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

На правах рукописи

Вишняков Алексей Вадимович

Поиск ошибок в бинарном коде методами динамической символьной интерпретации

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

Диссертация на соискание учёной степени
кандидата физико-математических наук

Научный руководитель:
кандидат технических наук
Федотов Андрей Николаевич

Москва — 2022

Оглавление

Стр.

Введение 5

Глава 1. Обзор работ 13
1.1 Динамическая символьная интерпретация 13
1.2 Существующие инструменты символьной интерпретации 15
1.2.1 KLEE (2008) 15
1.2.2 SAGE (2008) 16
1.2.3 S^2E (2011) 18
1.2.4 Mayhem (2012) 19
1.2.5 Triton (2015) 21
1.2.6 Driller (2016) 22
1.2.7 QSYM (2018) 23
1.2.8 SymCC (2020) 24
1.2.9 SymQEMU (2021) 25
1.2.10 Fuzzolic (2021) 26
1.3 Преодоление избыточной и недостаточной помеченности 28
1.3.1 Устранение избыточных ограничений в KLEE 29
1.3.2 Оптимистичные решения в QSYM 29
1.3.3 Пропуск базовых блоков в QSYM 30
1.3.4 Моделирование символьных адресов 31
1.3.5 Частичная пометка символьного входа в LeanSym 32
1.3.6 Моделирование семантики функций 33
1.3.7 Выводы 36
1.4 Поиск ошибок с помощью символьной интерпретации 39
1.4.1 Санитайзеры 39
1.4.2 Поиск ошибок одновременно с открытием путей в KLEE . . 41
1.4.3 Автоматическая генерация эксплойтов в Mayhem 41
1.4.4 Гибридное тестирование с прицелом на поиск ошибок в SAVIOR 42
1.4.5 Поиск целочисленного переполнения в IntScope 44
1.4.6 Обнаружение уязвимостей целочисленного переполнения с помощью графов программ 45

Стр.

1.4.7 Фаззинг с обратной связью по санитайзерам в ParmeSan . . 47

1.4.8 Выводы 48

Глава 2. Алгоритм слайсинга предиката пути 51

2.1 Построение предиката пути 51

2.2 Алгоритм устранения избыточных ограничений из предиката пути 55

2.3 Исследование свойств и характеристик представленного алгоритма 60

2.4 Преодоление избыточной и недостаточной помеченности 65

2.5 Экспериментальная оценка 67

Глава 3. Метод моделирования семантики функций 72

3.1 Моделирование функций стандартной библиотеки 73

3.1.1 Пропуск символьной интерпретации функции 74

3.1.2 Моделирование строкового сравнения и поиска 76

3.1.3 Моделирование преобразования строки в число 79

3.2 Экспериментальная оценка 81

Глава 4. Метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности 85

4.1 Метод построения предикатов безопасности 85

4.1.1 Обнаружение ошибок выхода за границу массива 88

4.1.2 Обнаружение ошибок целочисленного переполнения 90

4.2 Метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности в гибридном фаззинге 95

4.3 Экспериментальная оценка точности метода построения предикатов безопасности на наборе тестов Juliet 96

4.4 Апробация методов путем поиска новых ошибок в проектах с открытым исходным кодом 101

4.4.1 FreeImage 102

4.4.2 xlnt 104

4.4.3 unbound 105

4.4.4 hdp 106

4.4.5 miniz 107

4.4.6 EXRS 108

Стр.

4.4.7 Goblin 108
4.4.8 OpenJPEG 109
4.4.9 Poppler 110
4.4.10 Rizin 110
4.4.11 TensorFlow 111

Глава 5. Реализация предложенных методов в программных инструментах 112

5.1 Инструмент динамической символьной интерпретации Sydr 112
5.1.1 Реализация слайсинга предиката пути 115
5.1.2 Реализация моделирования семантики функций 116
5.1.3 Реализация предикатов безопасности 117
5.2 Реализация метода автоматизированного поиска ошибок при помощи символьных предикатов безопасности в контексте гибридного фаззинга 117

Заключение 119

Список литературы 120

Список рисунков 130

Список таблиц 131

Введение

Современное программное обеспечение стремительно развивается. Кодовая база продуктов постоянно увеличивается. Новый код неизбежно приносит с собой ошибки и уязвимости [1]. Согласно отчету компании Synopsys 2022 года в 2400 коммерческих кодовых базах 81 % проанализированных программ содержали хотя бы одну уязвимость [2].

Уязвимости, являющиеся следствием ошибок в коде программ, могут приводить к серьезным последствиям: денежным убыткам, разрыву связи, отказу в обслуживании сервисов, компрометации шифрованной переписки, утечке персональных данных и др. Число найденных уязвимостей растет с каждым годом [3]. Так в 2021 году было сообщено о 20169 уязвимостях (по сравнению с 18325 в 2020 г.) в интернет браузерах, редакторах офисных документов, мобильных и настольных операционных системах, сетевых маршрутизаторах. Активное развитие технологий «умного дома» и «интернета вещей» расширяет поверхность атаки. Теперь даже обычные бытовые предметы (светильники, обогреватели, холодильники, чайники, водопроводные системы, роботы-пылесосы) могут быть захвачены злоумышленником. Более того, была показана возможность эксплуатации медицинского оборудования, а именно имплантируемых сердечных дефибрилляторов [4].

Наличие ошибок — это неизбежное свойство «живой» программы, разработка которой продолжается. Однако не каждая ошибка может быть эксплуатируема атакующим. Кроме того, существуют различные механизмы защиты операционных систем, которые затрудняют эксплуатацию. Например, широко применяется рандомизация адресного пространства программы [5], которая усложняет переносимость эксплойта на разные компьютеры.

В настоящее время распространен подход для обнаружения ошибок и уязвимостей непосредственно во время процесса разработки программного обеспечения — безопасный цикл разработки ПО (SDL), который становится стандартом индустрии [6—8]. Следуя практикам SDL, разработчики обязаны применять различные инструменты анализа кода для повышения качества и безопасности продукта. Таким образом, многие ошибки обнаруживаются во время разработки до того, как программа введена в эксплуатацию. Ошибки обнаруживают широко распространенными методами статического [9—11] и динамического [12] анализа

программ. Такие инструменты могут исследовать программу как на уровне исходного кода [10; 11], так и машинного [9; 12].

Инструменты статического анализа, не запуская приложение, исследуют его код на предмет наличия в нем слабых мест. В результате работы анализатора будет выдан набор предупреждений с пояснениями, в чем заключаются ошибки. Следует отметить, что ни один анализатор общего применения не может продемонстрировать стопроцентную точность. Таким образом, анализатор может выдавать ложные срабатывания или же наоборот пропускать некоторые ошибки. Поэтому пользователь просматривает выданные предупреждения и размечает, какие из них являются истинными ошибками.

Динамический подход предполагает либо реальный запуск анализируемой программы, либо интерпретацию (или эмуляцию) ее кода. При таком подходе ошибочные ситуации обнаруживаются во время выполнения программы. Такой подход, как правило, предоставляет возможность воспроизвести проявление дефекта.

Во время безопасного цикла разработки ПО непрерывно применяется фаззинг-тестирование [13; 14]. Фаззинг является динамическим методом анализа программы, во время которого порождаются новые входные данные. Фаззер мутирует входные данные программы и наблюдает за ее выполнением. Таким образом, могут быть получены входные данные программы, на которых она зависает или аварийно завершается.

Повсеместно распространен фаззинг с обратной связью по покрытию [13; 14]. При таком фаззинге не только осуществляется наблюдение за результатом выполнения программы, но и собирается информация о покрытом коде. Для организации обратной связи используются генетические алгоритмы. Наиболее «приспособленными» считаются входные файлы, открывающие как можно больше нового кода. Входные данные «скрещиваются» между собой и оставляются наиболее «приспособленные» с точки зрения покрытия.

Более продвинутые методы гибридного фаззинга [15—20] помимо обратной связи по покрытию применяют методы динамической символьной интерпретации (DSE, dynamic symbolic execution, где execution по сути является интерпретацией программы) [16—25], которые позволяют обнаруживать сложные состояния программы, труднодоступные для обычного фаззинга. Для этого строится математическая модель программы, которая при генерации новых входных данных позволяет учитывать семантику программы.

Благодаря динамической символьной интерпретации гибридный фаззинг решает две задачи: (1) генерации новых входных данных для расширения тестового покрытия программы и (2) обнаружения ошибок. Процесс фаззинга может выглядеть следующим образом. Динамическая символьная интерпретация помогает исследовать новые состояния программы благодаря инвертированию условных переходов, встреченных на пути выполнения. Более того, DSE позволяет накладывать дополнительные ограничения для генерации новых входных данных, активирующих дефекты. Все полученные таким способом входные данные передаются фаззеру, который в свою очередь проверяет, открывают ли они новый код или же приводят к ошибкам. Таким образом, фаззер покрывает новые пути и обнаруживает ошибки благодаря учету семантики программы.

Динамический анализ бинарного кода дополняет классические методы статического анализа и фаззинга: появляется возможность обнаружения новых классов ошибок и уязвимостей (например, внесенных на этапе компиляции). Анализ на уровне бинарного кода позволяет разработать решение, применимое к широкому классу программ — достаточно скомпилировать программу для ее анализа. Кроме того, такой подход позволяет анализировать программы, у которых отсутствует исходный код.

В данной работе рассматривается задача поиска ошибок в бинарном коде методами динамической символьной интерпретации в контексте гибридного фаззинга. Требуется разработать методы, позволяющие обнаруживать ошибки неопределенного поведения и работы с памятью в результате динамической символьный интерпретации программы с входными данными, при запуске на которых ошибка изначально не проявляется. Такие методы должны генерировать входные данные для воспроизведения ошибок и автоматически выделять истинные срабатывания.

Отдельно стоит отметить вопрос борьбы с недостаточной и избыточной помеченностью во время динамической символьной интерпретации, когда в символьной модели программы часть формул может отсутствовать, или наоборот могут присутствовать избыточные ограничения. Решение проблем недостаточной и избыточной помеченности напрямую влияет на точность генерируемых входных данных и воспроизведение ошибок.

Существующие решения поиска ошибок с помощью символьной интерпретации либо работают на уровне исходного кода, либо используют статический анализ, а также часто сосредотачиваются на поиске некоторого одного типа оши-

бок. Данная работа решает поставленные задачи специфичные непосредственно для динамического анализа бинарного кода (в частности, задачи определения границ массивов и знаковости арифметических операций). Аналогичные работы не уделяют должного внимания точности генерируемых входных данных для воспроизведения ошибки. Отдельно следует отметить, что часть аналогичных решений являются закрытыми и недоступны для использования. Во время поиска ошибок моделируется семантика библиотечных функций, что в частности позволяет обнаруживать переполнение буфера на уровне вызова функций. Более того, данная работа предлагает автоматизированный комплексный метод поиска ошибок, применяемый в контексте гибридного фаззинга и позволяющий обнаруживать новые ошибки различного типа (деление на нуль, целочисленное переполнение, выход за границу массива и др.) в сложных программных системах. В данной работе дополнительно производится автоматическая верификация срабатываний и методы придерживаются точности 95.59 % на наборе тестов Juliet [26].

Целью данной работы является разработка метода поиска ошибок в бинарном коде методами динамической символьной интерпретации. Метод должен иметь возможность применения в контексте гибридного фаззинга, сочетающего в себе динамическую символьную интерпретацию и фаззинг с обратной связью по покрытию.

Для достижения поставленной цели решаются следующие **задачи**:

1. Разработать метод моделирования семантики функций для расширения анализируемых символьных состояний в рамках одного пути выполнения.
2. Разработать алгоритм слайсинга предиката пути, который устраняет избыточные ограничения пути и ускоряет поиск моделей для предикатов математическим решателем. Исследовать свойства алгоритма слайсинга пути и доказать соответствующие теоремы.
3. Разработать символьные предикаты безопасности, позволяющие обнаруживать ошибки деления на нуль, выхода за границу массива и целочисленного переполнения в бинарном коде.
4. Разработать метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности, который анализирует программу на корпусе входных файлов, полученных в результате гибридного фаззинга, и позволяет генерировать входные данные для проявления новых

ошибок, а также производит верификацию полученных срабатываний на санитайзерах.

Научная новизна:

1. Предложенный алгоритм слайсинга предиката пути, который устраняет избыточные ограничения, полученные в результате динамической символьной интерпретации бинарного кода. Для алгоритма доказаны теоремы о его конечности, корректности, и проведена оценка его вычислительной сложности.
2. Разработанный метод построения предикатов безопасности для ошибок деления на нуль, выхода за границу массива и целочисленного переполнения во время динамической символьной интерпретации пути выполнения программы.
3. Разработанный метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности. Метод применяется после гибридного фаззинга. Сгенерированные входные данные для активации дефектов валидируются с помощью санитайзеров.

Теоретическая и практическая значимость Теоретическая значимость заключается в разработанных методах построения предикатов безопасности для обнаружения ошибок деления на нуль, выхода за границу массива и целочисленного переполнения. Данные методы применимы для анализа больших программных систем в контексте гибридного фаззинга. Кроме того, был разработан алгоритм слайсинга предиката пути, устраняющий избыточные ограничения во время динамической символьной интерпретации. Были формально исследованы свойства алгоритма и доказаны необходимые теоремы.

Практическая значимость заключается в том, что разработанные методы позволяют обнаруживать новые программные дефекты и генерировать входные данные для их воспроизведения. Метод позволяет автоматически выделить истинно положительные срабатывания символьных предикатов безопасности. Предложенные методы встраиваются в системы непрерывной интеграции (CI) в рамках следования безопасному циклу разработки ПО (SDL). Разработанные методы позволили обнаружить новые ошибки в различных проектах с открытым исходным кодом. Разработанные методы используется в Центре доверенного искусственного интеллекта ИСП РАН. Методы могут применяться в будущем для сертификации и безопасного цикла разработки ПО.

Методология и методы исследования. Результаты диссертационной работы получены на базе использования методов динамического анализа программ, динамической символьной интерпретации, анализа бинарного кода, теории компиляторов, анализа потока данных, гибридного фаззинга и динамической двоичной трансляции. Математическую основу исследования составляют математическая логика, дискретная математика, теория множеств и теория алгоритмов.

Основные положения, выносимые на защиту:

1. Алгоритм слайсинга предиката пути, позволяющий устранять избыточные ограничения во время динамической символьной интерпретации бинарного кода. Алгоритм основан на анализе зависимостей символьных переменных по данным.
2. Метод построения предикатов безопасности для обнаружения ошибок деления на нуль, целочисленного переполнения и выхода за границу массива во время динамической символьной интерпретации пути выполнения программы. Метод добавляет к предикату пути дополнительные ограничения, которые описывают достаточные условия возникновения ошибки.
3. Метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности во время динамического анализа программ на корпусе входных файлов, полученного в результате гибридного фаззинга.

Достоверность полученных результатов обеспечивается формальным исследованием свойств предложенного алгоритма слайсинга предиката пути: доказаны теоремы о его конечности, корректности, и проведена оценка его вычислительной сложности. Дополнительно была проведена экспериментальная оценка точности и скорости работы алгоритма. Было показано, что оба показателя растут при применении алгоритма во время динамической символьной интерпретации. Точность метода построения предикатов безопасности для обнаружения ошибок была измерена на наборе тестов Juliet. Качественная оценка метода автоматизированного поиска ошибок при помощи символьных предикатов безопасности заключается в том, что с помощью данного метода были обнаружены новые ошибки в различных проектах с открытым исходным кодом.

Апробация работы. Основные результаты работы докладывались на конференциях:

1. Открытая конференция ИСП РАН имени В.П. Иванникова. Москва. 11 декабря 2020.
2. Открытая конференция ИСП РАН имени В.П. Иванникова. Москва. 3 декабря 2021.
3. 31-я научно-техническая конференция «Методы и технические средства обеспечения безопасности информации» памяти П.Д. Зегжды (МиТСОБИ). Санкт-Петербург. 28–29 июня 2022.

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

Публикации. Основные результаты по теме диссертации изложены в 6 печатных изданиях, 2 из которых изданы в журналах, рекомендованных ВАК [27; 28], 2 — в периодических научных журналах, индексируемых Web of Science и Scopus [25; 29], 2 — в тезисах докладов [30; 31]. Зарегистрированы 2 программы для ЭВМ [32; 33]. В работе [27] автором предложены методы борьбы с недостаточной и избыточной помеченностью при построении предиката пути по трассе машинных инструкций путем добавления дополнительных ограничений и пропуска функций аллокаторов соответственно. В статье [29] автором представлен метод моделирования семантики функций. В статье [25] автором предложен разработанный алгоритм слайсинга предиката пути, устраняющий избыточные ограничения, и проведена его экспериментальная оценка. В статье [29] автором представлен метод построения предикатов безопасности для ошибок деления на нуль, целочисленного переполнения и выхода за границу массива, а также проведена оценка его точности на наборе тестов Juliet. В статье [28] автором описан обобщенный автоматизированный метод поиска ошибок при помощи символьных предикатов безопасности и представлены найденные новые ошибки в проектах с открытым исходным кодом. В докладе [30] автором описывается практическое применения автоматизированного метода поиска ошибок к проектам с открытым исходным кодом. В докладе [31] автором представляется применение разработанного метода поиска ошибок для повышения безопасности фреймворка машинного обучения TensorFlow, а также внедрение метода в системы непрерывной интеграции (CI) в рамках безопасного цикла разработки ПО (SDL). Зарегистрирована программа для ЭВМ [32], в которой автором реализован алгоритм слайсинга предиката пути, а также методы моделирования семантики

функций и построения предикатов безопасности. Зарегистрирована программа для ЭВМ [33], содержащая реализованный автором автоматизированный метод поиска ошибок при помощи символьных предикатов безопасности в результате анализа программы на корпусе входных файлов, полученных в результате гибридного фаззинга.

Объем и структура работы. Диссертация состоит из введения, 5 глав и заключения. Полный объём диссертации составляет 131 страницу, включая 1 рисунок и 9 таблиц. Список литературы содержит 111 наименований.

Глава 1. Обзор работ

1.1 Динамическая символьная интерпретация

В 1976 году J. C. King предложил метод символьной интерпретации программ для автоматической генерации тестов [21]. Реальные входные данные подменяются математическими символами — свободными символьными переменными. Преобразования в программе описываются в виде формул над этими переменными и константами. Условные ветвления в свою очередь порождают предикаты, истинные для некоторого выбранного направления. Следовательно, пути в программе можно описывать конъюнкцией выполненных условий переходов на этом пути. Такая конъюнкция называется предикатом пути. Если полученную конъюнкцию подать на вход математическому решателю [34; 35], то он подберет модель — подстановку конкретных значений символьным переменным, при которой предикат истинный. Выполнение программы на конкретных входных данных из модели пройдет по описанному предикатом пути. Чтобы отклонить выполнение программы в другую сторону, можно взять условие перехода с отрицанием. Таким образом, перебирая пути в программе и символьно их интерпретируя, можно получать новые входные данные для тестирования. Однако следует отметить, что такому подходу присущ экспоненциальных рост числа путей при переборе.

Как это часто бывает в науке, теоретическая идея символьной интерпретации смогла получить свое реальное практическое применение только спустя десятки лет. Свое развитие область получила в связи с ростом производительности вычислительной техники. Это обстоятельство позволило математическим решателям подбирать новые входные данные для сложных программных систем за удобоваримое время. Так в 2005 году Godefroid и др. [36] предложили систему DART для автоматической генерации направленных тестов с помощью символьной интерпретации.

Затем в том же году Koushik и др. [37] предложили метод конколической (concolic) интерпретации в инструменте CUTE. При таком подходе направление (путь) при символьной интерпретации задается некоторым конкретным выполнением. Направления условных переходов и значения некоторых констант берутся

непосредственно из реальных значений на фиксированном пути программы. Иначе данный метод называют динамической символьной интерпретацией.

В 2006 году Cadar и др. [38] представят миру свою разработку EXE, которая генерирует входные данные, приводящие к аварийному завершению программы. В дальнейшем эта работа разовьется в символьный интерпретатор KLEE [39] для сложных программных систем. В 2008 году Godefroid и др. [40] впервые предложат идею использовать символьную интерпретацию для фаззинга. Авторы воплотят ее в инструменте SAGE, который с помощью динамической символьной интерпретации инвертирует условные переходы из реального выполнения программы. Перебирая пути выполнения, SAGE способен обнаруживать аварийные завершения программы.

В 2010 году Schwartz и др. [41] формализовали динамическую символьную интерпретацию. Динамическая символьная интерпретация исследует вариацию изначальных входных данных на некотором фиксированном пути выполнения. Изначально каждый байт входных данных моделируется свободной символьной переменной. Каждая инструкция моделируется SMT [42] формулой над константами и символьными переменными в соответствии со своей операционной семантикой. Символьное состояние содержит текущее отображение регистров и памяти в формулы. Все изменения регистров (или памяти) обновляют символьное состояние. Условия переходов на исследуемом пути представляются булевыми предикатами и формируют предикат пути. Таким образом, предикат пути содержит ограничения, которые описывают исследуемый путь. Решением конъюнкции всех таких ограничений являются входные данные, которые проведут программу по тому же пути выполнения. Для инвертирования некоторого перехода необходимо взять его ограничение с логическим отрицанием. Таким образом, можно открывать новые пути выполнения, ответвляясь от изначального. Описанная Schwartz и др. [41] модель по сути использовалась в инструменте Mayhem [43; 44] (2011–2012 гг.). В дальнейшем мы будем использовать именно эту терминологию, когда будем говорить о динамической символьной интерпретации.

В 2016 году Stephens и др. [16] реализовали подход гибридного фаззинга [15] в инструменте Driller. В качестве фаззера использовался AFL [45] с обратной связью по покрытию. Фаззер позволяет быстро открывать новые пути выполнения. Однако, когда фаззер долгое время не открывает новые пути, запускается более медленная динамическая символьная интерпретация (DSE). Полученные новые

входные данные добавляются в корпус фаззера и расширяют его покрытие. Таким образом, DSE используется для помощи фаззеру, когда тот перестал открывать новые пути.

В 2018 году Yun и др. [17] показали, что динамическая символьная интерпретация может быть достаточно быстрой для одновременной работы фаззера и DSE. Гибридный фаззинг в инструменте QSYM показал себя эффективнее, чем обычный фаззинг с обратной связью по покрытию. Дальнейшие работы [18—20] еще сильнее разовьют скорость динамической символьной интерпретации в гибридном фаззинге. А в 2021 году независимо будет подтверждено, что гибридный фаззинг, совмещающий в себе один динамический символьный интерпретатор и один фаззер с обратной связью по покрытию AFL++ [14], открывают новое покрытие кода быстрее двух фаззеров [46].

1.2 Существующие инструменты символьной интерпретации

1.2.1 KLEE (2008)

В 2008 году Cadar и др. [39] опубликовали статью, описывающую внутреннее устройство символьной виртуальной машины KLEE, которая является широко используемым фреймворком символьной интерпретации с открытым исходным кодом. Инструмент KLEE по сути появился в результате переосмысления архитектуры системы EXE [38] 2006 года от тех же авторов. Мы опустим рассмотрение EXE, т. к. KLEE включает в себя лучшие идеи из EXE и полностью его заменяет.

KLEE осуществляет символьную интерпретацию на уровне промежуточного представления LLVM [47], что позволяет анализировать все, что транслируется в LLVM-IR (в частности такие компилируемые языки, как C, C++, Go, Rust, Swift). Более того, KLEE эмулирует окружение операционной системы Linux. Таким образом, часть системного окружения обрабатывается символьно.

KLEE совмещает в себе операционную систему для символьных процессов и интерпретатор. Каждый процесс описывается символьным состоянием, которое содержит регистры, стек, кучу, счетчик инструкций и предикат пути. Анализиру-

емая программа компилируется в промежуточное представление LLVM, которое в дальнейшем символьно интерпретируется.

Интерпретатор переключается между символьными процессами и интерпретирует по одной инструкции в контексте выбранного процесса. Регистры, стек и куча в символьных состояниях хранят деревья выражений, листьями которых являются символьные переменные и константы. Остальные вершины содержат операции из LLVM-IR. В результате интерпретации инструкций символьное состояние обновляется деревьями выражений, которые в свою очередь могут использовать другие выражения из состояния. Для ускорения интерпретации значения выражений над константами вычисляются конкретно.

На каждом условном переходе KLEE проверяет, какое направление достижимо и соответственно обновляет счетчик инструкций в состоянии процесса. Если же достижимы оба направления перехода, то KLEE копирует состояние для дальнейшего исследования обоих путей. Для повышения производительности используется механизм аналогичный copy-on-write во время системного вызова `fork`.

Для оптимизаций запросов к решателю KLEE применяет тождественные преобразования к выражениям, которые уменьшают размер формул и упрощают их решение. Более того, KLEE уменьшает набор ограничений в предикате пути за счет сужения интервалов значений символьных переменных. Например, $x < 10 \wedge x = 5$ можно упростить до $x = 5$. Также KLEE кэширует ответы решателя для уменьшения времени, проводимого в решателе:

1. если подмножество ограничений не имеет решений, то и любое множество, которое содержит это подмножество тоже не имеет решений,
2. когда у множества ограничений есть решение, оно так же удовлетворяет любому его подмножеству,
3. если подмножество ограничений имеет решение, то вероятно это же решение будет удовлетворять его надмножеству.

1.2.2 SAGE (2008)

Компания Microsoft в 2008 году предложила использовать динамическую символьную интерпретацию для фаззинг тестирования приложений [40]. Реа-

лизацию эта идея получила в инструменте SAGE, который позволяет находить в программах ошибки и уязвимости. SAGE сначала записывает трассу выполнения анализируемой программы, запущенную на корректных входных данных. Затем происходит символьная интерпретация бинарного кода из трассы (последовательности инструкций). В результате интерпретации собирается предикат пути, состоящий из условий переходов на конкретной трассе. Полученные условия берутся с отрицанием один за другим и отдаются на вход математическому решателю. Таким образом, генерируются новые входные данные, которые проводят программу по разным путям выполнения. Описанный процесс повторяется с использованием эвристик, максимизирующих покрытие кода.

С целью максимизации покрытия, поиска ошибок и борьбы с экспоненциальным ростом путей SAGE применяет алгоритм генерационного поиска. Алгоритм позволяет обрабатывать большие входные данные (тысячи символьных переменных) и исследовать глубокие пути выполнения (сотни миллионов инструкций).

Алгоритм выполняет следующие действия. Запуск программы проверяется на наличие ошибок и нежелательного поведения (например, чрезмерного потребления памяти) на каждом входном файле, полученном в результате работы алгоритма. Сгенерированные файлы добавляются в очередь с приоритетом. Больший приоритет имеют файлы, которые принесли больше нового покрытия (в терминах базовых блоков) относительно уже достигнутого текущего покрытия. На наиболее приоритетных файлах происходит символьная интерпретация программы и построение предиката пути. Переходы в предикате пути инвертируются. Файлы, которые инвертируют переходы также добавляются в очередь. Однако для полученных файлов инвертирование переходов будет начинаться с глубины, на которой был инвертирован переход у родительского файла из которого они были получены. Это позволяет избежать повторного обхода одних и тех же путей.

Более того, авторы утверждают, что алгоритм генерационного поиска позволяет бороться с неточностями символьной модели. Если сгенерированный файл не приводит к инвертированию целевого перехода, то с большой вероятностью он не принесет нового покрытия и его приоритет будет низкий.

Отличительной особенностью SAGE является то, что инструмент осуществляет символьную интерпретацию на уровне машинных инструкций x86, а не на уровне исходного кода как KLEE. SAGE повторяет воспроизведение уже

записанной трассы выполнения программы, во время чего обновляет конкретное и символьное состояния. Конкретное состояние содержит текущие значения регистров и памяти, а символьное состояние отображает байты (регистров и памяти) в символьные переменные.

Изначально для входных байтов создаются символьные переменные. Для каждой неконстантной операции создается новая символьная переменная, значение которой равняется формуле, описывающей эту операцию. Когда SAGE встречает условный переход, он добавляет условие прохождения по направлению из трассы в предикат пути. Для получения условия перехода SAGE поддерживает специальный битовый вектор для регистра флагов, который содержит условия выставления флагов в единицу.

1.2.3 S^{2}E (2011)

В 2011 году Chipounov и др. [48; 49] представили платформу S^{2}E для разработки инструментов анализа свойств и поведения сложных программных систем. S^{2}E является модульной библиотекой для полносистемной символьной интерпретации и анализа программ. S^{2}E позволяет запускать сложные программные комплексы (приложения, библиотеки, ядра, драйверы и др.) внутри виртуальной машины. При этом не требуется вносить изменения в анализируемый код. Символьная интерпретация позволяет исследовать множественные пути исполнения, в то время как различные анализаторы проверяют выполнение желаемых свойств кода на этих путях. Более того, S^{2}E использует специальные механизмы выбора пути, чтобы сфокусироваться на исследовании интересующих компонентов.

S^{2}E предлагает выборочную символьную интерпретацию в качестве решения проблемы экспоненциального роста путей. Описанный метод базируется на наблюдении о том, что часто представляют интерес только некоторые семейства путей. Например, ветвления в коде библиотек и ядра можно опустить для исследования путей выполнения программы. Таким образом, S^{2}E интерпретирует много путей выполнения в анализируемой части когда, а при попадании в другие части системы интерпретация сужается до одного пути. Технически это реализуется постоянным переключением между символьной и конкретной интерпретациями.

Для соблюдения баланса между конкретной и символьной интерпретациями S^2E предлагает различные модели целостности. Под полной целостностью понимается, что в каждой точке выполнения существует достижимый путь от начального состояния к текущему. Однако большинство алгоритмов анализа не требуют наличия полной целостности. Например, во время автоматической генерации тестов (входных данных) не требуется, чтобы все исследуемые пути были достижимы, а точный обход всех путей может быть накладным с точки зрения ресурсов. С ростом числа анализируемых путей растет и алгоритмическая сложность получения модели математическим решателем. Кроме того, чрезмерные ограничения на предикат пути уменьшают число потенциально достижимых путей для дальнейшей символьной интерпретации. С другой стороны, недостаточное моделирование окружения может привести к неполноте символьной модели и неверным решениям. Таким образом, S^2E предоставляет пользователю возможность выбора уровня целостности системы в зависимости от конкретной прикладной задачи.

