академии наук.

Научный руководитель: кандидат технических наук
Федотов Андрей Николаевич

Официальные оппоненты: **Шабанов Борис Михайлович,**
член-корреспондент РАН, доктор технических наук, доцент,
директор Межведомственного Суперкомпьютерного Центра РАН,
заместитель директора по научной работе Федерального государственного учреждения «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

Дмитрий Олегович Маркин,
кандидат технических наук,
сотрудник ФГКВОУ ВО «Академия Федеральной службы охраны Российской Федерации»

Ведущая организация: Федеральное государственное учреждение ”Федеральный исследовательский центр Институт прикладной математики им. М.В. Келдыша Российской академии наук”

Защита состоится 19 октября 2023 г. в 16 часов на заседании диссертационного совета 24.1.120.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В. П. Иванникова Российской Академии Наук по адресу: 109004, г. Москва, ул. А. Солженицына, д. 25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Института системного программирования им. В. П. Иванникова Российской академии наук.

Автореферат разослан «___»_____ 2023 года.

Ученый секретарь
диссертационного совета
24.1.120.01,
кандидат физико-математических наук Зеленов С. В.

Общая характеристика работы

Актуальность темы. 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 косвенной адресации также важен, поскольку это вляет 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, четырех глав и заключения. Полный объем диссертации составляет 113 страниц, включая 5 рисунков и 7 таблиц. Список литературы содержит 75 наименований.

Содержание работы

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

Первая глава посвящена обзору работ по теме диссертации. Приводится описание метода динамической символьной интерпретации и обзор существующих инструментов, реализующих данный подход к динамическому анализу программ. В главе рассматриваются существующие подходы к решению проблемы моделирования косвенной адресации при анализе бинарных программ.

В разделе 1.1 описывается метод динамической символьной интерпретации для анализа бинарных программ. В разделе 1.2 приводится обзор основных инструментов динамической символьной интерпретации, внесших свой вклад в развитие метода. 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. Символьная интерпретация в нем происходит вдоль одного пути исполнения, определяемого конкретным исполнением программы на начальных входных данных.

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. Другим подходом, который называется онлайн-режимом, является выполнение символьной интерпретации одновременно вдоль нескольких путей исполнения. Однако из-за экспоненциального роста числа путей исполнения в реальных программах, такой подход потребляет много памяти и требует внушительных вычислительных ресурсов. Гибридный подход, предложенный в Mayhem, представляет собой компромисс между двумя стратегиями и заключается в чередовании двух режимов. Анализ начинается с онлайн-режима, и, по исчерпании определенного лимита вычислительных ресурсов, переходит в офлайн-режим.

В настоящее время все большее распространение приобретает подход гибридного фаззинга, при котором символьная интерпретация используется совместно с классическими инструментами фаззинга. Основная концепция гибридного фаззинга была представлена в инструменте QSYM. Она заключается в параллельном запуске фаззера и символьного интерпретатора, когда оба инструмента работают одновременно и помогают друг другу в режиме реального времени. QSYM реализует конкретно-символьное исполнение (конколик) в режиме офлайн. Основным преимуществом инструмента является скорость анализа, которой авторам удалось добиться за счет эффективной архитектуры. Альтернативный подход используется в гибридном фаззере SymCC, который вместо динамической бинарной инструментации внедряет код для проведения символьной интерпретации на этапе компиляции тестируемой программы. Инструмент SymQEMU использует платформу QEMU для инструментации программы. Инструментация осуществляется на этапе трансляции бинарного кода в промежуточное представление TCG IR, используемое в QEMU для эмуляции. Чтобы компенсировать потери в производительности, связанные с трансляцией кода, SymQEMU компилирует инструментированный промежуточный код обратно в бинарный код, который затем исполняется непосредственно на процессоре. Такой же подход, основанный на использовании QEMU для проведения инструментации, используется в гибридном фаззере Fuzzolic. 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.

