На правах рукописи

САРГСЯН СЕВАК СЕНИКОВИЧ

МЕТОДЫ ОПТИМИЗАЦИИ АЛГОРИТМОВ СТАТИЧЕСКОГО И ДИНАМИЧЕСКОГО АНАЛИЗА ПРОГРАММ

2.3.5 – математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей

Автореферат

диссертации на соискание ученой степени
доктора технических наук

Москва – 2024

Работа выполнена в государственном образовательном учреждении высшего профессионального образования Российско-Армянский (Славянский) университет.

Научный консультант: **Аветисян Арутюн Ишханович**, академик РАН, д.ф.-м.н.

Официальные оппоненты: **Шабанов Борис Михайлович**, доктор технических наук, член-корреспондент РАН, Национальный исследовательский центр "Курчатовский институт", заместитель директора по исследованиям в области информационных технологий

Кознов Дмитрий Владимирович, доктор технических наук, доцент, Санкт-Петербургский государственный университет, профессор

Кореньков Владимир Васильевич, доктор технических наук, старший научный сотрудник, Объединенный институт ядерных исследований, научный руководитель

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

Защита состоится 17 декабря 2024 г. в 15 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 "___"_____2024 г.

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

Актуальность. Несмотря на быстрые темпы развития инструментов статического и динамического анализа программ и постоянно расширяющееся внедрение цикла разработки безопасного программного обеспечения (ПО), количество находимых уязвимостей ежегодно только увеличивается. Можно указать несколько причин сложившейся ситуации. Во-первых, сложность программных систем постоянно растет; современные программные системы, в том числе дистрибутивы ОС, могут содержать десятки, и даже сотни миллионов строк кода. Это приводит к ошибкам во время проектирования систем, а также при реализации конкретного функционала. Во-вторых, жесткий график реализации проектов и серьезная конкуренция не позволяют обеспечить полное покрытие тестами. В-третьих, высокая потребность в ИТ-специалистах на рынке труда приводит к вовлечению в проекты специалистов с недостаточно высокой квалификацией, что отражается на качестве кода. В-четвертых, сами компиляторы и другие средства разработки могут содержать ошибки, и из корректного исходного кода может генерироваться бинарный код с дефектами, что, в свою очередь, порождает проблему анализа непосредственно бинарного кода, а это сложнее, чем анализ исходного кода. В дополнение к вышесказанному, широкое использование открытого ПО приводит к многократному тиражированию одной ошибки. Одним из известных примеров является дефект в коде библиотеки *openssl*, известный как "*HeartBleed*", который уже привел к уязвимости около полумиллиона сайтов.

Крупные игроки индустрии уже давно осознали важность разработки безопасного ПО, и многие из них предлагают решения, а также платформы для анализа программ, в частности, открытого ПО. Одной из известных и широко используемых платформ статического анализа исходного кода является *CodeQL* от компании *GitHub*, которая способна выполнять различные запросы к коду и выявлять уязвимости. Для запросов доступны инструкции программ, граф потока данных и управления, и т.д. Для динамического анализа бинарного кода компания *Google* предлагает проект *OSS-Fuzz*, который позволяет выполнять фаззинг на кластерных мощностях для открытого ПО. В наборе компиляторной инфраструктуры *LLVM* содержится проект libFuzzer для фаззинга отдельных бинарных функций. Для каждой целевой функции пишется специальная оболочка, которая обеспечивает произвольно сгенерированные/мутированные входные данные. Вместе с тем можно заключить, что все доступные инструменты решают проблемы безопасности кода только частично и нацелены на узкий круг задач.

Каждый из существующих подходов анализа имеет свои преимущества и ограничения, что не позволяет при применении одного метода найти все возможные

ошибки в программах. Например, методы статического анализа могут учитывать все возможные пути потока управления программы, но не всегда способны отличать выполнимые пути от невыполнимых, что часто приводит к ложным срабатываниям. Методы динамического анализа могут предоставлять данные, которые позволяют выполнять конкретные пути в программе, но, как правило, они не масштабируются и обеспечивают только частичное покрытие кода. Одним из подходов к решению описанных проблем стала комбинация методов статического и динамического анализа. Существуют различные сценарии комбинирования. Динамический анализ может быть использован для верификации результатов, полученных статическим анализатором, производя проверку выполнимости путей найденных ошибок. Также возможно комбинирование разных методов динамического анализа. Одним из часто встречающихся вариантов подобного комбинирования является использование фаззинга с символьным выполнением, целью которого является увеличение покрытия кода за счет способности символьного выполнения обеспечивать проход по путям выполнения со сложными условиями.

Множество нерешенных проблем можно условно разделить на два класса. Первый класс связан с ограничениями отдельных методов анализа, а второй – со сложностью комбинирования различных методов и технологий анализа и интеграции их в единый программный комплекс поддержки жизненного цикла разработки безопасного ПО. Методы статического анализа почти всегда не учитывают информацию об известных уязвимостях в открытом программном обеспечении, а поиск известных ошибок – это отдельный класс инструментов. Не существует качественных и точных инструментов поиска клонов кода, включая клоны известных уязвимостей. Также отсутствуют высококачественные инструменты сопоставления исходного и бинарного кода, что могло бы решить множество задач обратной инженерии, возникающих при анализе кода. Методы фаззинга сталкиваются с серьезными ограничениями при генерации структурированных данных и цепочек вызовов системных/библиотечных функций. Во время фаззинга информация, полученная статическим анализом, используется не в полном объеме. Не существует платформы, в рамках которой можно было бы собрать, хранить и удобно использовать артефакты большого объема открытого ПО, некоторыми унифицированными способами обмениваться данными между различными методами анализа, комбинировать единообразным подходом разные виды анализов в зависимости от конкретной задачи. Это позволило бы увеличить эффективность отдельно используемых методов и их комбинаций в целом.

Можно заключить, что развитие отдельных методов анализа, способов комбинирования различных технологий анализа и создание платформы для совместного использования этих методов является важной задачей, при этом требующей решения сложных теоретических и инженерных проблем.

Цель и задачи работы. Разработка и реализация эффективных методов статического и динамического анализа программ для выявления ошибок и уязвимостей в программном обеспечении, а также создание платформы, обеспечивающей: обработку большого объема ПО; эффективную интеграцию и комбинацию разных видов анализа.

Для достижения поставленной цели необходимо решить следующие задачи:

- Провести анализ существующих методов и подходов обеспечения безопасности ПО для выявления их недостатков, и определения основных направлений исследования, а также определить функциональные требования разрабатываемой платформы.
- Разработать и реализовать методы статического анализа для сопоставления исходных и бинарных файлов, а также для поиска клонов кода, неисправленных уязвимостей, утечек динамической памяти, ошибок повторного освобождения динамической памяти, ошибок использования форматной строки.
- Разработать и реализовать методы динамического анализа, позволяющих интегрировать символьное выполнение с фаззингом, интегрировать статический анализ с фаззингом, проводить эффективный фаззинг программ, принимающих структурированные данные, и фаззинг интерфейсных функций.
- Проанализировать требования, разработать и реализовать платформу, которая позволяет: собирать артефакты для большого объема открытого ПО, включая графы зависимостей программ, промежуточные представления программ, исходный/бинарный код и информацию об известных уязвимостях; комбинировать единообразным подходом различные методы анализа кода в зависимости от конкретной задачи.

Методы исследования. Для решения поставленных задач использовались методы теории графов, теории решеток и теории компиляции.

Положения, выносимые на защиту и научная новизна. В диссертации получены следующие новые результаты, которые **выносятся на защиту**:

- Архитектура и экспериментальный образец платформы анализа программ, обеспечивающая: сбор артефактов для большого объема открытого ПО и

информации об известных уязвимостях; единообразный подход комбинирования различных методов анализа кода в зависимости от конкретной задачи.