Полносистемный анализ в S^2E реализуется с использованием эмулятора QEMU [50]. Символьная интерпретация осуществляется с помощью KLEE [39]. QEMU осуществляет эмулирование окружения и конкретную интерпретацию. Промежуточное представление QEMU транслируется в LLVM, а затем KLEE его символьно интерпретирует. Таким образом, QEMU реально исполняет код программной системы и ничего не знает о символьном состоянии, а KLEE в свою очередь интерпретирует LLVM представление полностью символьно. S^2E же реализует переключение между конкретным и символьным состояниями, а также осуществляет контроль за их целостностью. Следует отметить, что множественные трансляции между промежуточными представлениями негативно сказываются на производительности.

1.2.4 Mayhem (2012)

В 2012 году в университете Карнеги — Меллона был разработан инструмент Mayhem [44] для автоматического обнаружения и эксплуатации уязвимостей. Mayhem обходит пути выполнения программы и исследует их на предмет наличия уязвимостей методами символьной интерпретации. Авто-

матическая генерация эксплойта (входных данных, приводящих к выполнению произвольного кода) осуществляется следующим образом. Сначала происходит построение предиката пути от точки получения входных данных до точки проявления уязвимости. Затем к полученному предикату пути добавляется предикат безопасности — дополнительные ограничения, описывающие размещение некоторого кода в памяти процесса и передачу ему управления. Таким образом, решением полученного предиката будет эксплойт. Предикат пути отвечает за прохождение программой того же пути до точки проявления уязвимости. А предикат безопасности обеспечит перехват потока управления.

Описанный подход подтвердил свою состоятельность, когда Mayhem в 2016 году победил в соревновании DARPA Cyber Grand Challenge [51]. На этом мероприятии машины соревновались в умении автоматически искать ошибки и эксплуатировать уязвимости без помощи человека.

Mayhem разделен на два процесса: конкретный и символьный вычислители. Конкретный вычислитель осуществляет динамическую бинарную инструментацию анализируемой программы с помощью Pin [52]. Кроме того, конкретный вычислитель производит динамический анализ помеченных данных и отправляет поток отобранных инструкций символьному вычислителю, который отвечает за их символьную интерпретацию. Машинные инструкции сначала транслируются в промежуточное представление BAP, а затем происходит его символьная интерпретация. Более того, символьный вычислитель строит предикат пути, объединяет его с предикатом безопасности и отдает математическому решателю Z3 [34].

Mayhem применяет гибридный подход, сочетающий в себе конколическую (concolic) и полностью символьную интерпретации. Таким образом, инструмент балансирует между двумя подходами, переключение между которыми осуществляется при достижении заданного ограничения потребления памяти и числа выполняющихся процессов. Таким образом, Mayhem конколически интерпретирует программу через символьную интерпретацию трассы инструкций, заданной конкретным путем выполнения, аналогично SAGE. Однако в точках ветвления могут порождаться новые символьные процессы, клонирующие состояние и исследующие различные направления условного перехода, как это делается в KLEE.

1.2.5 Triton (2015)

Triton [23] представляет из себя фреймворк для динамической символьной интерпретации. Его устройство базируется на теоретической модели, описанной Schwartz и др. [41]. Triton интерпретирует инструкции машинного кода напрямую, без промежуточного представления, что позволяет получить выигрыш в скорости символьной интерпретации. Таким образом, поддерживаются различные процессорные архитектуры: x86, x86-64, ARM, AARCH64.

Triton позволяет реализовывать различные инструменты анализа программ и обратной инженерии. Он дает возможность по одной интерпретировать подаваемые ему на вход инструкции. Способ получения этих инструкций перекладывается на плечи разработчика.

Triton позволяет присвоить символьные переменные произвольным регистрам и памяти, которые моделируются битовыми векторами. В фреймворке поддерживаются конкретное и символьное состояния. Первое содержит реальные константные значения регистров и памяти, в то время как второе — отображение регистров и байтов памяти в символьные выражения. Символьная интерпретация инструкций порождает новые выражения в SSA-форме [53], а также обновляет соответствующие символьные и конкретные состояния. В результате интерпретации условного перехода в предикат пути добавляется символьное выражение для его условия. Символьные выражения представляют из себя нумерованные символьные формулы в форме абстрактного синтаксического дерева (AST), с которыми Triton позволяет производить различные манипуляции (арифметические и логические операции над битовыми векторами). AST-представление в дальнейшем может быть транслировано в SMT [42] и передано математическому решателю [34; 35].

Для каждой ассемблерной инструкции в Triton написан код, который строит для нее символьные выражения в виде AST на основе построенных ранее выражений, а также обновляет необходимые конкретные и символьные состояния. Например, если инструкция складывает два символьных регистра, то выражения для этих регистров берутся из символьного состояния. Затем конструируется новое выражение, которое состоит из AST вершины, складывающей эти два выражения. После этого полученное выражение присваивается результирующему регистру.

1.2.6 Driller (2016)

В 2016 году Stephens и др. [16] реализовали и показали эффективность гибридного фаззинга [15], который комбинирует фаззинг тестирование с обратной связью по покрытию и динамическую символьную интерпретацию. Идея заключается в следующем. Во время фаззинга сложно перебирать входные данные, открывающие новые пути. Символьная интерпретация в свою очередь страдает от проблемы экспоненциального роста исследуемых путей. Разработанный авторами инструмент Driller объединяет оба подхода. Фаззинг позволяет легко обходить пути программы, а символьная интерпретация открывает новые пути программы, которые труднодоступны фаззеру. Более того, совместное использование фаззинга и символьной интерпретации приводит к открытию новых ошибок, пропущенных при использовании методов по отдельности.

Driller использует фаззер AFL [45], а символьную интерпретацию осуществляет фреймворк angr [54]. При этом символьная интерпретация запускается по надобности, когда фаззер «застрял» и перестал открывать новые пути. Таким образом, сначала запускается фаззинг. Фаззинг останавливается, когда покрытие перестает расти, и запускается символьная интерпретация на входных данных, которые недавно открыли пути во время фаззинга. Символьная интерпретация добавляет дополнительные предусловия, которые фиксируют символьные переменные на конкретном входе, чтобы избежать экспоненциального роста путей. Далее во время символьной интерпретации обнаруживаются пути, раннее не открытые фаззером, и для них генерируются новые входные данные. Файлы от символьного вычислителя затем передаются в очередь фаззера. Затем Driller переключается обратно на фаззинг. Такие переключения между символьным вычислителем и фаззером происходят до тех пор, пока не будет найдено аварийное завершение.

Символьный вычислитель сначала транслирует бинарный код в промежуточное представление VEX [55]. Далее VEX символьно интерпретируется аналогично Mayhem и S^2E. В результате символьной интерпретации angr порождает формулы в виде внутреннего абстрактного синтаксического дерева (AST). Помимо прочего angr реализует широкий спектр оптимизаций над AST [56]. Однако у angr сильно страдает производительность из-за множественных транс-

ляций между промежуточными представлениями (как и в S^2E) и реализации на «медленном» языке программирования Python.

1.2.7 QSYM (2018)

В 2018 году был представлен миру инструмент QSYM [17]. Авторы впервые показали, что динамическая символьная интерпретация может быть настолько производительная, что ее можно запускать непрерывно вместе с фаззингом. Более того, гибридный подход достигает большее покрытие, чем обычный фаззер с обратной связью по покрытию (AFL).

QSYM использует динамическую инструментацию (Pin [52]) для достижения большей производительности по сравнению с полностью символьной эмуляцией каждой инструкции. При взаимодействии с окружением берутся реальные значения регистров и памяти, полученные в результате системных вызовов. Также для тех же целей авторы отказались от промежуточного представления — машинные инструкции напрямую транслируются в формулы. Более того, QSYM впервые предложили потерять часть математической строгости в символьной интерпретации в угоду производительности и большей скорости открытия новых путей. В частности, QSYM пропускает символьную интерпретацию часто повторяющихся блоков, т. к. они в большинстве случаев только усложняют модель. Кроме того, авторы предложили идею оптимистичных решений. Если конъюнкция условия для инвертирования перехода и предиката пути невыполнима, то производится попытка решить просто условие инвертирования без контекста пути. Иногда сгенерированные таким образом данные позволяют инвертировать переход. Другими словами, используя оптимистичные решения для невыполнимых предикатов, можно за то же время открыть больше путей, чем при консервативной символьной интерпретации.

В отличие от Driller, QSYM запускает фаззинг тестирование и динамическую символьною интерпретацию параллельно. Фаззер и символьный вычислитель постоянно обмениваются входными данными. В первую очередь символьный вычислитель запускается на файлах, недавно принесших покрытие.

Для учета уже решенных переходов QSYM накапливает кэш — байтовый массив, хранящий счетчик уже выполненных запросов к решателю на инверти-

рование переходов. QSYM представил два режима кэширования: статический и контекстный. При статическом кэшировании производится попытка инвертировать каждый переход один раз. При контекстном один и тот же переход инвертируется повторно, если он был встречен на другой глубине, или же до него выполнялся символьный переход, который ранее перед ним обнаружен не был. Таким образом, переходы, содержащиеся в кэше, повторно не инвертируются, что позволяет сосредоточится на исследовании новых путей.

1.2.8 SymCC (2020)

В 2020 году S. Poeplau и др. [18] предложили компилировать конколическую (concolic) интерпретацию непосредственно внутрь исполняемого файла вместо символьной интерпретации инструкций во время выполнения программы. Представленный инструмент SymCC, разработанный на фреймворке LLVM [47], по сути является заменой компиляторов `clang` и `clang++`. Таким образом, SymCC встраивает динамическую символьную интерпретацию на уровне LLVM непосредственно в анализируемую программу.

Авторы предложили подход компилируемой символьной интерпретации. Основная идея заключается в ускорении символьной интерпретации за счет компилирования символьных вычислений вместе с кодом исследуемой программы. При таком подходе код инструментации, который конструирует формулы и обновляет символьное состояние, встраивается рядом с оригинальными вычислениями программы. Следовательно, генерация кода, обновляющего символьное состояние, происходит один раз во время компиляции. Однако все библиотеки, которые использует программа, должны быть проинструментированы. Описанный метод позволяет использовать все преимущества оптимизаций промежуточного представления LLVM, т. к. инструментация компилируется вместе с кодом приложения. Более того, осуществляющий символьную интерпретацию код подвержен оптимизациям процессора, таким как кэширование и предсказание переходов.

SymCC уменьшает размер инструментирующего кода за счет анализа константных значений. Если во время компиляции известно, что вычисления константные, то для них не порождается код символьной интерпретации. Для остальных вычислений добавляются проверки во время выполнения, чтобы

символьно интерпретировать только вычисления, оперирующие с символьными данными.

SymCC реализован в виде прохода компилятора, который обходит промежуточное представление LLVM и вставляет инструментирующий код, который либо напрямую конструирует формулы в математическом решателе Z3 [34] или же использует бэкенд символьного вычислителя QSYM [17]. Таким образом, код символьной интерпретации выполняется в качестве части целевой программы и не требует переключения между кодом программы и интерпретатором (как, например, в KLEE [39]).

Следует отметить, что SymCC разрешает собирать только часть библиотек собственным компилятором. Проинструментированный код может вызывать функции из библиотек без символьной интерпретации. Тогда код этих библиотек будет просто выполняться конкретно. Более того, авторы реализовали обертки для части функций из стандартной библиотеки Си, которые добавляют необходимые ограничения в предикат пути, описывающие исследуемый путь выполнения.

Авторы реализовали примитивное взаимодействие с фаззером AFL [45] аналогично QSYM. Символьный вычислитель запускается на файлах от фаззера и отправляет новые сгенерированные входные данные обратно фаззеру. Обмен файлами происходит через очереди фаззера. QSYM притворяется фаззером AFL, создает свою очередь файлов и использует механизм распределенного фаззинга для синхронизации очередей. Очередь входных файлов для символьного вычислителя сортируется следующим образом. Символьный вычислитель в первую очередь запускается на новых (недавно созданных) файлах, которые принесли покрытие (открыли новые пути).

1.2.9 SymQEMU (2021)

В 2021 году авторы SymCC развили подход компилируемой символьной интерпретации применительно к бинарному коду, когда исходный код отсутствует [19]. Они разработали инструмент SymQEMU поверх эмулятора QEMU [50]. Инструмент модифицирует промежуточное представление целевой программы перед его трансляцией в машинный код. Такой метод позволяет SymQEMU компилировать символьную интерпретацию внутрь исполняемых файлов. Тем самым

авторы получают преимущество от скорости компилируемого подхода, в то же время поддерживая независимость от целевой архитектуры.

Авторы расширили функционал компоненты QEMU под названием TCG. Компонента отвечает за трансляцию блоков машинных инструкций гостевой системы в архитектурно-независимое промежуточное представление TCG ops, которое в дальнейшем компилируется в машинный код хоста. Транслируемые блоки кэшируются для большей производительности. Следовательно, трансляция блока происходит один раз за исполнение. На этапе трансляции в TCG ops SymQEMU добавляет новые инструкции TCG ops для конструирования символьных выражений. Код построения выражений и решения запросов к решателю заимствован из SymCC (который в свою очередь использует QSYM [17]). Авторы используют эмуляцию QEMU на уровне пользовательского пространства. Таким образом, SymQEMU останавливает символьный анализ на границе системных вызовов (аналогично QSYM [17] и angr [54]), что позволяет достигать большей производительности в отличие от полносистемной эмуляции (как в S^2E [48]).

С целью экономии потребляемой памяти в SymQEMU реализован сборщик мусора для символьных выражений, которые больше не используются. Если выражение больше не содержится в регистрах, памяти или предикате пути, то оно освобождается.

Гибридный фаззинг SymQEMU и AFL [45] реализован абсолютно так же, как и в SymCC. Обмен файлами происходит через очереди фаззера.

1.2.10 Fuzzolic (2021)

Немногим позже в 2021 году L. Borzacchiello и др. [20] представили инструмент Fuzzolic, который был разработан независимо, но частично повторяет идею SymQEMU [19]. Fuzzolic так же предлагает реализацию динамической символьной интерпретации на основе модификации QEMU TCG. Однако он показал большую производительность с точки зрения скорости прироста покрытия, чем QSYM [17], SymCC [18] и SymQEMU [19].

Авторы предложили ускорить конколическую интерпретацию с двух сторон. С одной стороны, они используют QEMU для быстрой эмуляции. С другой стороны, авторы разработали Fuzzy-Sat решатель [57], который применяет техни-

ки фаззинг-тестирования для решения SMT формул. В частности, используются специальные мутации, которые учитывают структуру формулы. Таким образом, приближенное решение подбирается фаззингом значений символьных переменных.

Fuzzolic разделяет анализ на два компонента. Первый компонент отвечает за трассировку инструкций в QEMU и построение символьных выражений. Второй компонент осуществляет решение формул. Оба процесса обмениваются символьными выражениями через разделяемую память. Такое решение было принято, чтобы облегчить эмуляцию программы и по максимуму уменьшить число побочных эффектов во время интерпретации. Таким образом, в QEMU выполняется минимальное количество инструментирующего кода, который порождает символьные выражения в компактном представлении и сохраняет их в разделяемой памяти. Второй процесс уже конструирует тяжеловесные формулы математического решателя и осуществляет их решение.

JIT-компиляция QEMU позволяет добиться высокой производительности инструментированной программы. Такой подход позволяет компилировать инструментирующий код один раз. Более того, Fuzzolic пользуется всеми преимуществами инлайнинга инструментациии в код приложения. А инструментация кода на уровне базового блока (SymQEMU инструментирует на уровне инструкций) позволяет получать выгоду от анализа блока целиком. Кроме того, в отличие от SymQEMU, Fuzzolic реализует функции обработчики для символьной интерпретации векторных инструкций.

Fuzzolic использует схему гибридного фаззинга аналогичную QSYM. Инструмент реализует интеграцию с фаззером AFL++ [14] с обратной связью по покрытию. Fuzzolic запускается на файлах из очереди фаззера, а складывает новые сгенерированные входные данные в свою собственную очередь в формате AFL [45]. AFL++, со своей стороны, синхронизируется с очередью Fuzzolic и загружает интересующие его файлы (которые открывают новые пути). Следует отметить, что Fuzzolic берет файлы только от одного запущенного экземпляра фаззера, чтобы избежать запуска на повторяющихся входных данных. Более того, инструмент аналогично QSYM пропускает инвертирование переходов, которые уже были ранее исследованы символьным вычислителем.

1.3 Преодоление избыточной и недостаточной помеченности

Символьной интерпретации свойственны схожие проблемы, которые встречаются во время динамического анализа помеченных данных [58], а именно: избыточная и недостаточная помеченность. В таком случае помеченными считаются данные, которые зависят от пользовательского ввода.

Недостаточная помеченность возникает, когда значение, вычисление которого зависит от входных данных, не помечено. У этого может быть несколько причин. Одной из них является проблема символьных адресов. Когда индекс массива (разыменуемый адрес) зависит от пользовательских данных, в худшем случае он может указывать на произвольную ячейку адресуемой памяти. Поэтому большинство инструментов динамической символьной интерпретации конкретизуют такой адрес значением из реального выполнения, чтобы он указывал на одну конкретную ячейку памяти. Однако это приводит к тому, что часть ограничений, зависящих от значения по символьному адресу, отсутствуют в предикате пути. В итоге символьная модель теряет точность, и полученное решение может не провести программу по тому же пути выполнения. Другой причиной недостаточной помеченности может являться недостаточное моделирование системного окружения. В частности, инструмент может пропускать часть системных вызовов, которые работают с символьными данными. Использование генератора псевдослучайных чисел приводит к недетерминированной модели, когда случайное значение участвует в символьных вычислениях. Более того, описать трансляцию всех инструкций в символьные формулы не всегда представляется возможным, поэтому отсутствие их трансляций также может приводить к недостаточной помеченности.

Избыточная помеченность — это ситуация, когда в предикат пути попадает чрезмерное число ограничений. Однако эти ограничения могут совсем не оказывать влияния на входные данные, а только замедлять время решения предиката пути. Наиболее опасно, когда такие избыточные ограничения приводят к невыполнимости запросов к математическому решателю. Частое распространение пометок в коде библиотечных функций является одной из причин избыточной помеченности. Более того, некоторые ограничения в предикате пути никак не влияют на факт инвертирования перехода. Такие ограничения желательно устранять из запросов к решателю.

1.3.1 Устранение избыточных ограничений в KLEE

Многие ограничения в предикате пути на самом деле не пересекаются в терминах памяти (символьных переменных), к которой они обращаются. KLEE [39] (как и ранее EXE [38]) разбивает ограничения в предикате пути на независимые подмножества. Таким образом, KLEE устраняет нерелевантные для запроса ограничения. Например, для запроса $i = 20$ требуются только первые два ограничения из предиката пути $\{i < j, j < 20, k > 0\}$.

Идея подхода заключается в том, что ограничения из предиката пути разбиваются на независимые подмножества ограничений. Два ограничения считаются независимыми, если ни одна символьная переменная из одного ограничения не содержится во втором. Следует отметить, что KLEE определяет ограничения и содержащиеся в них символьные переменные непосредственно из условных переходов в исходном коде.

Разбиение на независимые подмножества ограничений реализуется с помощью алгоритма поиска компонентов связности на графе следующего вида. Вершинами графа являются символьные переменные, а ребра отражают факт, что символьные переменные содержатся вместе в некотором ограничении. Таким образом, для каждой компоненты связности графа можно построить независимое подмножество ограничений, которое состоит из ограничений, содержащих символьные переменные из компоненты связности.

Описанный выше метод помогает KLEE бороться с избыточной помеченностью: для инвертирования целевого перехода отбираются только релевантные ограничения из предиката пути. Более того, такой подход позволяет эффективно использовать кэширование ответов математического решателя (описанное в разделе 1.2.1), т. к. запросы состоят из меньшего числа ограничений.

1.3.2 Оптимистичные решения в QSYM

Авторы QSYM [17] заметили, что во время исследования новых путей с помощью динамической символьной интерпретации доля невыполнимых (UNSAT) запросов к решателю достаточно высока. Они предложили решать только огра-

ничение инвертируемого перехода без предиката пути, если запрос к решателю с учетом пути не разрешим. Такое решение без контекста пути было названо оптимистичным. Следует отметить, что точность инвертирования переходов падает. Однако оптимистичные решения иногда успешно инвертируют переход. Запросы имеют более простую форму, и математический решатель способен их обрабатывать быстрее. Таким образом, число успешно инвертированных за единицу времени переходов растет. Поэтому такой подход эффективен во время гибридного фаззинга, т. к. позволяет открывать больше путей, а фаззер в состоянии сам отбросить входные данные, которые не приводят к росту путей.

Для иллюстрации подхода авторы приводят следующий мотивационный пример:

```
1 if (ch >= 0x20 && ch < 0x7f)
2   ...
3 if (ch == 0x7f)
4   ...
```

Если во время построения предиката пути первый условный переход выполнился, а второй нет, то запрос на инвертирование второго перехода $ch \geqslant \texttt{0x20} \wedge ch < \texttt{0x7f} \wedge ch = \texttt{0x7f}$ был бы невыполним. Однако условие выполнения кода внутри второго перехода никак не зависит от первого. Таким образом, осуществление оптимистичного решения запроса $ch = \texttt{0x7f}$ подберет входные данные программы, которые успешно приведут выполнение внутрь второго перехода.

1.3.3 Пропуск базовых блоков в QSYM

Символьная интерпретация часто повторяющихся базовых блоков может приводить к избыточной помеченности. Интерпретация таких блоков порождает тяжелые и даже невыполнимые ограничения на предикат пути. QSYM [17] пропускает символьную интерпретацию блоков с высокой частотой, которые блокируют открытие новых путей. Таким образом, ограничения из пропущенных блоков не добавляются в предикат пути.

Во время динамической символьной интерпретации QSYM измеряет частоту выполнения каждого базового блока. Таким образом, QSYM интерпретирует только те базовые блоки, чья частота является степенью двойки. Исключением

являются блоки, которые не порождают новых ограничений. Например, блоки, содержащие только инструкции перемещения символьных данных в памяти, всегда интерпретируются.

Чтобы избежать чрезмерного пропуска базовых блоков, QSYM группирует несколько выполнений блока в одно при подсчете частоты. Более того, базовые блоки, которые выполнялись в разном контексте (с различными стеками вызовов), имеют раздельные счетчики частоты.

1.3.4 Моделирование символьных адресов

Проблема символьных адресов [24; 41] проявляется, когда значение указателя зависит от пользовательских данных. Такое часто происходит в бинарном коде. Например, конструкция языка Си `switch(c)` компилируется в таблицу переходов, где по индексу `c` лежит указатель на код некоторого `case`. Различные функции конвертации строк (`tolower`, ASCII в UNICODE и т. д.) также используют табличные преобразования, где по входному символу лежит его отображение.

Моделирование символьных адресов является сложной задачей. В худшем случае такой адрес может указывать в произвольное место адресуемой памяти. Поэтому большинство инструментов динамической символьной интерпретации кокретизуют символьный указатель значением из реального выполнения программы (в частности, так делает SAGE [40]). Такой подход позволяет ускорить символьную интерпретацию и сократить время решения запросов к SMT-решателю. Однако конкретизация символьных адресов, которые встречаются часто в программах, неизбежно приводит к недостаточной помеченности.

Во время разыменования символьного указателя S^2E [48] определяет страницы памяти, к которым можно обращаться по этому указателю, и передает их содержимое SMT-решателю. Поиск допустимых границ символьного указателя осуществляется бинарным поиском с последовательными запросами к SMT-решателю, ограничивающими границы адреса в контексте пути. Обращения к памяти моделируются теорией SMT Array, которая позволяет выполнять операции сохранения (*store*) и чтения (*select*) массива по произвольному индексу.

Mayhem [44] предложил индексную модель памяти для решение проблемы символьных указателей. Символьные адреса, по которым производится чтение, рассматриваются как символьные. Символьный адрес записи при этом всегда конкретизируется. Таким образом, предложенная модель памяти позволяет описывать чтения по символьным адресам, где чтение моделируется объектом памяти, который хранит значения памяти для различных индексов. Границы адресуемой памяти определяются бинарным поиском с использованием математического решателя. Для ускорения работы решателя предварительный допустимый набор значений символьного индекса определяется алгоритмом VSA [59]. Далее эти границы уточняются решателем. Значения памяти представляются в виде бинарного дерева поиска по индексу массива: вложенные if-then-else выражения SMT $ite(i < 8, ite(i < 2, 5, 6), 7)$. Для сокращения числа листьев в дереве подряд идущие точки объединяются в линию, когда это возможно. Таким образом, линию можно описать одним листом, параметризованным индексом i: $k * i + b$. Например, формула для функции `tolower` выглядит следующим образом: $ite(n < 91, ite(n < 64, n, n + 32), n)$. В дальнейшем метод индексной модели памяти будет развит до применения в контексте гибридного фаззинга [22].

1.3.5 Частичная пометка символьного входа в LeanSym

Авторы LeanSym [60] предложили создавать символьные переменные только для части входных байтов программы, которые влияют на выполнение условия целевого перехода. Определение входных байтов, влияющих на инвертирование перехода, осуществляется с использованием динамического анализа помеченных данных [58].

Изначально динамический анализ помеченных данных позволяет определить множества символьных переменных, которые влияют на выполнимость каждого условного перехода. LeanSym помечает весь входной файл и осуществляет прямое распространение пометок. Для каждого условного перехода сохраняются смещения входных байтов, которые привели к пометке операндов инструкции `cmp`, отвечающей за условие выполнения перехода.

Затем происходит инвертирование условных переходов. Создаются символьные переменные для байтов, которые приводят к пометке операндов сравнения (`cmp`) для целевого перехода. Эти байты были получены на предыдущем шаге. Происходит символьная интерпретация инструкций до целевого перехода. В результате частичной пометки символьного входа происходит интерпретация только части инструкций, которые обрабатывают помеченные данные. А в предикат пути отбираются только необходимые ограничения для инвертирования перехода.

LeanSym отправляет запрос к SMT-решателю для инвертирования целевого перехода. Математический решатель возвращает модель для помеченных входных байтов. Значения этих байтов заменяются в изначальном входном файле. В результате, генерируется входной файл для инвертирования перехода.

Частичная пометка символьного входа значительно ускоряет символьную интерпретацию и решение запросов. С другой стороны, запуск символьной интерпретации с нуля для инвертирования каждого условного перехода может негативно влиять на скорость открытия новых путей выполнения.

1.3.6 Моделирование семантики функций

Производительность имеет решающее значение применительно к динамической символьной интерпретации. Реализовано множество оптимизаций для сокращения времени анализа. Следует уменьшать размер инструментирующего кода там, где это возможно, т. к. инструментация программы медленнее обычного выполнения на несколько порядков.

Нет никакой необходимости символьно интерпретировать каждую инструкцию внутри функций из стандартной библиотеки Си, которые имеют специфицированную семантику. Многие инструменты символьной интерпретации моделируют семантики функций и тем самым уменьшают число избыточных ограничений в предикате пути [18; 20; 48; 54]. Например, функция `tolower` может принимать как заглавные, так и строчные символы. Однако динамическая символьная интерпретация трассы инструкций внутри функции приводит к тому, что символ ограничивается регистром из реального выполнения. Таким образом, возникает избыточная помеченность. Этого можно избежать, если моделировать

данную функцию следующим if-then-else (*ite*) выражением: $ite(ch < \texttt{'a'}, ch + (\texttt{'a'} - \texttt{'A'}), ch)$. Моделирование семантики функций позволяет описать сразу несколько символьных состояний одной формулой и ускорить символьную интерпретацию. Таким образом, можно открывать больше путей в результате символьной интерпретации одной конкретной трассы программы.

Методы моделирования семантики функций, которые доминируют или же комбинируются в различных инструментах, можно разделить на пять групп:

1. Пропуск интерпретации функции и конкретизация ее выполнения.
2. Символьная интерпретация собственной упрощенной реализации функции, которая подменяет оригинальную функцию из стандартной библиотеки.
3. Фиксация конкретной трассы внутри функции путем наложения ограничений на предикат пути.
4. Моделирование функции условным переходом, который описывает характерные результаты выполнения функции (например, равенство строк в `strcmp`).
5. Построение единой символьной формулы для возвращаемого значения, которая описывает сразу несколько результатов выполнения функции.

Пропуск функций

Функции работы с динамической памятью (`malloc`, `calloc`, `realloc`, `free` и др.) часто обрабатывают символьные данные, а также их аргументы иногда оказываются помеченными. Таким образом, интерпретация условных переходов в коде данных функций добавляет избыточные ограничения в предикат пути. Однако путь выполнения внутри этих функций никак не влияет на инвертирование целевых переходов. Поэтому можно пропустить символьную интерпретацию внутри функций аллокаторов и взять необходимые значения из конкретного выполнения.

Функции, которые пишут в лог программы или стандартный поток вывода, обычно не представляют интереса для анализа [20]. Их код не оказывает влияния на символьную модель за пределами этих функций. Поэтому такие функции также могут быть пропущены.

Интерпретация упрощенной реализации функции

KLEE [39] заменяет вызовы стандартной библиотеки Си на единожды скомпилированный LLVM код встроенных библиотек (в частности, на код uClibc [61] и собственную упрощенную реализацию). Это направлено на сокращение компиляции дополнительных компонентов и упрощение анализируемого кода. Такой подход упрощает эмуляцию окружения для внешних вызовов. Таким образом, KLEE не требуется компилировать код произвольной стандартной библиотеки, а достаточно использовать уже имеющийся LLVM-IR для стандартных функций. Более того, такой LLVM код короче и проще для дальнейшей символьной интерпретации, что позволяет генерировать более простые формулы.

Добавление ограничений в предикат пути

Авторы SymCC [18] реализовали обертки для части функций из стандартной библиотеки Си, которые добавляют необходимые ограничения в предикат пути, описывающие исследуемый путь выполнения. Например, для функции `memcmp` в предикат пути добавляется условный переход с двумя ветвями: строки различаются или же полностью совпадают. Инвертирование такого перехода позволяет за один запуск символьной интерпретации приравнять обе строки.

Fuzzolic [20] предлагает три режима анализа для моделирования семантики функций. Первый режим предполагает полный пропуск символьной интерпретации функций. Например, пропуск функций печати (`printf`) может значительно ускорить анализ и уменьшить число избыточных ограничений на предикат пути. Второй режим осуществляет символьную интерпретацию функции без инвертирования переходов внутри нее. Это полезно для библиотечных функций, когда важно обновить символьное состояние, но исследование кода внутри них не представляет интерес. В частности, так можно пропускать код аллокаторов. Кроме того, для строковых сравнений формируются запросы к решателю, чтобы приравнять строки сразу, а не перебирать побайтовые сравнения в случае инвертирования отдельно взятых переходов внутри функции. Третий режим включает полную символьную интерпретацию кода и является наиболее тяжеловесным.

Балансируя между режимами, Fuzzolic добивается приемлемой производительности.