В разделе 1.3 приводится описание существующих подходов к обработке косвенной адресации в динамической символьной интерпретации. В классическом алгоритме символьной интерпретации поддерживаются только прямые зависимости по данным – когда в вычислении результата операций непосредственно участвуют символьные переменные. Однако при анализе бинарного кода весьма распространена косвенная адресация, когда символьные данные участвуют в вычислении адресов, которые затем используются для доступа к нужным значениям. Такие зависимости в бинарном коде на высоком уровне абстракции, как правило, соответствуют табличным преобразованиям данных в программе. Среди таких зависимостей можно выделить косвенные переходы, которые имеют более двух направлений и организованы через доступ к специальной таблице, в которой находятся целевые адреса для перехода.

В инструменте QSYM для моделирования косвенной адресации в целом и инвертирования косвенных переходов в частности 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. Поскольку число различных значений символьного адреса может быть очень велико, инструмент перебирает не все возможные варианты, а последовательно обходит возможные значения в большую и меньшую сторону. Аналогичный подход используется в инструментах SymQEMU и SymCC. В них при доступе к памяти на чтение или запись по символьному адресу создается запрос к SMT-решателю с целью подобрать одно альтернативное значение адреса доступа. В SymCC, в отличие от SymQEMU, косвенные переходы обрабатываются отдельно от остальных косвенных зависимостей. Благодаря символьной интерпретации на уровне LLVM IR, SymCC имеет доступ к высокоуровневой информации о программном коде, в том числе и к операторам множественного ветвления `switch`, являющихся источниками таких конструкций в бинарном коде. Инструмент Fuzzolic также выполняет перебор значений символьного адреса и делает это для моделирования неявных зависимостей и косвенных переходов одинаково. Для перебора значений инструмент сначала применяет ряд мутаций к выражению символьного адреса, и только затем выполняет проверку в SMT-решателе на совместимость решения.

Полноценная поддержка косвенной адресации представлена в таких инструментах, как KLEE, angr и Mayhem. В инструменте KLEE символьная память представляется в виде набора независимых объектов в памяти программы. Каждый объект имеет свой базовый адрес и размер. KLEE интерпретирует как чтения, так и записи по символьному адресу, что позволяет инструменту моделировать все аспекты поведения программы. При обработке таких операций, KLEE прибегает к помощи SMT-решателя, чтобы определить, какие из объектов в памяти программы могут быть затронуты текущим доступом по символьному адресу. Процесс анализа программы разветвляется для каждого объекта, к которому возможен доступ.

Для улучшения масштабируемости анализа можно пожертвовать полнотой моделирования и интерпретировать только часть зависимостей, организованных с помощью косвенной адресации. Такой подход используется в инструментах angr и Mayhem, где интерпретируются только операции чтения по символьному адресу. В случае записи происходит конкретизация символьного адреса. В angr применяется плоская модель памяти, когда каждый байт представляется в качестве отдельного элемента. С помощью SMT-решателя определяется список возможных значений символьного адреса, на основе которого angr составляет формулу, устанавливающую взаимосвязь между значениями адреса и соответствующих ячеек памяти. В Mayhem определение границ чтения по символьному адресу определяется методом бинарного поиска с помощью запросов к SMT-решателю. В целях оптимизации, инструмент предварительно выполняет анализ VSA, а результаты запросов кэширует. Вместо перечисления всех возможных значений адреса подряд и составления вложенного `if-then-else` дерева как в angr, Mayhem конструирует формулу в виде бинарного дерева поиска над диапазоном адресов. Для уменьшения размеров и упрощения формулы применяется линеаризация – объединение части листьев дерева в одно линейное выражение.

Инструмент Sydr, разрабатываемый в ИСП РАН, реализует конкретно-символьное выполнение, для чего использует библиотеку символьной интерпретации Triton. Результатом моделирования инструкции в Triton является выражение, которое задает 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 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 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.

