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»

Автореферат
диссертации на соискание учёной степени
кандидата физико-математических наук

Москва — 2022

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

Научный руководитель: **Федотов Андрей Николаевич,**
кандидат технических наук

Официальные оппоненты: **Ильин Вячеслав Анатольевич,**
доктор физико-математических наук,
главный научный сотрудник Курчатовского комплекса НБИКС-природоподобных технологий Федерального государственного бюджетного учреждения «Национальный исследовательский центр «Курчатовский институт»

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

Ведущая организация: Федеральное государственное учреждение «Федеральный исследовательский центр «Информатика и управление» Российской академии наук»

Защита состоится 15 декабря 2022 г. в 14 часов на заседании диссертационного совета 24.1.120.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В. П. Иванникова Российской Академии Наук по адресу: 109004, г. Москва, ул. А. Солженицына, д. 25.

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

Автореферат разослан «___»_____ 2022 года.

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

Общая характеристика работы

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

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. 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.

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, 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. 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 и динамического анализа программ. Такие инструменты могут исследовать программу как на уровне исходного кода, так и машинного.

Во время безопасного цикла разработки ПО непрерывно применяется фаззинг-тестирование. 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. 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 помимо обратной связи по покрытию применяют методы динамической символьной интерпретации (DSE, dynamic symbolic execution, где execution по сути является интерпретацией программы), 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.

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 [1; 2], 2 — в периодических научных журналах, индексируемых Web of Science и Scopus [3; 4], 2 — в тезисах докладов [5; 6]. Зарегистрированы 2 программы для ЭВМ [7; 8]. В работе [1] 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 [4] автором представлен метод моделирования семантики функций. В статье [3] автором предложен разработанный алгоритм слайсинга предиката пути, устраняющий избыточные ограничения, и проведена его экспериментальная оценка. В статье [4] 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 [2] 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 [5] автором описывается практическое применения автоматизированного метода поиска ошибок к проектам с открытым исходным кодом. В докладе [6] 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 [7], в которой автором реализован алгоритм слайсинга предиката пути, а также методы моделирования семантики функций и построения предикатов безопасности. Зарегистрирована программа для ЭВМ [8], 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 объем диссертации составляет 131 страницу, включая 1 рисунок и 9 таблиц. Список литературы содержит 111 наименований.

Содержание работы

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

Первая глава посвящена обзору работ по теме диссертации. В главе рассмотрены существующие инструменты символьной интерпретации. Приводится обзор методов преодоления избыточной и недостаточной помеченности, когда предикат пути содержит чрезмерное количество ограничений или же, наоборот, часть необходимых ограничений отсутствуют. Анализируются методы поиска ошибок с помощью символьной интерпретации.

В разделе 1.1 описывается метод динамической символьной интерпретации программ. Раздел 1.2 содержит обзор существующих инструментов символьной интерпретации. В 2010 году Schwartz и др. формализовали динамическую символьную интерпретацию, которая исследует вариацию изначальных входных данных на некотором фиксированном пути выполнения. Изначально каждый байт входных данных моделируется свободной символьной переменной. Каждая инструкция моделируется SMT 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 этих ограничений являются входные данные, которые проведут программу по тому же пути выполнения. Для инвертирования некоторого перехода необходимо взять его ограничение с логическим отрицанием. Так можно открывать новые пути выполнения, ответвляясь от изначального.

В 2016 году Stephens и др. реализовали подход гибридного фаззинга в инструменте Driller. В качестве фаззера использовался AFL 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. показали, что динамическая символьная интерпретация может быть достаточно быстрой для одновременной

работы фаззера и DSE. Гибридный фаззинг в инструменте QSYM оказался эффективнее, чем обычный фаззинг с обратной связью по покрытию. Дальнейшие работы Poeplau, Borzacchiello и др. 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++, открывают новое покрытие кода быстрее двух фаззеров.