Символьное выражение для возвращаемого значения

S^2E [48] включает в себя модуль для моделирования символьной семантики функций. При вызове библиотечной функции, которая обрабатывает символьные данные, управление передается на специальную функцию-обработчик, которая подставляет в возвращаемое значение символьную формулу. Таким образом, можно моделировать сразу несколько символьных состояний (путей) одной формулой. Поддерживается моделирование около десятка функций из стандартной библиотеки. Формулы по своей структуре представляют из себя вложенные if-then-else (*ite*) выражения SMT, которые учитывают различные варианты возвращаемого значения. Например, функция сравнения строк `strcmp` моделируется вложенными *ite*, которые попарно сравнивают байты при условии равенства всех предыдущих байтов.

В angr [54] реализованы символьные модели для большого числа библиотечных функций (libc, glibc, POSIX). Часть моделей просто заменяют вызов функций на собственную упрощенную реализацию. Другая часть же полностью моделируется, исходя из семантики (`strtol`, `strlen`, `strchr`, `memcmp` и т. д.). Пожалуй, angr имеет самое большую библиотеку поддерживаемых стандартных функций. Однако это обусловлено тем, что для осуществления символьной интерпретации в angr должны быть определены все вызываемые программой функции из стандартной библиотеки и системные вызовы.

1.3.7 Выводы

Проблемы недостаточной и избыточной помеченности присущи динамической символьной интерпретации. Они напрямую влияют на точность и выполнимость запросов к математическому решателю. А следовательно, влияют на возможность обнаружения ошибок методами динамической символьной

интерпретации и на их воспроизводимость. Необходимо решать проблемы недостаточной и избыточной помеченности для повышения точности разрабатываемого метода поиска ошибок.

Использование оптимистичных решений QSYM [17] неизбежно приведет к большому числу ложно положительных срабатываний, т. к. факт наличия ошибки будет проверяться без чувствительности к пути. В частности, искомый путь, который содержит ошибку, может быть вовсе недостижим. Пропуск базовых блоков влечет за собой потерю консервативности анализа: ошибки могут быть вовсе пропущены, или же для них не будут сгенерированы корректные входные данные для воспроизведения.

Моделирование символьных адресов [22; 44] является эффективным методом для борьбы с недостаточной помеченностью. Однако во время символьной интерпретации порождаются гораздо более сложные формулы, что на порядок замедляет решение запросов [22]. Поэтому мы оставим рассмотрение методов моделирования символьных адресов за пределами данной работы.

Частичная пометка символьного входа в LeanSym [60] не является консервативным анализом. В частности, такой подход может приводить к невыполнимым запросам к математическому решателю. Предположим, что следующая программа была запущена на входных данных 0, 5:

```
1 signed char s[2];
2 read(0, s, 2);
3 if (s[0] + s[1] > 4)
4   if (s[0] + s[1] < 6)
5     if (s[1] > 10)
6       printf("OK\n");
```

Попробуем инвертировать условный переход на строке 5, чтобы программа вывела `OK`. С помощью динамического анализа помеченных данных мы выясним, что целевой переход на строке 5 зависит только от входного байта `s[1]`. Пометим этот байт как символьный. В результате динамической символьной интерпретации мы получим следующий запрос к математическому решателю: $0 + s_1 > 4 \wedge 0 + s_1 < 6 \wedge s_1 > 10$. Полученный предикат не имеет решения. Однако, если мы пометим все входные байты символьными (чего избегает LeanSym), то полученный предикат $s_0 + s_1 > 4 \wedge s_0 + s_1 < 6 \wedge s_1 > 10$ выполним (например, $s[0] = -6$, $s[1] = 11$).

Более того, LeanSym использует фреймворк динамического анализа помеченных данных libdft64 [62], который работает не со всеми сложными

программными системами. Также анализ помеченных данных выявляет зависимости по данным менее точно, чем это делает символьная интерпретация. Это происходит потому, что анализ помеченных данных не учитывает вычисляемые значения (семантику инструкций) при распространении пометок. Например, символьный интерпретатор может понять, что формула $(a - a)$ не символьная. А точность анализа помеченных данных напрямую влияет на отобранные входные байты для дальнейшей их пометки во время символьной интерпретации.

Наиболее подходящим для повышения точности решения задачи поиска ошибок является метод устранения избыточных ограничений, используемый в KLEE [38; 39]. Однако он применим только к исходному коду. В данной работе получен новый алгоритм (вдохновленный KLEE), который устраняет избыточные ограничения во время динамической символьной интерпретации бинарного кода. Алгоритм позволяет консервативно удалять ограничения из предиката пути, сохраняя достижимость пути до точки проявления ошибки. Таким образом, число полученных ограничений лежит между одним ограничением (в оптимистичных решениях) и полным предикатом пути. Если некоторые базовые блоки действительно не влияют на достижимость пути, то они будут устранены в результате работы алгоритма, а не во время символьной интерпретации.

Методы моделирования семантики функций помогают избегать как избыточной, так и недостаточной помеченности. А моделирование функций одной формулой возвращаемого значения позволяет исследовать большее число состояний в рамках анализа одного пути выполнения, что повышает шансы обнаружения ошибок методами динамической символьной интерпретации. Кроме того, учет семантики функций предоставляет возможность детектировать ошибки переполнения буфера на уровне функций.

Как видно из таблицы 1, ни один существующий инструмент не поддерживает моделирование формулой возвращаемого значения всех основных функций из стандартной библиотеки Си. Часть инструментов могут подменять анализ функции символьной интерпретацией собственной реализации этой функции, добавлять дополнительные ограничения в предикат пути для описания трассы внутри функции, либо моделировать функцию условным переходом, учитывающим ее характерные возвращаемые значения. В данной работе необходимо разработать семантические модели для возвращаемого значения функций, чтобы повысить вероятность обнаружения ошибок. Отдельно следует разработать модель для функции конвертации строки в число `strtol`, т. к. она вызывается

Таблица 1 — Сравнение методов моделирования семантики функций в различных инструментах

Инструмент	`strlen/strchr`	`strcmp/memcmp`	`strtol`
KLEE [39]	подмена реализации (на собственную и µclibc [61])		
S^2E [48]	формула	формула	—
angr [54]	ограничение	формула	формула
QSYM [17]	конкретизация всех функций libc		
SymCC [18]	ограничение	ограничение	—
Fuzzolic [20]	ветвление	ветвление	—

внутри функции `scanf` для чтения целых чисел со стандартного потока ввода. Функция `scanf` повсеместно используется в наборах тестов Juliet [26], на которых требуется измерить точность разработанного метода поиска ошибок. Также для ускорения символьной интерпретации и борьбы с избыточной помеченностью стоит пропускать интерпретацию функций аллокаторов и печати.

1.4 Поиск ошибок с помощью символьной интерпретации

1.4.1 Санитайзеры

Прежде чем приступить к описанию методов обнаружения ошибок с помощью символьной интерпретации, необходимо осветить, пожалуй, самую распространенную технику динамического анализа для обнаружения ошибок во время выполнения программы. Санитайзеры [12], разработанные в компании Google [63], позволяют обнаруживать различные ошибки во время работы программы, в т. ч. ошибки работы с памятью и неопределенного поведения. Следует отметить, что санитайзеры способны детектировать ошибку только в том случае, когда эта ошибка проявляется на входных данных, на которых была запущена программа. Методы символьной интерпретации, напротив, позволяют генерировать входные данные для воспроизведения ошибки в результате интерпретации программы с входными данными, изначально не приводящи-

ми к проявлению ошибки. Однако санитайзеры могут быть использованы для верификации истинно положительных срабатываний, полученных в результате символьной интерпретации. А также могут служить вспомогательным средством для поиска ошибок [39; 64].

Реализация санитайзеров основана на инструментации кода программы на этапе его компиляции. Например, AdressSanitizer [65] дублирует эффекты выполнения инструкций, работающих с памятью программы, в теневой памяти, которая позволяет следить за корректностью осуществления различных операций с памятью. При использовании UndefinedBehaviorSanitizer в код программы на этапе компиляции встраиваются различные проверки на целочисленные переполнения и другие ошибки неопределенного поведения.

Существует несколько различных санитайзеров: AddressSanitizer, LeakSanitizer, ThreadSanitizer, UndefinedBehaviorSanitizer и MemorySanitizer.

AddressSanitizer [65] позволяет обнаруживать ошибки работы с памятью во время выполнения программы: переполнение буфера на стеке, куче, переполнение глобальных массивов, использование указателя после освобождения памяти, на которую он ссылается, двойное или некорректное освобождение памяти и др.

LeakSanitizer позволяет обнаруживать утечки памяти. Для использования этого санитайзера отдельная инструментация программы не требуется. Однако также он встроен в AddressSanitizer, что позволяет использовать эти два санитайзера совместно.

ThreadSanitizer [66] позволяет обнаруживать ошибки гонки данных, которые происходят, когда разные потоки выполнения обращаются к одним и тем же областям памяти без синхронизации и где хотя бы одно из обращений является записью в эту область памяти.

UndefinedBehaviorSanitizer позволяет обнаруживать ошибки неопределенного поведения, являющиеся результатом выполнения операций с неспецифицированными семантиками, таких как деление на нуль, целочисленное переполнение, использование неинициализированной нестатической переменной и т. п.

MemorySanitizer [67] позволяет обнаруживать ошибки неинициализированного чтения из памяти. Этот санитайзер находит случаи, когда производится чтение из памяти, выделенной на стеке или куче, до того, как в эту память было что-либо записано.

1.4.2 Поиск ошибок одновременно с открытием путей в KLEE

KLEE [39] позволяет искать ошибки во время символьной интерпретации. Потенциально опасные операции порождают дополнительные условные переходы, которые проверяют, существуют ли входные данные, которые могут привести к ошибке. Например, операция деления порождает переход, который проверяет равенство делителя нулю. KLEE обрабатывает такие условные переходы аналогично обычным. Если предикат, содержащий условие возникновения ошибки, выполним, то KLEE генерирует входные данные для воспроизведения ошибки и завершает символьную интерпретацию на этом пути. Однако интерпретация продолжается на пути, который не содержит ошибки. При этом в предикат пути дополнительно добавляется условие на отсутствие ошибки.

Обнаружение ошибки доступа к памяти происходит путем создания перехода с условием того, что разыменуемый адрес лежит за пределами границ массива. Для этого KLEE поддерживает теневую память, где хранятся адреса и размеры всех аллоцированных объектов памяти. Размер статического массива является константой в промежуточном представлении LLVM [47], полученного из исходного кода. А значения размера динамического массива на куче определяется в процессе интерпретации.

Для обнаружения целочисленного переполнения требуется собрать анализируемую программу с санитайзерами. KLEE клонирует символьные состояния на условных переходах внутри кода санитайзеров, которые проверяют факт переполнения. Таким образом, обнаруживаются пути, на которых результат целочисленной арифметической операции переполняется.

1.4.3 Автоматическая генерация эксплойтов в Mayhem

Mayhem — это система для автоматической эксплуатации уязвимостей в бинарном коде [44]. Mayhem генерирует эксплойты для каждой возможной перезаписи счетчика инструкций, обычно вызываемой переполнением буфера (общий метод генерации эксплойтов описан в разделе 1.2.4). Когда Mayhem находит символьный адрес инструкции, он пытается сгенерировать эксплуатацию перехода по

регистру. В таком случае счетчик инструкций должен указывать на инструкции вида `jmp eax`, а регистр `eax` должен указывать на место в памяти, где может быть размещен некоторый произвольный код. Такие условия конструируются в виде предиката безопасности, конъюнкция которого с предикатом пути передается математическому решателю. Если решение существует, то ошибка может быть проэксплуатирована на входных данных, полученных от решателя. Если решения не существует, то генерируется более простой предикат безопасности, где счетчик инструкций должен указывать конкретно на место в памяти, где может быть размещен произвольный код.

Для форматных строк Mayhem проверяет, содержат ли аргументы для форматной строки или сама форматная строка символьные байты. Если так, то Mayhem пытается разместить в аргументе данные, которые приведут к перезаписи адреса возврата из форматирующей функции.

Следует отметить, что Mayhem применяет предикаты безопасности именно для эксплуатации уже найденных ошибок, а не для поиска новых дефектов. Поиск аварийных завершений осуществляется посредством открытия нового кода путем инвертирования переходов с помощью символьной интерпретации.

1.4.4 Гибридное тестирование с прицелом на поиск ошибок в SAVIOR

SAVIOR — это фреймворк для гибридного фаззинга программного обеспечения, нацеленный на поиск ошибок [68]. Символьная интерпретация в SAVIOR осуществляется с помощью KLEE [39]. Для этого авторы модифицировали код KLEE так, чтобы он поддерживал конколическую (concolic, динамическую) символьную интерпретацию. Для этого в начальном состоянии они присваивают символьным переменным значения конкретных входных данных. В то же время KLEE по-прежнему интерпретирует каждую встреченную инструкцию. Однако при достижении условного перехода символьная интерпретация продолжается только в одном направлении, заданном входными данными. А для противоположного условия производится запрос к математическому решателю, и генерируются входные данные для исследования альтернативной ветви условного перехода. При этом значения байтов, для которых отсутствуют ограничения, берутся из изначальных входных данных. Схема работы SAVIOR основывается на двух

механизмах: приоритизации файлов в корпусе для повышения вероятности обнаружения ошибок и верификации найденных ошибок.

Суть механизма приоритизации заключается в том, чтобы производить символьную интерпретацию не на всех наборах входных данных, а отдавать приоритет таким входным данным, которые с большей вероятностью приведут к обнаружению программных ошибок. До тестирования программного обеспечения SAVIOR анализирует исходный код и статически помечает потенциально опасные места в программе. Также SAVIOR высчитывает набор базовых блоков, достижимых от каждого условного перехода. Во время символьной интерпретации SAVIOR считает приоритетными те наборы входных данных, которые смогут привести к посещению наибольшего числа условных переходов, ведущих к еще не исследованным потенциально опасным частям кода программы. Таким образом, SAVIOR ускоряет обнаружение новых ошибок в программах. SAVIOR работает одновременно с фаззером AFL [45], а приоритизированные входные файлы для поиска ошибок берет из очереди фаззера.

Суть механизма верификации заключается в следующем. Получая наборы входных данных в результате фаззинг-тестирования, SAVIOR символьно интерпретирует программу на этих входных данных и для каждого ранее помеченного потенциально опасного места верифицирует соответствующий ему предикат, проверяя его с учетом пути выполнения. Если предикат удовлетворим, то ошибка подтверждена. Это позволяет SAVIOR генерировать доказательство обнаружения ошибки в виде конкретных входных данных или же сообщать о ее отсутствии на конкретном пути выполнения программы.

Пометка потенциально опасных мест производится с помощью Undefined Behavior Sanitizer, используя который SAVIOR инструментирует потенциально опасные места в программе предикатами, удобными для математического решателя, которые будут проверяться на этапе верификации. SAVIOR поддерживает предикаты для ошибок выхода за границы массива, битового сдвига на слишком большое значение, знакового и беззнакового целочисленного переполнения. С помощью статического анализа исходного кода SAVIOR учитывает знаковость операндов и их размеры при составлении предикатов для дальнейшей проверки на этапе верификации.

1.4.5 Поиск целочисленного переполнения в IntScope

IntScope — инструмент для обнаружения ошибок целочисленного переполнения при помощи символьной интерпретации программного обеспечения [69]. IntScope предлагает следующий подход для обнаружения ошибок целочисленного переполнения. При помощи символьной интерпретации анализируются инструкции, реализующие арифметические операции. Далее найденные места с потенциальной ошибкой целочисленного переполнения проверяются лениво, то есть арифметическая инструкция проверяется не сразу, а лишь когда помеченное символьное значение используется в чувствительных точках, таких как выделение памяти функциями `malloc`, `calloc`. IntScope предупреждает о потенциальной ошибке целочисленного переполнения, только если помеченное символьное значение, используемое в чувствительных местах, может переполниться.

Источник ошибки — арифметическая инструкция в программе, где потенциально может произойти ошибка целочисленного переполнения, и один из операндов получен из помеченных данных. Помеченные данные могут быть получены из недоверенных источников входных данных, таких как сеть, файлы или опции командной строки. Функциями, с помощью которых могут быть получены помеченные данные, считаются `read`, `fread`, `recv`.

Сток ошибки — место в программе, в котором использование переполненного значения может привести к дальнейшим программным уязвимостям. В IntScope выделяются следующие стоки ошибок:

- выделение памяти (использование переполненного значения в аргументах таких функций, как `malloc`);
- доступ к памяти (переполненное значение использовано в качестве индекса или смещения при разыменовании адреса);
- условный переход (использование переполненного значения в условном переходе, который может привести к пропуску проверок на безопасность).

IntScope осуществляет поиск ошибок методами статического анализа бинарного кода. Сначала ассемблерный код транслируется в промежуточное представление, разработанное авторами. Затем строятся граф потока управления и граф вызовов. Используя анализ помеченных данных из графов получаются

срезы — подграфы, которые содержат только помеченные пути от недоверенных источников до потенциальных стоков ошибок. IntScope обходит такие срезы графов в глубину и производит символьную интерпретацию промежуточного представления. При достижении потенциального стока ошибки, использующего уже найденный источник ошибки, происходит соответствующая проверка. Если полученный предикат выполним, то IntScope сообщает о наличии ошибки. При этом входные данные для ее подтверждения не генерируются из-за использования статического анализа. Ее подтверждение может быть осуществлено вручную или же методами направленного фаззинга и динамического анализа для достижения точки проявления ошибки. Однако направленный поиск далеко не всегда способен подобрать входные данные для воспроизведения ошибки, в то время как ошибка все равно присутствует. Отдельно следует отметить, что подход, использующий промежуточное представление бинарного кода, неминуемо влечет за собой падение скорости символьной интерпретации.

В бинарном коде отсутствует явная информация о знаковости операндов арифметической инструкции. Чтобы избежать ложных срабатываний, IntScope пытается косвенно определить знаковость. Если сток ошибки — аргумент функции выделения памяти, то значение проверяется как беззнаковое. Также информацию о знаковости символьного значения IntScope получает на основе анализа помеченных инструкций условных переходов, таких как `JG` (знаковый), `JA` (беззнаковый) и подобных.

1.4.6 Обнаружение уязвимостей целочисленного переполнения с помощью графов программ

В данной работе авторы предлагают свой подход для поиска ошибок целочисленного переполнения в бинарном коде [70]. Для машинного кода строится усеченный граф потока управления. Слои графа проверяются на выполнение условий для обнаружения уязвимостей. Главная идея подхода заключается в использовании символьной интерпретации кода для конструирования условий на входные переменные, которые проявляют уязвимость. Выполнимость полученных условий проверяется математическим решателем. Представленный подход можно разделить на следующую последовательность фаз:

- **Построение графа потока управления.** Входной файл используется для построения графа потока управления тестируемой программы, в результате чего должны быть обнаружены входные и выходные вершины. Входная вершина отображает части кода, где происходит ввод данных, а выходная вершина отображает части кода, где происходит вызов функций выделения памяти.
- **Отсеивание неиспользованного пути в графе потока управления.** Все вершины, которые не были использованы в любом из путей от входной до выходной вершины, должны быть удалены.
- **Построение символьного входа.** Входные данные и неинициаллизированные ячейки памяти полагаются символьными. Каждая ячейка изначально отображается как пара, состоящая из символьного числа и символьного адреса.
- **Символьная эмуляция.** Каждый путь обрабатывается независимо. Обработка включает в себя символьную эмуляцию машинных инструкций. Она затрагивает обе части отображения ячеек (число и адрес). Символьные ячейки могут потерять один из этих параметров, если в процессе интерпретации становится понятно, является ли символьная переменная адресом или числом. Например, если переменная была разыменована, то она является адресом, тогда параметр числа отбрасывается.
- **Построение системы условий.** Система условий описывает достижимость выходной вершины от входной. Ограничения добавляются в систему по мере исследования путей выполнения. Условия для проверки возможности ошибки используются в качестве последнего условия в системе.
- **Проверка существования решения системы.** Созданные условия проверяются с помощью автоматической системы доказательства теорем. Если система имеет решение, то программа уязвима, а сгенерированное решение является подтверждением найденной ошибки.

Символьная эмуляция программы применяется к каждому пути в графе от входной вершины до выходной. Уравнения и неравенства на символьные переменные встречаются между вершинами графа и являются условиями операторов ветвления или условиями на переполнение в арифметических инструкциях, таких как `add`, `sub`, `mul`, `imul`, `shl` и т. д. Составленная система уравнений и неравенств на символьные переменные проверяется решателем последовательно по

достижению выходной вершины, то есть вызова некоторой опасной функции, использование переполненного значения в которой может привести к серьезным последствиям. Если система имеет решение, то это говорит об уязвимости в вызове функции.

1.4.7 Фаззинг с обратной связью по санитайзерам в ParmeSan

ParmeSan — инструмент для фаззинга с обратной связью по санитайзерам, который оптимизирован специально для покрытия ошибок [64]. Основной идеей является использование инструментации, производимой санитайзерами, для обеспечения эффективного механизма поиска интересных базовых блоков для обратной связи с фаззером. В отличие от фаззеров с обратной связью по покрытию, которые вслепую покрывают все базовые блоки программы, ParmeSan направлен на исследование путей выполнения программы, имеющих наибольший шанс проявления ошибок за кратчайшее время.

Первым делом ParmeSan исследует интересные целевые точки кода из санитайзера, которым была проинструментирована программа, убирая неинтересные проверки. Затем динамически конструируется граф потока управления, чтобы направить фаззинг на конкретные целевые точки. Сам процесс фаззинга основан на двух этапах: фаззинга для конструирования графа потока управления и фаззинга целевых точек программы, нацеленного на поиск ошибок в программе. Для ускорения работы второго этапа ParmeSan использует доступную из анализа потока данных информацию.

Одной из основных идей, заложенных в работу ParmeSan, является то, как система направляет фаззинг по путям выполнения, ведущим к интересным блокам. ParmeSan в процессе работы строит граф потока управления, выполняет программу на исходных наборах входных данных и смотрит, до каких базовых блоков дошло выполнение программы. Основываясь на построенном графе потока управления, ParmeSan с помощью введенной авторами метрики вычисляет дистанцию от пройденных базовых блоков до целевых блоков. На основе полученных значений метрики и графа потока управления фаззинг направляется так, чтобы как можно скорее дойти до целевых блоков, содержащих потенциальную ошибку. Для этого представленный авторами фаззер выбирает следующим для

мутации такой набор входных данных, при выполнении которого были посещены базовые блоки, имеющие наименьшую дистанцию с целевыми блоками. Фаззер повышает приоритет инвертирования тех условных переходов, которые наиболее близко расположены к целевым базовым блокам.

Санитайзеры инструментируют программу двумя разными способами. Некоторые инструментации просто обновляют внутренние структуры данных (например, теневую память). Другие же обнаруживают ошибку, используя условный переход, зависящий от внутренних структур данных или непосредственного состояния программы. ParmeSan преследует две цели. Во-первых, направить фаззинг в те точки программы, где санитайзер обновляет свои внутренние структуры данных. Во-вторых, решить вставленные санитайзерами предикаты (выполнимость которых означает наличие ошибки) с помощью мутаций на основе анализа потока данных.

Таким образом, процесс работы ParmeSan состоит из трех фаз. Первая фаза — быстрое исследование путей выполнения с опорой на покрытие и трассировка для получения графа потока управления. В течение первой фазы ParmeSan динамически собирает трассы выполнения и пытается построить граф потока управления настолько точно, насколько это возможно. Вторая фаза — направленное исследование для достижения целевых базовых блоков. Третья фаза генерации входных данных (с помощью мутаций), проявляющих ошибку, начинается, когда целевой блок был достигнут.

1.4.8 Выводы

Санитайзеры (разд. 1.4.1) являются надежным средством для динамического детектирования ошибок работы с памятью и неопределенного поведения. Они позволяют обнаруживать ошибки, которые проявляются во время работы программы, запущенной на конкретных входных данных. Однако они не способны обнаружить ошибку, если она не проявляется на этих входных данных. Напротив, в результате динамической символьной интерпретации программы на входных данных, которые не приводят ошибке, можно генерировать новые входные данные для проявления ошибки. А санитайзеры могут быть использованы для верификации того, что сгенерированные входные данные действительно выявля-

ют ошибку. В данной работе санитайзеры будут применяться для верификации истинно положительных срабатываний, полученных во время динамической символьной интерпретации.

KLEE [39] работает только с исходным кодом и требует сборки анализируемого кода (в т. ч. стандартных библиотек) специальным образом. Более того, KLEE осуществляет полностью символьную интерпретацию (без реального запуска программы) с клонированием состояний, что влечет за собой проблемы с масштабированием скорости и потребления памяти на больших программах.

Mayhem [44] — инструмент с закрытым кодом, который нацелен на эксплуатацию уязвимостей. Методами динамической символьной интерпретации возможно обнаружить ошибочное состояние программы. Однако это состояние может быть далеко от эксплуатируемого (в частности, от перезаписи счетчика инструкций). Такие ошибки Mayhem не обнаружит. В то же время Mayhem одним из первых начал применять символьные предикаты безопасности к бинарному коду.

SAVIOR [68] работает с исходным кодом и использует файлы из корпуса фаззера в качестве входных данных для динамической символьной интерпретации. В роли символьного интерпретатора выступает модифицированная версия KLEE [39], которая была адаптирована для динамической символьной интерпретации одного пути. Однако все встреченные инструкции по-прежнему эмулируются через KLEE, что влечет за собой понижение скорости (сопоставимое с QSYM [17]) и точности генерируемых входных данных для подтверждения ошибок по сравнению с подходом, когда конкретные значения берутся из реального выполнения программы. SAVIOR требует инструментации используемых анализируемой программой библиотек. Также требуется применение патчей к коду LLVM [47], что не является переносимым решением. Более того, авторы отмечают проблемы с анализом C++ кода.

IntScope [69] применяют статический анализ бинарного кода и символьную интерпретацию для поиска ошибок целочисленного переполнения. К сожалению, инструмент и его код не доступны. Авторы предлагают концепцию источников и стоков ошибки для уменьшения числа ложных срабатываний. Также они определяют знаковость операндов арифметических инструкций по знаковости помеченных инструкций условных переходов. Однако IntScope реализует медленную символьную интерпретацию собственного промежуточного представления машинного кода путем обхода в глубину графов программы. Аналогичные проблемы присущи и статье [70], где так же производится символьная эмуляция на

графах программы. Код обоих решений закрыт. Кроме того, IntScope не генерирует входные данные для воспроизведения ошибок из-за применения статического анализа. В данной работе будут использованы схожие концепции источника и стока ошибки, а также определения знаковости операндов арифметических инструкций.

Авторы ParmeSan [64] предложили использовать информацию от инструментации санитайзеров для направленного фаззинга в сторону детектирования ошибок. Однако предложенный алгоритм является переборным (мутационным) и не позволяет сходу генерировать входные данные для воспроизведения ошибки с учетом семантики программы.

Глава 2. Алгоритм слайсинга предиката пути

Динамической символьной интерпретации свойственны проблемы избыточной и недостаточной помеченности. Избыточная помеченность влечет за собой неминуемый рост числа ограничений в предикате пути, из-за чего некоторые запросы к решателю становятся невыполнимыми. Недостаточная помеченность, в свою очередь, приводит к отсутствию необходимых ограничений и неточным решениям. Подробнее о проблемах недостаточной и избыточной помеченности рассказывается в разделе 1.3. В данной главе предлагается алгоритм слайсинга предиката пути, который позволяет преодолевать обе проблемы. Алгоритм устраняет избыточные ограничения из предиката пути на основе анализа зависимостей символьных переменных по данным. Представленный алгоритм вдохновлен аналогичным методом в KLEE [38; 39] (разд. 1.3.1), который применим только к исходному коду. В данной работе получен новый алгоритм, устраняющий избыточные ограничения во время динамической символьной интерпретации бинарного кода программы.

2.1 Построение предиката пути

Динамическая символьная интерпретация [23; 41] позволяет исследовать вариацию входных данных на некотором фиксированном пути выполнения программы. Изначально производится запуск исследуемой программы на конкретных входных данных. Во время выполнения детектируется точка получения пользовательских данных, например, системный вызов `read` для чтения файла. На обнаруженной точке определяются адрес и размер буфера с входными данными. Каждый байт входных данных моделируется свободной символьной переменной, которая может принимать произвольное значение. Далее производится символьная интерпретация выполненных инструкций на исследуемом пути. Каждая машинная инструкция моделируется SMT [42] формулой в соответствии с ее операционной семантикой. Символьный интерпретатор поддерживает символьное состояние, которое является отображением из регистров и байтов памяти в символьные формулы. Все изменения регистров (или памяти) обновляют сим-

вольное состояние новыми формулами. Во время интерпретации инструкции значения регистров и памяти, зависящие от символьных переменных, берутся из символьного состояния, а конкретные значения — из реального выполнения. Условия переходов представляются булевыми предикатами и формируют предикат пути. Таким образом, предикат пути содержит ограничения, которые описывают исследуемый путь. Решением конъюнкции всех таких ограничений являются входные данные, которые проведут программу по тому же пути выполнения.

Построив предикат пути, можно инвертировать встреченные на исследуемом пути условные переходы. Для этого условие инвертируемого перехода берется с отрицанием. А решение составленной конъюнкции отрицания целевого перехода с ограничениями предыдущих переходов из предиката пути позволит получить входные данные, которые проведут выполнение в другую сторону. Таким образом, можно открывать новые пути.

Аналогичным образом производится обнаружение ошибок. Для этого составляется конъюнкция предиката пути и предиката безопасности. При этом предикат пути отвечает за достижимость точки проявления ошибки. А предикат безопасности описывает условие возникновения ошибки, например, равенство делителя нулю. Если полученный предикат выполним, то его модель (решение) по сути является входными данными для воспроизведения найденной ошибки.