- Масштабируемые и точные методы нахождения клонов кода, основанные на поиске схожих подграфов максимального размера для графов зависимостей программ, построенных на основе промежуточных представлений исходного и бинарного кода.
- Метод сопоставления исходных и бинарных файлов, который из входного исходного кода получает множество бинарных файлов, скомпилированных с разными уровнями оптимизации и содержащих отладочную информацию, после чего производит сопоставление инструкций полученных и выходных бинарных файлов на основе разработанного инструмента поиска клонов бинарного кода, а в конце выполняет сопоставление исходного кода с инструкциями входных бинарных файлов на основе отладочной информации сопоставленных бинарных инструкций.
- Метод поиска утечек памяти для языков Си/Си++, который на первом этапе производит поиск утечек на специальном представлении программы, содержащей поток управления и данных со смещениями доступа к указателям и полям структур, а на втором этапе производит проверку выполнимости путей ошибок методом направленного символьного выполнения.
- Метод фаззинга программ, генерирующий структурированные данные на основе специализированных автоматов БНФ грамматик, где веса автоматов динамическим образом меняются в процессе фаззинга, что обеспечивает адаптацию шаблонов генерируемых программ в зависимости от их эффективности для увеличения покрытия кода.
- Метод фаззинга интерфейсных функций, который позволяет генерировать цепочки вызовов функций и использовать возвращаемые значения одних функций в качестве аргументов для других, что обеспечивает возможность подготовки необходимых ресурсов для тестирования сложных сценариев использования нескольких функций в среде выполнения.
- Метод направленного фаззинга для быстрой генерации входных данных с целью выполнения конкретных инструкций или фрагментов целевой программы, содержащий потенциальные уязвимости или дефекты.
- Метод интеграции статического анализа с фаззингом, который применяет статический анализ для получения константных значений, используемых в

условных операторах, и затем использует эти константы для генерации входных данных, покрывающих соответствующие ветви кода.

Теоретическая и практическая значимость. Теоретическая значимость данной диссертационной работы заключается в предложенной концепции платформы анализа программ, а также в методах и алгоритмах статического и динамического анализа программ, которые в ходе экспериментального тестирования показали свое превосходство по сравнению с существующими решениями.

Практическая значимость определяется тем, что на базе разработанных методов реализована программная платформа *GENES ISP*, включающая в себя функциональность сбора артефактов ПО, инструменты, реализующие предложенные методы статического и динамического анализа, а также возможность комбинированного применения всех инструментов для анализа в режиме непрерывной интеграции. *GENES ISP* внедрен в цикл разработки ПО в ИСП РАН и ЦППТ РАУ с 2021. Разработанное средство *GENES ISP* может применяться в жизненном цикле разработки безопасного ПО, что покрывает многие из требований ГОСТ Р 56939-2016 и "*Методики выявления уязвимостей и недекларированных возможностей в программном обеспечении*" ФСТЭК Российской Федерации. Концепция предложенной платформы может быть использована при создании отраслевых и корпоративных репозиториев безопасного ПО. Отдельно разработанные методы также реализованы в инструментах *Svace* и *ISP-Fuzzer*, входящий в состав *Crusher*, которые являются индустриальным стандартом для жизненного цикла разработки безопасного ПО. Акты о внедрении результатов диссертации в "*Базальт СПО*" и ООО "*РусБИТех-Астра*" приведены в приложении к диссертации.

Апробация работы. Основные результаты диссертационной работы обсуждались на 12 международных конференциях: Открытая конференция по компиляторным технологиям, Москва, 2 декабря, 2015; FOSDEM-2015, Brussels, 31 January - 5 February, 2015; International Conference on Computer Science and Information Technologies CSIT 2015, Yerevan, 28 September - 2 October, 2015; International Conference on Computer Science and Information Technologies CSIT 2017, Yerevan, 25-29 September, 2017; Ivannikov Memorial Workshop (IVMEM), Yerevan, 3-4 May 2018; International Conference on Engineering Technologies and Computer Science (EnT), Moscow, 26-27 March, 2019; Ivannikov Memorial Workshop (IVMEM), Velikiy Novgorod, Russia, 13-14 September, 2019; Ivannikov ISP RAS Open Conference, Moscow, 11-12 December, 2020; Ivannikov Memorial Workshop (IVMEM), Nizhny Novgorod, 24-25 September 2021; Ivannikov ISP RAS Open Conference, Moscow, 2-3 December, 2021;

Ivannikov Memorial Workshop (IVMEM), Kazan, 23-24 September, 2022; Ivannikov ISP RAS Open Conference, Moscow, 1-2 December, 2022.

Гранты и контракты. Работа по теме диссертации проводилась в соответствии с планами исследований по проектам, поддержанными: совместным грантом КН Армении и РФФИ, 20RF-033 "*Разработка и реализация масштабируемых методов анализа современных операционных систем*"; грантом КН Армении 21SCG-1B003 "*Разработать и реализовать систему анализа безопасности и сертификации программного обеспечения*".

Личный вклад. Выносимые на защиту результаты получены соискателем лично. В опубликованных совместных работах постановка и исследование задач осуществлялись совместными усилиями соавторов при непосредственном участии соискателя. Статьи [1-16] полностью написаны автором лично. В статьях [17-23] автором написаны обзорные разделы. В статье [24] автором написаны обзорный раздел и разделы, описывающие алгоритмы для построения аннотаций и детекторов поиска утечек памяти. Разработка зарегистрированных программных систем [25-31] была произведена непосредственно под руководством соискателя и с его личным участием в процессе разработки.

Публикации. Автором опубликовано более 30 научных печатных трудов по теме диссертации, включая работы по теории компиляции и анализу программного кода. В том числе по материалам диссертации опубликовано 12 работ в изданиях, входящих в список изданий, рекомендованных ВАК РФ, кроме того, 11 статей опубликовано в изданиях, индексируемых *Scopus* и *Web of Science*. Статья [24] опубликована в *IEEE Access*, входящем в первый квартиль *SJR*. Получено 7 свидетельств [25-31] о регистрации программ для ЭВМ.

Структура и объем диссертационной работы. Диссертация состоит из введения, семи глав и заключения, изложенных на 268 страницах, списка литературы из 271 наименований, содержит 56 рисунков и 35 таблиц.

СОДЕРЖАНИЕ РАБОТЫ

Во введении обосновывается актуальность исследований, приводятся цель и задачи работы, формулируется научная новизна и практическая значимость полученных результатов, приводятся сведения о результатах внедрения и использования.

В первой главе обосновывается важность обеспечения безопасности ПО. Рассматриваются области применения существующих методов. Приводятся основные ограничения существующих методов. Выделяются основные приоритеты для дальнейших исследований, а также описываются предлагаемые подходы решений. В частности, формулируется концепция платформы анализа программ, обеспечивающая: сбор артефактов для большого объема открытого ПО и информации об известных уязвимостях, единообразный подход комбинирования различных методов анализа кода в зависимости от конкретной задачи, очерчиваются основные элементы архитектуры экспериментального образца платформы.

В **разделе 1.1** были проанализированы функциональные требования и проведены исследования, по результатам которых предложена *архитектура и реализована платформа, для обеспечения комбинированного применения нескольких методов анализа в зависимости от конкретной задачи*. При комбинированном использовании различных методов анализа появляется возможность обнаружения тех ошибок, которые ранее не удавалось обнаружить каждым из методов в отдельности. Это, в свою очередь, позволяет более эффективно находить и исправлять уязвимости на ранних стадиях жизненного цикла разработки безопасного ПО, тем самым повышая его стабильность и безопасность. Одним из примеров совместного использования нескольких инструментов в рамках предлагаемой платформы – это поиск копий известных уязвимостей в сочетании с направленным фаззингом. Этот подход привел к обнаружению ряда ошибок в пакетах операционной системы (ОС) *Debian*, что подтверждает эффективность комбинирования различных методов анализа.

Функциональные требования к платформе и ее архитектура разрабатывались с целью преодоления ограничений отдельных технологий анализа путем их единообразной комбинации в зависимости от конкретной задачи. Также учитывалась возможность применения огромной базы доступного открытого ПО и информации об известных уязвимостях во время анализа.