На основе исследования существующих подходов можно сделать следующие выводы. Все рассмотренные методы моделирования косвенной адресации можно разделить на два класса, каждый из которых имеет свои преимущества и недостатки. Методы ограниченной поддержки неявных зависимостей имеют высокую производительность анализа и, потому, используются в инструментах гибридного фаззинга, таких как QSYM, SymQEMU и Fuzzolic. Однако они не позволяют строить точную модель исполнения программы и пропускают потенциальные пути исполнения. С другой стороны, методы полноценной поддержки косвенной адресации позволяют моделировать все аспекты поведения программ и открывать множество новых путей исполнения. Такие подходы используются в инструментах динамической символьной интерпретации angr, KLEE (онлайн-режим анализа) и Mayhem (гибридный режим). Из-за большой дополнительной нагрузки на анализ, а также из-за жестких ограничений режима анализа онлайн, данные инструменты имеют проблемы с производительностью и не могут использоваться для гибридного фаззинга. Исходя из этого, в данной работе предлагается разработать метод полноценной обработки косвенной адресации, который позволяет учитывать такие зависимости в символьной модели программы, и при этом не приводить к значительному снижению производительности анализа, делая его применимым в гибридном фаззинге. Моделирование косвенных переходов позволяет напрямую открывать множество новых путей исполнения, поэтому такой случай предлагается обрабатывать отдельно и с использованием более точного метода. Для остальных случаев косвенной адресации необходимо повысить эффективность существующих подходов. Для этого необходимо разработать алгоритм поиска границ памяти, к которой осуществляется доступ по вычисляемому адресу, и определить наиболее эффективный способ построения ограничений при моделировании.

Вторая глава посвящена описанию предлагаемого метода поиска и моделирования косвенных переходов. Алгоритм поиска косвенных переходов в бинарном коде приведен в разделе 2.2, моделирование косвенных переходов описывается в раздел 2.3. Раздел 2.1 содержит подробное описание косвенных переходов в бинарном коде. Такие переходы представляют собой инструкции безусловной передачи управления. Для получения адреса назначения перехода в

них используются табличные преобразования. Чаще всего косвенные переходы образуются при компиляции оператора ветвления `switch`. Линейная последовательность операторов `if`, при соответствии определенным критериям, также может быть представлена в виде косвенного перехода. Такие переходы преобразуют условное выражение в индекс, который затем используется для получения нужного адреса перехода из заранее составленной таблицы. Таблица с целевыми адресами переходов обычно располагается в секциях `.rodata` или `.data` исполняемого файла. 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.

Существует 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`). Пример такого перехода представлен на листинге 2.