Рассмотрим построение предиката пути и инвертирование последнего перехода на примере в таблице 2. Изначально символьное состояние содержит отображение из регистров `rax`, `rbx` и `rcx` в свободные символьные переменные φ_i. Символьная интерпретация условного перехода сравнения регистра `rcx` с нулем добавляет в предикат пути соответствующее ограничение $\varphi_3 \neq 0$. Затем происходит сложение регистров `rax` и `rbx`, которое порождает новую символьную формулу $\varphi_4 = \varphi_1 + \varphi_2$. Эта формула присваивается результирующему регистру $rax = \varphi_4$ в символьном состоянии. Далее аналогично значение `rax` уменьшается на 2, что порождает очередную формулу φ_5, которая в свою очередь снова обновляет отображение `rax` в символьном состоянии. Таким образом, вычисления инструкций порождают новые формулы в SSA-форме [53] согласно их операционной семантике, а присваивание результата в регистр влечет за собой обновление отображения для этого регистра в символьном состоянии на новую формулу. В результате символьной интерпретации в предикат пути добавляются еще два ограничения $\varphi_2 \geqslant 10$ и $\varphi_5 < 30$ для условных переходов `jl` и `jge`

Таблица 2 — Пример построения предиката пути и инвертирования перехода

Символьное состояние	**Инструкция**	**Множество формул**	**Предикат пути** Π
$rax = \varphi_1$ $rbx = \varphi_2$ $rcx = \varphi_3$	—	$\varnothing$	$true$
$rax = \varphi_1$ $rbx = \varphi_2$ $rcx = \varphi_3$	`cmp rcx, 0` `jz .exit`	$\varnothing$	$\varphi_3 \neq 0$
$rax = \varphi_4$ $rbx = \varphi_2$ $rcx = \varphi_3$	`add rax, rbx`	$\varphi_4 = \varphi_1 + \varphi_2$	$\varphi_3 \neq 0$
$rax = \varphi_5$ $rbx = \varphi_2$ $rcx = \varphi_3$	`sub rax, 2`	$\varphi_4 = \varphi_1 + \varphi_2$ $\varphi_5 = \varphi_4 - 2$	$\varphi_3 \neq 0$
$rax = \varphi_5$ $rbx = \varphi_2$ $rcx = \varphi_3$	`cmp rbx, 10` `jl .exit`	$\varphi_4 = \varphi_1 + \varphi_2$ $\varphi_5 = \varphi_4 - 2$	$\varphi_3 \neq 0 \wedge \varphi_2 \geqslant 10$
$rax = \varphi_5$ $rbx = \varphi_2$ $rcx = \varphi_3$	`cmp rax, 30` `jge .exit`	$\varphi_4 = \varphi_1 + \varphi_2$ $\varphi_5 = \varphi_4 - 2$	$\varphi_3 \neq 0 \wedge \varphi_2 \geqslant 10 \wedge \varphi_5 < 30$
Инвертирование последнего перехода $\varphi_3 \neq 0 \wedge \varphi_2 \geqslant 10 \wedge \varphi_5 \geqslant 30$			

соответственно. Получившийся предикат пути $\varphi_3 \neq 0 \wedge \varphi_2 \geqslant 10 \wedge \varphi_5 < 30$ ($\varphi_3 \neq 0 \wedge \varphi_2 \geqslant 10 \wedge \varphi_1 + \varphi_2 - 2 < 30$ после подстановки формул) описывает ограничения на значения регистров `rax`, `rbx` и `rcx` для прохождения программы по пути, в рамках которого производилась динамическая символьная интерпретация. Чтобы инвертировать последний условный переход `jge`, его условие берется с отрицанием и составляется конъюнкция с предикатов пути: $\varphi_3 \neq 0 \wedge \varphi_2 \geqslant 10 \wedge \varphi_5 \geqslant 30$. Сконструированный предикат передается SMT-решателю, который возвращает модель — значения переменных φ_1, φ_2 и φ_3, при которых предикат истинен. Если присвоить значения из модели регистрам `rax`,

`rbx` и `rcx`, то исследуемая программа пройдет по тому же пути до перехода jge, который выполнится и передаст управление на метку `.exit`.

В данной работе используется логика битовых векторов SMT (QF_BV) для описания значений регистров и памяти. Такая логика реализует модульную арифметику и позволяет моделировать работу реального процессора. Каждый байт входных данных моделируется свободной переменной типа 8-битового вектора: `(declare-fun sym_i () (_ BitVec 8))`. Тогда регистр `rax` из примера в таблице 2 описывается конкатенацией 8 таких переменных (битовых векторов): `(define-fun phi_1 () (_ BitVec 64) (concat sym_8 sym_7 sym_6 sym_5 sym_4 sym_3 sym_2 sym_1))`. А сложение регистров `rax` и `rbx` представляется сложением битовых векторов по модулю их размера: `(define-fun phi_4 () (_ BitVec 64) (bvadd phi_1 phi_2))`. В действительности флаги (CF, OF, ZF и др.) моделируются 1-битовыми векторами, и символьное состояние хранит отображение из каждого флага в текущую формулу. Интерпретация инструкции `cmp rcx, 0` по сути конструирует формулы для флагов (ZF, AF, CF, OF, PF, SF) и обновляет их отображения в символьном состоянии. При этом формула для флага ZF выглядит следующим образом: `(define-fun zf_1 () (_ BitVec 1) (ite (= phi_3 (_ bv0 64)) (_ bv1 1) (_ bv0 1)))`. В свою очередь, интерпретация инструкции условного перехода `jz .exit` добавляет ограничение в предикат пути `(assert (not (= zf_1 (_ bv0 1))))`, которое указывает на то, что флаг ZF не должен равняться нулю.

Динамическая символьная интерпретация осуществляется при помощи фреймворка Triton [23] (разд. 1.2.5). Разберем детали его работы на том же примере в таблице 2. Изначально конкретное состояние содержит отображение из регистров `rax`, `rbx` и `rcx` в их значения из реального выполнения. Для каждого байта этих регистров создается символьная переменная, которая аннотирована конкретным значением, чтобы в дальнейшем можно было вычислять значения символьных данных в реальном выполнении. После этого символьное состояние будет хранить отображение этих регистров в символьные выражения конкатенации их отдельных байтов (символьных переменных). Triton позволяет символьно интерпретировать инструкцию одну за другой. Вычисления инструкций моделируются абстрактными синтаксическими деревьями (AST), вершинами которых являются SMT операции, символьные переменные и константы. Результат инструкции записывается в символьное выражение, которое по сути является

нумерованной формулой в виде AST. Абстрактные синтаксические деревья также могут иметь специальную вершину, ссылающуюся по номеру на другое символьное выражение. Так вычисления могут производится в SSA-форме и ссылаться на ранее вычисленные значения. Для интерпретации инструкции `add rax, rbx` Triton получает символьные выражения φ_1 и φ_2 из символьного состояния, конструирует AST сложения этих выражений и сохраняет его в символьное выражение φ_4, создает символьные выражения для флагов, обновляет отображение `rax` и флагов на новые выражение в символьном состоянии, вычисляет конкретные значения AST и обновляет регистры в конкретном состоянии. Во время интерпретации условного перехода `jl .exit` выражения для флагов SF и OF извлекаются из символьного состояния. Далее составляется AST для условия перехода. Реально выполненное направление перехода вычисляется по AST. При этом предикат пути хранит для каждого условного перехода обе ветви, описанные ребрами: адрес перехода, адрес назначения, AST условия выполнимости ребра и флаг, выполнился ли переход в реальном выполнении. Для решения составляется AST конъюнкции формул из предиката пути. Затем производится топологическая сортировка AST с раскрытием всех ссылок на другие символьные выражения. Полученный список вершин AST транслируется во внутреннее SMT представление математического решателя [34; 35], который в свою очередь пытается найти модель для описанного предиката. Такой подход позволяет транслировать в SMT только используемые в AST символьные выражения.

2.2 Алгоритм устранения избыточных ограничений из предиката пути

Заметим, что на примере в таблице 2 можно без потери точности устранить конъюнкт $\varphi_3 \neq 0$ из итогового предиката для инвертирования последнего перехода. Тогда математическому решателю будет передан предикат $\varphi_2 \geqslant 10 \wedge \varphi_5 \geqslant 30$ ($\varphi_2 \geqslant 10 \wedge \varphi_1 + \varphi_2 - 2 \geqslant 30$ после подстановки). SMT-решатель подберет модель для переменных φ_1 и φ_2. А значение φ_3 можно взять из реального выполнения. Оно будет удовлетворять устраненному конъюнкту, т. к. предикат пути строился в рамках конкретного пути, заданного входными значениями. По сути входные данные уже являются решением предиката пути. Таким образом, возможно устранять часть ограничений из предиката пути, а значения недостающих символьных

Листинг 2.1 Пример недостаточной помеченности, устраняемой слайсингом предиката пути

```
 1 char* syms = "SLICING FIX IT!\n";
 2 
 3 int main(int argc, char **argv) {
 4   FILE *ptr = fopen(argv[1], "rb");
 5   unsigned *b = malloc(6 * sizeof(int));
 6   fread(b, 6, sizeof(int), ptr);
 7   int len = strlen(syms);
 8   if (b[0] < len)
 9     if (syms[b[0] % len] == '!')
10       if (b[2] > '@')
11         if (b[5] + b[4] < 'B')
12           if (b[3] + b[5] > '@')
13             if (b[1] + b[3] > '@')
14               if (b[4] < '9')
15                 if (b[1] > '@')
16                   printf("OK\n");
17                 else
18                   printf("FAIL\n");
19 }
```

переменных брать из входных данных. Следовательно, математический решатель будет быстрее обрабатывать запросы, содержащие меньше ограничений и свободных переменных. Кроме того, устранение избыточных ограничений позволяет бороться с проблемами недостаточной и избыточной помеченности (разд. 1.3). Так удаление части ограничений по определению уменьшает избыточную помеченность.

Рассмотрим пример на листинге 2.1, где присутствует недостаточная помеченность. Выполнение программы на изначальных входных данных приводит к печати `FAIL` на строке 18. Зададимся целью инвертировать переход на строке 15, чтобы было напечатано сообщение `OK`. Строка 9 содержит символьный индекс `b[0] % len` (разд. 1.3.4), который зависит от пользовательских данных. Во время динамической символьной интерпретации этот индекс будет конкретизован значением из реального выполнения. Тогда выражение `syms[b[0] % len]` является константным и содержит один конкретный символ из массива `syms`. Таким образом, переход на строке 9 недостаточно помечен (не символьный) и не будет добавлен в предикат пути. Построенный (согласно разделу 2.1)

предикат для инвертирования целевого перехода будет иметь следующий вид: $\texttt{b[0]} < \texttt{len} \wedge \texttt{b[2]} > \texttt{'@'} \wedge \texttt{b[5]} + \texttt{b[4]} < \texttt{'B'} \wedge \texttt{b[3]} + \texttt{b[5]} > \texttt{'@'} \wedge \texttt{b[1]} + \texttt{b[3]} > \texttt{'@'} \wedge \texttt{b[4]} < \texttt{'9'} \wedge \texttt{b[1]} > \texttt{'@'}$. В итоге ограничение для условного перехода на строке 8 добавлено в предикат пути, и математический решатель может подобрать такую модель для `b[0]`, что она не будет удовлетворять условию `syms[b[0] % len] == '!'` перехода на строке 9. Таким образом, в предикате пути отсутствует необходимое ограничение на `b[0]` из строки 9. Если запустить программу на входных данных из полученной модели, то условие на строке 9 не выполнится, и программа вообще ничего не напечатает.

Однако инвертирование перехода на строке 15 по сути не зависит от символьной переменной `b[0]`. Если убрать ограничение $\texttt{b[0]} < \texttt{len}$ из полученного выше предиката и взять значение `b[0]` из начальных входных данных, то полученные входные данные приведут программу к печати `OK`. Ниже предложен алгоритм слайсинга предиката пути, который позволит устранять избыточные ограничения из предиката пути на основе анализа зависимостей символьных переменных по данным. Представленный алгоритм уменьшает (или вообще сводит на нет) влияние избыточной и недостаточной помеченности на точность генерируемых входных данных.

Алгоритм 1 слайсинга предиката пути [25] позволяет устранять избыточные ограничения из предиката пути без потери точности. Алгоритм принимает на вход целевое ограничение $cond$ (условие для инвертирования перехода или предикат безопасности) и последовательность ограничений П из предиката пути до точки проверки целевого ограничения. Предварительно вычисляются используемые символьные переменные для каждого ограничения в предикате пути и целевого условия. Их вычисление производится путем обхода абстрактных синтаксических деревьев для символьных выражений. Изначально множество переменных слайсинга $vars$ содержит символьные переменные, от которых зависит $cond$. Далее происходят проходы по всем ограничениям c из предиката пути П. Если переменные $vars$ пересекаются с используемыми переменными в c ($used_variables(c)$), то множество $vars$ пополняется символьными переменными из c. Проходы происходят до тех пор, пока множество $vars$ пополняется новыми элементами. В результате будут получены все переменные $vars$, которые транзитивно зависят от переменных в целевом ограничении $cond$. В конце производится последний проход по ограничениям из предиката пути П. Ограничения из П, используемые переменные которых пересекаются с $vars$, составляются в конъ-

Алгоритм 1: Алгоритм слайсинга предиката пути

Входные данные: $cond$ — ограничение для инвертирования целевого перехода (или предикат безопасности), Π — предикат пути (ограничения пути до целевого перехода).

Результат: Π_S — предикат с устраненными избыточными ограничениями.

```
vars ← used_variables(cond)          /* переменные слайсинга */
change ← vars
while change ≠ ∅ do
    change ← vars
    /* итерирование по ограничениям пути                     */
    forall c ∈ Π do
        if vars ∩ used_variables(c) ≠ ∅ then
            vars ← vars ∪ used_variables(c)
    change ← vars \ change
/* ограничение для инвертирования/предикат
   безопасности                                              */
Π_S ← cond
/* итерирование по ограничениями пути                        */
forall c ∈ Π do
    if vars ∩ used_variables(c) ≠ ∅ then
        Π_S ← Π_S ∧ c
return Π_S
```

юнкцию вместе с ограничением $cond$. Полученный предикат Π_S будет отвечать решаемой задаче: инвертированию перехода или проверке предиката безопасности. Таким образом, берется лишь часть ограничений из предиката пути. При этом далее в разделе 2.3 будет показано, что алгоритм решает ту же задачу.

В результате применения алгоритма слайсинга SMT-решатель возвращает модель для некоторого подмножества символьных переменных. Значения недостающих переменных получаются из начальных входных данных. Полученное таким образом решение корректно, т. к. изначальные входные данные уже являются решением предиката пути.

Как было упомянуто выше, алгоритму слайсинга предиката пути требуется предварительно вычислить используемые символьные переменные в ограни-

Таблица 3 — Применение алгоритма слайсинга предиката пути к примеру на листинге 2.1

Строка	Переменные слайсинга $vars$
15	`b[1]`
13	`b[1],b[3]`
12	`b[1],b[3],b[5]`
11	`b[1],b[3],b[4],b[5]`
14	`b[1],b[3],b[4],b[5]`

чениях из предиката пути. Следует отметить, что используемые переменные в ограничениях предиката пути вычисляются единожды. При инвертировании разных условных переходов используются уже раннее сохраненные множества используемых символьных переменных. Более того, поддерживается кэш, хранящий по номеру символьного выражения множество используемых им переменных. Определение используемых переменных осуществляется по абстрактному синтаксическому дереву ограничения. Используемые в AST символьные переменные объединяются с переменными из ссылок на другие выражения. Переменные для выражений берутся либо из кэша, либо для них происходит аналогичное вычисление. Поскольку Triton поддерживает SSA-форму для символьных выражений, вычисление используемых переменных в выражениях происходит в порядке возрастания их номера.

В таблице 3 описывается применение алгоритма 1 слайсинга предиката пути к приведенному выше примеру на листинге 2.1. Изначально переменные слайсинга $vars$ содержат единственную используемую в целевом переходе на строке 15 переменную `b[1]`. Затем осуществляется проход по ограничениям из предиката пути. Переход на строке 13 тоже использует переменную `b[1]`, поэтому множество $vars$ пополняется переменной `b[3]` из условия перехода `b[1] + b[3] > '@'`. В результате еще одного прохода добавляется переменная `b[5]` из ограничения перехода на строке 12, использующего также переменную `b[3]`. Аналогичным образом отбираются условные переходы на строках 11 и 14. Результирующий предикат будет содержать конъюнкции ограничений всех переходов, которые используют переменные слайсинга `b[1]`, `b[3]`, `b[4]`, `b[5]`. Это переходы на строках 11–15: $\texttt{b[2]} > \texttt{'@'} \land \texttt{b[5]} + \texttt{b[4]} < \texttt{'B'} \land \texttt{b[3]} + \texttt{b[5]} > \texttt{'@'} \land \texttt{b[1]} + \texttt{b[3]} > \texttt{'@'} \land \texttt{b[4]} < \texttt{'9'} \land \texttt{b[1]} > \texttt{'@'}$. Обратим внима-

ние, что предикат не содержит символьной переменной `b[0]`, которая является недостаточно помеченной. В результате применения слайсинга был устранен переход на строке 8, а значение `b[0]` было взято из начальных входных данных. Таким образом, запуск программы на сгенерированных данных успешно приведет к печати `OK`.

В итоге, слайсинг предиката пути позволяет осуществлять более эффективную динамическую символьную интерпретацию. Разработанный алгоритм имеет следующие преимущества:

1. Слайсинг позволяет избегать избыточной помеченности и пропускать нерелевантные ограничения в предикате пути. Математический решатель требует меньше памяти и времени для решения запросов. Возвращаемая модель содержит только часть входных байтов, непосредственно отвечающих за инвертирование перехода (или истинность предиката безопасности). Таким образом, во входных данных подменяются значения лишь нескольких байтов.
2. Недостаточная помеченность приводит к нехватке необходимых ограничений в предикате пути. Так сгенерированные входные данные могут не провести программу по желаемому пути. Слайсинг удаляет потенциально недостаточно помеченные символьные переменные из запроса к решателю. Значения этих переменных берутся из начальных входных данных.

2.3 Исследование свойств и характеристик представленного алгоритма

В данном разделе исследуются формальные свойства и характеристики алгоритма слайсинга предиката пути. В разделе доказываются теоремы, что алгоритм конечен и не теряет решений. Более того, производится оценка асимптотической сложности алгоритма. Перед тем как приступить к изучению свойств и характеристик алгоритма 1, приведем необходимые определения.

Определение 1. *Модель для предиката* $P(v_1, ..., v_n)$ *— это такая подстановка* $v_i \leftarrow c_i, i = 1..n$*, где* c_i *— константы, при которой предикат истинен:* $P(c_1, ..., c_n) \equiv 1$.

Определение 2. *Символьная переменная — свободная переменная, которая ставится в соответствие с каждым входным байтом программы.*

Определение 3. *Предикат пути* Π*, построенный для пути выполнения, заданного конкретными значениями символьных переменных* α_i *(входными байтами программы) — это конъюнкция ограничений над символьными переменными и константами, каждая модель которой проведет программу по тому же пути выполнения.*

Следует отметить, что из определения предиката пути следует, что значения символьных переменных α_i являются для него моделью, т.к. это изначальные входные байты программы, которые проводят ее по заданному пути.

Теорема 1. *Алгоритм 1 слайсинга предиката пути конечен, т.е. завершается за конечное число итераций.*

Доказательство. Алгоритму требуются предварительно вычисленные множества используемых символьных переменных $used_variables(c)$ для всех ограничений в предикате пути и ограничения $cond$ ($c \in \Pi \vee c = cond$). Производится обход деревьев символьных выражений (для ограничений c) в ширину и составляется множество номеров, используемых в них переменных. Алгоритм поиска в ширину является конечным для деревьев с конечным числом вершин, что справедливо для деревьев символьных выражений. Множество ограничений в предикате пути Π также конечно. Следовательно, алгоритм вычисления используемых символьных переменных конечен.

Проход по всем ограничениям $c \in \Pi$ осуществляется за конечное число шагов, т.к. множество ограничений в предикате пути Π конечно. Из этого следует, что вложенный цикл на каждой итерации **while**, который обновляет $vars$, и цикл, конструирующий итоговый предикат Π_S, конечны. Для доказательства конечности всего алгоритма 1 остается только доказать, что итераций цикла **while** с условием завершения $change = \varnothing$ конечное число.

Каждая итерация цикла **while** заключается в переборе всех ограничений c из предиката пути Π с возможным обновлением множества $vars$. При этом, если на итерации не происходит изменение множества $vars$, выполняется условие завершения цикла.

Для $i \in \mathbb{N}$ обозначим $V_i = vars_i$ — множество переменных слайсинга $vars$ в начале i-й итерации. Тогда $V_1 = used_variables(cond)$, а $V_i \subseteq V_{i+1}$, так как

множество переменных $vars$ на каждой итерации включает в себя множество переменных предыдущей итерации: $V_{i+1} = V_i \cup U_i$, где $V_i \cap U_i = \varnothing$, и U_i — множество добавленных в $vars$ переменных.

Заметим, что на каждой итерации возможны 2 варианта работы алгоритма:

1. происходит перебор всех ограничений из предиката пути Π без изменения множества V_i, и цикл завершается;
2. множество V_i пополняется новыми элементами.

Покажем, что цикл **while**, отбирающий символьные переменные для слайсинга, имеет конечное число итераций. Докажем «от противного»: предположим, что цикл бесконечен. Следовательно, никогда не выполняется условие завершения цикла. Это возможно (в соответствии с вариантами его работы) в том и только в том случае, когда на каждой итерации i множество V_i пополняется хотя бы одним новым элементом x_i (при этом x_i может быть не единственным новым элементом на i-й итерации: $V_{i+1} = V_i \cup U_i$, где $V_i \cap U_i = \varnothing$, а $x_i \in U_i$). Более того, множества V_i строго вложены ($V_i \subset V_{i+1}$), в противном случае стало бы истинным условие завершения цикла $change = \varnothing$ (другими словами, $V_{i+1} = V_i$). Исходя из строгой вложенности множеств V_i, все элементы x_i различны. Тогда существует бесконечная последовательность $\{x\}_i$, такая что $x_i \in V_{i+1}$, состоящая из различных элементов. Обозначим множество элементов последовательности как $X = \{x | x \in \{x\}_i\}$. Поскольку каждое из множеств V_i является подмножеством множества V_P всех символьных переменных программы, то множество элементов последовательности $X \subseteq V_P$. Следовательно мощность множества $|X| \leqslant |V_P|$. Но множество всех символьных переменных V_P конечно: $|V_P| = N$, где $N \in \mathbb{N}$. А значит, $|X| \leqslant N$, и последовательность $\{x\}_i$ не может содержать бесконечное число различных элементов. Получили противоречие с изначальным определением $\{x\}_i$, из чего следует, что цикл отбора символьных переменных для слайсинга конечен, а значит, и алгоритм 1 слайсинга предиката пути конечен. ■

Теорема 2. *Пусть предикат пути* Π *строился по пути выполнения, заданному конкретными значениями символьных переменных* $v_i \leftarrow \alpha_i, i = 1..n, n \in \mathbb{N}$, *где* v_i *— символьная переменная, а* α_i *— ее значение (входной байт программы).*

Если конъюнкция ограничений предиката пути Π *и ограничения* $cond$ *(*$\Pi \wedge cond$*) выполнима, то предикат* Π_S *(полученный в результате применения алгоритма 1 слайсинга предиката пути к* Π *и* $cond$*) тоже выполним, и для любой его модели* $v_j \leftarrow \beta_j, j \in J \subseteq \{1, ..., n\}$ *подстановка* $v_j \leftarrow \beta_j, v_k \leftarrow \alpha_k, k \in K =$

$\{1, ..., n\} \setminus J$ *(α_k — входные байты программы) является моделью для предиката* $\Pi \wedge cond$.

Доказательство. Исходя из описания алгоритма, предикат Π_S состоит из конъюнкции части ограничений из предиката пути Π и ограничения $cond$: $\Pi_S = P \wedge cond, \Pi = P \wedge Q$. Предикат $\Pi \wedge cond$ выполним, а значит, и предикат $P \wedge Q \wedge cond$ выполним. Следовательно, предикаты $P \wedge cond = \Pi_S$ и Q тоже выполнимы.

Обратим внимание на последнюю итерацию цикла **while**, когда выполняется условие выхода из цикла, а именно множество переменных $vars$ остается неизменным в результате итерирования по всем ограничениям c из предиката пути Π. Для этого на каждой итерации вложенного цикла **for** не должно происходить пополнение множества $vars$. Это возможно в том и только том случае, когда либо c не использует ни одну переменную из $vars$, либо $used_variables(c) \subseteq vars$ (в противном случае множество $vars$ пополнилось бы переменными из $used_variables(c) \setminus vars$). Другими словами, $\forall c \in \Pi\ (used_variables(c) \subseteq vars) \vee (used_variables(c) \cap vars = \varnothing)$.

Из последнего цикла **for** алгоритма следует, что конъюнкты P зависят хотя бы от одной переменной из множества переменных $vars$, а конъюнкты Q напротив не зависят ни от одной переменной из $vars$. Из доказанного выше утверждения следует, что каждый конъюнкт P зависит от подмножества переменных из $vars$. Таким образом, каждый конъюнкт P содержит только переменные из $vars$ и не содержит других переменных. Значит, множества переменных в предикатах P и Q не пересекаются. Из первой строчки алгоритма следует, что множество переменных в $cond$ является подмножеством $vars$, а значит, множества переменных в $cond$ и Q так же не пересекаются. Следовательно, множество переменных предиката $P \wedge cond = \Pi_S$ не пересекается с множеством переменных предиката Q.

Пусть $v_j \leftarrow \beta_j, j \in J \subseteq \{1, ..., n\}$ — модель для предиката Π_S. Тогда $v_k \leftarrow \alpha_k, k \in K = \{1, ..., n\} \setminus J$ — модель для предиката Q, т.к. Q выполним, множества переменных в Π_S и Q не пересекаются, $\Pi = P \wedge Q$, а $v_i \leftarrow \alpha_i$ — модель для предиката пути Π из определения. Осталось найти модель для предиката $\Pi \wedge cond = P \wedge Q \wedge cond = (P \wedge cond) \wedge Q = \Pi_S \wedge Q$. Моделью является только что найденная подстановка непересекающихся множеств переменных: $v_j \leftarrow \beta_j, v_k \leftarrow \alpha_k, j \in J \subseteq \{1, ..., n\}, k \in K = \{1, ..., n\} \setminus J$. ■

Следствие 2.1. *Для получения входных байтов программы, приводящих к инвертированию целевого перехода (или проявлению ошибки), достаточно заменить часть входных байтов* v_j *на значения* β_j *из модели для предиката* Π_S*, полученного в результате применения алгоритма 1 слайсинга предиката пути.*

Теорема 3. *Асимптотическая сложность алгоритма 1 —* $O(|\Pi|{*}|V_P|^2{*}log(|V_P|))$*, где* $|\Pi|$ *— число ограничений в предикате пути, а* $|V_P|$ *— число всех символьных переменных (входных байтов) программы.*

Доказательство. Номера используемых символьных переменных хранятся в двоичных (красно-черных) деревьях поиска. Операция объединения двух деревьев имеет сложность $O(|V_P| * log(|V_P|))$. Операция определения факта пересечения (пересекаются/не пересекаются) двух отсортированных контейнеров имеет линейную сложность $O(|V_P|)$. Определение факта изменения множества $vars$ имеет константную сложность: для этого достаточно сравнить количество переменных $vars$ в начале и в конце итерации цикла. Конъюнкция двух предикатов так же имеет константную сложность, т.к. предикаты представляются абстрактными синтаксическими деревьями, а их конъюнкция равносильна созданию одной вершины.

Предварительно до начала работы алгоритма вычисляются множества используемых символьных переменных для ограничений c из предиката пути Π и ограничения $cond$. Для этого производится обход по абстрактным синтаксическим деревьям. Таким образом, вычисление используемых переменных имеет сложность $O(|\Pi| * |V|)$, где $|V|$ — максимальное число вершин в дереве. Следует отметить, что используемые переменные в ограничениях предиката пути Π могут быть вычислены единожды, а алгоритм применен многократно.

Оценим вычислительную сложность непосредственно самого алгоритма 1 слайсинга предиката пути. В худшем случае цикл **while** будет иметь $|V_P|$ итераций. Это возможно, когда на каждой итерации множество $vars$ пополняется ровно одной переменной. Таким образом, асимптотическая сложность цикла **while**: $O(|V_P|{*}|\Pi|{*}(|V_P|+|V_P|{*}log(|V_P|))) = O(|\Pi|{*}|V_P|^2{*}log(|V_P|))$. Вычислительная сложность каждой итерации внутреннего цикла **for** состоит из суммы сложности пересечения и объединения множеств. Последний цикл **for** осуществляет $|\Pi|$ пересечений множеств: $O(|\Pi| * |V_P|)$. Итого асимптотическая сложность алгоритма — $O(|\Pi| * |V_P|^2 * log(|V_P|) + |\Pi| * |V_P|) = O(|\Pi| * |V_P|^2 * log(|V_P|))$. ■

2.4 Преодоление избыточной и недостаточной помеченности

Алгоритм слайсинга предиката пути позволяет уменьшать избыточную помеченность. Рассмотрим следующий пример:

```
char buf[1024];
read(0, buf, sizeof(buf));
encrypt(buf + 16, sizeof(buf) - 16);
if (buf[0] > 'a' & buf[0] < 'f')
  if (buf[1] > 'a' & buf[1] < 'f' & buf[1] > buf[0])
    if (buf[1] - buf[0] < 3)
      printf("OK\n");
    else
      printf("FAIL\n");
```

Предположим, что начальные входные данные приводят к печати FAIL. Попробуем инвертировать переход на строке 6, чтобы было выведено OK. Во время динамической символьной интерпретации в предикат пути попадут ограничения из функции encrypt, которая осуществляет сложные криптографические преобразования. Математический решатель часто не может подобрать модель для такого рода вычислений, т. к. данная задача имеет высокую временную сложность. Однако функция encrypt не применяется к первым 16 байтам буфера buf, а условие перехода на строке 6 зависит только от переменных buf[0] и buf[1]. Полученный в результате применения алгоритма слайсинга предикат не будет содержать ограничения из функции encrypt. Таким образом, будут получены входные данные, которые успешно инвертируют переход и приведут к печати OK.

Проиллюстрировать результат работы алгоритма слайсинга предиката пути можно на реальном примере. На листинге 2.2 приводится фрагмент SMT формул для инвертирования целевого перехода после применения слайсинга. Полный пример можно найти в бенчмарке QF_BV 20210219-Sydr, который был подан на соревнование математических решателей SMT-COMP 2021 [71]. Можно заметить, что даже условный переход на большой глубине выполнения программы по сути зависит от 4 символьных переменных: file_file.ppm_3, file_file.ppm_4, file_file.ppm_6, file_file.ppm_7. Следовательно, только эти четыре байта будут подменены в изначальных входных данных для инвертирования целевого перехода.