Платформа должна обеспечивать:

1. *Сбор, хранение и удобное использование артефактов большого объема открытого ПО, включая различные дистрибутивы ОС (Debian, CentOS,*

FreeBSD) и соответствующие им доступные пакеты. Полученные данные будут использованы для поиска информации о конкретных файлах, пакетах и дистрибутивах ОС, включая наличие в них копий известных уязвимостей или устаревших версий библиотек. Собранная информация также будет применяться для восстановления исходного кода по заданному бинарному коду, что, в свою очередь, позволит проводить различные анализы уже непосредственно на исходном коде. Бинарные артефакты будут использованы для проведения автоматического фаззинга и поиска ошибок.

2. *Единый подход к обмену данными между различными методами анализа*. Это позволит легко комбинировать и запускать множество доступных методов анализа для поиска сложных дефектов.
3. *Возможность комбинировать единообразным подходом доступные методы анализа в зависимости от конкретной задачи*. Это позволит увеличить эффективность отдельно используемых методов за счет полученной информации из других анализов, а также обнаружить сложные дефекты за счет их комбинаций.
4. *Хранение и дальнейшее использование результатов анализов, в том числе и в режиме непрерывной интеграции*. Это позволит итеративно улучшать результаты анализов, не дублируя уже выполненную работу.

Для обеспечения функциональных требований сервисы предлагаемой платформы разделены на две группы. Первая группа предназначена для сбора и пополнения базы данных артефактов ПО. В базе данных артефактов хранится информация о разных дистрибутивах ОС (*Debian*, *CentOS*, *FreeBSD*) и соответствующих им, доступных пакетах. Вторая группа предназначена для анализа проектов с комбинированным применением всех разработанных методов в рамках данной работы. Проектами могут являться добавленные в базу данных дистрибутивы ОС и их пакеты, а также другие проекты пользователя.

Для выбора способа реализации единообразного комбинирования нескольких методов анализа необходимо учитывать несколько важных особенностей. Во-первых, входные данные и результаты различных методов анализа могут значительно отличаться как по структуре, так и по содержанию. Это означает, что при передаче результатов одного метода в качестве входных параметров другому может потребоваться предварительная обработка и конвертация данных, что требует соответствующих программных средств. Во-вторых, потенциальный набор инструментальных средств в предлагаемой платформе не ограничивается только нашими разработками, набор может быть расширен за счет других инструментов

анализа, в частности, программных средств с открытым кодом. Теоретически существует бесконечное количество вариантов комбинаций даже для ограниченного числа методов анализа, учитывая, что каждый метод может быть применен более одного раза. Таким образом, использование плагинов для каждого отдельного сценария представляется неудобным (создание нового плагина требует слишком много ресурсов). Разработка и использование собственного *DSL* также неудобны из-за ограничений таких языков, так как добавление новых инструментов анализа может потребовать поддержки новых функциональных возможностей в самом *DSL*. Кроме того, возникнет необходимость обучения пользователей новому языку. Учитывая вышеизложенное, было принято решение использовать язык программирования общего назначения *Python* для реализации единообразного комбинирования нескольких методов анализа. Выбор языка *Python* был обусловлен двумя факторами: он достаточно популярен и содержит множество библиотек для обработки различных типов данных. В результате реализован *Python*-терминал, который предоставляет доступ ко всем инструментам анализа через специально разработанные программные интерфейсы. Он обеспечивает единообразный и гибкий подход передачи данных, между разними инструментами анализа, средствами самого языка *Python*. С помощью *Python*-терминал можно написать скрипты/плагины для комбинированного запуска нескольких инструментов анализа. Возможно комбинировать любое количество инструментов анализа под конкретную задачу.

На рисунке 1 приводится схема работы предлагаемой платформы. Архитектура платформы состоит из двух основных компонентов. Первый компонент обеспечивает сбор необходимой информации для различных дистрибутивов ОС и соответствующих пакетов. Также имеется возможность добавить отдельные проекты и вручную указать зависимости (по сборке, установке и запуску) между ними. Для сбора информации платформа загружает исходные файлы дистрибутивов ОС и пакетов, после чего производит их сборку. Сборка производится в специально разработанной нами виртуальной среде. Процесс сборки осуществляется параллельно и распределенно для эффективной обработки большого объема данных. В ходе сборки строятся графы зависимостей программы для исходного и бинарного кода, сохраняется отладочная информация и т.д. В случае сборки отдельных проектов пользователь должен предоставить команды сборки или соответствующие скрипты. В систему можно добавить также бинарные проекты; в этом случае строятся только графы зависимостей

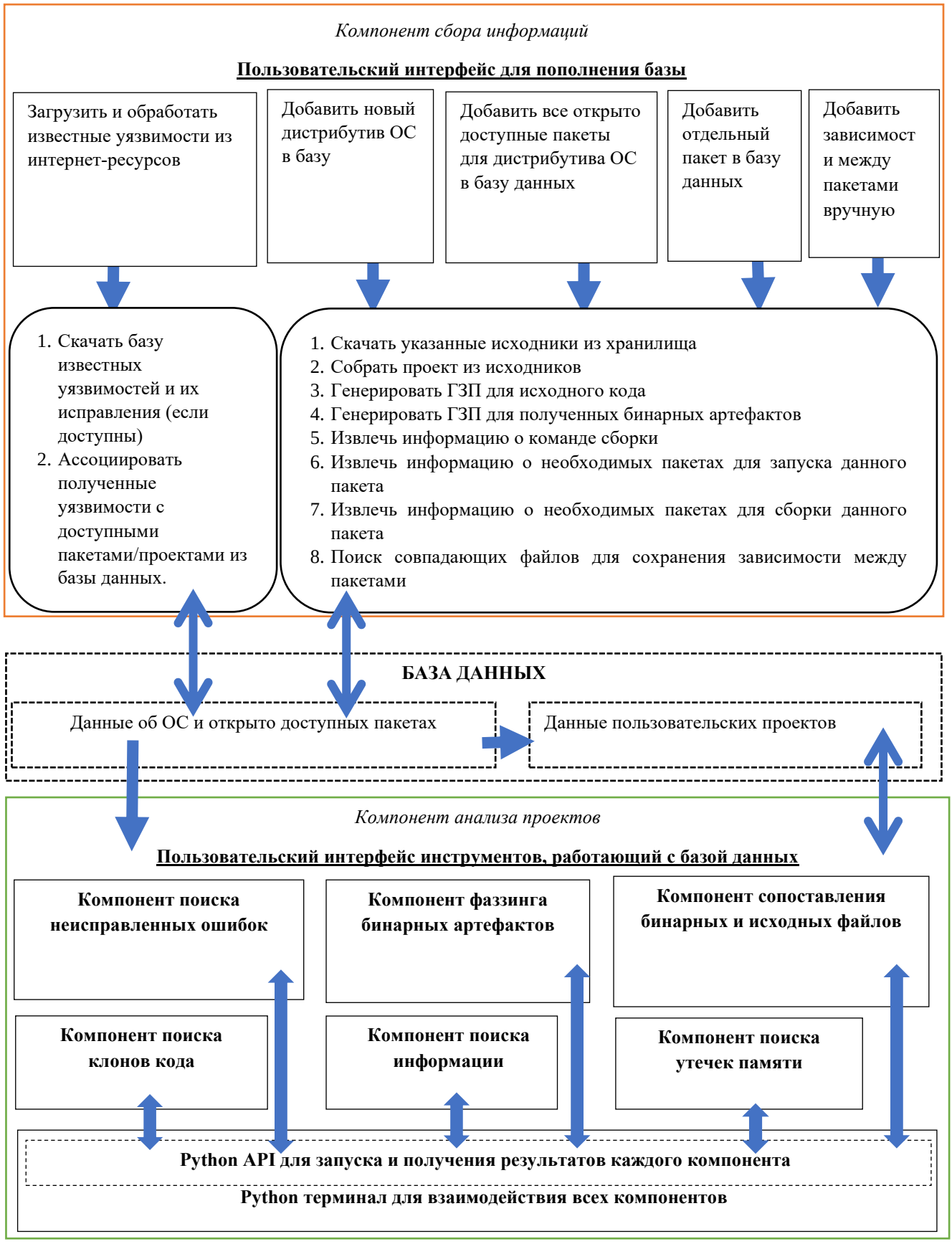

Рисунок 1. Схема работы предлагаемой платформы.