Листинг 2 Косвенный 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
```

Задача обнаружения косвенных переходов в бинарном коде возникает из-за того, что для повышения точности анализа такие конструкции должны моделироваться отдельно от остальных случаев косвенной адресации. Алгоритм поиска косвенных переходов в бинарном коде является частью разработанного метода, его описанию посвящен раздел 2.2. Разработанный метод поиска и моделирования косвенных переходов использует подход, который заключается в определении основной схемы работы косвенного перехода. Поиск выполняется в два этапа. Во-первых, производится быстрая и легковесная проверка, позволяющая определить, может ли базовый блок потенциально содержать косвенный 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. 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 алгоритма выполняется обратный слайсинг по инструкциям базового блока. Алгоритм 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, который соответствует определенным критериям (иметь вычисляемый адрес доступа, не зависящий от программного счетчика), то такая инструкция распознается как доступ к таблице переходов, и алгоритм завершается, а косвенный переход считается обнаруженным. В противном случае слайсинг продолжается до тех пор, пока не будет достигнуто начало базового блока. Результатом работы алгоритм поиска косвенных переходов в базовом блоке является адрес инструкции, осуществляющей доступ к таблице переходов.

В разделе 2.3 приводится подробное описание разработанного метода моделирования косвенных переходов. Метод заключается в построении таких символьных ограничений для косвенных переходов, 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.

В результате работы данного метода, косвенные переходы инвертируются наряду с остальными условными переходами программы.

Таблица переходов представляет собой набор адресов или смещений, записанный в памяти программы. 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 косвенных 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).

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. Символьный интерпретатор отслеживает символьное состояние программы, производит интерпретацию символьных инструкций и построение предиката пути. Эти данные используются им для построения ограничений пути, необходимых для корректного инвертирования условного перехода, организованного через косвенную адресацию. В методе моделирования косвенных переходов 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 косвенном 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.

Полученные ограничения добавляются в предикат пути как условный переход с несколькими направлениями перехода. При этом ограничения добавляются на инструкции передачи управления, а не в точке доступа к таблице. При инвертировании перехода запросы к SMT-решателю будут создаваться для каждого направления, кроме соответствующего текущему пути исполнения. Таким образом, инвертирование косвенного перехода происходит практически идентично обычным условным переходам.

В разделе 2.3.3 приведены результаты экспериментальной оценки разработанного метода поиска и моделирования косвенных переходов. Для оценки метода запуск анализа проводился дважды: с использованием стандартного инструмента Sydr и с включенным методом инвертирования косвенных переходов. В таблице 1 приведены результаты эксперимента. 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

Таблица 1 — 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

(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. По числу инвертированных условных переходов и корректности инвертирования можно сделать вывод об общем эффекте от поддержки косвенных переходов. В целом, заметный прирост корректно инвертированных переходов наблюдается на шести примерах (`jasper`, `muraster`, `histmap_pgm`, `readelf`, `yices-smt2`, `yodl`). Еще одним интересным параметром анализа может быть соотношение между числом инвертированных и правильно инвертированных условных переходов. 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.

Третья глава посвящена описанию разработанного метода моделирования чтений памяти по символьно вычисляемому адресу. Метод 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.

Алгоритм определения области памяти рассматривается в разделе 3.2, описание разработанного алгоритма моделирования символьных чтений приводится в разделе 3.4.

В разделе 3.1 подробно рассматривается косвенная адресация в бинарном коде. Чтение памяти по символьному адресу является частным случаем косвенной адресации, далее в работе для описания данных операций применяются такие термины, как ”символьные указатели”и ”символьные адреса”.

В разделе 3.2 описываются различные подходы к определению границ участка памяти, к которому производится доступ по символьному адресу. Результатом моделирования чтений памяти по символьно вычисляемому адресу является предикат пути, для построения которого необходимо определить область памяти, в пределах которой может изменяться символьный адрес. В рамках метода было реализовано несколько способов определения границ доступа к памяти:

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

Определение области памяти при моделировании символьного доступа является важным этапом метода моделирования символьных чтений памяти, поскольку корректность и размер выбранных границ доступа сильно влияют на производительность 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 подход является самым быстрым и легковесным из всех, но при этом имеет наименьшую точность.

Алгоритм бинарного поиска границ с помощью запросов к SMT-решателю заключается в определении максимального и минимального значения, которое может принимать символьное выражение адреса в контексте текущего пути исполнения. Бинарный поиск точной левой и правой границы памяти осуществляется над диапазоном памяти в 100 ячеек доступа в соответствующую сторону от текущего адреса доступа. 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. Применение SMT-решателя невозможно в том случае, когда построенный предикат пути накладывает слишком сильные ограничения на выражение символьного адреса. Также алгоритм бинарного поиска требует множества дополнительных запросов

к SMT-решателю во время анализа, что оборачивается значительным падением производительности анализа, практически сводя на нет все преимущества точного определения границ.

Листинг 3 Анализ символьного выражения адреса

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

Алгоритм анализа символьного выражения адреса 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. 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. Для определения константной части адреса алгоритм выполняет рекурсивный обход SMT-формулы, представляющей символьный адрес (листинг 3). Во время этого обхода собираются все константные составляющие, удовлетворяющие определенным условиям. Сложив все обнаруженные таким образом значения, можно получить предполагаемую левую границу доступа. 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, поэтому при интерпретации символьных указателей в ходе анализа применяется именно этот подход.

В разделе 3.3 приводится описание существующих подходов к интерпретации символьного чтения. Следующим шагом после определения участка памяти, из которого производится чтение, является конструирование соответствующего ограничения пути. Оно должно ставить в соответствие результат чтения в зависимости от символьного выражения адреса. В рамках работы было исследовано и реализовано для сравнения три существующих 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.

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

Листинг 4 Линеаризованное BST-дерево

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

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. Во многих случаях 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. Получившаяся в итоге формула, так же как и стандартное BST-дерево, имеет небольшую вложенность, но при этом гораздо меньшие размеры. Логическая организация такого дерева представлена на листинге 4.

В разделе 3.4 приводится описание разработанного алгоритма моделирования чтений памяти по символьно вычисляемому адресу. Для построения соответствующих ограничений пути был предложен комбинированный подход, который является улучшением существующего метода построения линеаризованных BST-деревьев. Разработанный комбинированный алгоритм устраняет такие недостатки оригинального подхода, как отсутствие поддержки символьных значений в читаемой области памяти и слишком большие коэффициенты линейных уравнений, ведущих к увеличенному потреблению памяти. Разработанный подход также включает в себя дополнительные улучшения, направленные на оптимизацию и повышение эффективности алгоритма при его применении в динамическом анализе.

Одним из улучшений является обработка символьных чтений из небольшого участка памяти. Иногда границы области памяти определяются так, что в ней содержится небольшое число элементов доступа. Конструируемое линеаризованное BST-дерево не будет давать заметного улучшения производительности в таком случае, 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. Алгоритм 1 описывает процесс линеаризации в комбинированном методе моделирования символьных указателей.

Разработанный алгоритм моделирования также позволяет учитывать символьные ячейки памяти и при этом сохранять все преимущества линеаризованных BST-деревьев. Для этого производится раздельное моделирование 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.

В разделе 3.5 приводится экспериментальная оценка разработанного метода моделирования чтений по символьному адресу. Целью одного из проведенных

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
```