В разделе 1.3 описываются существующие методы преодоления избыточной и недостаточной помеченности. Помеченными считаются данные, которые зависят от пользовательского ввода. Недостаточная помеченность возникает, когда значение, вычисление которого зависит от входных данных, не помечено. В итоге символьная модель теряет точность, и полученное решение может не провести программу по тому же пути выполнения. Избыточная помеченность — это ситуация, когда в предикат пути попадает чрезмерное число ограничений. Однако эти ограничения могут совсем не оказывать влияния на генерацию новых входных данных, а только замедлять время решения предиката пути. Наиболее опасно, когда такие избыточные ограничения приводят к невыполнимости запросов к математическому решателю.

KLEE (как и ранее EXE) разбивает ограничения в предикате пути на независимые подмножества. Два ограничения считаются независимыми, если ни одна символьная переменная из одного ограничения не содержится во втором. Таким образом, KLEE устраняет нерелевантные для запроса ограничения. Например, для запроса $i = 20$ требуются только первые два ограничения из предиката пути $\{i < j, j < 20, k > 0\}$. 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. Вершинами графа являются символьные переменные, а ребра отражают факт, что символьные переменные содержатся вместе в некотором ограничении.

Многие инструменты символьной интерпретации пропускают интерпретацию функций со специфицированной семантикой и моделируют их. Так они уменьшают число избыточных ограничений в предикате пути. Моделирование семантики функций позволяет описать сразу несколько символьных состояний одной формулой и ускорить символьную интерпретацию. Для функции `memcmp` SymCC добавляет в предикат пути условный переход с двумя ветвями: строки различаются или же полностью совпадают. При вызове библиотечной функции, которая обрабатывает символьные данные, S^2E передает управление на специальную функцию-обработчик, которая подставляет в возвращаемое значение символьную формулу. 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.

На основе проведенного обзора можно сделать следующие выводы. Проблемы недостаточной и избыточной помеченности 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.

Наиболее подходящим для повышения точности решения задачи поиска ошибок является метод устранения избыточных ограничений, используемый в KLEE. 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.

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. В данной работе необходимо разработать семантические модели для возвращаемого значения функций, чтобы повысить вероятность обнаружения ошибок.

В разделе 1.4 проводится обзор методов поиска ошибок с помощью символьной интерпретации. KLEE 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.

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, 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.

SAVIOR — это фреймворк для гибридного фаззинга программного обеспечения, нацеленный на поиск ошибок. Символьная интерпретация в SAVIOR осуществляется с помощью KLEE. Для этого авторы модифицировали код KLEE так, чтобы он поддерживал динамическую символьную интерпретацию. 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 в дальнейшем. 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.

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 пользовательских данных. Сток ошибки — место в программе, в котором использование переполненного значения может привести к дальнейшим программным уязвимостям:

- 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 пользовательского ввода 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. 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.

На основе исследования аналогичных методов можно сделать следующие выводы. KLEE 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.

SAVIOR работает с исходным кодом и использует файлы из корпуса фаззера в качестве входных данных для динамической символьной интерпретации. Однако все встреченные инструкции по-прежнему эмулируются через KLEE, что влечет за собой понижение скорости (сопоставимое с QSYM) 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, что не является переносимым решением. Авторы отмечают проблемы с анализом C++ кода.

IntScope 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. 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.