бинарного кода. В рамках первого компонента платформы также собирается информация об известных уязвимостях из интернет-ресурсов. После сбора информации производится ассоциация уязвимостей с пакетами и проектами из базы данных. Собранная база данных считается общей, и пользователи имеют доступ к ней только в режиме чтения. Только администратор системы может обновить общую базу данных. Если пользователю необходимо добавить свои пакеты и проекты для анализа, в базе данных выделяется локальное пространство для пользователя. Результаты анализов также сохраняются в локальном пространстве пользователя в базе данных.

Второй компонент платформы содержит множество методов анализа, разработанных в рамках данной работы. Каждый из разработанных методов может быть запущен отдельно – через пользовательский интерфейс. Во вторую компоненту платформы также входит *Python*-терминал, который обеспечивает возможность единообразного подхода объединения различных методов анализа. Для каждого доступного метода анализа разработан программный интерфейс, с помощью которого он может быть запущен из *Python*-терминала. По сути, все методы анализа доступны в виде удобной библиотеки в среде *Python*-терминала, а разработанное удобное автодополнение на каждом шаге подсказывает пользователю возможные интерфейсные функции. Вдобавок к этому, каждый сценарий объединения методов анализа, разработанный пользователем, может быть сохранен в виде плагина/скрипта и, далее, использован в режиме непрерывной интеграции.

В **разделах 1.2**, **1.3** и **1.4** приводятся перспективные направления исследований различных методов анализа кода и возможные улучшения. В целевые направления включаются: методы поиска клонов исходного/бинарного кода и базирующиеся на них инструменты поиска копий известных уязвимостей, сопоставления исходного и бинарного кода; методы поиска ошибок, связанных с использованием форматной строки и динамической памяти, включая утечку памяти и использование освобожденной памяти; методы оптимизации фаззинга программ для разных сценариев анализа.

В **разделе 1.5** приводятся новые направления исследований, которые определяют новые возможности по оснащению платформы анализа программ на основе предложенной концепции. Возможность собирать артефакты для большого объема открытого ПО и информацию об известных уязвимостях, а также единообразный подход к комбинации различных методов анализа кода, включая разработанные нами методы сопоставления исходных и бинарных файлов, поиска утечек памяти и фаззинга программ для разных сценариев, позволяют считать перспективными следующие направления исследований:

1. ***Улучшение модели взаимодействия методов статического и динамического анализа программ***: передача информации между инструментами статического и динамического анализа, разработка новых метрик переключения между разными методами во время комбинированного анализа.
2. ***Интеграция в разработанную платформу доступных инструментов статического и динамического анализа программ***, а также поддержка соответствующих программных интерфейсов.
3. ***Поддержка разработанных методов статического анализа кода для других языков программирования (Java, Rust, Solidity и т.д.)***.
4. ***Автоматический фаззинг интерфейсных функций***. Платформа умеет автоматически собирать артефакты разных дистрибутивов ОС и соответствующих пакетов, а также открыто-доступного ПО. Доступные методы статического анализа могут находить функции с потенциальными ошибками. Это делает актуальной задачу автоматической генерации драйверов фаззинга (функции оболочки, которые подают на вход мутированные данные).
5. ***Применение МО для генерации эффективных входных данных для фаззинга***. При фаззинге доступных в базе программ платформа собирает большое количество данных о бинарных артефактах целевой программы, входных данных и соответствующих покрытиях кода. Это делает возможным применение методов машинного обучения для генерации эффективных входных данных с учетом бинарного кода целевой программы.
6. ***Поддержка множества сценариев комбинаций различных методов анализа по умолчанию***. В текущей реализации в платформе поддерживается малое количество комбинаций нескольких методов анализа. В основном, это совершалось для демонстрации эффективности платформы "*proof of concept*". Поддержка новых сценариев, по умолчанию, позволит иметь набор плагинов, которые можно включать в режиме непрерывной интеграции.
7. ***Автоматическое определение наилучшего набора комбинаций методов анализов – в зависимости от целевой программы.*** Так как сценариев комбинаций различных методов анализа достаточно много, важной задачей становится автоматическое определение конкретного набора методов анализа и их последовательность, в случае которой можно найти ошибки в целевой проекте.

Вторая глава представляет собой обзор современного состояния методов статического и динамического анализа, важных для обеспечения безопасности программ. Для каждого вида методов описываются основные подходы, сравниваются их функциональные возможности и доступные инструменты. **Раздел 2.1** посвящен методам поиска клонов кода в исходных файлах программ. В нем вводятся основные типы клонов и по отношению к этим типам описываются подходы к поиску клонов и реализующие эти подходы инструменты. **Раздел 2.2** представляет задачу поиска клонов для бинарного кода. Делается вывод об актуальности задачи эффективного поиска бинарных клонов в значительно отличающихся бинарных файлах. **В разделе 2.3** приводится описание методов анализа изменений между версиями ПО, включая поиск неисправленных ошибок; оказывается, что существующие инструменты опираются только на сообщения, записанные в системе контроля версий при исправлении ошибок, и на доступные базы данных с известными ошибками. **В разделе 2.4** описываются методы сопоставления исходного и бинарного кода и результаты сравнения соответствующих инструментов, из которых следует, что эта задача удовлетворительного решения не получила. **Разделы 2.5** и **2.6** посвящены описанию и анализу методов поиска утечек динамической памяти и ошибок использования освобожденной памяти соответственно. **Раздел 2.7** содержит описание методов фаззинг-тестирования и известных инструментов фаззинга для различных сценариев, включая фаззинг протоколов, грамматик и системных вызовов.

В третьей главе приводится описание разработанных автором методов поиска клонов кода на исходных и бинарных файлах. Кроме этого, приводится описание инструментов, которые были разработаны автором на базе технологий поиска клонов кода. Приводится описание инструментов для поиска копии неисправленных ошибок, а также сопоставления исходного и бинарного кода.

В разделе 3.1 приводится описание разработанных автором методов поиска клонов исходного кода. *Поиск клонов кода производится на основе ГЗП (граф зависимостей программы), который строится на базе промежуточного представления компиляторной инфраструктуры LLVM. Клоны определяются как максимальные схожие подграфы рассматриваемой пары ГЗП*. Поиск максимальных схожих подграфов производится в четыре этапа, что позволяет эффективно и точно решить задачу. Сначала ГЗП разделяются на подграфы (единицы сравнения – ЕС, рассматриваемые как потенциальные клоны друг друга), после чего для всех возможных пар ЕС применяются алгоритмы линейной сложности, которые проверяют, что данная пара ЕС не является клоном. Если эти алгоритмы не смогли доказать, что пара ЕС не является клоном, применяется приближенный алгоритм

поиска максимальных схожих подграфов. После того, как максимальные схожие подграфы найдены, производится дополнительная проверка соответствия исходного кода и найденных подграфов. Если строки исходного кода, соответствующие схожему подграфу, находятся рядом (на незначительном расстоянии) и длина участка больше размера минимального клона, то они считаются клонами.

В ходе экспериментального тестирования разработанный инструмент (*CCD*) показал лучшие результаты по сравнению с существующими аналогами. В таблице 1 приводится описание тестового набора, а в таблице 2 — полученные результаты.

Таблица 1. Набор тестов, содержащий разные типы клонов кода.