Листинг 2.2 Пример SMT формул для инвертирования условного перехода после применения алгоритма слайсинга предиката пути

```
(declare-fun file_file.ppm_3 () (_ BitVec 8))
(declare-fun file_file.ppm_4 () (_ BitVec 8))
(declare-fun file_file.ppm_6 () (_ BitVec 8))
(declare-fun file_file.ppm_7 () (_ BitVec 8))
(define-fun ref!3 () (_ BitVec 8) file_file.ppm_3) ; Byte reference
(define-fun ref!4 () (_ BitVec 8) file_file.ppm_4) ; Byte reference
(define-fun ref!6 () (_ BitVec 8) file_file.ppm_6) ; Byte reference
(define-fun ref!7 () (_ BitVec 8) file_file.ppm_7) ; Byte reference
(define-fun ref!4450 () (_ BitVec 32) ((_ zero_extend 24) ref!3)) ; Extended part - MOVZX operation - 0x7f67900390b5: movzx esi, byte ptr [rax]
(define-fun ref!4453 () (_ BitVec 64) ((_ zero_extend 32) ref!4450)) ; MOV operation - 0x7f6790039107: mov eax, esi
(define-fun ref!4455 () (_ BitVec 32) (bvsub ref!4450 (_ bv35 32))) ; CMP operation - 0x402448: cmp eax, 0x23
(define-fun ref!4461 () (_ BitVec 1) (ite (= ref!4455 (_ bv0 32)) (_ bv1 1) (_ bv0 1))) ; Zero flag - 0x402448: cmp eax, 0x23

...

(define-fun ref!268038 () (_ BitVec 64) (bvsub ref!268034 (_ bv256 64))) ; CMP operation - 0x7f6790110f63: cmp rdx, 0x100
(define-fun ref!268040 () (_ BitVec 1) ((_ extract 63 63) (bvxor (bvxor ref!268034 (bvxor (_ bv256 64) ref!268038)) (bvand (bvxor ref!268034 ref!268038) (bvxor ref!268034 (_ bv256 64)))))) ; Carry flag - 0x7f6790110f63: cmp rdx, 0x100
(define-fun ref!268055 () (_ BitVec 8) (bvsub ((_ extract 7 0) ref!4583) (_ bv128 8))) ; CMP operation - 0x7f6790110f79: cmp dl, 0x80
(define-fun ref!268057 () (_ BitVec 1) ((_ extract 7 7) (bvxor (bvxor ((_ extract 7 0) ref!4583) (bvxor (_ bv128 8) ref!268055)) (bvand (bvxor ((_ extract 7 0) ref!4583) ref!268055) (bvxor ((_ extract 7 0) ref!4583) (_ bv128 8)))))) ; Carry flag - 0x7f6790110f79: cmp dl, 0x80

...

(assert (= ref!267667 (_ bv0 1)))
(assert (not (= ref!267682 (_ bv0 1))))
(assert (or (or (= (bvadd ref!267768 (_ bv24 64)) (_ bv6340264 64)) (= (bvadd ref!267768 (_ bv24 64)) (_ bv6340272 64))) (= (bvadd ref!267768 (_ bv24 64)) (_ bv6340280 64))))
(assert (not (= ref!267886 (_ bv0 1))))
(assert (= ref!267903 (_ bv1 1)))
(assert (= ref!267912 (_ bv1 1)))
(assert (= ref!268011 (_ bv0 1)))
(assert (not (= ref!268040 (_ bv0 1))))
(assert (not (= ref!268057 (_ bv1 1))))
```

Недостаточная помеченность может возникать, когда условные переходы зависят от значения, читаемого по символьному адресу. Однако условные переходы в реальных программах часто оперируют независимыми множествами символьных переменных. Символьный интерпретатор не сможет инвертировать переход, зависящий от символьного указателя. Однако большое количество переходов не будут зависеть транзитивно по переменным от таких переходов. Слайсинг позволит устранять недостаточно помеченные части предиката пути, которые не влияют на инвертирование целевого перехода (или решение предиката безопасности). Более того, слайсинг позволяет устранять формулы, где частично отсутствуют трансляции инструкций в символьные выражения или же в трансляциях присутствуют ошибки. Особенно слайсинг полезен для инвертирования переходов на большой глубине, где при обычном подходе (разд. 2.1) ошибка от недостаточной помеченности только накапливается с глубиной. Алгоритм слайсинга инвариантен относительно глубины и учитывает только зависимости символьных переменных по данным.

2.5 Экспериментальная оценка

В данном разделе приводится экспериментальная оценка эффективности алгоритма слайсинга предиката пути [25]. Измерения производились для 64-битных Linux программ [72] на сервере с двумя 64-ядерными процессорами AMD EPYC 7702 и 256 Гб оперативной памяти.

Во время эксперимента производится динамическая символьная интерпретация каждой программы на некоторых фиксированных характерных для нее входных данных. Сначала для каждой программы осуществляется построение предиката пути с ограничением по времени в 20 минут. В результате получается последовательность ограничений для встреченных символьных условных переходов. Затем происходит последовательное инвертирование условных переходов от встреченных первыми к самым глубоким. Суммарное время анализа программы ограничено 2 часами.

Сгенерированные входные данные для инвертирования условных переходов проверяются на корректность. Для этого был разработан клиент для инструмента динамической инструментации DynamoRIO [73], который во вре-

Таблица 4 — Результаты инвертирования переходов без применения слайсинга предиката пути

Приложение	Корректные	SAT	Запросы	Ветвления	Время
bzip2recover	2101	2101	5131	5131	52m42s
cjpeg	50	50	198	8010	120m
faad	386	389	585	470588	120m
foo2lava	27	31	6252	910725	120m
hdp	116	464	2427	67475	120m
jasper	1	1987	5639	837669	120m
libxml2	130	1043	13520	53700	120m
minigzip	425	3961	4183	8977	120m
muraster	3234	3235	4739	7102	120m
pk2bm	181	183	3672	3673	21m48s
pnmhistmap_pgm	3158	3159	4681	967187	120m
pnmhistmap_ppm	106	107	8247	8121	40m15s
readelf	135	218	2046	64196	120m
yices-smt2	13	521	2135	19543	120m
yodl	26	313	5201	4831	43m24s

мя работы программы на сгенерированных входных данных записывает трассу выполненных ребер условных переходов. Полученная трасса ребер сравнивается с трассой символьных переходов, которая сохраняется во время построения предиката пути. Входные данные считаются **корректными**, когда трасса содержит ту же последовательность ребер символьных переходов за исключением целевого в инвертированном направлении. Таким образом, корректность проверяется с учетом пути выполнения.

Эксперимент производится для двух методов инвертирования переходов: с полным предикатом пути как в разделе 2.1 (табл. 4) и с применением алгоритма слайсинга предиката пути (табл. 5). Для каждого **приложения** измеряются следующие показатели:

- **Корректные** — число корректно сгенерированных входных данных, приводящих к инвертированию целевого перехода с учетом пути.
- **SAT** — суммарное число сгенерированных входных данных.

Таблица 5 — Результаты инвертирования переходов с применением слайсинга предиката пути

Приложение	**Корректные**	**SAT**	**Запросы**	**Ветвления**	**Время**
bzip2recover	2101	2101	5131	5131	51m3s
cjpeg	50	50	197	8010	120m
faad	**426**	430	**652**	458145	120m
foo2lava	27	31	6127	910725	120m
hdp	**809**	1037	**3828**	67476	120m
jasper	**6766**	6798	**18207**	837669	120m
libxml2	**545**	1069	**17532**	53699	120m
minigzip	**3896**	7569	**8977**	8977	**29m42s**
muraster	3227	3228	4726	7102	120m
pk2bm	182	183	3673	3673	21m39s
pnmhistmap_pgm	**17088**	17089	**25446**	967187	120m
pnmhistmap_ppm	106	107	8247	8121	**28m52s**
readelf	**639**	739	**6141**	64196	120m
yices-smt2	**2114**	2699	**9647**	19543	120m
yodl	**180**	313	5201	4831	**34m59s**

– **Запросы** — число запросов (на инвертирование перехода) к математическому решателю.
– **Ветвления** — число обнаруженных условных переходов во время динамической символьной интерпретации. Следует отметить, что суммарное число символьных переходов может быть меньше числа запросов потому, что каждая таблица переходов (`switch`) рассматривается как один переход, но может производить несколько запросов на каждый `case`.
– **Время** — суммарное время анализа приложения. Если указано время 120m, то анализ программы был завершен по тайм-ауту в два часа.

Из таблиц 4 и 5 видно, что для большей части приложений (выделены жирным в таблице 5: *faad*, *hdp*, *jasper*, *libxml2*, *minigzip*, *pnmhistmap* с входным файлом .pgm, *readelf*, *yices-smt2*, *yodl*) значительно выросло число корректно инвертированных переходов (с учетом пути). Более того, для этих же приложений за исключением *yodl* возросло число обработанных SMT-решателем запросов, а для программ *minigzip*, *pnmhistmap* (с файлом .ppm) и *yodl* заметно сократи-

Таблица 6 — Сравнение точности и скорости инвертирования переходов с применением и без применения слайсинга предиката пути

Приложение	**Слайсинг выключен**			**Слайсинг включен**		
	Точность	**SAT**	**Запросы**	**Точность**	**SAT**	**Запросы**
bzip2recover	100.0 %	2101	5131	100.0 %	2101	5131
cjpeg	100.0 %	50	198	100.0 %	50	197
faad	99.23 %	389	585	99.07 %	430	652
foo2lava	87.1 %	31	6252	87.1 %	31	6127
hdp	25.0 %	464	2427	**78.01 %**	1037	3828
jasper	0.05 %	1987	5639	**99.53 %**	6798	18207
libxml2	12.46 %	1043	13520	**50.98 %**	1069	17532
minigzip	10.73 %	3961	4183	**51.47 %**	7569	8977
muraster	99.97 %	3235	4739	99.97 %	3228	4726
pk2bm	98.91 %	183	3672	99.45 %	183	3673
pnmhistmap_pgm	99.97 %	3159	4681	**99.99 %**	17089	25446
pnmhistmap_ppm	99.07 %	107	8247	99.07 %	107	8247
readelf	61.93 %	218	2046	**86.47 %**	739	6141
yices-smt2	2.5 %	521	2135	**78.33 %**	2699	9647
yodl	8.31 %	313	5201	**57.51 %**	313	5201

лось суммарное время анализа. Число корректных переходов для приложений *bzip2recover*, *cjpeg*, *foo2lava* и *pnmhistmap* (с файлом .ppm) осталось неизменно, а для *muraster* и *pk2bm* оно колеблется в пределах погрешности измерений между двумя запусками. В результате применения алгоритма слайсинга предиката пути повышается скорость инвертирования переходов во время динамической символьной интерпретации: генерируется больше корректных входных данных для инвертирования переходов, математический решатель успевает обработать большее число запросов за меньшее время. Таким образом, было экспериментально показано, что алгоритм слайсинга позволяет бороться с избыточной помеченностью путем устранения избыточных ограничений, вследствие чего SMT-решатель быстрее обрабатывает запросы.

В таблице 6 приводится сравнение точности сгенерированных входных данных для инвертирования переходов с применением и без применения алгоритма слайсинга предиката пути. **Точность** — это процентное отношение числа

корректных входных данных для инвертирования переходов (с учетом пути) к общему числу сгенерированных входных данных (**SAT**). Точность возросла для большинства приложений, которые выделены жирным в таблице 6: *hdp*, *jasper*, *libxml2*, *minigzip*, *pnmhistmap* с файлом .pgm, *readelf*, *yices-smt2* и *yodl*. При этом для ряда программ точность возросла на порядок. В частности, точность входных данных для *jasper* возросла с 0.05 до 99.53 %. Следует отметить, что для *faad* точность незначительно упала, однако абсолютное число корректных входных данных увеличилось (табл. 5). Таким образом, применение алгоритма слайсинга позволяет повысить точность генерируемых входных данных, в некоторых случаях даже на порядки. Это особенно критично для генерации точных входных данных для воспроизведения обнаруженных ошибок. Экспериментально показано, что представленный алгоритм преодолевает недостаточную помеченность так, что из запросов устраняются нерелевантные символьные переменные с недостаточными ограничениями, и точность генерируемых данных возрастает.

В итоге алгоритм слайсинга предиката пути нигде не показал худший результат по сравнению с решением полного предиката пути как в разделе 2.1 (за исключением незначительного ухудшения из-за погрешности измерений на *muraster*). Следовательно, для повышения точности и скорости решения всегда необходимо применять алгоритм слайсинга к целевому предикату перед отправкой запроса математическому решателю. В дальнейшем в данной работе алгоритм слайсинга всегда будет применяться перед запросом к SMT-решателю (в т. ч. перед решением предикатов безопасности).

Глава 3. Метод моделирования семантики функций

Моделирование семантики функций (разд. 1.3.6) позволяет пропускать интерпретацию некоторых функций и тем самым ускорять динамическую символьную интерпретацию. Более того, моделирование функции одним символьным выражением дает возможность объединить несколько символьных состояний и исследовать больше путей выполнения программы. Так моделирование семантики функций уменьшает недостаточную и избыточную помеченность. Функции, потенциально содержащие табличные преобразования (такие как `tolower`), могут быть обработаны на уровне семантики функций без анализа символьных адресов (разд. 1.3.4). Такие функции больше не будут привносить недостаточную помеченность. В свою очередь, пропуск функций, добавляющих избыточные ограничения в предикат пути, которые не влияют на решение целевой задачи (инвертирование перехода или проверку предиката безопасности), позволяет избегать избыточной помеченности.

В данной главе предлагается метод моделирования семантики функций [29], который повышает производительность динамической символьной интерпретации, а также преодолевает избыточную и недостаточную помеченность. Идея метода заключается в следующем. Все функции реально выполняются для обновления конкретного состояния. Однако символьная интерпретация части функций пропускается. Такие функции моделируются необходимыми символьными выражениями, которые сохраняются в символьное состояние и предикат пути. При этом символьная интерпретация начинается с момента первого чтения символьных входных данных. Также необходимо поддерживать согласованность символьного и конкретного состояний. Таким образом, вычисляемые значения символьных выражений должны соответствовать значениям из конкретного состояния.

Символьные модели функций можно разделить на несколько групп. Некоторые из них перемещают уже существующие в памяти символьные выражения и добавляют необходимые ограничения в предикат пути, в то время как другие просто пропускают символьную интерпретацию тела функции. Более сложные модели обрабатывают строковые сравнения и преобразования строк в числа, что требует конструирования новых символьных выражений, описывающий возвращаемые значения.

Следует отметить, что предлагаемый метод конкретизует длины строк значениями из реального выполнения, т. к. он разрабатывался для применения в рамках динамической символьной интерпретации. Таким образом, метод не позволяет генерировать строки большего размера, чем они были во входных данных. Однако в некоторых случаях метод, напротив, может уменьшить размеры изначальных строк.

Моделирование семантики функций уточняет предикаты безопасности и расширяет, анализируемые ими символьные состояния. Так некоторые предикаты безопасности можно проверять на уровне функций. Моделирование функции `strtol`, вызываемой в `scanf`, позволяет конструировать предикаты безопасности над числами, представленными строками, а также оценивать точность реализованных предикатов безопасности на наборах тестов Juliet [26], которые используют функцию `scanf` для чтения чисел.

3.1 Моделирование функций стандартной библиотеки

В данном разделе приводятся символьные формулы, моделирующие семантику стандартных библиотечных функций. Набор функций для моделирования был получен путем анализа результатов динамической символьной интерпретации различных Linux программ [72]. В результате были получены наборы символьных условных переходов и соответствующие имена функций, которые их содержат. Таким образом, были созданы символьные модели, которые почти полностью покрывают функции, обрабатывающие символьные данные. В итоге было промоделировано более 30 функций стандартной библиотеки.

Следует отметить, что в формулах используется модульная арифметика битовых векторов SMT [42]. Таким образом, значения могут переполняться, как на реальном процессоре. Более того, для простоты знаковые и беззнаковые расширения в формулах опущены.

В частности, функция `tolower` (аналогично `toupper`) моделируется следующим if-then-else (*ite*) выражением для входного символа *ch*:

$$ite(ch - \texttt{'A'} < 26, ch + 32, ch) \quad (1)$$

Представленная выше формула разрешает строчные и заглавные символы в символьных входных данных. Таким образом, обходится избыточная помеченность, ограничивающая регистр символа тем, что был в реальном выполнении.

3.1.1 Пропуск символьной интерпретации функции

Пропуск интерпретации определенных функций может значительно ускорить динамическую символьную интерпретацию и уменьшить избыточную помеченность [27]. В данном разделе описываются функции с тривиальными побочными эффектами. Они моделируются так, чтобы повысить производительность интерпретации и упростить порождаемые формулы.

Моделирование динамической памяти

Функции динамического выделения памяти (`malloc`, `calloc`, `realloc`, `free`) часто обрабатывают символьные данные. Аргументы таких функций могут оказаться помеченными, что приводит к добавлению избыточных ограничений в предикат пути из-за символьной интерпретации тел функций. Однако путь выполнения внутри аллокаторов не оказывает влияние на инвертирование переходов или проверку предикатов безопасности.

Моделирование функций работы с динамической памятью осуществляется посредством их конкретного выполнения и поддержки согласованности символьного состояния. При этом аргументы функции конкретизуются, а символьная интерпретация приостанавливается до возврата из функции. Дополнительно для функции `realloc` происходит копирование символьных выражений для памяти в символьном состоянии по новому адресу. Таким образом, число избыточно помеченных условных переходов и ограничений в предикате пути значительно уменьшается. На практике пропуск символьной интерпретации кода внутри этих функций влечет за собой значительный рост производительности.

Помимо прочего, во время моделирования функций аллокаторов поддерживается теневая память, которая содержит адреса и размеры буферов, выделенных

в динамической памяти. Теневая память используется для определения границ массивов при конструировании предикатов безопасности. Более того, моделирование динамической памяти позволяет обнаруживать ошибки некорректной работы с кучей: двойное освобождение, освобождение или реаллокация по неверному адресу и другие.

Копирование данных

Символьная модель операций копирования (таких как `strcpy`, `strncpy`, `memcpy`, `memmove`) просто копирует формулы в символьном состоянии без их модификации. Дополнительно для функций копирования строк `str(n)cpy` в предикат пути добавляются ветвления на равенство каждого байта нулю. Таким образом, для каждого байта строки добавляется отдельное ограничение, и при инвертировании дальнейших переходов будет учитываться, что строка не содержит нуль-терминатора. Более того, символьный интерпретатор будет воспринимать наложенные ограничения как обычные условные ветвления и пытаться подставлять нуль-терминатор на различные позиции строки.

Для моделирования функции `memset` достаточно присвоить соответствующее символьное выражение символа по диапазону адресов в символьном состоянии.

Пропуск печати

Функции, которые пишут лог программы в стандартный поток вывода, не представляют интереса для процесса динамической символьной интерпретации и исследования новых путей выполнения программы [20]. Следовательно, их можно пропускать ради повышения производительности и борьбы с избыточной помеченностью. Таким образом, из предиката пути устраняются избыточные ограничения над печатаемыми данными. Так к функциям не представляющих интереса относятся `printf`, `std::cout`, `std::cerr`, `std::clog`, `putchar`, `putc`, `puts`, `perror` и другие. Более того, аналогичным образом можно пропус-

кать функции работы с файлами, если они пишут в стандартные потоки вывода (`stdout`) или ошибок (`stderr`): `fprintf`, `vfprintf`, `fputs`, `fflush` и т. д.

3.1.2 Моделирование строкового сравнения и поиска

Приведенные в данном разделе функции моделируются символьной формулой возвращаемого значения. Так сразу несколько состояний функции моделируются одной формулой, что позволяет исследовать больше путей выполнения программы. В частности, формула для функции `memchr` позволяет одновременно учесть как наличие искомого символа на некоторой позиции строки, так и его отсутствие. При этом в предикат пути не будет добавлено ни одного ограничения. В результате, устраняются избыточные ограничения (помеченность) на значения, которые может принимать строка. Возвращаемое значение становится символьным и может далее участвовать в символьных вычисления и условных переходах. Например, условный переход `if (memchr(ptr, ch, count) == ptr + 5)` породит ветвление, в котором символ `ch` либо будет обнаружен на 5 позиции, либо нет. Символьный интерпретатор сможет исследовать оба направления. При классическом подходе без моделирования семантики функций результат выполнения функции `memchr` ограничен реальным путем выполнения, и символ `ch` не может быть подставлен на произвольные позиции строки. Таким образом, моделирование семантики функции символьным выражением для возвращаемого значения позволяет избегать добавления избыточных ограничений, а также рассматривать возвращаемое значение как символьное в дальнейших условных переходах и вычислениях. Ослабляя ограничения в предикате пути и обобщая символьные состояния, можно открывать больше новых путей выполнения и повышать вероятность срабатывания предикатов безопасности.

В данной работе используется фреймворк динамической символьной интерпретации Triton [23] для конструирования символьных выражений возвращаемых значений таких функций, как `memchr`, `strlen`, `strcmp`, `memcmp`, `strstr` и др. В этой группе функций можно выделить два основных типа семантик:

1. поиск символа в строке или области памяти (например, `memchr`);

2. лексикографическое сравнение двух строк или буферов памяти (в частности, memcmp).

Функция strstr в таком случае может быть рассмотрена как strchr с расширением искомого символа *ch* до подстроки *substr*.

Поиск символа или подстроки в строке

Формула для таких функций, как memchr, должна выражать указатель на найденный символ в строке или нулевой указатель, когда символ в строке отсутствует. Как строка, так и искомый символ могут быть символьными. Разработанная формула вычисляет позицию символа как сумму if-then-else (*ite*) выражений, которые могут принимать значения 1 или 0. Ниже приводится формула для возвращаемого значения функции memchr(ptr, ch, count):

$$ite\left(\bigwedge_{k=0}^{count-1} ptr[k] \neq ch, 0, ptr + \sum_{i=0}^{count-1} ite\left(\bigwedge_{k=0}^{i} ptr[k] \neq ch, 1, 0\right)\right) \qquad (2)$$

Внешнее *ite*-выражение проверяет, присутствует ли искомый символ *ch* в строке *ptr* длины *count*. Если символ *ch* отсутствует в строке, это *ite*-выражение возвращает нулевой указатель. В альтернативном случае стоит формула, каждый слагаемый которой инкрементирует указатель *ptr* до тех пор, пока не будет встречен искомый символ *ch*. Следует отметить, что семантика функции strlen(str) похожа на семантику memchr(str, 0, length + 1).

Моделирование функций, принимающих нуль-терминированные строки (таких как strchr и strrchr), добавляет дополнительные ограничения в предикат пути, чтобы учесть нуль-терминатор.

В формуле для функции strstr поиск символа $ptr[k] \neq ch$ заменяется на соответствующее сравнение с подстрокой: $\bigvee str[k+n] \neq substr[n]$. Идея вычисления индекса найденной подстроки аналогична memchr и заключается в суммировании *ite*-выражений, принимающих значения 0 и 1. Такое *ite*-выражение принимает значение 1, когда искомая подстрока *substr* не совпала с префиксами строк $str + k$, где $k \in [0, i]$, а $i \geqslant 0$ — номер слагаемого. Ниже представлена формула для функции strstr(str, substr), где $len(str) \geqslant$

$len(substr)$:

$$count = len(str) - len(substr) \tag{3}$$

$$\begin{gathered} ite\left(\bigwedge_{k=0}^{count} \bigvee_{n=0}^{len(substr)-1} str[k+n] \neq substr[n], 0, \right. \\ \left. str + \sum_{i=0}^{count} ite\left(\bigwedge_{k=0}^{i} \bigvee_{n=0}^{len(substr)-1} str[k+n] \neq substr[n], 1, 0 \right) \right) \end{gathered} \tag{4}$$

Внешнее *ite*-выражение проверяет, присутствует ли подстрока *substr* в строке *str*. Альтернативная ветка *ite* содержит формулу, инкрементирующую *str*, пока подстрока не была найдена на предыдущих индексах $str + i$. В целях упрощения полагается, что строка и подстрока имеют фиксированные длины. Для этого в предикат пути добавляются дополнительные ограничения на неравенство нулю каждого байта строки и подстроки.

Лексикографическое сравнение строк

Задача лексикографического сравнения строк (`memcmp`, `strcmp`, `strncmp`) может быть сведена к поиску первых различающихся символов и вычислению их разности. Ниже приводится формула для функции `memcmp(lhs, rhs, count)`:

$$lhs[0] - rhs[0] + \sum_{i=1}^{count-1} (lhs[i] - rhs[i]) * ite\left(\bigwedge_{k=0}^{i-1} lhs[k] = rhs[k], 1, 0 \right) \tag{5}$$

Формула составлена так, что только одно слагаемое имеет значение, в то время как оставшиеся равны нулю. Для обнаружения первой пары различающихся символов составляется конъюнкция попарного сравнения на равенство всех предыдущих пар символов. Если хотя бы в одной предыдущий паре символы неравны, то множитель $ite\left(\bigwedge lhs[k] = rhs[k], 1, 0\right)$ будет равняться нулю. А для равных символов множитель $lhs[i] - rhs[i]$ равняется нулю. Таким образом, единственное ненулевое слагаемое сделает формулу равной разнице первых различающихся символов. Для моделирования функций сравнения нуль-терминированных строк `str(n)cmp` дополнительно проверяется, что ранее не было обнаружено нулевых байтов.

Описанная выше семантика функции определяет знак возвращаемого значения. Однако она не гарантирует согласованность символьного и конкретного состояний. Так в 32-битных Linux приложениях из функции возвращаются значения $\{1, 0, -1\}$. В то же время в приложениях x86_64 возвращается разница символов. Поэтому формула возвращаемого значения (5) (обозначим как F) оборачивается дополнительным *ite*, чтобы значения из конкретного и символьного состояний совпадали. Например, если в реальном выполнении функция вернула значение меньшее нуля, то составляется следующая формула: $ite(F < 0, ret_val, F)$, где *ret_val* — возвращаемое значение из реального выполнения программы.

Разработанная модель лексикографического сравнения строк позволяет исследовать пути выполнения программы, содержащие сравнения со строковыми константами. В частности, подбирать значения, которые удовлетворяют следующему условному переходу: `if (!memcmp(buf, "magic", 5))`.

3.1.3 Моделирование преобразования строки в число

В данном разделе рассматривается семантика функций `strtol`, `strtoul`, `strtoll`, `strtoull` и т. д. Эти функции также вызываются внутри других функций, таких как `atoi` и `scanf("%d", &x)`. Чтение входных чисел в наборе тестов Juliet [26] осуществляется с использованием `scanf`. Поэтому необходимо моделировать семантику преобразования строки в число, чтобы возможно было измерить точность работы предикатов безопасности на тестах Juliet. Более того, похожая семантика применяется во время чтения целых чисел через `std::cin`.

Строка, которую обрабатывает `strtol`, может иметь различную конфигурацию в зависимости от опциональных компонентов. Таким образом, формат строки может быть промоделирован путем разбиения его на несколько частей:
`[whitespaces]`*`[sign][prefix]`*
`[valid symbols]``[invalid symbols]nullbyte`, где

– `[whitespaces]` — пробельные символы;
– *`[sign]`* — символ знака (+ или –);
– *`[prefix]`* — префикс, задающий основание системы счисления (`0`, `0x` или `0X`);

– *[valid symbols]* — допустимые символы цифр, задающие число;
– [invalid symbols] — недопустимые символы числа;
– nullbyte — нуль-терминатор.

Парсинг входной строки str функции strtol(str, end, base) осуществляется до первого недопустимого символа. Множество допустимых символов цифр задается аргументом base и может состоять из цифр и латинских символов. Когда base равняется нулю, по умолчанию выбирается десятичная система счисления или же она может быть определена из восьмеричного или шестнадцатеричного префикса (0 и 0x/0X соответственно). Строка подвергается предварительному анализу для извлечения частей формата строки, отвечающих за преобразование в число (выделено *курсивом*). При этом изначальная структура формата сохраняется при генерации новых входных данных.

$$\pm\,(c_n c_{n-1} ... c_1 c_0)_b \longrightarrow x \tag{6}$$

$$\begin{aligned} a_k = ite(c_k \geqslant \texttt{'0'} \wedge\ c_k \leqslant \texttt{'9'} \wedge\ c_k < \texttt{'0'} + b,& \\ c_k - \texttt{'0'},& \\ ite(c_k \geqslant \texttt{'a'} \wedge\ c_k < \texttt{'a'} + b - 10,& \\ c_k - \texttt{'a'} + 10, c_k - \texttt{'A'} + 10))& \end{aligned} \tag{7}$$

$$|x| = \sum_{k=0}^{n} a_k b^k,\ x = ite(sign = \texttt{'-'}, -|x|, |x|) \tag{8}$$

$$\begin{aligned} &(c_k \geqslant \texttt{'0'} \wedge\ c_k \leqslant \texttt{'9'} \wedge\ c_k < \texttt{'0'} + b) \vee \\ &\qquad (c_k \geqslant \texttt{'a'} \wedge c_k < \texttt{'a'} + b - 10) \vee \\ &\qquad (c_k \geqslant \texttt{'A'} \wedge c_k < \texttt{'A'} + b - 10) \end{aligned} \tag{9}$$

Чтобы реализовать преобразование (6) строки в число необходимо добавить ограничение (9) для каждого символа, чтобы он лежал в пределах допустимых значений в соответствии с основанием системы счисления b, которое определяется из аргумента base или предварительного анализа префикса строки str. Формула (7) отражает вычисление цифры из символа, который может быть как цифрой, так и латинской буквой (строчной или заглавной). Вычисление всего числа представлено классическим алгоритмом перевода между системами счисления в формуле (8).

Предлагаемая модель рассматривает аргумент base как фиксированный потому, что символьный аргумент base порождает множество сценариев с ис-

ключительными случаями и значительно усложняет символьные выражения. Аргумент `base` не рассматривается как символьный ради упрощения модели. Таким образом, на задающий основание системы счисления префикс накладываются дополнительные ограничения в предикате пути.

Если аргумент `base` равняется нулю, ведущие нули могут привести к неверному рассмотрению десятичного числа как восьмеричного. Например, для строки 4567 ($base = 0$) будет определено десятичное основание системы счисления $b = 10$, а строка 0123 не должна быть сгенерирована для числа 123_{10}, т. к. она будет преобразована в число 123_{8}. Описанный исключительный случай учитывается путем добавления дополнительных ограничений, которые заменяют ведущие нули на пробелы. Более того, добавляются дополнительные ограничения, чтобы убедиться, что пробелы не будут присутствовать между цифрами. Следует отметить, что такой проблемы не возникает, когда значение аргумента `base` отлично от нуля.

Полученная формула (8) может переполниться, т. е. вычисленное значение не поместится в тип возвращаемого значения (`long` для `strtol`). Поэтому вычисление формулы (8) осуществляется битовом векторе с размером в два раза большим, чем тип возвращаемого значения. А в предикат пути добавляются ограничения, что вычисленное значение должно лежать в пределах возвращаемого типа, например, $LONG_MIN \leqslant x \leqslant LONG_MAX$.