Таблица 2 — Сравнение производительности Sydr при работе в двух режимах

Application	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

экспериментов является сравнение производительности инструмента анализа при работе в стандартном режиме и с поддержкой символьных указателей. Итоги такого сравнения приведены в таблице 2. Результаты показывают, что и время построения предиката пути, и общее время анализа увеличилось для всех программ. Для оценки производительности анализа была рассчитана скорость обработки запросов в SMT-решателе (столбец *Queries/min*). Скорость решения обычных запросов, за рядом исключений, в разы выше тех, которые содержат ограничения для символьных чтений.

Еще один эксперимент был проведен для оценки увеличения тестового покрытия программы в результате интерпретации косвенной адресации. Для этого с помощью инструмента Ida Pro и его плагина Lighthouse было рассчитано покрытие, достигнутое на сгенерированных входных данных, полученных в результате двух запусков инструмента Sydr. Использование Ida Pro и Lighthouse позволяет измерить покрытие по базовым блокам в процентном соотношении от всего кода программы. Также возможно вычисление разницы между покрытиями, собранными на разных корпусах входных данных. Результаты эксперимента приведены в таблице 3. Столбец *Coverage* показывает достигнутое покрытие при анализе одного пути в стандартном режиме и с поддержкой символьных указателей. В столбце *Coverage diff* приведена разница между достигнутыми в двух режимах покрытиями. Результаты показывают, что включение поддержки символьных указателей всегда приводит к обнаружению нового уникального покрытия, при этом для большинства программ никаких потерь покрытия по сравнению с обычным анализом нет.