Имя теста	**Описание**

copy00.cpp – оригинал	1.void foo(float sum, float prod) {
	2. float result = sum + prod;
	3.}
	4.void sumProd(int n) {
	5. float sum = 0.0; //C1
	6. float prod = 1.0;
	7. for (int i = 1; i <= n; i++) {
	8. sum = sum + i;
	9. prod = prod * i;
	10. foo(sum, prod);
	11. }
	12.}

copy01.cpp	**copy00.cpp:** были добавлены пробелы ст. 8, 9
copy02.cpp	**copy00.cpp:** были добавлены комментарии ст. 6, 9
copy03.cpp	**copy00.cpp:** переменные **sum** и **prod** были переименованы в **s** и **p**
copy04.cpp	**copy00.cpp:** аргументы **foo** поменялись местами, ст. 10
copy05.cpp	**copy00.cpp:** тип **sum** и **prod** изменен на **int**, ст. 5,6
copy06.cpp	**copy00.cpp: i** заменен на **i * i**, ст. 8,9
copy07.cpp	**copy00.cpp:** строки 5 и 6 поменялись местами
copy08.cpp	**copy00.cpp:** строки 8 и 9 поменялись местами
copy09.cpp	**copy00.cpp:** lines 9 and 10 поменялись местами
copy10.cpp	**copy00.cpp: for** заменен на **while**
copy11.cpp	**copy00.cpp:** добавлено условие (**if(i%2)**) для выполнения инструкций на 8-й строке
copy12.cpp	**copy00.cpp:** инструкция на 9-й строке удалена
copy13.cpp	**copy00.cpp:** добавлено условие (**if(i%2)**) для выполнения инструкций на 10-й строке
copy14.cpp	**copy00.cpp:** для второго аргумента **foo** добавлено значение по умолчанию, **prod** удален ст. 10
copy15.cpp	**copy00.cpp:** дополнительный аргумент добавлен в **foo** ст. 1, 10

В разделе 3.2 приводится описание разработанного автором *метода поиска клонов бинарного кода. Разработанный инструмент на базе предложенного метода является платформенно-независимым и может анализировать исполняемые файлы архитектур: x86, x86-64, ARM. Работа инструмента состоит из трех основных этапов: генерация ГЗП для бинарных функций, разделение ГЗП на ЕС и поиск клонов кода*.

Таблица 2. Результаты сравнения доступных инструментов поиска клонов кода.

Имя теста	MOSS	CloneDR	CCFinder	PMD/CPD	SourcerCC	VUDDY	Deckard	CCD
copy01.cpp	+	+	+	+	+	+	+	+
copy02.cpp	+	+	+	+	-	+	+	+
copy03.cpp	+	+	+	-	-	+	-	+
copy04.cpp	+	+	+	-	-	+	-	+
copy05.cpp	+	+	+	+	-	+	+	+
copy06.cpp	-	+	-	-	-	-	-	+
copy07.cpp	+	+	-	+	+	-	+	+
copy08.cpp	-	-	-	-	-	-	+	+
copy09.cpp	-	+	-	+	+	-	+	+
copy10.cpp	-	+	-	-	+	-	+	+
copy11.cpp	-	-	-	-	+	-	-	+
copy12.cpp	+	+	-	-	-	-	-	+
copy13.cpp	+	+	-	-	+	-	-	+
copy14.cpp	+	+	+	+	+	-	+	+
copy15.cpp	+	+	+	+	+	-	+	+

Для генераций ГЗП сначала исполняемые файлы дизассемблируются с помощью *IDA Pro* и транслируются в *REIL* (*Reverse Engineering Intermediate Language*) представление платформы обратной инженерии Binnavi. Узлам ГЗП соответствуют инструкции *REIL*, а ребрам – зависимости по управлению и по данным. Зависимости по управлению получаются из дизассемблера *IDA Pro* (граф потока управления). Зависимости по данным восстанавливает разработанный алгоритм анализа *use-def* цепочек на *REIL*-представлении. Для производительности весь процесс генерации ГЗП распараллелен. На втором этапе ГЗП разделяются на ЕС, которые представляют собой подграфы ГЗП. Доступны два подхода разделения на ЕС – по базовым блокам исполняемого кода, и по слабо связанным компонентам ГЗП. На последнем этапе происходит поиск клонов исполняемого кода на основе полученных ЕС. Сначала производится фильтрация пар ГЗП на основе минимального числа инструкций для конечных клонов. Если пара ЕС не содержит заданного количества совпадающих инструкций с одинаковыми кодами операций, тогда поиск наибольшего общего подграфа не производится. Большинство пар ЕС обрабатывается именно этим алгоритмом. Сложность алгоритма фильтраций доказывается в соответствующей теореме:

Теорема 1. *Сложность алгоритма фильтраций пар ЕС составляет* $O(n * log_2 n)$*, где n – сумма всех вершин в паре ЕС.*

Далее, применяется эвристический алгоритм для поиска максимального общего подграфа в парах ЕС. Если находится общий подграф, соответствующий нужным критериям (количество инструкций, степень схожести), то соответствующие фрагменты кода выдаются как клоны. Преимуществом предлагаемого метода

является то, что он основан на семантическом подходе, что позволяет достичь высокой точности.

В таблице 3 приводятся результаты сравнения разработанного инструмента *binCCD* с доступным наилучшим (см. раздел *2.2.3*, обзор инструментов) инструментом *BinDiff*. Из результатов становится очевидно, что разработанный инструмент в большинстве случаев охватывает и превосходит результаты *BinDiff*.

Таблица 3. Сравнения разработанного инструмента binCCD с BinDiff.

Бинарный файл 1	**Бинарный файл 2**	**BinDiff**		**binCCD**		**Количество совпадений**
		Количество правильных сопоставлений	**Количество неправильных сопоставлений**	**Количество правильных сопоставлений**	**Количество неправильных сопоставлений**	
python-3.5.1	python-3.5.2	3895	36	3936	8	3895
python-3.5.2	python-3.6.3	2981	903	3456	489	2981
openssl-1.0.1f	openssl-1.0.1s	5305	108	5367	57	5305
openssl-1.0.1r	openssl-1.0.1s	5389	6	5379	16	5373
rsync-3.0.9	rsync-3.1.1	421	148	529	79	420
git-2.6.0	git-2.9.5	3147	288	3164	168	3046
libXML-2.9.2	libXML-2.9.3	2577	4	2581	3	2577

В разделе 3.3 приводится описание разработанных автором *методов сопоставления исходных и бинарных файлов*. Для этого исходный код компилируется в бинарные файлы – с использованием разных оптимизаций компилятора (*-O0, -O1, -O2, -O3*). При компиляции сохраняется также отладочная информация. Затем производится сопоставление всех полученных и входных бинарных файлов. Для этого используется разработанный инструмент поиска клонов бинарного кода (**раздел 3.2**), который предоставляет пары сопоставленных бинарных инструкций. Сопоставление инструкций бинарного кода со строками исходного кода получается из сопоставленных пар бинарных инструкций и отладочной информации.

Один из ключевых моментов этого подхода состоит в переборе параметров компиляции для исходного кода и получении максимально схожих с входными бинарных файлов. Перебор опций компилятора позволяет получать аналогичные сценарии встраивания функций при компиляции, что позволяет улучшить качество сопоставления бинарных файлов (одна из основных проблем сопоставления исходных и бинарных файлов – это разные сценарии встраивания функций).

Для проверки качества инструмента разработана специальная тестовая система. На входе она получает две версии (из истории разработки) исходного кода одной программы. Одна версия программы сначала компилируется в бинарные файлы с отладочной информацией, на основе чего производится точное сопоставление

бинарных функций (на основе имен) с функциями исходного кода. Далее, из бинарного файла удаляется отладочная информация и запускается разработанный инструмент для него и второй версии исходного кода. В результате тестирования разработанный инструмент показал, в среднем, 85%-ую точность и 83%-ую полноту. В случаях, когда опции компиляции исходного кода совпадали с опциями компиляции входного бинарного файла, точность и полнота превысили 96%.

В разделе 3.4 приводится описание разработанных автором метода поиска неисправленных уязвимостей. Он состоит из четырех основных шагов. На первом шаге производится сбор информации обо всех известных уязвимостях (*CVE - Common Vulnerabilities and Exposures*) из таких онлайн-ресурсов, как *GitHub, NVD, Mitre*. На втором шаге для проекта, соответствующего каждой уязвимости, производится поиск соответствующих репозиториев. Если репозиторий проекта существует, инструмент проверяет историю изменений кода, извлекает исправления этой уязвимости и сохраняет в базе данных. Дополнительно извлекаются все исправления для всех пакетов *Debian*. Из исправлений строится уязвимая версия данного фрагмента, которая затем используется для поиска неисправленных уязвимостей. На третьем шаге собирается большая база открыто доступных проектов, в том числе из популярных проектов *GitHub*. На последнем шаге производится поиск построенных уязвимых фрагментов кода в собранной базе данных. Поиск производится улучшенной версией известного инструмента *ReDeBug*, в которой добавлен параллелизм и произведен ряд улучшений, сокращающих ложные срабатывания.