3.2 Экспериментальная оценка

В данном разделе приводится экспериментальная оценка разработанного метода моделирования семантики функций [29]. Измерения производительности и эффективности осуществлялись для 64-битных Linux приложений [72] аналогично разделу 2.5. Использовался сервер с двумя 64-ядерными процессорами AMD EPYC 7702 и 256 Гб оперативной памяти. Для каждой программы производилась динамическая символьная интерпретация одной конкретной трассы выполнения. Результаты сравнивались для символьной интерпретации с моделированием семантики функций и без.

В начале измерялись время построения предиката пути и число обнаруженных символьных переходов (табл. 7). Как можно заметить, моделирование

Таблица 7 — Сравнение построения предиката пути с моделированием и без моделирования семантики функций

Приложение	**Без моделирования**		**С моделированием**	
	Ветвления	**Время**	**Ветвления**	**Время**
bzip2recover	5131	6s	5131	6s
cjpeg	8008	19s	**6992**	**18s**
faad	470585	21m	**466697**	**15m52s**
foo2lava	910737	21m9s	**905592**	**18m20s**
hdp	66070	43s	**29265**	**20s**
jasper	837643	14m47s	**771806**	**10m37s**
libxml2	53400	40s	**8873**	**12s**
minigzip	8977	1m4s	8977	1m3s
muraster	7102	5s	**4453**	**4s**
pk2bm	3665	2s	**658**	**1s**
pnmhistmap_pgm	967187	9m21s	**967155**	**9m2s**
pnmhistmap_ppm	7864	12s	**7822**	**11s**
readelf	62713	41s	**13649**	**10s**
yices-smt2	19352	17s	**10340**	**11s**
yodl	8329	9s	**5340**	**5s**

семантики функций повлекло за собой уменьшение обеих измеренных величин (выделены **жирным**) для всех приложений, кроме *bzip2recover* и *minigzip*, где значения не изменились. Таким образом, за счет уменьшения времени построения предиката пути больше времени остается на исследование новых путей выполнения программы путем инвертирования условных переходов. Чем меньше символьных переходов, тем меньше формулы будут содержать избыточной помеченности. Более того, пропускается инвертирование внутренних переходов в поддерживаемых библиотечных функциях, которые теперь моделируются единственной (или несколькими) формулой. Следовательно, можно исследовать большее число логических состояний программы в результате символьной интерпретации одной трассы выполнения. Например, возможно приравнять строки `memcmp` в рамках одного запуска динамической символьной интерпретации.

Затем производилось инвертирование обнаруженных символьных условных переходов. Суммарное время построения предиката пути и инвертирования

Таблица 8 — Сравнение результатов инвертирования переходов с моделированием и без моделирования семантики функций

Приложение	Без моделирования				С моделированием			
	Точность	**SAT**	**Запросы**	**Время**	**Точность**	**SAT**	**Запросы**	**Время**
bzip2recover	100 %	2101	5131	47m35s	100 %	2101	5131	45m38s
cjpeg	100 %	50	2656	120m	100 %	50	**3750**	120m
faad	97.11 %	1974	3072	120m	98.91 %	1560	2414	120m
foo2lava	87.1 %	31	5998	120m	**99.02 %**	**205**	**6668**	120m
hdp	76.69 %	1171	4122	120m	72.22 %	**5893**	**12172**	120m
jasper	99.62 %	8457	22538	120m	96.61 %	9528	**24472**	120m
libxml2	51.27 %	1063	18485	120m	**82.44 %**	**1247**	8970	**5m53s**
minigzip	51.47 %	7569	8977	16m16s	51.47 %	7569	8977	16m16s
muraster	99.94 %	3304	6041	120m	100 %	360	470	120m
pk2bm	99.45 %	183	3664	15m55s	**100 %**	**189**	657	**4m55s**
pnmhistmap_pgm	99.99 %	19351	28932	120m	**100 %**	**19964**	**29369**	120m
pnmhistmap_ppm	99.07 %	107	7990	27m26s	**99.12 %**	**114**	7948	**25m31s**
readelf	87.38 %	1022	9541	120m	85.82 %	**2363**	6541	120m
yices-smt2	73.79 %	4258	16222	120m	70.27 %	**5534**	11753	**11m5s**
yodl	36.25 %	1153	9403	51m3s	**98.26 %**	1150	6414	**1m50s**

переходов ограничивалось 2 часами. Каждый запрос к математическому решателю был ограничен 10 секундами. Инвертирование переходов происходило в прямом порядке (от первого к последнему в предикате пути). Для каждых сгенерированных входных данных (**SAT**) проверялась корректность (разд. 2.5). **Точность** — это процент сгенерированных входных данных, которые успешно открывают ожидаемый путь, т. е. проводят программу по тому же пути выполнения за исключением последнего перехода в инвертированном направлении. В таблице 8 для большинства приложений были показаны лучшие результаты (выделены **жирным**) с использованием динамической символьной интерпретации с моделированием семантики функций. В результате, либо уменьшилось суммарное время анализа, либо увеличилось количество корректно инвертированных переходов за те же 2 часа. Число открытых путей для *faad* и *muraster* уменьшилось потому, что эти программы содержат много строковых сравнений. Каждое строковое сравнение моделируется одной формулой. Таким образом, пропускаются условные переходы внутри соответствующих функций. Эти переходы по сути неинтересны для анализа, т. к. сравнивают единичные байты, а их инвертирование едва ли приведет к открытию новых путей в анализируемой программе. Напротив, разработанный метод осуществляет сравнение строк целиком.

В ручную было проверено, что сложные условные переходы, зависящие от строковых сравнений, действительно инвертируются, например, `if (!strcmp(s, "abc"))`. Однако некоторые запросы к решателю усложняются, т. к. в результате слайсинга предиката пути (разд. 2.2) отбирается большее число ограничений, которые зависят от хотя бы одного байта целой строки. Это негативно сказывается на производительности.

В результате, метод моделирования семантики функций ускоряет динамическую символьную интерпретацию и открытие новых путей выполнения за счет устранения избыточной помеченности.

Глава 4. Метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности

Динамическая символьная интерпретация является эффективным методом не только открытия новых путей выполнения программы [17—20; 22; 25], но и обнаружения дефектов. Разработанный метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности [28—30] позволяет детектировать программные дефекты на изначально корректном пути выполнения. Для этого производится символьная интерпретация программы, запущенной на входных данных, которые не влекут за собой аварийное завершение и могут не приводить к проявлению ошибок (работы с памятью или неопределенного поведения) во время выполнения. Во время динамической символьной интерпретации конструируется предикат пути, как описано в разделе 2.1. Если интерпретируемая инструкция или функция обрабатывает символьные данные (зависящие от пользовательских входных данных), осуществляется построение предикатов безопасности, которые проверяют возможность неопределенного поведения или нарушения доступа к памяти.

Более того, разработанный метод генерирует входные данные для воспроизведения обнаруженной ошибки, что позволяет автоматически отличать истинные срабатывания путем запуска программы, скомпилированной с санитайзерами, на сгенерированных входных данных. Это является значительным преимуществом метода по сравнению со статическим анализом.

Таким образом, в результате динамической символьной интерпретации пути программы, на котором ошибка не проявляется, будут получены новые входные данные, запуск программы на которых приведет к ошибке. Активация найденных дефектов зависит от пользовательских данных, что может привести к проявлению серьезных уязвимостей в программе.

4.1 Метод построения предикатов безопасности

Предикат безопасности для некоторого класса ошибок является булевым предикатом, который истинен, когда выполнение инструкции (или функции)

Листинг 4.1 Выход за границу массива (Linux x64)

```
1 char a[] = {'1', '2', '3', '4', '5'};
2 int main() {
3     long i;
4     scanf("%ld", &i);
5     if (i >= 5) return 1;
6     printf("%c\n", a[i]);
7 }
```

приводит к ошибке [29]. Для обнаружения дефекта составляется конъюнкция предиката безопасности и предиката пути после применения алгоритма слайсинга (разд. 2.2), который устраняет избыточные ограничения на символьные переменные, не зависящие по данным от переменных в предикате безопасности [25]. Таким образом, предикат пути отвечает за достижимость анализируемой инструкции (функции), а предикат безопасности — за проявление дефекта. Полученный объединенный предикат передается SMT-решателю [35]. Если предикат выполним, то разработанный метод сообщает об обнаруженной ошибке и генерирует новые входные данные для воспроизведения этой ошибки в анализируемой программе.

Одним из разработанных предикатов безопасности является предикат безопасности для поиска ошибок целочисленного деления на нуль. Во время символьной интерпретации программы каждая символьная инструкция деления, такая как `div` или `idiv`, проверяется предикатом безопасности для поиска ошибки деления на нуль. Если делитель инструкции является символьным, то составляется уравнение на равенство делителя нулю. Затем получившееся уравнение конъюнктируется с предикатом пути, и выполнимость полученного предиката проверяется математическим решателем.

Листинг 4.1 содержит потенциальную ошибку разыменования нулевого указателя на строке 6. Предикат безопасности, обнаруживающий такую ошибку, сравнивает разыменуемый адрес с нулем. Пример на листинге 4.1 компилируется под 64-битную архитектуру.

В стандартный поток ввода программы подается строка +0000000000000000001, чтобы можно было сгенерировать любое знаковое 64-битное целое число типа `long`. Это связано с ограничением метода динамической символьной интерпретации, которая не может увеличить размер

входных данных, т. к. в противном случае не были бы интерпретированы инструкции, обрабатывающие дополнительные входные байты, и символьная модель содержала бы недостаточное число ограничений.

Символьные входные данные читаются с помощью системного вызова `read` внутри функции `scanf`. Для каждого входного байта создается свободная символьная переменная. Начиная с точки чтения входных данных производится символьная интерпретация ассемблерных инструкций, обрабатывающих символьные данные (в результате, обновляется символьное состояние регистров и памяти). Необходимые конкретные значения регистров и памяти получаются из двоичного транслятора [73]. Функция `scanf` вызывает функцию `__strtoll_internal`, которая осуществляет преобразование входной строки в число. Семантика данного преобразования моделируется необходимыми формулами и ограничениями (разд. 3.1.3). В дальнейшем полученная формула числа сравнивается с размером массива *a* на строке 5, и соответствующее ограничение добавляется в предикат пути: $i < 5$. На строке 6 значение индекса *i* является символьным. Поэтому проверяется, может ли символьный адрес $a+i$ равняться нулю. Предикат безопасности для ошибки разыменования нулевого указателя объединяется с предикатом пути. Полученный предикат $i < 5 \wedge a + i = 0$ выполним, потому что на строке 5 отсутствует проверка на неотрицательность индекса. Таким образом, разработанный метод сообщает о найденной ошибке разыменования нулевого указателя инструкцией `movzx eax, byte ptr [rdx + rax]`, где `rax` — это базовый адрес массива *a*, а `rdx` — индекс *i*. Более того, сохраняется файл с новыми входными данными -0000000000006295616, запуск программы на которой приводит к ошибке разыменования нулевого указателя.

Следует отметить, что метод позволяет генерировать входные данные, приводящие к разыменованию нулевого указателя, когда программа скомпилирована без рандомизации адресного пространства (ASLR) [5]. Однако предикат безопасности для разыменования нулевого указателя порождает простой и быстро решаемый запрос к SMT-решателю, который по-прежнему позволяет генерировать входные данные, запуск программы на которых приводит к ошибке сегментации даже с включенным ASLR.

Однако некоторые ошибки разыменования нулевого указателя не могут быть обнаружены при помощи предикатов безопасности. Такие ошибки стоит находить в результате открытия новых путей выполнения программы с исполь-

зованием символьной интерпретации или фаззинга. Например, указатель `p` на листинге ниже не символьный, но может равняться нулю:

```
int *p = 0;
if (some_condition) p = malloc(4);
*p = 3;
```

4.1.1 Обнаружение ошибок выхода за границу массива

Выход за границы массива — одна из самых опасных и распространенных программных ошибок [74]. Для обнаружения таких ошибок конструируется предикат безопасности для каждого разыменования символьного адреса *sym_addr* (который зависит от пользовательских данных).

Чтобы построить предикат безопасности для ошибки выхода за границы массива, сначала требуется определить границы массива $[lower_bound, uppper_bound)$. Определив границы массива, конструируется предикат безопасности, который возвращает истину, когда символьный адрес выходит за границы массива: $sym_addr < lower_bound \lor sym_addr \geqslant upper_bound$. Далее получившиеся неравенства объединяются с предикатом пути (после слайсинга), а итоговый предикат проверяется на выполнимость. Если предикат выполним, то генерируются входные данные, приводящие к выходу за границу массива. Для этого, как описано в разделе 2.2, часть байтов в изначальных входных данных заменяются на значения переменных из модели, полученной от SMT-решателя.

Однако обе границы массива не всегда могут быть определены по бинарному коду. В таком случае предикат безопасности пытается направить символьный адрес за пределы одной из границ (нижней или верхней). В частности, на листинге 4.1 возможна ошибка выхода за границу массива, но по бинарному коду может быть определена только нижняя граница массива `a`. Базовый адрес массива вычисляется эвристически из символьного адресного выражения `[rdx + rax]`, где `rax` — конкретный базовый адрес массива, а `rdx` — символьный индекс. Конкретная часть адресного выражения `rax` полагается за нижнюю границу. Таким образом, возможно сгенерировать входные данные для выхода за нижнюю границу массива (например, -1).

Для определения границ символьных адресов поддерживаются теневая куча и теневой стек. Во время символьной интерпретации моделируются функции для работы с динамической памятью, такие как `malloc`, `calloc`, `realloc`, `free` и т. д. При вызове таких функций обновляется теневая куча, которая хранит границы выделенных в памяти буферов (разд. 3.1.1). Для каждой встреченной инструкции `call` на теневой стек сохраняется адрес, по которому располагается адрес возврата (указатель стека). А для каждой инструкции возврата `ret` элементы с теневого стека снимаются в соответствии с текущим значением указателя стека.

Когда происходит разыменование символьного адреса, предложенный метод обнаруживает границы соответствующего буфера следующим образом. Если текущее конкретное значение адреса находится в теневой куче, то обе границы можно получить из нее. Если адрес указывает на стек, то ближайший адрес на теневом стеке (где расположен адрес возврата), больший текущего адреса, будет считаться верхней границей массива. Нижняя граница вычисляется эвристически из конкретной части формулы символьного адреса. Основной идеей такого способа является суммирование конкретных частей символьной формулы. Более того, учитываются некоторые исключительные ситуации. Например, необходимо различать вычисление индекса массива `a[i - 0x20]` и базовый адрес массива на стеке `[ebp - 0x20]`. Схожие эвристики используются и для вычисления нижней границы глобального массива.

Более того, моделируются функции копирования, такие как `memcpy`, `memmove`, `memset` и т. д., чтобы обнаруживать ошибки переполнения буфера, к которым может привести небезопасное использование таких функций (разд. 3.1.1). Если аргумент размера копирования является символьным, разработанный метод пытается сделать его значение таким, чтобы выйти за верхнюю границу.

Перед решением предиката безопасности к нему добавляются дополнительные ограничения, которые позволяют обнаружить ошибку, которая вероятнее приведет к аварийному завершению программы, перезаписав адрес возврата или обратившись к отрицательному адресу. Если такой более сильный предикат не выполним, метод возвращается к решению изначального предиката безопасности. Если при этом и адрес, и значение, которое будет записано по адресу, являются символьными, дополнительно предупреждается о возможности ошибки записи произвольного значения по произвольному адресу (CWE-123 [75]).

4.1.2 Обнаружение ошибок целочисленного переполнения

Ошибка целочисленного переполнения является одной из самых распространенных программных ошибок [74]. Целочисленное переполнение достаточно часто встречается в бинарном коде. Динамический символьный интерпретатор будет работать слишком долго, если он будет проверять каждое место в программе, где может возникнуть целочисленное переполнение. Более того, в некоторых ситуациях, таких как вычисление значения хэш-функции, целочисленное переполнение является корректным. Поэтому разработанный метод выделяет только критические части программы и для них проверяет предикат безопасности для обнаружения ошибок целочисленного переполнения. Для этого введем следующие понятия. Источник ошибки — это инструкция, где может произойти целочисленное переполнение (например, различные арифметические инструкции, такие как сложение или умножение). Сток ошибки — это место в коде, где предшествующее целочисленное переполнение может привести к критичному дефекту. Таким образом, в отличие от других предикатов безопасности, сток ошибки отделяется от ее источника.

Предикат безопасности для поиска ошибок целочисленного переполнения решается на стоках ошибок, которые используют потенциально переполненное значение. Можно выделить следующие стоки ошибок:

- условные переходы, изменяющие поток управления в зависимости от переполненного значения;
- адреса доступа к памяти;
- аргументы функций.

Особенно критично может быть использование переполненного значения в аргументах таких функций, как `malloc`, `memcpy` и т. п. [69; 70] Все символьные аргументы некоторых моделируемых функций стандартной библиотеки (разд. 3.1) считаются потенциальными стоками. Для остальных функций проверяются первые три аргумента в соответствии со стандартным соглашением о вызовах.

При анализе конкретной инструкции проверяется, является ли она потенциальным источником, т. е. является ли она арифметической и является ли хотя бы один из ее операндов символьным. Если так, то конструируются предикаты безопасности для беззнакового (флаг `CF`) и знакового (`OF`) целочисленного пере-

полнения. Для большинства арифметических инструкций предикат безопасности будет истинным, когда соответствующий флаг выставлен в единицу. Исключением являются инструкции битовых сдвигов, для которых предикат безопасности составляется другим способом так, чтобы он соответствовал целочисленному переполнению в таких случаях [29].

Затем проверяется, задействовано ли потенциально переполненное значение, т. е. источник ошибки, в вычислении стока. Для этого выясняется, является ли абстрактное синтаксическое дерево (AST) источника (результат переполненной арифметической операции) ребенком AST стока. Если так, то вычисление стока содержит потенциально переполненное значение.

Далее определяется знаковость арифметической операции с помощью обратного слайсинга [76]. Это необходимо для того, чтобы можно было выбрать один из двух предикатов безопасности для знакового или беззнакового переполнения. Важно определить знаковость, т. к. отсутствие знания о знаковости может повлечь за собой большое число ложно положительных срабатываний.

Предложенный алгоритм 2 позволяет определить знаковость арифметической операции [29]. В точке стока ошибки производится итерирование в обратном порядке, начиная с последнего перехода в предикате пути П. В результате, обнаруживается первый условный переход, который может указать на знаковость [69]. Например, условный переход `JL` говорит о том, что значение знаковое. Помимо этого проверяется, что условный переход использует хотя бы одну символьную переменную из AST стока ошибки *sink*, а место вызова функции *s*, содержащей инструкцию перехода *inst*, находится в текущем стеке вызовов *call_stack*.

Более того, знаковость может быть определена, когда символьное число получается из функции `strto*l`. Например, функция `strtol` используется для знаковых целых чисел, а `strtoul` — для беззнаковых.

Наконец, когда информация о знаковости получена (или нет), осуществляется проверка выполнимости предиката безопасности. Если знаковость известна, то проверяется один предикат безопасности для этой знаковости. Производится слайсинг предиката пути (разд. 2.2) в точке стока ошибки. Полученные релевантные ограничения объединяются с предикатом безопасности. Если предикат выполним, то сообщается об ошибке знакового или беззнакового целочисленного переполнения и генерируются соответствующие входные данные.

Если же знаковость определить не удалось, проверяется, могут ли одновременно произойти целочисленные переполнения обеих знаковостей. Другими

Алгоритм 2: Определение знаковости через обратный слайсинг

Входные данные: $sink$ — AST с переполненным значением (сток), $call_stack$ — стек вызовов в месте стока ошибки, Π — предикат пути (ограничения пути до стока ошибки).

Результат: True, если тип знаковый; False, если тип беззнаковый; None, если знаковость не может быть определена.

$vars \leftarrow used_variables(sink)$ `/* переменные слайсинга */`

`/* Итерирование по ограничениям пути в обратном порядке. */`

forall $c \in reversed(\Pi)$ **do**

  `/* Использует ли ограничение` c `переменные` $vars$ `из AST стока` $sink$`? */`

  if $vars \cap used_variables(c) \neq \varnothing$ **then**

    `/* Получение инструкции, соответствующей ограничению` c`. */`

    $inst \leftarrow instruction(c)$

    `/* Получение места вызова функции, содержащей инструкцию` $inst$`. */`

    $s \leftarrow callsite(function(inst))$

    if $inst$ *is branch* **and** $s \in call_stack$ **then**

      if $inst$ *is signed* **then** `// js/jns/jg/jge/jl/jle`

        return True

      if $inst$ *is unsigned* **then** `// ja/jae/jb/jbe`

        return False

      `/* Знаковость не может быть определена (например, jz). */`

return None

словами, могут ли некоторые входные данные повлечь и знаковое, и беззнаковое переполнение одновременно. Для этого оба предиката безопасности для знакового и беззнакового переполнения объединяются. Если полученный предикат выполним, то генерируются соответствующие входные данные и сообщается об обеих ошибках знакового и беззнакового целочисленного переполнения. Если же предикат невыполним, то оба предиката проверяются раздельно. Если оба предиката по отдельности выполнимы, то выводится два предупреждения

об ошибке и сохраняются два набора входных данных для каждого случая. Метод не сообщает об ошибке, когда только один из двух предикатов выполним, т. к. без знания знаковости это может привести к ложно положительным срабатываниям.

Кроме того, добавляются дополнительные ограничения на предикат безопасности для функций выделения памяти, таких как `malloc`, `calloc`, и функций копирования, таких как `memcpy`, т. к. переполнение в таких функциях может привести к более серьезным последствиям. Для функций выделения памяти добавляются дополнительные ограничения, чтобы значение, получившееся в результате переполнения, оказалось меньше чем значение, на котором происходило конкретное выполнение, но при этом не равнялось нулю, что позволяет переполнить размер выделяемой памяти и при этом ее успешно выделить (если переполненное число будет больше количества памяти на машине, то `malloc` вернет нуль). Для функций копирования накладываются ограничения, чтобы переполненное значение было больше чем то, что было при конкретном выполнении программы. Если предикат с дополнительными ограничениями не выполним, проверяется изначальный предикат безопасности.

Также обрабатывается случай, когда арифметическая операция производится с типами, размер которых меньше, чем `sizeof(int)`. В таком случае в силу расширения целочисленных типов данных (integer promotion) при вычислениях оригинальная инструкция (например, `add eax, 1`) не может привести к переполнению, поэтому конструируются новые предикаты безопасности для целочисленного переполнения с корректным размером операндов.

Более того, учитывается длинная арифметика для сложения и вычитания типов `int64_t` на 32-битной архитектуре. А также оптимизации компилятора, которые подменяют вычитание сложением, например, заменяют инструкцию `sub eax, 1` на `add eax, 0xffffffff` [69].

На листинге 4.2 приводится пример программы, содержащей ошибку целочисленного переполнения, которая влечет за собой переполнение буфера (на 32-битной архитектуре). Программа запускается с входными данными +00000000002, которых достаточно для генерации любого знакового целого 32-битного числа. Размер выделяемого массива *size* читается функцией `scanf` с помощью функции `strtol`. Символьная формула для *size* получается в результате моделирования семантики преобразования строки в число (разд. 3.1.3). Размер массива *size* умножается на `sizeof(int)`, что является потенциальным источником ошибки целочисленного переполнения. В этой точке конструиру-

Листинг 4.2 Целочисленное переполнение в размере `malloc` (32 бита)

```
int main() {
    int size;
    fscanf(stdin, "%d", &size);
    if (size <= 0) return 1;
    size_t i;
    int *p = malloc(size * sizeof(int));
    if (p == NULL) return 1;
    for (i = 0; i < (size_t)size; i++) {
        p[i] = 0;
    }
    printf("%d\n", p[0]);
    free(p);
}
```

ется предикат безопасности для знакового целочисленного переполнения при умножении (знаковость определяется из функции `strtol`). Потенциальный сток ошибки — это аргумент функции `malloc`, который использует переполненное значение. Дополнительно предикат безопасности объединяется с ограничением, что значение $size * sizeof(int)$ после переполнения должно быть меньше, чем изначальное $2 * 4 = 8$ из реального выполнения. Это дополнительное ограничение помогает обойти проверку на строке 7, т. к. размер выделяемой памяти должен быть меньше суммарной памяти, доступной на машине. Затем составляется конъюнкция предиката безопасности, дополнительного ограничения и предиката пути после слайсинга (разд. 2.2): $(long\ long)size * sizeof(int) \neq size * sizeof(int) \land (unsigned)size * sizeof(int) < 8 \land size > 0$, где $size$ — 32-битное целое число. Полученный предикат передается математическому решателю, который сообщит, что предикат выполним, и вернет соответствующую модель. Таким образом, метод сгенерирует новые входные данные +01073741825, которые приведут программу к целочисленному переполнению на строке 6. А функция `malloc` выделит только $(unsigned)1073741825 * sizeof(int) = 4$ байта из-за переполнения. Следовательно, на строке 9 произойдет ошибка переполнения буфера, т. к. цикл будет итерироваться по 1073741825 элементам массива, в то время как его настоящий размер равен 1.

4.2 Метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности в гибридном фаззинге

Для системного поиска ошибок в программах был разработан следующий метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности в гибридном фаззинге [28; 30]. Сначала производится гибридный фаззинг исследуемого проекта, при котором совместно работают фаззер [13; 14] и инструмент динамической символьной интерпретации [25] для открытия путей выполнения программы (разд. 2.1). После этого производится минимизация корпуса входных файлов, полученных в результате гибридного фаззинга. Затем идет этап проверки предикатов безопасности на всех файлах, полученных после минимизации корпуса. Срабатывания предикатов безопасности верифицируются с помощью исполняемого файла, собранного с санитайзерами. Подтвержденные санитайзерами срабатывания уникализируются по источнику ошибки. Уникальные верифицированные срабатывания оцениваются человеком на критичность и корректность.

Для подготовки к этапу гибридного фаззинга требуется скомпилировать два исполняемых файла для каждой фаззинг-цели исследуемого проекта: первый должен быть скомпилирован с санитайзерами и инструментацией подсчета покрытия для фаззера, второй — без санитайзеров, но с отладочной информацией (он будет использоваться символьным интерпретатором). Отладочная информация необходима для сопоставления срабатывания предикатов безопасности и предупреждения санитайзера на этапе верификации. После подготовки начинается этап гибридного фаззинга, в результате которого получается корпус входных файлов, полученных от фаззера и символьного интерпретатора.

Во время конструирования предикатов безопасности анализируется один конкретный путь выполнения. Поэтому для проверки как можно большего количества инструкций программы необходимо осуществить символьную интерпретацию программы на как можно большем числе входных файлов. Однако получившийся в результате гибридного фаззинга корпус может содержать большое количество файлов, похожих друг на друга и покрывающих один и тот же код в исследуемом проекте. Поэтому перед проверкой предикатов безопасности необходимо минимизировать корпус, чтобы оставить в нем наиболее репрезен-

тативное подмножество файлов, дающее такое же покрытие кода программы, но при этом содержащее намного меньше файлов, чем изначальный корпус.

После минимизации корпуса начинается этап проверки предикатов безопасности. Проверка предикатов безопасности осуществляется в результате динамической символьной интерпретации программы (без санитайзеров) на каждом файле из минимизированного корпуса. Новые файлы, генерируемые предикатами безопасности, верифицируются с помощью исполняемого файла, собранного с санитайзерами. Если при запуске этого исполняемого файла санитайзеры выдают предупреждение для строчки программы, аналогичной той, для которой был сгенерирован входной файл, то результат считается верифицированным и сохраняется отдельно. Более того, если запуск собранного с санитайзерами исполняемого файла на проверяемом файле приводит к выводу предупреждения санитайзеров, отсутствовавшему при запуске этого исполняемого файла на оригинальном входном файле (на котором проводилась символьная интерпретация и проверка предикатов безопасности), результат также считается верифицированным. Отдельно проверяется, приводит ли проверяемый файл к аварийному завершению. Обнаруженные верифицированные срабатывания уникализируются по имени файла и строчке исходного кода источника ошибки.

После этапа проверки предикатов безопасности стоит вручную проверить уникальные подтвержденные срабатывания. Зачастую найденные ошибки действительно приводят к срабатыванию санитайзеров, но с точки зрения логики работы программы могут ошибками не являться. В частности, хеширующие функции часто содержат целочисленные переполнения.

4.3 Экспериментальная оценка точности метода построения предикатов безопасности на наборе тестов Juliet

Набор тестов Juliet для C/C++ [26] покрывает 118 различных типов ошибок CWE [1]. Тесты Juliet изначально нацелены на измерение точности обнаружения дефектов инструментами статического анализа. Поэтому сборочная система Juliet была адаптирована так, чтобы она подходила для измерения точности инструментов динамического анализа [29; 77]. При этом исходный код самих тестов не модифицировался. Производилась сборка только тех тестов, которые читают

пользовательские данные, чтобы их возможно было модифицировать для вызова программной ошибки. В Juliet каждый файл с исходным кодом содержит два теста: положительный и отрицательный. Положительный тест содержит ошибку, в то время как отрицательный тест избегает потенциального дефекта с помощью дополнительных проверок. Эти тесты реализованы в функциях `bad` (положительный) и `good` (отрицательный), которые вызываются напрямую из функции `main`. Оба теста компилируются в два отдельных исполняемых файла: один имеет дефект, а второй не содержит ошибок. Таким образом, каждый исполняемый файл выполняет один (положительный или отрицательный) тест. Дополнительно для каждого теста компилируется его версия с санитайзерами (разд. 1.4.1). Проверка символьных предикатов безопасности осуществляется на исполняемом файле без санитайзеров. Затем проверяется, что запуск программы с санитайзерами на сгенерированных входных данных действительно приводит к срабатыванию ошибки. Таким образом, был разработан Juliet C/C++ Dynamic Test Suite [77], который позволяет измерять долю истинно положительных (чувствительность) и истинно отрицательных (специфичность) срабатываний инструмента динамического обнаружения ошибок.

Следует отметить, что тесты Juliet читают символьные входные данные с помощью функций `fscanf` или `fgets+atoi`. Оба варианта используют функцию `strto*l` для преобразования строки в число. Поэтому моделирование семантики соответствующей функции (разд. 3.1.3) необходимо для обнаружения ошибок в наборе тестов Juliet.

Результаты тестирования на Juliet разработанного метода обнаружения ошибок при помощи символьных предикатов безопасности приводятся в таблице 9. Измерения производились для подмножества CWE [1] из Juliet. Предикат безопасности для выхода за границу массива покрывает CWE 121, 122, 124, 126, 127, 194 и 195. Предикат безопасности для целочисленного переполнения обнаруживает ошибки для CWE 190, 191 и 680. CWE369 обрабатывается предикатом безопасности для деления на нуль.