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
```

Вторая глава посвящена описанию предлагаемого **алгоритма слайсинга предиката пути**, который позволяет устранять избыточные ограничения из предиката пути. В разделе 2.1 подробно описывается метод построения предиката пути во время динамической символьной интерпретации. А в разделе 2.2 описывается сам предложенный алгоритм слайсинга предиката пути.

Алгоритм 1 слайсинга предиката пути [3] 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 Π 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 Π. 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 Π. Ограничения из Π, используемые переменные которых пересекаются с $vars$,

составляются в конъюнкцию 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.

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.

В разделе 2.3 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. Доказательства теорем представлены в тексте диссертации. 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.*

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 $\Pi \wedge cond$.

Таблица 1 — 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

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.*

В разделе 2.4 описываются конкретные примеры того, как алгоритм слайсинга предиката пути позволяет преодолевать избыточную и недостаточную помеченность.

В разделе 2.5 приводится экспериментальная оценка эффективности предложенного алгоритма. Во время эксперимента производится динамическая символьная интерпретация набора программ с последовательным инвертированием встреченных условных переходов. Суммарное время анализа каждой программы ограничено 2 часами. Сгенерированные входные данные для инвертирования условных переходов проверяются на корректность. Входные данные считаются корректными, когда они проводят программу по тому же пути выполнения за исключением целевого перехода в инвертированном направлении. Для большей части анализируемых приложений значительно выросло число корректно инвертированных переходов (с учетом пути выполнения).

В таблице 1 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 1. 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. 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.

В итоге, алгоритм слайсинга предиката пути нигде не показал худший результат по сравнению с решением полного предиката пути. Следовательно, для повышения точности и скорости решения всегда необходимо применять алгоритм слайсинга к целевому предикату перед отправкой запроса математическому решателю.

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.

Третья глава посвящена разработанному методу моделирования семантики функций, который позволяет пропускать символьную интерпретацию функций стандартной библиотеки и описывать эффекты их выполнения символьными выражениями [4]. Так моделирование функций дает возможность объединять несколько символьных состояний и исследовать больше путей выполнения программы. Набор функций для моделирования был получен путем анализа результатов динамической символьной интерпретации различных Linux программ. 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.

В разделе 3.1 приводятся разработанные символьные модели функций. В частности, функция `tolower` (аналогично `toupper`) моделируется следующим if-then-else (*ite*) выражением для входного символа *ch*: $ite(ch - \texttt{'A'} < 26, ch + 32, ch)$. Представленная формула разрешает строчные и заглавные символы в символьных входных данных и тем самым предотвращает избыточную

помеченность. Если бы модель для этой функции отсутствовала, то регистр символа был бы ограничен регистром из реального выполнения.

Пропуск интерпретации определенных функций может значительно ускорить динамическую символьную интерпретацию и уменьшить избыточную помеченность [1]. Функции динамического выделения памяти (`malloc` и др.) 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. Поэтому символьную интерпретацию таких функций можно пропустить. Аналогично можно поступить с функциями, которые пишут лог программы в стандартный поток вывода (например, `printf`). Они не представляют интереса для процесса динамической символьной интерпретации и исследования новых путей выполнения программы. Символьная модель операций копирования (`strcpy`, `memcpy` и т. д.) 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.

Оставшиеся функции моделируются символьной формулой возвращаемого значения. 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. Ослабляя ограничения в предикате пути и обобщая символьные состояния, можно открывать больше новых путей выполнения и повышать вероятность срабатывания предикатов безопасности.

Ниже приводится формула для возвращаемого значения функции `memchr(ptr, ch, count)`:

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

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(str, substr)` поиск символа $ptr[k] \neq ch$ заменяется на соответствующее сравнение с подстрокой: $\bigvee str[k+n] \neq substr[n]$.

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

$$lhs[0] - rhs[0] + \sum_{i=1}^{count-1} (lhs[i] - rhs[i]) * ite\left(\bigwedge_{k=0}^{i-1} lhs[k] = rhs[k], 1, 0\right) \quad (2)$$

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 $ite\,(\bigwedge lhs[k] = rhs[k], 1, 0)$ 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.