В ходе тестирования разработанного метода на реальных проектах было выявлено множество неисправленных уязвимостей, в том числе в *grub2*, являющемся загрузчиком ядра ОС *Linux*. В частности, инструмент обнаружил, что:

- *4Pane*, менеджер файлов в ОС *Linux*, использует старую версию архиватора *bzip2*, содержащую известную уязвимость CVE-2019-12900. Схожая ошибка была найдена также в проекте *doublecmd* (менеджер файлов);
- *AdAway*, блокировщик рекламы, использует старую версию *tcpdump*, содержащую известные уязвимости CVE-2018-16452, CVE-2017-13030, CVE-2018-14879 и CVE-2017-5486;
- *Praat*, инструмент анализа речи, использует библиотеку аудиокодеков без потерь *FLAC*, содержащую известную уязвимость CVE-2014-9028.

В проектах *LuaJIT*, *Grub2* и *MoarVM* метод сумел найти копию уязвимого фрагмента кода из проекта *minilua*, которому соответствует CVE-2014-5461.

В четвертой главе приводится описание разработанных автором методов поиска утечек памяти, поиска проблем, связанных с некорректным использованием динамической памяти и анализа помеченных данных.

В разделе 4.1 приводится описание разработанного автором метода анализа помеченных данных, на основе которого реализуется поиск ошибок форматной строки. Также приводится описание метода поиска ошибок использования освобожденной динамической памяти. Оба метода базируются на ГЗП всей программы, который в себе содержит граф потока данных и управления всех доступных функций, а также межпроцедурные зависимости по данным (прослеживается поток данных в случаях передачи аргументов другим функциям, а также через глобальные переменные). Доказываются сложности предложенных алгоритмов.

Теорема 2. *Сложность алгоритма поиска ошибок форматной строки составляет* $O(n^2 * (n + e))$, *где* n *– количество вершин в ГЗП, а* e *– количество ребер.*

Теорема 3. *Сложность алгоритма поиска использования освобожденной памяти составляет* $O(n^3 * (n + e))$, *где* n *– количество вершин в ГЗП, а* e *– количество ребер.*

В ходе экспериментального запуска разработанный инструмент сумел найти известные уязвимости CVE-2016-0705 и CVE-2016-0799 в проекте *openssl*. Также было выявлено, что применение данного подхода дает высокий уровень ложных срабатываний. На ядре *Linux* были найдены более 5000 ошибок использования освобожденной динамической памяти и 300 ошибок форматной строки. Дальнейшее улучшение метода приведет к снижению уровня ложных срабатываний, но достичь необходимого уровня (меньше 40% ложных срабатываний) данным методом не получится. Одна из основных причин – это нечувствительность анализа к выполнимости путей. В разделе 4.3 приводится описание более сложного подхода, частично использующего данный метод для поиска утечек памяти, который уже обеспечивает приемлемый уровень ложных срабатываний и внедрен в процесс безопасной разработки. Данный метод может быть адаптирован для поиска ошибок использования освобожденной динамической памяти и анализа помеченных данных.

В **разделе 4.2** приводится описание разработанного автором метода динамического поиска ошибок использования освобожденной динамической памяти. Для каждого пути выполнения программы проверяются корректность операций создания, доступа и освобождения динамической памяти. Для этого создаются два множества: *allocSet* и *freeSet*. При операциях выделения памяти в *allocSet* добавляются выделенные адреса. При освобождениях памяти соответствующий адрес переходит из

allocSet в *freeSet*. Метод выдает ошибку использования освобожденной памяти, если производится доступ к памяти, адрес которой находится в *freeSet*. Для прослеживания всех операций с памятью производится инструментация целевой программы.

Дополнительно разработан новый алгоритм подбора текущих входных данных для обработки (точек в целевой программе, откуда символьное выполнение должно продолжаться). В ходе динамического символьного выполнения те входные данные, которые обеспечили большее покрытие кода, получают высокий приоритет, что обеспечивает их использование при следующей итерации анализа. Данный подход позволяет ускорить рост покрытия кода.

Несмотря на то, что предложенный метод практический не имеет ложных срабатываний, его применение для анализа реальных проектов представляется невозможным. Этому есть две основные причины: во-первых, динамический анализ не может покрыть все пути выполнения; во-вторых, он работает достаточно медленно для анализа реальных проектов.

С учетом полученного опыта во время разработки и реализации данного метода и метода статического анализа, описанного в **разделе 4.1** удалось разработать комбинированный метод анализа утечек памяти, который может производить качественный анализ реальных проектов.

В разделе 4.3 приводится описание разработанного автором *метода поиска утечек памяти на базе графов потока данных и управления с последующей верификаций результатов на основе символьного выполнения*. Для этого используется специальная структура данных под названием *ProcedureGraph*, которая строится на базе промежуточного представления *LLVM*. Она содержит в себе граф потока управления, где вершинам соответствуют базовые блоки программы, а ребрам соответствуют переходы по управлению между этими блоками. *ProcedureGraph*, также содержит в себе граф потока данных со смещениями доступа к указателям и полям структур. Каждой вершине соответствует инструкция промежуточного представления *LLVM*, а ребрам соответствуют зависимости по данным, маркированные соответствующими смещениями доступов. Каждая вершина потока управления содержит в себе набор вершин потока данных, соответствующих инструкциям данного базового блока. Это представление программы является ключевой для разработанного метода.

Метод обнаружения ошибок состоит из трех основных этапов. На первом этапе из графов вызовов функций удаляются циклы. После этого производится восходящий анализ функций, где начальным уровнем являются функции, из которых вызываются библиотечные функции, либо не производится вызов вообще. Далее, рекурсивно

анализируются все функции, из которых производятся только вызовы функций предыдущих уровней. Во время анализа каждой функции строится ее аннотация. Аннотация содержит информацию о выделениях, освобождениях, присваиваниях динамической памяти и соответствующих условий, при которых данные операции производятся. Аннотации позволяют пропускать повторный анализ одной функции при множестве его вызовов и ускорять анализ. Пользователь может также предоставить аннотации использованных библиотечных функций для улучшения результатов анализа.

На втором этапе происходит поиск утечек памяти. Разработано несколько детекторов утечек памяти для разных сценариев. Для каждой найденной ошибки детекторы также сохраняют поток управления, выполнение которого будет воспроизводить ошибку. Для поиска утечек памяти внутри функций используется дополнительно разработанная структура данных под названием *MemoryOperationGraph*. Приводится доказательство некоторых свойств предложенной структуры данных:

Определение 1. *Входной точкой в MemoryOperationGraph графе назовем вершину, не имеющую входящих ребер.*

Определение 2. *Выходными точками в MemoryOperationGraph графе назовем вершины, не имеющие исходящих ребер.*

Теорема 4. *Каждому пути от входной точки до некоторой выходной в MemoryOperationGraph соответствует, как минимум, один путь в потоке управления функций.*

MemoryOperationGraph строится на базе *ProcedureGraph*. В работе проводится доказательство сложности алгоритма:

Теорема 5. *Сложность алгоритма построения MemoryOperationGraph составляет* $O(n * (n + e))$*, где n – количество вершин в ProcedureGraph, а e – количество ребер*.

На третьем этапе происходит верификация найденных ошибок. Для этого разработан движок символьного выполнения на базе *KLEE*. На входе движок получает поток управления (множество базовых блоков промежуточного представления *LLVM*), базовый блок, содержащий инструкцию выделения памяти, и базовый блок, выполнение которого приведет к утечке памяти. Далее, движок производит символьное выполнение только в рамках указанного потока управления, что обеспечивает быстродействие анализа.

В таблице 4 приводится сравнение разработанного инструмента (*ml-hunter*) с доступными аналогами на тестовом наборе *CWE401_Memory_Leak* из проекта *Juliet*.