Тесты для некоторых CWE пропущены, потому что они не читают пользовательские входные данные (например, CWE476: разыменование нулевого указателя). Это является следствием того, что изначально тесты Juliet разрабатывались для статического анализа.

Каждое поддерживаемое предикатами безопасности CWE компилируется для 32-битной и 64-битной архитектуры. Единственным исключением является

Таблица 9 — Результаты тестирования на Juliet

CWE	P=N	Текстовые срабатывания			Верификация санитайзерами		
		TPR	**TNR**	**ACC**	**TPR**	**TNR**	**ACC**
CWE121: Stack Based Buffer Overflow	188	100 %	100 %	100 %	100 %	100 %	100 %
CWE122: Heap Based Buffer Overflow	376	100 %	100 %	100 %	100 %	100 %	100 %
CWE124: Buffer Underwrite	188	100 %	100 %	100 %	100 %	100 %	100 %
CWE126: Buffer Overread	188	100 %	100 %	100 %	100 %	100 %	100 %
CWE127: Buffer Underread	188	100 %	100 %	100 %	100 %	100 %	100 %
CWE190: Integer Overflow	2580	99.92 %	90.89 %	95.41 %	98.10 %	90.89 %	94.50 %
CWE191: Integer Underflow	1922	99.90 %	91 %	95.45 %	97.45 %	91 %	94.22 %
CWE194: Unexpected Sign Extension	752	100 %	100 %	100 %	100 %	100 %	100 %
CWE195: Signed to Unsigned Conversation Error	752	99.87 %	100 %	99.93 %	99.87 %	100 %	99.93 %
CWE369: Divide by Zero	564	66.67 %	100 %	83.33 %	66.67 %	100 %	83.33 %
CWE680: Integer Overflow to Buffer Overflow	188	100 %	100 %	100 %	100 %	100 %	100 %
ИТОГО	7886	97.55 %	94.83 %	96.19 %	96.36 %	94.83 %	95.59 %

CWE680 (целочисленное переполнение, приводящее к переполнению буфера), которое компилируется только для 32-битной архитектуры. Это происходит из-за того, что тесты Juliet для CWE680 разработаны для 32-битной архитектуры, при компиляции их для 64-битной архитектуры ошибка попросту отсутствует. Например, на листинге 4.2 входная переменная `size` является 32-битным целым числом. Потенциальное целочисленное переполнение происходит в аргументе функции `malloc`. Входная переменная `size` типа `int` умножается на `sizeof(int)`, который имеет тип `size_t`. В 64-битной архитектуре тип `size_t` эквивалентен типу `unsigned long long`. Поэтому 32-битное значение `size` сначала знаково расширяется до 64 бит, а потом умножается на 4 (`sizeof(int)`). В таком случае знаковое целочисленное переполнение невозможно.

Разработанная тестовая система Juliet Dynamic [77] измеряет число истинно положительных (*TP*) и истинно отрицательных (*TN*) срабатываний. Затем вычисляются доля истинно положительных срабатываний (чувствительность) $TPR = \frac{TP}{P}$, доля истинно отрицательных срабатываний (специфичность) $TNR = \frac{TN}{N}$ и точность $ACC = \frac{TP+TN}{P+N}$ (Juliet содержит равное число истинных и отрицательных тестов $P = N$). Результаты измеряются для двух типов срабатываний:

- **Текстовые срабатывания** — инструмент, реализующий символьные предикаты безопасности, выводит текстовое предупреждение о наличии ошибки в тесте.
- **Верификация санитайзерами** — сгенерированные входные данные для обнаруженных ошибок приводят к срабатыванию санитайзеров.

Ниже описывается, как производятся измерения.

Изначально собираются пути до исполняемых файлов с тестами и выбираются соответствующие входные данные. Для этого специально были составлены входные данные для типов `int64_t`, `int`, `short` и `char`. В частности, для типа `int` в стандартный поток ввода передается строка +00000000002. Знак плюса ('+') позволяет изменить знак в результате динамической символьной интерпретации. А ведущие нули требуются для генерации больших чисел, т. к. динамическая символьная интерпретация не может увеличивать размер входных данных. Однако дополнительный первый ведущий нуль (сразу после знака плюса) необходим, т. к. функция `scanf` конкретизует первый нуль:

```
1 if (buf[i] == '0') buf[i] = '0';
```

Затем производится проверка предикатов безопасности для каждого теста с соответствующими входными данными. Если для положительного теста (P) выдается текстовое предупреждение, то срабатывание классифицируется как истинно положительное (TP). Если срабатывание происходит на отрицательном тесте (N), то оно классифицируется как ложно положительное (FP). Срабатывание считается истинно отрицательным (TN), если для отрицательного теста (N) не было обнаружено ошибок. Ложно отрицательное срабатывание (FN) происходит, когда отсутствуют предупреждения для положительного теста (P).

Наконец, сгенерированные на предыдущем шаге входные данные верифицируются с помощью санитайзеров. Для истинно положительного срабатывания (TP) мог быть сгенерирован как один, так и несколько входных файлов для воспроизведения ошибки. Например, могло быть обнаружено несколько ошибок для вложенных функций-стоков с одним и тем же источником ошибки. Также генерируется пара входных данных для знакового и беззнакового переполнения, когда не удалось определить знаковость арифметической операции. Версия теста с санитайзерами запускается на всех сгенерированных для этого теста входных данных. Если хотя бы на одних входных данных программа выводит предупреждение от санитайзеров, то срабатывание для теста остается истинно положительным (TP). Однако если предупреждения от санитайзеров отсутствуют, класс срабатывания изменяется на ложно отрицательный (FN) после верификации санитайзерами. Число ложно положительных (FP) и истинно отрицательных (TN) срабатываний остается таким же, как на предыдущем шаге.

В таблице 9 приводятся результаты измерений TPR, TNR и ACC для текстовых срабатываний и их верификации санитайзерами. Разработанный метод поиска ошибок при помощи символьных предикатов безопасности показал общую точность 95.59 % для 11 CWE [1] (15772 теста) из набора тестов Juliet [26]. Предложенный метод полностью покрывает ($ACC = 100\ \%$) тестовые наборы для CWE 121, 122, 124, 126, 127, 194 и 680. Однако предикаты безопасности пропускают часть ошибок деления на нуль (CWE369), т. к. используемый фреймворк динамической символьной интерпретации Triton [23] не поддерживает арифметику с плавающей точкой.

Как можно заметить, тесты на целочисленное переполнение (CWE190/191) показывают несколько ложно положительных (FP) и ложно отрицательных (FN) срабатываний. В 32-битной архитектуре тип `int64_t` порождает длинную ариф-

метику, которую разработанный метод учитывает для сложения и вычитания. Однако поддержка длинной арифметики для умножения пока не реализована.

Другие ложные срабатывания вызваны вычитанием из типа `int64_t` в 32-битной архитектуре. Некоторые оптимизации компилятора подменяют вычитание `int64_t` сложением, например, заменяют `sub eax, 1` на `add eax, 0xffffffff`. Разработанный метод учитывает такую ситуацию для обычной арифметики, но не учитывает для длинной арифметики.

Более того, ложно положительные срабатывания присутствуют для тестов Juliet, возводящих в квадрат переменную `data` типа `unsigned int`:

```
1 if (abs((long)data) < (long)sqrt((double)UINT_MAX))
2 {
3     unsigned int result = data * data;
4     printUnsignedLine(result);
5 }
```

Алгоритм определения знаковости указывает, что тип переменной `data` знаковый. Это происходит из-за того, что тест явно приводит переменную `data` к знаковому типу `long` в условном переходе. Следовательно, алгоритм обратного слайсинга от операции умножения первым обнаружит знаковый условный переход. Таким образом, разработанный метод в данном случае выведет ложно положительное срабатывание с ошибкой знакового целочисленного переполнения, в то время как арифметическая операция на самом деле беззнаковая. Данный пример иллюстрирует ограничение метода динамического анализа бинарного кода при отсутствии исходного кода и отладочной информации. Однако остается открытым вопрос, насколько вероятен такой условный переход из синтетического теста в реальных программах.

Артефакты описанных в данном разделе измерений доступны в репозитории Juliet Dynamic [77]. Репозиторий хранит скрипт для воспроизведения результатов.

4.4 Апробация методов путем поиска новых ошибок в проектах с открытым исходным кодом

Разработанный метод автоматизированного поиска ошибок был применен к ряду проектов с открытым исходным кодом [28; 30; 31]. В результате апробации

Листинг 4.3 Беззнаковое целочисленное переполнение в `fseek` (BMP)

```
1 unsigned off_head, off_setup, off_image, i, temp;
2 ...
3 fseek(ifp, off_setup + 792, SEEK_SET);
```

Листинг 4.4 Знаковое целочисленное переполнение в `fseek` (TIFF)

```
1 fseek(ifp, offset + length - 4, SEEK_SET);
```

разработанного метода было обнаружено 17 новых ошибок в 10 различных проектах. Информация обо всех ошибках [78—90] была доведена до разработчиков, часть из них уже исправлены.

4.4.1 FreeImage

FreeImage — это библиотека с открытым исходным кодом [91], которая поддерживает работу с изображениями различных форматов, таких как PNG, BMP, JPEG, TIFF и т. д. С помощью разработанного метода в коде FreeImage было найдено 5 ошибок целочисленного переполнения на этапе проверки предикатов безопасности [78].

На листинге 4.3 приводится фрагмент кода, относящийся к обработке изображений формата BMP, в котором была найдена ошибка беззнакового целочисленного переполнения при вызове функции `fseek`, смещающей текущую позицию в файле. Так как в коде отсутствуют проверки введенных значений, выражение `off_setup + 792` может привести к переполнению, что в свою очередь может привести к неправильной позиции в файле, измененной с помощью функции `fseek`. Источником ошибки, определенным проверкой предикатов безопасности, является инструкция сложения, реализующая вычисление выражения `off_setup + 792`, а стоком ошибки является аргумент смещения в функции `fseek`.

Аналогичная ошибка представлена на листинге 4.4, где также была найдена ошибка целочисленного переполнения при вызове функции `fseek`, но уже в части кода, отвечающей за обработку изображений формата TIFF. Найденная

Листинг 4.5 Неявное преобразование в `parse_tiff`

```
int parse_tiff(int base);
...
parse_tiff(thumb_offset + 12);
```

Листинг 4.6 Беззнаковое переполнение в условном переходе

```
if (*len * tagtype_dataunit_bytes[(*type <=
    LIBRAW_EXIFTAG_TYPE_IFD8) ? *type : 0] > 4)
    fseek(ifp, get4() + base, SEEK_SET);
```

Листинг 4.7 Беззнаковое целочисленное переполнение при вычислении ширины

```
width = raw_width - left_margin - (get4() & 7);
```

ошибка целочисленного переполнения происходит в аргументе смещения функции `fseek` при вычислении выражения `offset + length - 4`.

На листинге 4.5 представлен фрагмент кода FreeImage, где в аргументе функции `parse_tiff` происходит неявное преобразование типа `long` к типу `int`, где переменная `thumb_offset` имеет тип `long`. Так как предикат безопасности для поиска ошибок целочисленного переполнения определяет размер операндов по размеру типа стока, являющегося аргументом функции, то предикат безопасности посчитал эту ошибку как целочисленное переполнение, которое по факту является неявным преобразованием типа, где значение, получающееся из выражения `thumb_offset + 12` типа `long`, не может быть помещено в аргумент функции типа `int` меньшего размера.

На листинге 4.6 приводится фрагмент когда, где в условии оператора ветвления была найдена ошибка беззнакового целочисленного переполнения при умножении `*len` на `tagtype_dataunit_bytes[(*type <= LIBRAW_EXIFTAG_TYPE_IFD8) ? *type : 0]`. Инструкцией источника является умножение, а стоком — инструкция условного перехода. Были подобраны такие входные данные, что при целочисленном переполнении меняется поток управления, что приводит к ошибочной работе программы.

На листинге 4.7 приводится фрагмент кода, где вычисляется некоторая ширина. При ее вычислении может произойти ошибка беззнакового целочисленного переполнения. Значение позже используется в условных переходах и многих

Листинг 4.8 Беззнаковое целочисленное переполнение в xlnt

```
1 in_->seekg(static_cast<std::ptrdiff_t>(sector_data_start() +
2         sector_size() * static_cast<std::size_t>(id)));
3 std::vector<byte> sector(sector_size(), 0);
```

других местах, и найденная ошибка может привести к неправильной работе программы.

4.4.2 xlnt

xlnt — это библиотека с открытым исходным кодом для языка C++ для управления электронными таблицами и их чтения/записи из/в файлы формата XLSX [92]. С помощью разработанного метода в xlnt была найдены две ошибки целочисленного переполнения [83; 84] и ошибка выхода за границы массива [85].

На листинге 4.8 приведен фрагмент кода, где были найдены обе ошибки беззнакового целочисленного переполнения в аргументе функции `seekg` в выражении `sector_data_start() + sector_size() * static_cast <std::size_t>(id)` при умножении и сложении соответственно. Источником ошибки является вычисление приведенного выражения, а стоком — аргумент вызываемой функции, являющийся результатом вычислений. Сама ошибка не приводит к серьезным последствиям, однако подобранные для воспроизведения переполнения значения могут привести к следующим исходам. Был найден набор входных данных, при котором значения `sector_size()` и `static_cast<std::size_t>(id)` достаточно велики, чтобы при перемножении привести к переполнению. Так как значение `sector_size()` очень велико, то вызов конструктора `std::vector<byte> sector(sector_size(), 0)` приводит к аварийному завершению работы программы из-за невозможности выделить слишком большое количество памяти. Также был найден набор входных данных, при котором значение `id` равняется -1, и при приведении этого значения к типу `size_t` получается очень большое значение. Это также приводит к переполнению в указанном выражении. Однако, так как значение `sector_size()` не было подобрано

Листинг 4.9 Выход за границы массива в xlnt

```
sector_chain
compound_document::follow_chain(sector_id start,
                                const sector_chain &table)
{
    auto chain = sector_chain();
    auto current = start;
    while (current >= 0)
    {
        chain.push_back(current);
        current = table[static_cast<std::size_t>(current)];
    }
    return chain;
}
```

большим, то аварийного завершения программы при вызове конструктора на следующей строке не происходит. Но далее в том же файле исходного кода в функции `read_directory` происходит обращение на чтение из памяти: `entries_[static_cast<std::size_t>(current_entry_id)]`. При обращении в память в силу подобранных для переполнения значений происходит выход за границы массива и аварийное завершение.

На листинге 4.9 приведен фрагмент кода, где была найдена ошибка выхода за границы буфера. Ошибка происходит при обращении по адресу на строке `current = table[static_cast<std::size_t>(current)]`. Значение `current` было подобрано равным 2147483647, что приводит к обращению в память за границами буфера. Ошибка была найдена независимо как фаззером, так и с использованием предикатов безопасности.

4.4.3 unbound

unbound — это проверяющий, рекурсивный, кэширующий распознаватель DNS [93]. В коде проекта unbound при помощи разработанного метода была найдена ошибка беззнакового целочисленного переполнения. Фрагмент кода, в котором была найдена ошибка, представлен на листинге 4.10.

Листинг 4.10 Беззнаковое целочисленное переполнение в unbound

```
int sign = 0;
uint32_t i = 0;
uint32_t seconds = 0;
for(*endptr = nptr; **endptr; (*endptr)++) {
    switch (**endptr) {
    ...
    case '9':
        i *= 10;
    ...
}
```

Листинг 4.11 Деление на нуль в hdp

```
int32 buf_size;
/* we are bounded above by VDATA_BUFFER_MAX */
buf_size = MIN(total_bytes, VDATA_BUFFER_MAX);
/* make sure there is at least room for one record in our buffer*/
chunk = buf_size / hsize + 1;
```

Переполнение происходит в функции `sldns_str2period`, которая обрабатывает строку и переводит ее в некоторое значение времени. Переполнение происходит в переменной `i` внутри цикла обработки строки. Переполнение может привести к неправильной работе со строкой и вследствие к неправильному результату работы функции. Ошибка была исправлена автором проекта [86].

4.4.4 hdp

HDF — это библиотека, предоставляющая набор утилит для работы с файлами формата hdf, который используется для хранения научных данных [94]. Одной из таких утилит является hdp, которая обеспечивает быстрое получение основной информации из hdf-файлов. В ходе применения разработанного метода автоматизированного поиска ошибок в hdp была найдена ошибка целочисленного деления на нуль, приводящая к аварийной остановке работы утилиты [87].

Листинг 4.12 Целочисленное переполнение в miniz

```
1 if (cdir_size < pZip->m_total_files * MZ_ZIP_CENTRAL_DIR_HEADER_SIZE)
2     return mz_zip_set_error(pZip, MZ_ZIP_INVALID_HEADER_OR_CORRUPTED);
```

На листинге 4.11 приведен фрагмент кода, в котором происходит деление на нуль. Код приведен из функции `VSread`, которая считывает данные в некоторый буфер. Приведенный фрагмент кода вычисляет, какое количество элементов можно считать за раз. Так как в коде отсутствует проверка на равенство делителя нулю, на этапе проверки предикатов безопасности подбирается значение `hsize`, равное 0, что приводит к ошибке деления на нуль.

4.4.5 miniz

miniz — это высокопроизводительная библиотека сжатия данных без потерь в одном исходном файле, реализующая стандарты формата сжатия данных zlib и Deflate [95]. Также miniz входит в зависимости PyTorch — популярного фреймворка для машинного обучения [96]. В результате применения разработанного метода тестирования в miniz была найдена ошибка целочисленного переполнения и предложено исправление [88], которые было принято разработчиками.

На листинге 4.12 приведен фрагмент кода, в котором может произойти целочисленное переполнение. Приведенное условие проверяет заголовок файла на корректность, и в случае, если заголовок некорректен, возвращается ошибка. В результате применения разработанного метода удалось найти входные данные, которые приводят к ошибке целочисленного переполнения. Однако для условных переходов накладываются дополнительные условия на предикат безопасности, чтобы при проявлении ошибки был изменен поток управления. Так как при конкретном исполнении проверка проходила успешно, то были подобраны такие входные данные, при которых происходит ошибка целочисленного переполнения и меняется поток управления, в результате чего происходит возврат из функции с сообщением об ошибке. Для такого случая были изменены дополнительные условия на предикат безопасности так, чтобы при проявлении ошибки поток управления сохранялся. В результате этого эксперимента получились такие вход-

Листинг 4.13 Целочисленное переполнение в EXRS

```
1 let pixel_data = i8::read_vec(read, width * height *
    components_per_pixel, 1024*1024*4, None, "preview attribute
    pixel count")?;
```

ные данные, при которых заголовок файла является некорректным, но благодаря целочисленному переполнению булево выражение в условии принимает значение истины, и проверка проходит успешно.

4.4.6 EXRS

EXRS [97] — это Rust библиотека для чтения и записи изображений формата OpenEXR, который используется для анимации. Данная библиотека входит в Rust пакет image-rs [98], который в свою очередь используется в движке веб-браузера Firefox Quantum [99]. Разработанный метод поиска ошибок при помощи символьных предикатов безопасности обнаружил ошибку целочисленного переполнения в размере выделяемой памяти [89].

На листинге 4.13 приводится строка, содержащая целочисленное переполнение. Переполнение происходит при вычислении размера выделяемого вектора для чтения изображения: `width * height * components_per_pixel`. При отсутствии дополнительных проверок результат умножения переполняется, что приводит к выделению вектора меньшего размера, который не может вместить всю читаемую картинку. Были добавлены необходимые проверки, и ошибка была успешно исправлена.

4.4.7 Goblin

Goblin — это Rust библиотека для парсинга исполняемых файлов (ELF, PE, Mach-O и т. д.). В данной библиотеке разработанным методом было найдено два целочисленных переполнения при вычитании [79; 90].

Листинг 4.14 Целочисленное переполнение при вычитании в Goblin

```
let read_size =
  ((section.pointer_to_raw_data as usize + size_of_raw_data +
    file_alignment - 1)
   & !(file_alignment - 1))
   - aligned_pointer_to_raw_data(section.pointer_to_raw_data as usize);
```

Листинг 4.15 Целочисленное переполнение при вычитании в Goblin

```
// adjust the offset and size accordingly
*offset = header_offset + SIZEOF_HEADER + len;
header.size -= len;
```

Листинг 4.16 Целочисленное переполнение в OpenJPEG

```
l_y1 = p_cp->ty0 + (p_cp->th - 1U) * p_cp->tdy; /* can't overflow */
```

На листинге 4.14 происходит целочисленное переполнение при вычитании единицы из `file_alignment`, равного нулю. Данная ошибка приводит к некорректному определению размера читаемой секции исполняемого файла формата PE.

На листинге 4.15 происходит целочисленное переполнение при вычитании `len` из `header.size`. Переполнение происходит, когда значение `len` больше `header.size`. Данная ошибка приводит к некорректному парсингу заголовка исполняемого файла.

4.4.8 OpenJPEG

OpenJPEG [100] — это кодек для изображений формата JPEG 2000. Разработанный метод был применен к библиотеке компьютерного зрения OpenCV [101], которая использует OpenJPEG для работы с JPEG изображениями. В результате, была обнаружена ошибка целочисленного переполнения в коде OpenJPEG [80].

На листинге 4.16 приводится строка, где было обнаружено переполнение при умножении в OpenJPEG. Следует отметить, что строка содержит комментарий, что переполнение невозможно. Однако с использованием символьных

Листинг 4.17 Целочисленное переполнение в Poppler

```
1 bitmap = readGenericBitmap(mmr, (grayMax + 1) * patternW, patternH
    , templ, false, false, nullptr, atx, aty, length - 7);
```

предикатов безопасности все же удалось подобрать входные данные, приводящие к переполнению.

4.4.9 Poppler

Poppler [102] — это библиотека для рендеринга PDF файлов. В ней было обнаружено целочисленное переполнение в функции `readPatternDictSeg` [81].

На листинге 4.17 приводится строка с целочисленным переполнением в умножении `(grayMax + 1) * patternW`. Данная ошибка была принята и исправлена разработчиками.

4.4.10 Rizin

Rizin [103] — это фреймворк для обратной разработки. Предложенный метод обнаружил в нем ошибку целочисленного переполнения [82], которая была исправлена разработчиками.

На листинге 4.18 происходит целочисленное переполнение при вычислении значения `symbols_size`, которое в дальнейшем используется как размер выделяемой памяти (`calloc`) для массива `symbols`. В результате, выделяется меньше памяти, чем необходимо для хранения всех элементов в массиве. А разыменование `symbols[j]` приводит к выходу за границу массива. Таким образом, из-за целочисленного переполнения была выделена память, недостаточная для хранения всех элементов, что привело к выходу за границу массива при попытке сохранения очередного элемента.

Листинг 4.18 Целочисленное переполнение в Rizin, приводящее к выходу за границы массива

```
symbols_size = (symbols_count + 1) * 2 * sizeof(struct symbol_t);
if (symbols_size < 1) {
    ht_pp_free(hash);
    return NULL;
}
if (!(symbols = calloc(1, symbols_size))) {
    ht_pp_free(hash);
    return NULL;
}
...
symbols[j].last = true;
```

Листинг 4.19 Целочисленное переполнение в TensorFlow

```
char* FastInt32ToBufferLeft(int32 i, char* buffer) {
  uint32 u = i;
  if (i < 0) {
    *buffer++ = '-';
    u = -i;
  }
  return FastUInt32ToBufferLeft(u, buffer);
}
```

4.4.11 TensorFlow

Во время проведения аудита фреймворка машинного обучения TensorFlow [104] была найдена ошибка целочисленного переполнения с использованием разработанных методов [31].

На листинге 4.19 возможно целочисленное переполнение при попытке получить отрицание `-i`, когда `i` равняется минимальному целочисленному значению `INT_MIN`. Однако ручной анализ ошибки показал, что на практике она не может привести к некорректному поведению программы.

Глава 5. Реализация предложенных методов в программных инструментах

В данной главе описываются детали реализации разработанных методов в программных инструментах [32; 33].

5.1 Инструмент динамической символьной интерпретации Sydr

Динамическая символьная интерпретация, а также разработанные алгоритм слайсинга предиката пути и методы моделирования семантики функций и построения предикатов безопасности были реализованы в инструменте Sydr (Symbolic DynamoRIO) [25; 29; 32]. Sydr использует инструмент динамической бинарной инструментации DynamoRIO [73] для конкретной интерпретации и фреймворк Triton [23] для динамической символьной интерпретации. Sydr осуществляет символьную интерпретацию одного конкретного пути выполнения программы, заданного изначальными входными данными, а также генерирует новые входные данные для инвертирования целевых переходов с целью открытия новых путей и воспроизведения обнаруженных ошибок с помощью предикатов безопасности.

Можно выделить два основных подхода к реализации динамического символьного интерпретатора:

1. сохранить трассу выполнения программы и произвести символьную интерпретацию этой трассы [27; 40],
2. осуществить символьную интерпретацию во время выполнения программы [25; 105].

Первый подход имеет накладные расходы на сохранение трассы выполнения в постоянной памяти и обработку этой трассы для порождения SMT формул. Второй подход не имеет таких накладных расходов, поэтому в данной работе используется именно он.

Динамическая бинарная инструментация позволяет встраивать анализирующий код перед каждой выполняемой инструкцией программы. Такой инструментирующий код может производить динамическую символьную интерпретацию. Для осуществления инструментации с помощью DynamoRIO разрабатываются специальные расширения в виде динамических библиотек, ко-

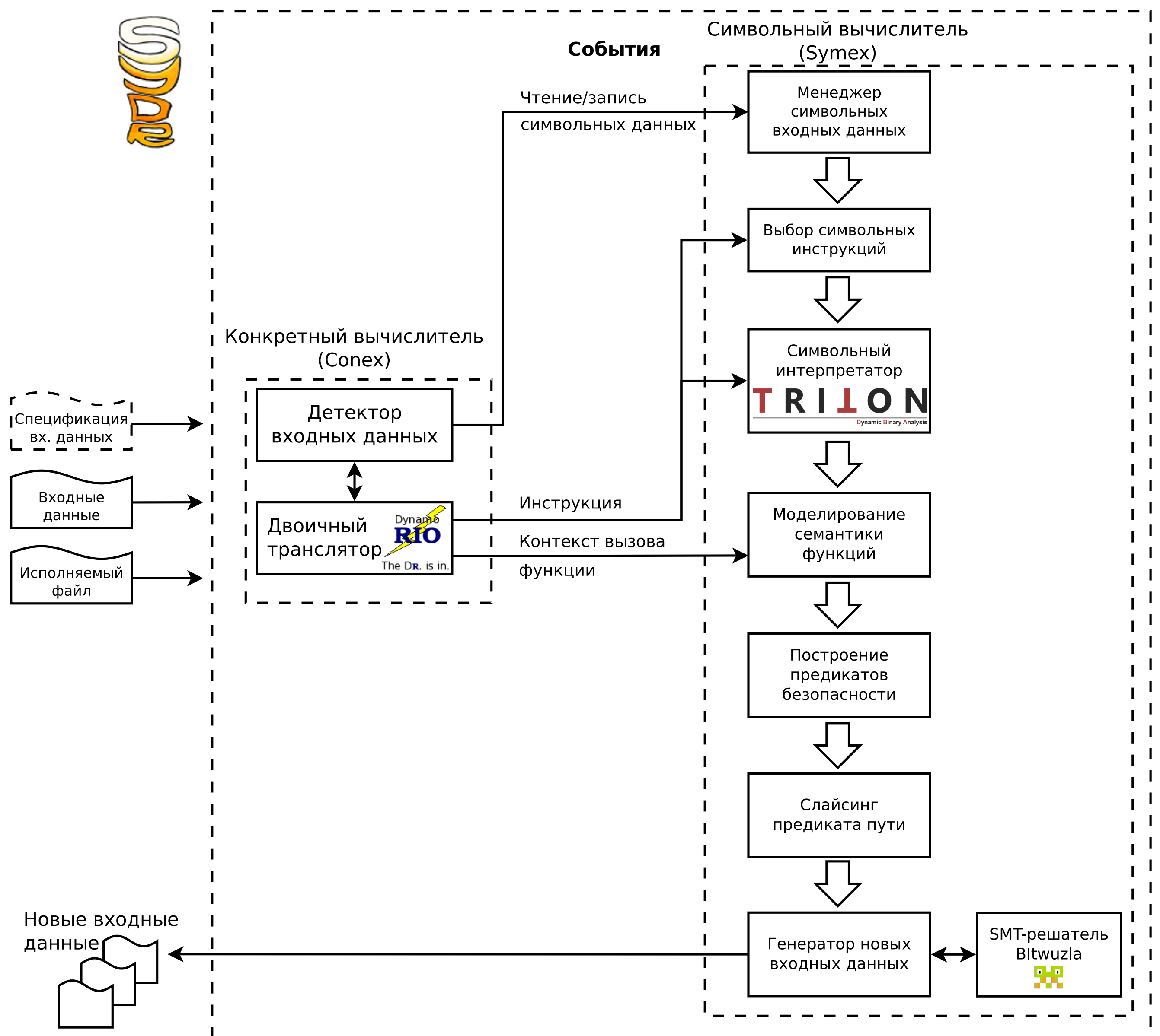

Рисунок 1 — Схема инструмента динамической символьной интерпретации Sydr

торые называются клиентами. Однако такой анализ ограничен 4 Гб оперативной памяти, когда применяется к 32-битным приложениям. Более того, проблемы могут возникнуть, когда инструментирующий код сложный и используют внешние библиотеки. В частности, клиент DynamoRIO аварийно завершается, если он использует библиотеку pthread [106]. Кроме того, выделение динамической памяти в DynamoRIO довольно медленное [107]. Описанные выше проблемы послужили мотивацией для разделения конкретной и символьной интерпретации на два процесса. Такое разделение позволяет уменьшить размер инструментирующего кода. Более того, символьный интерпретатор больше не ограничен 4 Гб оперативной памяти и может использовать произвольные внешние библиотеки.

На рисунке 1 представлена схема инструмента динамической символьной интерпретации Sydr [32]. Sydr осуществляет символьную интерпретацию во время выполнения программы. Конкретная и символьная интерпретация разделены на два процесса, которые обмениваются сообщениями (событиями) при помощи разделяемой памяти. Конкретный вычислитель (Conex) сохраняет описания событий в разделяемой памяти, которые в дальнейшем обрабатываются символьным вычислителем (Symex).