$$\pm\,(c_n c_{n-1} ... c_1 c_0)_b \longrightarrow x \quad (3)$$

$$\begin{aligned} a_k = ite(c_k \geqslant \texttt{'0'} \wedge\ c_k \leqslant \texttt{'9'} \wedge\ c_k < \texttt{'0'} + b,\\ c_k - \texttt{'0'},\\ ite(c_k \geqslant \texttt{'a'} \wedge\ c_k < \texttt{'a'} + b - 10,\\ c_k - \texttt{'a'} + 10, c_k - \texttt{'A'} + 10)) \end{aligned} \quad (4)$$

$$|x| = \sum_{k=0}^{n} a_k b^k, x = ite(sign = \texttt{'-'}, -|x|, |x|) \quad (5)$$

$$\begin{aligned} (c_k \geqslant \texttt{'0'} \wedge\ c_k \leqslant \texttt{'9'} \wedge\ c_k < \texttt{'0'} + b) \vee\\ (c_k \geqslant \texttt{'a'} \wedge c_k < \texttt{'a'} + b - 10) \vee\\ (c_k \geqslant \texttt{'A'} \wedge c_k < \texttt{'A'} + b - 10) \end{aligned} \quad (6)$$

Чтобы реализовать преобразование (3) строки в число (`strto*l(str, end, base)`) необходимо добавить ограничение (6) 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 (4) 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 (5).

Функции `strto*l` вызываются внутри других функций, таких как `atoi` и `scanf("%d", &x)`. Чтение входных чисел в наборе тестов Juliet 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`.

В разделе 3.2 приводится экспериментальная оценка разработанного метода моделирования семантики функций. Измерения производительности и эффективности осуществлялись аналогично тому, как это делалось для алгоритма 1 слайсинга предиката пути. Моделирование семантики функций повлекло за собой уменьшение времени построения предиката пути и числа обнаруженных символьных условных переходов. Таким образом, за счет уменьшения времени построения предиката пути остается больше времени 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. Кроме того, для большинства приложений либо уменьшилось суммарное время анализа, либо увеличилось количество корректно инвертированных переходов за то же время. В результате, метод моделирования семантики функций ускоряет динамическую символьную интерпретацию и открытие новых путей выполнения за счет устранения избыточной помеченности.

Четвертая глава содержит описание разработанных методов построения предикатов безопасности [4] и автоматизированного поиска ошибок при помощи символьных предикатов безопасности в гибридном фаззинге [2; 5; 6].

В разделе 4.1 представлен разработанный **метод построения предикатов безопасности** для обнаружения ошибок деления на нуль, целочисленного переполнения и выхода за границу массива [4]. 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. 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.

Предикат безопасности для некоторого класса ошибок является булевым предикатом, который истинен, когда выполнение инструкции (или функции) приводит к ошибке. Для обнаружения дефекта составляется конъюнкция предиката безопасности и предиката пути после применения алгоритма 1 слайсинга предиката пути, который устраняет избыточные ограничения на символьные переменные, не зависящие по данным от переменных в предикате безопасности. Таким образом, предикат пути отвечает за достижимость анализируемой инструкции (функции), а предикат безопасности — за проявление дефекта. Полученный объединенный предикат передается SMT-решателю. Если предикат выполним, то разработанный метод сообщает об обнаруженной ошибке и генерирует новые входные данные для воспроизведения этой ошибки в анализируемой программе. В частности, разработаны предикаты безопасности для поиска ошибок целочисленного деления на нуль и разыменования нулевого указателя. Предикаты безопасности для таких ошибок заключаются в сравнении символьного делителя и символьного адреса соответственно с нулем.

В разделе 4.1.1 содержится описание разработанного метода построения предиката безопасности для обнаружения ошибок выхода за границу массива. Предикат безопасности конструируется для каждого разыменования символьного адреса *sym_addr* (который зависит от пользовательских данных). 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: $sym\_addr < lower\_bound \vee sym\_addr \geqslant upper\_bound$. 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 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, по бинарному коду без отладочной информации может быть определена только нижняя граница глобального массива. Базовый адрес массива может вычисляться эвристически из символьного адресного выражения, например, `[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.

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. 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. 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).

В разделе 4.1.2 описывается разработанный метод построения предиката безопасности для ошибки целочисленного переполнения. 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. 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 решается только 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. Все символьные аргументы некоторых моделируемых функций стандартной библиотеки 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 переполнения. Для большинства арифметических инструкций предикат безопасности будет истинным, когда соответствующий флаг выставлен в единицу.