Из результатов становится очевидным, что разработанный инструмент по всем критериям превосходит доступные решения. Инструмент был также протестирован на реальных проектах и нашел множестве ошибок, включая в библиотеке *openssl*. В таблице 5 приводится список всех найденных и подтвержденных ошибок.

Таблица 4. Сравнение разработанного инструмента с доступными аналогами

Имя инструмента	**Истинно положительны**	**Истинно отрицательные**	**Ложноположительных**	**Ложноотрицательный**
CSA	477	4509	97	421
Infer	262	4392	214	606
SMOKE	496	4510	96	372
PCA	486	4342	264	382
SVF	452	4168	438	416
ml-hunter	**868**	**4586**	**20**	**0**

Таблица 5. Найденные ошибки в реальных проектах

Название проекта	**Ссылка репозиторий**	**Номер ошибки**
openssl	https://github.com/openssl/openssl	20870
ffmpeg	https://lists.ffmpeg.org/	10342
radare2	https://github.com/radareorg/radare2	21703, 21704
bind9	https://gitlab.isc.org/isc-projects/bind9	4282
clib	https://github.com/clibs/clib	292, 293, 295
coturn	https://github.com/coturn/coturn	1259
cups	https://github.com/apple/cups	6144
cyclonedds	https://github.com/eclipse-cyclonedds/cyclonedds	1814
gpac	https://github.com/gpac/gpac	2569
pupnp	https://github.com/pupnp/pupnp	430
varnish-cache	https://github.com/varnishcache/varnish-cache	3986
masscan	https://github.com/robertdavidgraham/masscan	730
FreeRDP	https://github.com/FreeRDP/FreeRDP	9410, 9411
libvips	https://github.com/libvips/libvips	3642
zstd	https://github.com/facebook/zstd	3764

В разделе 4.4 приводится сравнение разработанных методов и заключение.

В пятой главе приводится описание разработанных автором методов фаззинга для разных задач. Разработан общий инструмент фаззинга под названием *ISP-Fuzzer*, на базе которого реализуются все предлагаемые методы. Ядро написано на Си/Си++ для быстродействия. Поддерживается возможность прямого вызова *Python* кода из Си/Си++, на основе чего и реализован механизм драйверов. Такой подход делает инструмент легко расширяемым и эффективным для поддержки новых функциональных возможностей.

Важно отметить, что инструмент *ISP-Fuzzer* входит в состав *Crusher*, который является индустриальным стандартом и используется во многих компаниях в процессе разработки в соответствии с ГОСТ Р 56939-2016 и "*Методики выявления*

уязвимостей и недекларированных возможностей в программном обеспечении" ФСТЭК Российской Федерации.

В разделе 5.1 приводится описание разработанного автором *метода эффективного фаззинга программ, принимающих сложные структурированные данные*. Генерация входных данных происходит на основе формального описания грамматики. Тип/структура генерируемых данных/программ периодически меняется на основе покрытия кода целевой программы. Генератор данных использует БНФ (Форма Бэкуса-Наура) описания правил на платформе *ANTLR* (*ANother Tool for Language Recognition*), что позволяет автоматическую поддержку более 280 языков и форматов данных.

Каждое БНФ-правило в *ANTLR* представлено в виде специального универсального автомата, где каждое состояние для нетерминальных символов представляет собой такой же автомат. Изначально, *ANTLR* используется как генератор парсеров. После лексического анализа, если полученная последовательность лексем позволяет из начального состояния автомата попасть в конечное, тогда входные данные считаются синтаксически корректными. Основная идея генерации данных – это:

1. Произвольный проход от начального состояния автомата к конечному, что предоставит некую последовательность состояний автомата – путь состояний автомата;
2. Для каждого терминального состояния пути добавляются в буфер соответствующие ему символы;
3. Для каждого нетерминального состояния рассматривается соответствующий ему автомат и находится в нем путь от начального состояния до конечного, состоящий только из терминальных состояний. При невозможности повторить процесс рекурсивно для нетерминальных состояний, пока не получится. Добавить в буфер соответствующие символы терминальных состояний, которые получились для найденного пути.

Это обеспечит синтаксически корректные данные в полученном буфере.

В *ANTLR* была также добавлена поддержка весов для ребер автомата, что обеспечивает вероятностный контроль над выбором путей при генерациях данных. Надо отметить, что выбор путей обеспечивает тип сгенерированных данных/программ. Для каждой грамматики пользователь может предоставить файл, содержащий положительные целые значения для всех ребер автомата. Нулевые значения не допускаются для весов ребер, потому что это может привести к тому, что пути от начального до конечного состояния не будут найдены. Если в автомате есть

некоторое состояние X, из которого выходит N ребер, то вес i-го ребра отмечается значением w_i. Вероятность выбора i-го исходящего ребра из состояния X определяется формулой $P(i) = \frac{w_i}{(w_1 + \cdots + w_N)}$ (нужно обратить внимание, что если имеется только одно исходящее ребро, то $P(1) = \frac{w_1}{w_1} = 1$.

Разработанный генератор данных интегрирован в *ISP-Fuzzer* и добавлена специальная мутация, которая мутирует веса автомата во время фаззинга. Это обеспечивает динамическое изменение типа генерируемых программ. Мутация весов происходит только тогда, когда текущая конфигурация не позволяет увеличить покрытие кода на длительное время. В ходе экспериментального тестирования разработанный метод смог обеспечить большее покрытие кода для таких известных компиляторов/интерпретаторов, как: *gcc*, *python*, *fortran*. Кроме этого, разработанный метод смог привести к аварийному завершению последние версии *gcc-12* и *clang-14*. В обоих случаях инструмент фаззинга сумел эксплуатировать ограничения алгоритма синтаксического анализа рекурсивного спуска, превысив лимит стека, по умолчанию, для компилятора.

В разделе 5.2 приводится описание разработанного автором *метода фаззинга интерфейсных функций*, в том числе для интернета вещей, который выявил 15 уникальных дефектов, приводящих к аварийному завершению приложения *SmartThings* от компании "*Samsung Electronics Co. Ltd*".

Инструмент состоит из трех основных компонентов. Первый компонент производит статический анализ *JAR* файлов и получает сигнатуры всех функций с атрибутами инкапсуляций, список всех функций для классов, конструкторы (далее: аннотация API). Второй компонент производит генерацию специального интерпретатора (фаззинг-клиента), который получает последовательность команд из инструмента фаззинга и производит вызов интерфейсных функций на основе модели регистровой машины. Для генерации данного интерпретатора используется информация, полученная на первом этапе. Третий компонент уже сам инструмент фаззинга, который умеет генерировать различные сценарии использования интерфейсных функций. На рисунке 2 приводится полная архитектура разработанного инструмента. Интерпретатор (фаззинг-клиент) представляет собой оператор-переключатель, который, в зависимости от кода операций, будет выполнять соответствующую операцию. Также имеется буфер переменных, в котором хранятся необходимые переменные (это, по сути, регистры регистровой машины).

В таблице 6 приводятся доступные коды операций. Каждый метод и класс (также поля класса) имеют уникальный идентификатор (*ID*), который позволяет

определить нужный метод, класс и поля класса. На рисунке 3 приводится фрагмент кода и соответствующая последовательность команд регистровой машины.

Генерация последовательности вызовов интерфейсных функций состоит из двух основных этапов. На первом этапе производится вызов всех интерфейсных функций (разминка). Целью данного этапа – сбор начального покрытия кода, чтобы генетический алгоритм фаззинга работал эффективно. В противном случае, добавление вызова новой функции в цепочку вызовов всегда будет увеличивать покрытие кода, и генетический алгоритм не сможет отличить эффективные мутации последовательности вызовов и аргументов функций от неэффективных. Кроме этого, на данном этапе всем функциям передаются нулевые указатели на объекты, в целях определения необработанных случаев использования нулевых указателей (*NullPointerException*). На этапе разминки, как правило, достигается покрытие кода по базовым блокам в 30-40%.