Конкретный вычислитель (Conex) состоит из двух компонентов: детектора входных данных и двоичного транслятора DynamoRIO. Детектор входных данных перехватывает системные вызовы и библиотечные функции, которые обрабатывают пользовательские входные данные (файл, стандартный поток ввода, аргументы командной строки, переменные окружения, сеть). При выполнении соответствующих системных вызовов Conex отправляет события символьному вычислителю (Symex). Событие чтения символьных данных содержит информацию об областях памяти, для которых Symex должен создать новые символьные переменные. Событие записи символьных данных позволяет отслеживать символьные выражения, когда программа сохраняет данные в постоянную память. DynamoRIO реализует динамическую бинарную инструментацию программы и используется детектором входных данных для перехвата системных вызовов и библиотечных функций. Также с помощью DynamoRIO собирается информация о каждой выполненной инструкции: адрес, код операции, явные и неявные операнды и их конкретные значения. Эта информация отправляется Symex для последующей символьной интерпретации инструкции.

Символьный вычислитель (Symex) обрабатывает события от Conex для осуществления динамической символьной интерпретации. Менеджер символьных входных данных отвечает за создание символьных переменных. Он обновляет символьное состояние регистров, памяти и файлов, когда получает событие чтения или записи символьных данных. Менеджер также хранит изначальные конкретные значения байтов, соответствующие символьным переменным. Эти значения будут использоваться во время генерации новых входных данных. Как только Conex детектирует первое чтение символьных входных данных, он начинает отправлять машинные инструкции символьному вычислителю. Symex выбирает из них только те, которые имеют хотя бы один символьный операнд (явный или неявный). Отобранные инструкции символьно интерпретируются при помощи Triton [23]. Пропуск несимвольных инструкций позволяет значитель-

но ускорить символьную интерпретацию [25]. Более того, во время символьной интерпретации применяются различные оптимизации над абстрактными синтаксическими деревьями (AST) символьных выражений [25].

Символьный вычислитель постоянно поддерживает согласованность конкретного и символьного состояний. Если полученные от Conex конкретные значения символьных операндов инструкции не соответствуют их символьному значению, то такие операнды конкретизуются. Такое происходит из-за ограниченного моделирования системного окружения. В частности, системный вызов может перезаписать значения некоторых символьных регистров, а динамическому инструментатору не доступен код ядра операционной системы.

Symex работает в нескольких потоках. Один отвечает за динамическую символьную интерпретацию и построение предиката пути. И произвольное число потоков осуществляют параллельное решение запросов к математическому решателю. Для этого поддерживается очередь, куда попадают задания на инвертирование переходов, когда в предикат пути добавляется новый символьный условный переход, и предикаты безопасности. Как только задание поступает на обработку, применяется алгоритм слайсинга предиката пути для устранения избыточных ограничений. Затем полученный предикат (AST) транслируется в SMT и передается математическому решателю Bitwuzla [35]. Если SMT-решатель возвращает модель для запроса, то байты из модели подменяются в изначальных входных данных. В результате, генерируются новые входные данные для открытия новых путей выполнения или воспроизведения ошибок, обнаруженных при помощи предикатов безопасности.

5.1.1 Реализация слайсинга предиката пути

Как было сказано выше, алгоритм слайсинга предиката пути применяется непосредственно перед формированием запроса к математическому решателю.

Для эффективной работы с множествами символьных переменных используется контейнер `std::set<size_t>`. На практике он показал большую производительность, чем `boost::dynamic_bitset`, который представляет из себя вектор битов. Дело в том, что в реальных программах большинство ограничений зависят от малого числа символьных переменных. Поэтому эффективнее

хранить набор номеров небольшого числа переменных, чем битовый вектор, который неизбежно будет содержать огромное число нулей для индексов независимых переменных. Также был реализован эффективный алгоритм, который позволяет определить, пересекаются ли два упорядоченных множества.

С целью ускорения алгоритма слайсинга используется кэширование используемых символьных переменных. Предварительно вычисляются переменные, которые использует каждое ограничение в предикате пути. Более того, во время вычисления используемых ограничением переменных (обхода AST в ширину) запоминаются символьные переменные его подвыражений. Сохраненные переменные в дальнейшем повторно используются при вычислении переменных для других ограничений.

5.1.2 Реализация моделирования семантики функций

Моделируемые функции стандартной библиотеки перехватываются при помощи DynamoRIO. Для каждого вызова такой функции Conex отправляет Symex событие, которое содержит имя функции, ее аргументы и возвращаемое значение. Символьный вычислитель применяет соответствующую семантику путем конструирования новых символьных выражений и обновления символьного состояния Triton.

Более того, символьный вычислитель обновляет теневую память при получении событий для функций выделения динамической памяти. А теневой стек обновляется при получении инструкций `call` и `ret`.

Отдельно стоит отметить, что чтение целых чисел с помощью `std::cin` обрабатывается аналогично семантике функции преобразования строки в число `strto*l`. Получение указателя строки для `std::cin` требует дополнительных усилий на стороне конкретного вычислителя. Указатель извлекается из структуры для аргумента-итератора.

5.1.3 Реализация предикатов безопасности

При помощи Triton осуществляется построение символьных выражений для предикатов безопасности во время символьной интерпретации инструкции источника ошибки. В момент достижения инструкции стока ошибки соответствующие предикаты безопасности добавляются в очередь заданий. А непосредственно перед формированием запроса к математическому решателю к предикатами безопасности применяется алгоритм слайсинга предиката пути.

Чтобы пропустить проверку предикатов безопасности для уже обнаруженных ранее ошибок используется быстрый алгоритм хеширования xxHash [108]. Для этого обновляется битовая карта, содержащая найденные ранее ошибки, для которых не надо повторно проверять предикаты безопасности. Индекс битовой карты получается при помощи хеширующей функции от типа ошибки, а также адресов источника и стока ошибки. Более того, сохраняется индекс ограничения в предикате пути, начиная с которого предикат безопасности невыполним. Таким образом, можно пропустить проверку предикатов безопасности с большими индексами, т. к. они тоже будут невыполнимы.

5.2 Реализация метода автоматизированного поиска ошибок при помощи символьных предикатов безопасности в контексте гибридного фаззинга

Предложенный метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности реализован в виде программного инструмента [33]. Гибридный фаззинг осуществляется с использованием фаззера libFuzzer [13] (или AFL++ [14]) и инструмента динамической символьной интерпретации Sydr [25; 32], который помогает фаззеру открывать сложные пути выполнения. После фаззинга производится минимизация полученного корпуса входных файлов с использованием штатных средств фаззера. Затем на полученном корпусе запускается проверка предикатов безопасности. Верификация сгенерированных входных файлов производится на исполняемом файле, собранном с санитайзерами. Sydr возвращает имя модуля и смещение для источника

ошибки. Поэтому используется утилита addr2line [109] для получения имени файла и номера строки, которые ищутся в выводе санитайзеров.

Апробация разработанного метода производилась путем поиска ошибок в проектах с открытым исходным кодом [110]. Часть проектов были взяты из репозитория Google OSS-Fuzz [111] и адаптированы для гибридного фаззинга. В то же время другая часть проектов была подготовлена для фаззинга самостоятельно. Для проектов подготавливались фаззинг-цели, сборочное окружение и скрипты, начальные корпуса входных файлов и словари. Все фаззинг-цели компилировались с отладочной информацией, чтобы можно было провести верификацию срабатываний предикатов безопасности. Более того, каждая цель компилировалась в двух версиях: с санитайзерами для фаззера и без санитайзеров для Sydr.

Заключение

Основные результаты работы заключаются в следующем.

1. Разработан алгоритм слайсинга предиката пути, который позволяет устранять избыточные ограничения во время динамической символьной интерпретации. Для предложенного алгоритма были доказаны теоремы о его конечности и корректности. Была произведена формальная оценка асимптотической сложности алгоритма. Проведена экспериментальная оценка эффективности алгоритма, которая показала повышение скорости и точности инвертирования условных переходов.
2. Разработан метод построения предикатов безопасности для обнаружения ошибок деления на нуль, выхода за границу массива и целочисленного переполнения при помощи динамической символьной интерпретации. Экспериментальная оценка метода на наборе тестов Juliet [26] показала общую точность 95.59 % для 11 классов ошибок CWE [1] (15772 теста).
3. Разработан метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности после гибридного фаззинга. Предложенный метод позволил обнаружить 17 новых ошибок в 10 различных проектах с открытым исходным кодом.
4. Предложенные методы были реализованы в программных системах [32; 33], которые используются в Центре доверенного искусственного интеллекта ИСП РАН.

Список литературы

1. CWE - Common Weakness Enumeration [Electronic Resource]. — URL: https://cwe.mitre.org/.
2. 2022 Open Source Security and Risk Analysis Report [Electronic Resource]. — 2022. — URL: https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/rep-ossra-2022.pdf.
3. Статистика уязвимостей (CVE) по годам [Электронный ресурс]. — URL: https://www.cvedetails.com/browse-by-date.php.
4. Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses / D. Halperin [et al.] // 2008 IEEE Symposium on Security and Privacy (sp 2008). — IEEE. 2008. — P. 129—142. — DOI: 10.1109/SP.2008.31.
5. *Spengler*, *B.* PaX: The Guaranteed End of Arbitrary Code Execution [Electronic Resource] / B. Spengler. — URL: https://grsecurity.net/PaX-presentation.pdf.
6. *Howard*, *M.* The security development lifecycle. Vol. 8 / M. Howard, S. Lipner. — Microsoft Press Redmond, 2006. — URL: http://msdn.microsoft.com/en-us/library/ms995349.aspx.
7. ISO/IEC 15408-3:2008: Information technology – Security techniques – Evaluation criteria for IT security – Part 3: Security assurance components. — ISO Geneva, Switzerland, 2008. — URL: https://www.iso.org/standard/46413.html.
8. ГОСТ Р 56939-2016: Защита информации. Разработка безопасного программного обеспечения. Общие требования. — Национальный стандарт РФ, 2016.
9. CodeSurfer/x86—A Platform for Analyzing x86 Executables / G. Balakrishnan [et al.] // Compiler Construction. — Springer Berlin Heidelberg, 2005. — P. 250—254. — DOI: 10.1007/978-3-540-31985-6_19.
10. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World / A. Bessey [et al.] // Communications of the ACM. — 2010. — Vol. 53, no. 2. — P. 66—75. — DOI: 10.1145/1646353.1646374.

11. Static analyzer Svace for finding defects in a source program code / V. P. Ivannikov [et al.] // Programming and Computer Software. — 2014. — Vol. 40, no. 5. — P. 265—275. — DOI: 10.1134/S0361768814050041.
12. SoK: Sanitizing for Security / D. Song [et al.] // 2019 IEEE Symposium on Security and Privacy (SP). — 2019. — P. 1275—1295. — DOI: 10.1109/SP.2019.00010.
13. *Serebryany*, *K.* Continuous Fuzzing with libFuzzer and AddressSanitizer / K. Serebryany // 2016 IEEE Cybersecurity Development (SecDev). — IEEE. 2016. — P. 157. — DOI: 10.1109/SecDev.2016.043.
14. AFL++: Combining Incremental Steps of Fuzzing Research / A. Fioraldi [et al.] // 14th USENIX Workshop on Offensive Technologies (WOOT 20). — 2020. — URL: https://www.usenix.org/system/files/woot20-paper-fioraldi.pdf.
15. *Pak*, *B. S.* Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution : Master's thesis / Pak Brian S. — School of Computer Science Carnegie Mellon University, 2012.
16. Driller: Augmenting Fuzzing Through Selective Symbolic Execution / N. Stephens [et al.] // NDSS. Vol. 16. — 2016. — P. 1—16.
17. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing / I. Yun [et al.] // 27th USENIX Security Symposium. — 2018. — P. 745—761. — URL: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-yun.pdf.
18. *Poeplau*, *S.* Symbolic execution with SymCC: Don't interpret, compile! / S. Poeplau, A. Francillon // 29th USENIX Security Symposium (USENIX Security 20). — 2020. — P. 181—198. — URL: https://www.usenix.org/system/files/sec20-poeplau.pdf.
19. *Poeplau*, *S.* SymQEMU: Compilation-based symbolic execution for binaries / S. Poeplau, A. Francillon // Proceedings of the 2021 Network and Distributed System Security Symposium. — 2021. — DOI: 10.14722/ndss.2021.23118.
20. *Borzacchiello*, *L.* FUZZOLIC: Mixing fuzzing and concolic execution / L. Borzacchiello, E. Coppa, C. Demetrescu // Computers & Security. — 2021. — Vol. 108. — P. 102368. — DOI: 10.1016/j.cose.2021.102368.

21. *King*, *J. C.* Symbolic Execution and Program Testing / J. C. King // Communications of the ACM. — 1976. — Vol. 19, no. 7. — P. 385—394. — DOI: 10.1145/360248.360252.

22. *Kuts*, *D.* Towards Symbolic Pointers Reasoning in Dynamic Symbolic Execution / D. Kuts // 2021 Ivannikov Memorial Workshop (IVMEM). — IEEE. 2021. — P. 42—49. — DOI: 10.1109/IVMEM53963.2021.00014.

23. *Saudel*, *F.* Triton : A Dynamic Symbolic Execution Framework / F. Saudel, J. Salwan // Symposium sur la sécurité des technologies de l'information et des communications. — 2015. — P. 31-54. — (SSTIC). — URL : https://triton.quarkslab.com/files/sstic2015_slide_en_saudel_salwan.pdf.

24. A Survey of Symbolic Execution Techniques / R. Baldoni [et al.] // ACM Computing Surveys. — 2018. — Vol. 51, no. 3. — DOI: 10.1145/3182657.

25. Sydr: Cutting Edge Dynamic Symbolic Execution / A. Vishnyakov [et al.] // 2020 Ivannikov ISPRAS Open Conference (ISPRAS). — IEEE, 2020. — P. 46—54. — DOI: 10.1109/ISPRAS51486.2020.00014.

26. Software Assurance Reference Dataset - Juliet Test Suite [Electronic Resource]. — URL: https://samate.nist.gov/SRD/testsuite.php.

27. Оценка критичности программных дефектов в условиях работы современных защитных механизмов / А. Н. Федотов, В. А. Падарян, В. В. Каушан, Ш. Ф. Курмангалеев, А. В. Вишняков, А. Р. Нурмухаметов // Труды института системного программирования РАН. — 2016. — Т. 28, № 5. — С. 73—92. — DOI: 10.15514/ISPRAS-2016-28(5)-4.

28. *Вишняков*, *А. В.* Поиск ошибок в бинарном коде методами динамической символьной интерпретации / А. В. Вишняков, И. А. Кобрин, А. Н. Федотов // Труды института системного программирования РАН. — 2022. — Т. 34, № 2. — С. 25—42. — DOI: 10.15514/ISPRAS-2022-34(2)-3.

29. Symbolic Security Predicates: Hunt Program Weaknesses / A. Vishnyakov [et al.] // 2021 Ivannikov ISPRAS Open Conference (ISPRAS). — IEEE, 2021. — P. 76—85. — DOI: 10.1109/ISPRAS53967.2021.00016.

30. *Вишняков*, *А. В.* Символьные предикаты безопасности в гибридном фаззинге / А. В. Вишняков, И. А. Кобрин, А. Н. Федотов // Материалы 31-й научно-технической конференции «Методы и технические средства обеспечения безопасности информации» (МиТСОБИ). — 2022. — С. 74—75.

31. *Кобрин*, *И. А.* Гибридный фаззинг фреймворка машинного обучения TensorFlow / И. А. Кобрин, А. В. Вишняков, А. Н. Федотов // Материалы 31-й научно-технической конференции «Методы и технические средства обеспечения безопасности информации» (МиТСОБИ). — 2022. — С. 90—91.
32. *Свидетельство о гос. регистрации программы для ЭВМ*. Инструмент динамической символьной интерпретации «Sydr» / 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).
33. *Свидетельство о гос. регистрации программы для ЭВМ*. Sydr-fuzz / 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).
34. *Moura*, *L. de*. Z3: An Efficient SMT Solver / L. de Moura, N. Bjørner // Tools and Algorithms for the Construction and Analysis of Systems. — Springer Berlin Heidelberg, 2008. — P. 337—340. — DOI: 10.1007/978-3-540-78800-3_24.
35. *Niemetz*, *A.* Bitwuzla at the SMT-COMP 2020 / A. Niemetz, M. Preiner // CoRR. — 2020. — Vol. abs/2006.01621. — arXiv: 2006.01621. — URL: https://arxiv.org/abs/2006.01621.
36. *Godefroid*, *P.* DART: Directed Automated Random Testing / P. Godefroid, N. Klarlund, K. Sen // Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. — ACM, 2005. — P. 213—223. — (PLDI ’05). — DOI: 10.1145/1065010.1065036.
37. *Sen*, *K.* CUTE: A Concolic Unit Testing Engine for C / K. Sen, D. Marinov, G. Agha // SIGSOFT Software Engineering Notes. — 2005. — Vol. 30, no. 5. — P. 263—272. — DOI: 10.1145/1095430.1081750.
38. EXE: Automatically Generating Inputs of Death / C. Cadar [et al.] // Proceedings of the 13th ACM Conference on Computer and Communications Security. — ACM, 2006. — P. 322—335. — (CCS ’06). — DOI: 10.1145/1180405.1180445.

39. *Cadar*, *C.* KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs / C. Cadar, D. Dunbar, D. R. Engler // OSDI. Vol. 8. — 2008. — P. 209—224. — URL: https://static.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

40. *Godefroid*, *P.* Automated Whitebox Fuzz Testing / P. Godefroid, M. Y. Levin, D. A. Molnar // NDSS. Vol. 8. — 2008. — P. 151—166. — URL: https://www.microsoft.com/en-us/research/publication/automated-whitebox-fuzz-testing/.

41. *Schwartz*, *E. J.* All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask) / E. J. Schwartz, T. Avgerinos, D. Brumley // 2010 IEEE Symposium on Security and Privacy. — 2010. — P. 317—331. — DOI: 10.1109/SP.2010.26.

42. *Barrett*, *C.* The SMT-LIB Standard: Version 2.6 [Electronic Resource] / C. Barrett, P. Fontaine, C. Tinelli. — 2017. — URL: www.SMT-LIB.org.

43. AEG: Automatic Exploit Generation / T. Avgerinos [et al.] // Network and Distributed System Security Symposium. — 2011. — P. 283—300. — URL: http://security.ece.cmu.edu/aeg/aeg-current.pdf.

44. Unleashing Mayhem on Binary Code / S. K. Cha [et al.] // Proceedings of the 2012 IEEE Symposium on Security and Privacy. — IEEE Computer Society, 2012. — P. 380—394. — (SP ’12). — DOI: 10.1109/SP.2012.31.

45. *Zalewski*, *M.* AFL: American Fuzzy Lop technical ”whitepaper” [Electronic Resource] / M. Zalewski. — URL: https://lcamtuf.coredump.cx/afl/technical_details.txt.

46. FuzzBench (Google). DSE+Fuzzing Experiment Report [Electronic Resource]. — 2021. — URL: https://www.fuzzbench.com/reports/experimental/2021-07-03-symbolic/index.html.

47. *Lattner*, *C.* LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation / C. Lattner, V. Adve // Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO’04). Vol. 4. — 2004. — P. 75. — DOI: 10.1109/CGO.2004.1281665.

48. *Chipounov*, *V.* S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems / V. Chipounov, V. Kuznetsov, G. Candea // SIGPLAN Notices. — 2011. — Vol. 46, no. 3. — P. 265—278. — DOI: 10.1145/1961296.1950396.

49. *Chipounov*, *V.* The S2E Platform: Design, Implementation, and Applications / V. Chipounov, V. Kuznetsov, G. Candea // ACM Transactions on Computer Systems (TOCS). — 2012. — Vol. 30, no. 1. — P. 1—49. — DOI: 10.1145/2110356.2110358.

50. *Bellard*, *F.* QEMU, a fast and portable dynamic translator / F. Bellard // ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference. — USENIX Association, 2005. — P. 41.

51. 2016 DARPA Cyber Grand Challenge [Electronic Resource]. — URL: https://en.wikipedia.org/wiki/2016_Cyber_Grand_Challenge.

52. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation / C.-K. Luk [et al.] // SIGPLAN Notices. — 2005. — Vol. 40, no. 6. — P. 190—200. — DOI: 10.1145/1064978.1065034.

53. Compilers: Principles, Techniques, and Tools / A. V. Aho [et al.] //. — 2nd Edition. — Pearson Addison Wesley, 2007. — Chap. Static Single-Assignment Form. P. 369—370.

54. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis / Y. Shoshitaishvili [et al.] // 2016 IEEE Symposium on Security and Privacy (SP). — 2016. — P. 138—157. — DOI: 10.1109/SP.2016.17.

55. *Nethercote*, *N.* Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation / N. Nethercote, J. Seward // SIGPLAN Not. — 2007. — Vol. 42, no. 6. — P. 89—100. — DOI: 10.1145/1273442.1250746.

56. Claripy: An abstraction layer for constraint solvers [Electronic Resource]. — URL: https://github.com/angr/claripy.

57. *Borzacchiello*, *L.* Fuzzing Symbolic Expressions / L. Borzacchiello, E. Coppa, C. Demetrescu // Proceedings of the 43rd International Conference on Software Engineering. — 2021. — (ICSE '21). — DOI: 10.1109/ICSE43902.2021.00071.

58. DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation / M. G. Kang [et al.] // Proceedings of the Network and Distributed System Security Symposium. — 2011. — (NDSS '11).

59. *Balakrishnan*, *G.* Analyzing Memory Accesses in x86 Executables / G. Balakrishnan, T. Reps // International conference on compiler construction. — Springer Berlin Heidelberg, 2004. — P. 5—23. — DOI: 10.1007/978-3-540-24723-4_2.

60. LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating / X. Mi [et al.] // 24th International Symposium on Research in Attacks, Intrusions and Defenses (RAID ’21). — 2021. — P. 62—77. — DOI: 10.1145/3471621.3471852.

61. µclibc. A C library for embedded Linux [Electronic Resource]. — URL: https://uclibc.org/.

62. libdft64 [Electronic Resource]. — URL: https://github.com/vusec/vuzzer64.

63. Google Sanitizers [Electronic Resource]. — URL: https://github.com/google/sanitizers.

64. ParmeSan: Sanitizer-guided Greybox Fuzzing / S. Österlund [et al.] // 29th USENIX Security Symposium (USENIX Security 20). — 2020. — P. 2289—2306. — URL: https://www.usenix.org/system/files/sec20-osterlund.pdf.

65. AddressSanitizer: A Fast Address Sanity Checker / K. Serebryany [et al.] // 2012 USENIX Annual Technical Conference (USENIX ATC 12). — 2012. — P. 309—318. — URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

66. *Serebryany*, *K.* ThreadSanitizer: Data Race Detection in Practice / K. Serebryany, T. Iskhodzhanov // Proceedings of the Workshop on Binary Instrumentation and Applications. — 2009. — P. 62—71. — (WBIA ’09). — DOI: 10.1145/1791194.1791203.

67. *Stepanov*, *E.* MemorySanitizer: Fast detector of uninitialized memory use in C++ / E. Stepanov, K. Serebryany // 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). — 2015. — P. 46—55. — DOI: 10.1109/CGO.2015.7054186.

68. SAVIOR: Towards Bug-Driven Hybrid Testing / Y. Chen [et al.] // 2020 IEEE Symposium on Security and Privacy (SP). — 2020. — P. 1580—1596. — DOI: 10.1109/SP40000.2020.00002.

69. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution / T. Wang [et al.] // NDSS. — 2009.

70. *Demidov*, *R.* Integer Overflow Vulnerabilities Detection in Software Binary Code / R. Demidov, A. Pechenkin, P. Zegzhda // Proceedings of the 10th International Conference on Security of Information and Networks. — ACM, 2017. — P. 101—106. — (SIN ’17). — DOI: 10.1145/3136825.3136872.

71. SMT-COMP 2021 QF_BV Benchmark for cjpeg under Sydr [Electronic Resource]. — URL: https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_BV/-/blob/master/20210219-Sydr/master/cjpeg/predicate_2716.smt2.

72. Sydr benchmark [Electronic Resource]. — URL: https://github.com/ispras/sydr-benchmark.

73. *Bruening*, *D.* Efficient, Transparent, and Comprehensive Runtime Code Manipulation : PhD thesis / Bruening Derek. — Massachusetts Institute of Technology, Department of Electrical Engineering, Computer Science, 2004. — URL: https://www.burningcutlery.com/derek/docs/phd.pdf.

74. 2022 CWE Top 25 Most Dangerous Software Weaknesses [Electronic Resource]. — URL: https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25.html.

75. CWE-123: Write-what-where Condition [Electronic Resource]. — URL: https://cwe.mitre.org/data/definitions/123.html.

76. *Weiser*, *M.* Program Slicing / M. Weiser // IEEE Transactions on Software Engineering. — 1984. — Vol. SE—10, no. 4. — P. 352—357. — DOI: 10.1109/TSE.1984.5010248.

77. Juliet C/C++ Dynamic Test Suite [Electronic Resource]. — URL: https://github.com/ispras/juliet-dynamic.

78. 5 ошибок целочисленного переполнения в FreeImage [Электронный ресурс]. — URL: https://sourceforge.net/p/freeimage/bugs/347/.

79. Goblin: Fix substract with overflow for header.size [Electronic Resource]. — URL: https://github.com/m4b/goblin/pull/333.

80. OpenJPEG: Integer Overflow in openjpeg/openjp2/image.c [Electronic Resource]. — URL: https://github.com/opencv/opencv/issues/22284.

81. Poppler: Integer overflow in readPatternDictSeg [Electronic Resource]. — URL: https://gitlab.freedesktop.org/poppler/poppler/-/issues/1268.

82. Целочисленное переполнение в Rizin, приводящее к выходу за границы массива [Электронный ресурс]. — URL: https://github.com/rizinorg/rizin/issues/2935.
83. Целочисленное переполнение в xlnt [Электронный ресурс]. — URL: https://github.com/tfussell/xlnt/issues/616.
84. Целочисленное переполнение в xlnt, приводящее к выходу за границы массива [Электронный ресурс]. — URL: https://github.com/tfussell/xlnt/issues/626.
85. Выход за границы массива в xlnt [Электронный ресурс]. — URL: https://github.com/tfussell/xlnt/issues/595.
86. Исправление ошибки целочисленного переполнения в unbound [Электронный ресурс]. — URL: https://github.com/NLnetLabs/unbound/issues/637.
87. Деление на нуль в hdp [Электронный ресурс]. — URL: https://bugs.launchpad.net/ubuntu/+source/libhdf4/+bug/1915417.
88. Исправление ошибки целочисленного переполнения в miniz [Электронный ресурс]. — URL: https://github.com/richgel999/miniz/pull/238.
89. EXRS: Multiply with overflow [Electronic Resource]. — URL: https://github.com/johannesvollmer/exrs/issues/165.
90. Goblin: Subtruct with overflow [Electronic Resource]. — URL: https://github.com/m4b/goblin/issues/331.
91. FreeImage project [Electronic Resource]. — URL: https://freeimage.sourceforge.io/.
92. xlnt project [Electronic Resource]. — URL: https://github.com/tfussell/xlnt.
93. unbound project [Electronic Resource]. — URL: https://github.com/NLnetLabs/unbound.
94. HDF4 library [Electronic Resource]. — URL: https://support.hdfgroup.org/products/hdf4/whatishdf.html.
95. miniz library [Electronic Resource]. — URL: https://github.com/richgel999/miniz.
96. PyTorch project [Electronic Resource]. — URL: https://github.com/pytorch/pytorch.
97. EXRS: 100% Safe Rust OpenEXR file library [Electronic Resource]. — URL: https://github.com/johannesvollmer/exrs.

98. image-rs: Encoding and decoding images in Rust [Electronic Resource]. — URL: https://github.com/image-rs/image.
99. Firefox Quantum engine [Electronic Resource]. — URL: https://www.mozilla.org/en-US/firefox/browsers/quantum/.
100. OpenJPEG [Electronic Resource]. — URL: https://github.com/uclouvain/openjpeg.
101. OpenCV: Open Source Computer Vision Library [Electronic Resource]. — URL: https://github.com/opencv/opencv.
102. Poppler, a PDF rendering library [Electronic Resource]. — URL: https://gitlab.freedesktop.org/poppler/poppler.
103. Rizin reverse engineering framework [Electronic Resource]. — URL: https://github.com/rizinorg/rizin.
104. TensorFlow: An Open Source Machine Learning Framework for Everyone [Electronic Resource]. — URL: https://github.com/tensorflow/tensorflow.
105. *Molnar*, *D. A.* Catchconv: Symbolic execution and run-time type inference for integer conversion errors : tech. rep. / D. A. Molnar, D. Wagner ; UC Berkeley EECS. — 2007. — UCB/EECS-2007—23. — URL: https://digitalassets.lib.berkeley.edu/techreports/ucb/text/EECS-2007-23.pdf.
106. *Bruening*, *D.* Issue for DynamoRIO libpthread support [Electronic Resource] / D. Bruening. — URL: https://github.com/DynamoRIO/dynamorio/issues/2848.
107. *Bruening*, *D.* Issue for DynamoRIO heap slowdowns [Electronic Resource] / D. Bruening. — URL: https://github.com/DynamoRIO/dynamorio/issues/2115.
108. xxHash - Extremely fast non-cryptographic hash algorithm [Electronic Resource]. — URL: http://www.xxhash.com/.
109. addr2line - convert addresses into file names and line numbers [Electronic Resource]. — URL: https://linux.die.net/man/1/addr2line.
110. OSS-Sydr-Fuzz: Hybrid Fuzzing for Open Source Software [Electronic Resource]. — URL: https://github.com/ispras/oss-sydr-fuzz.
111. OSS-Fuzz: Continuous Fuzzing for Open Source Software [Electronic Resource]. — URL: https://github.com/google/oss-fuzz.

Список рисунков

1 Схема инструмента динамической символьной интерпретации Sydr . . 113

Список таблиц

1 Сравнение методов моделирования семантики функций в различных инструментах . . . 39

2 Пример построения предиката пути и инвертирования перехода . . . 53

3 Применение алгоритма слайсинга предиката пути к примеру на листинге 2.1 . . . 59

4 Результаты инвертирования переходов без применения слайсинга предиката пути . . . 68

5 Результаты инвертирования переходов с применением слайсинга предиката пути . . . 69

6 Сравнение точности и скорости инвертирования переходов с применением и без применения слайсинга предиката пути . . . 70

7 Сравнение построения предиката пути с моделированием и без моделирования семантики функций . . . 82

8 Сравнение результатов инвертирования переходов с моделированием и без моделирования семантики функций . . . 83

9 Результаты тестирования на Juliet . . . 98