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. 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. Разработанный алгоритм позволяет определить знаковость арифметической операции. В точке стока ошибки производится итерирование в обратном порядке, начиная с последнего перехода в предикате пути. В результате, обнаруживается первый условный переход, который может указать на знаковость. 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, а место вызова функции, содержащей инструкцию перехода, находится в текущем стеке вызовов. 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.

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.

В разделе 4.2 описан разработанный **метод автоматизированного поиска ошибок при помощи символьных предикатов безопасности после гибридного фаззинга** [2; 5; 6]. Сначала производится гибридный фаззинг исследуемого проекта, при котором совместно работают фаззер и инструмент динамической символьной интерпретации для открытия путей выполнения программы. 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.

В разделе 4.3 приводится экспериментальная оценка точности **метода построения предикатов безопасности** на наборе тестов Juliet. 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. При этом исходный код самих тестов не модифицировался. Разработанная тестовая система Juliet Dynamic 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.

В таблице 2 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 (15772 теста) из набора тестов Juliet.

Таблица 2 — Результаты тестирования на Juliet

CWE	P=N	Текстовые срабатывания			Верификация санитайзерами		
		TPR	TNR	ACC	TPR	TNR	ACC
121: Stack Based Buffer Overflow	188	100%	100%	100%	100%	100%	100%
122: Heap Based Buffer Overflow	376	100%	100%	100%	100%	100%	100%
124: Buffer Underwrite	188	100%	100%	100%	100%	100%	100%
126: Buffer Overread	188	100%	100%	100%	100%	100%	100%
127: Buffer Underread	188	100%	100%	100%	100%	100%	100%
190: Integer Overflow	2580	99.92%	90.89%	95.41%	98.10%	90.89%	94.50%
191: Integer Underflow	1922	99.90%	91%	95.45%	97.45%	91%	94.22%
194: Unexpected Sign Extension	752	100%	100%	100%	100%	100%	100%
195: Signed to Unsigned Conversation	752	99.87%	100%	99.93%	99.87%	100%	99.93%
369: Divide by Zero	564	66.67%	100%	83.33%	66.67%	100%	83.33%
680: 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%

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 не поддерживает арифметику с плавающей точкой.

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 арифметику, 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.

В разделе 4.4 содержится апробация предложенного **метода автоматизированного поиска ошибок при помощи символьных предикатов безопасности в контексте гибридного фаззинга**. Апробация проводилась путем поиска новых ошибок в проектах с открытым исходным кодом. Разработанный метод был применен к ряду проектов с открытым исходным кодом [2; 5; 6]. В результате апробации разработанного метода было обнаружено **17** новых ошибок в **10** различных проектах (OpenJPEG, Poppler, miniz, unbound и другие): 15 ошибок целочисленного переполнения (2 из них приводят к выходу за границы массива), 1 ошибка выхода за границы массива и 1 ошибка деления на нуль. Информация обо всех ошибках была доведена до разработчиков, часть из них уже исправлены.

В **пятой главе** приведено описание деталей реализации разработанных методов в программных инструментах [7; 8].

В разделе 5.1 представлен инструмент динамической символьной интерпретации Sydr (Symbolic DynamoRIO), в котором реализованы алгоритм слайсинга предиката пути, а также методы моделирования семантики функций

и построения предикатов безопасности. Sydr использует инструмент динамической бинарной инструментации DynamoRIO для конкретной интерпретации и фреймворк Triton для динамической символьной интерпретации. В роли математического решателя выступает Bitwuzla. 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. Конкретная и символьная интерпретация разделены на два процесса, которые обмениваются сообщениями (событиями) при помощи разделяемой памяти. Конкретный вычислитель сохраняет описания событий в разделяемой памяти, которые в дальнейшем обрабатываются символьным вычислителем.

В разделе 5.2 описана реализация метода автоматизированного поиска ошибок при помощи символьных предикатов безопасности в контексте гибридного фаззинга. Гибридный фаззинг осуществляется с использованием фаззера libFuzzer (или AFL++) и инструмента динамической символьной интерпретации Sydr, 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.

В **заключении** приведены основные результаты работы, которые 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 показала общую точность 95.59 % для 11 классов ошибок CWE (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, которые используются в Центре доверенного искусственного интеллекта ИСП РАН.

Публикации автора по теме диссертации

1. 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. — (ВАК).
2. *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. — (ВАК).
3. 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. — (Scopus, WoS).
4. 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. — (Scopus, WoS).
5. *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.
6. *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.
7. *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).
8. *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).