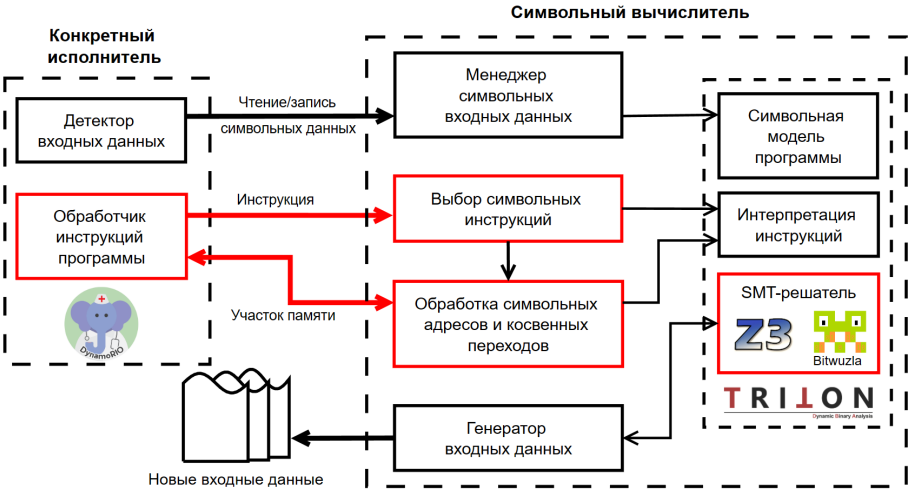
Таблица 3 — Оценка покрытия

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

Четвертая глава посвящена описанию программной реализации методов поиска и моделирования косвенных переходов и чтений памяти по символьно вычисляемым адресам. В разделе 4.1 приводится описание инструмента динамической символьной интерпретации Sydr, в котором были реализованы разработанные методы. 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. 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. На рисунке 1 приведена схема работы 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

Рис. 1 — Схема инструмента Sydr



формула для инвертирования его направления и выполняется соответствующий запрос к SMT-решателю. В качестве SMT-решателя Triton использует Z3. На основе моделей, получаемых от SMT-решателя происходит генерация новых входных данных, что и является основным результатом работы инструмента.

Раздел 4.2 посвящен описанию реализации метода моделирования косвенной адресации. Для этого в схему работы инструмента были внесены такие изменения, что для обработки инструкций с символьными указателями и косвенными переходами выполнялся двойной обмен сообщениями между процессами конкретного исполнителя и символьного вычислителя. Такое взаимодействие между процессами анализа отображено на рисунке 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.

Для обработки символьных указателей 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. 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.

Раздел 4.3 посвящен применению SMT-решателей в динамической символьной интерпретации. 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. В работе [2] приводится исследование о применении различных SMT-решателей для задач статической и динамической символьной интерпретации. В исследовании принимали участие SMT-решатели Yices2, Boolector, STP, Z3 и CVC4. Результаты измерения показывают, 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 и Boolector. Недостатком Yices2 является наличие неточных и ошибочных решений запросов, что имеет критическое значение для символьной интерпретации. Исходя из этих соображений, для использования в инструменте был выбран SMT-решатель Bitwuzla - более современный аналог от разработчиков Boolector.

В **заключении** приведены основные результаты работы, которые 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.

Публикации автора по теме диссертации

1. Anxiety: a dynamic symbolic execution framework [Text] / A. Gerasimov [et al.] // 2017 Ivannikov ISPRAS Open Conference (ISPRAS). — IEEE, 2017. — P. 16—21. — (Scopus, WoS).
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. — (Scopus, WoS).
3. Sydr: Cutting edge dynamic symbolic execution [Text] / A. Vishnyakov [et al.] // 2020 Ivannikov ISPRAS Open Conference (ISPRAS). — IEEE, 2020. — P. 46—54. — (Scopus, WoS).
4. *Kuts, D.* Towards symbolic pointers reasoning in dynamic symbolic execution [Text] / D. Kuts // 2021 Ivannikov Memorial Workshop (IVMEM). — IEEE, 2021. — P. 42—49. — (Scopus, WoS).
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).