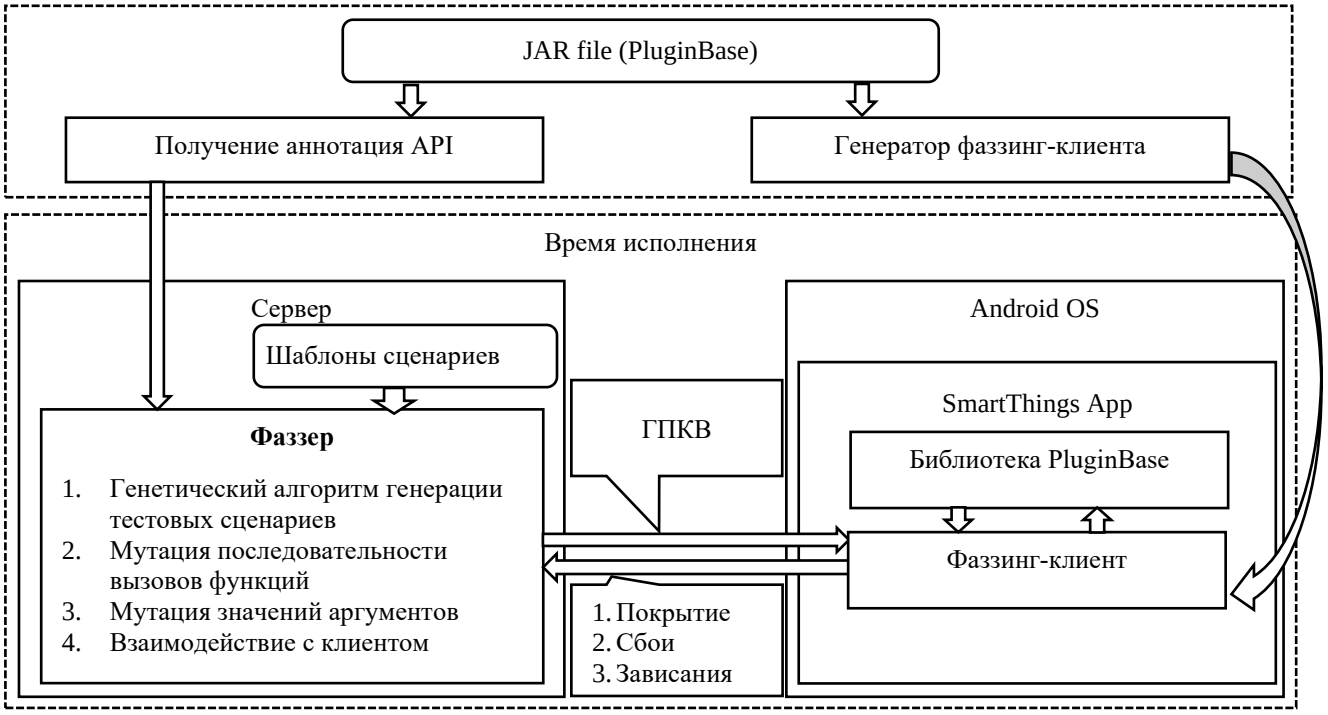

Рисунок 2. Схема работы фаззинга интерфейсных функций

На втором этапе начинается основной цикл фаззинга (мутация цепочки вызовов функций и их аргументов). Произвольным образом генерируется набор начальных цепочек. Одна из начальных цепочек вызовов функций загружается для обработки. Производятся мутации всех аргументов функций из цепочки, пока покрытие кода перестает увеличиваться. Тогда инструмент переходит к мутациям последовательности вызовов функций в цепочке. В процессе мутации могут быть добавлены или удалены некоторые вызовы. Если сгенерированная цепочка вызовов

увеличивает покрытие кода, то она сохраняется для дальнейшей обработки. Чем больше увеличивается покрытие кода, тем дольше будет обрабатываться соответствующая цепочка вызовов и аргументы функций. Пользователь может управлять процессом мутации, задав в шаблонном файле следующие параметры:

1. Последовательности вызовов функций, которые должны быть обработаны сначала;
2. Набор мутаций для аргументов функций;
3. Набор функций, которые должны быть вызваны до и после каждой цепочки вызовов;
4. Зависимости между функциями. В частности, возвращаемое значение одной функции должно быть использовано как аргумент вызова другой.

После того, как цепочка вызовов функций и соответствующие аргументы сгенерированы, они передаются следующему компоненту (генератор последовательности команд выполнения – ГПКВ). ГПКВ получает последовательность команд, выполнение которых на клиентской стороне обеспечит вызовы интерфейсных функций с соответствующими аргументами. Далее интерпретатор (фаззинг-клиент) выполняет последовательность команд и возвращает покрытие кода и информацию об аварийных завершениях.

Таблица 6. Доступные команды интерпретатора.

Описание команды	**Код операций**	**Операнды**
Положить значение в переменную	1-8 (в зависимости от типа)	переменная; значение
Положит UTF значение в переменную	9	переменная; UTF строка
Положить массив в переменную	10	переменная; идентификатор типа; длина; {значении}
Положить null в переменную	11	переменная
Положить строку в переменную	12	переменная; длина; {значении}
Положить массив переменных в переменную	13	переменная; длина; {переменные}
Вызов метода	"method ID"	[переменная]; ссылка объекта; {параметры метода}
Вызов статического метода	"method ID"	[переменная]; {параметры метода}
Получить значения поля объекта	"get field ID"	переменная; переменная, в которой ссылка на объект
Получить значения статического поля	"get field ID"	переменная
Положить значение в поле объекта	"put field ID"	переменная, в которой ссылка на объект; переменная, в которой значение
Положить значение в статическое поле	"put field ID"	переменная, в которой значение

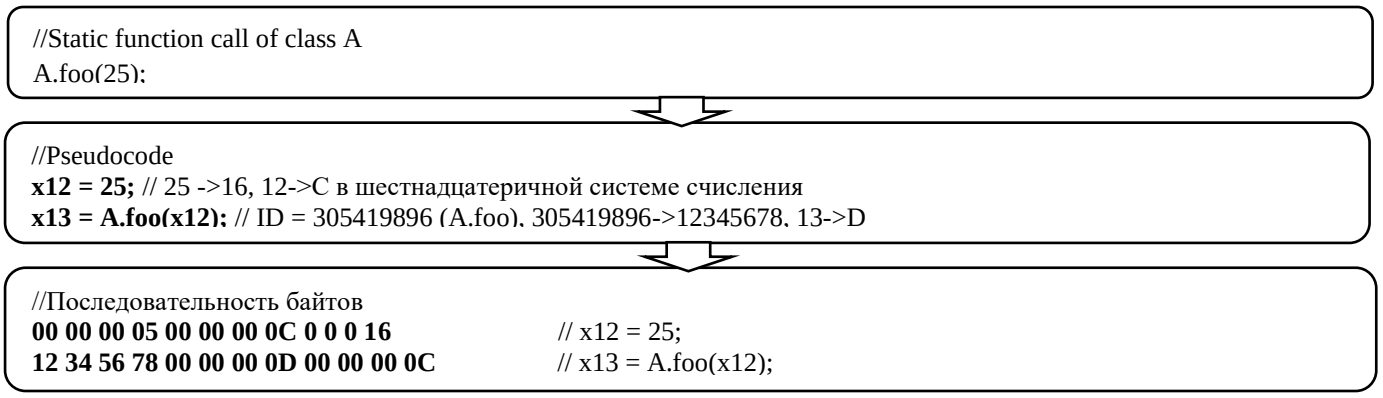

Рисунок 3. Пример последовательности байтов

В ходе экспериментального запуска на библиотеке *PluginBase*, входящий в состав платформы *SmartThings* (версия 1.7.9), разработанный инструмент нашел 15 уникальных аварийных завершений программы. Результаты были подтверждены командой тестирования "*Samsung Electronics Co. Ltd*".

В разделе 5.3 приводится описание разработанного автором *метода направленного фаззинга путем динамической инструментации программ*. Основной целью данного метода является быстрая генерация входных данных для выполнения конкретных инструкций/фрагментов целевой программы. Целевыми инструкциями и фрагментами обычно являются те, которые могут содержать уязвимости/дефекты. Для достижения цели сначала статическим анализом выявляются все пути в программе, которые соединяют точку входа программы с рассматриваемыми инструкциями. Затем применяется два типа динамической инструментации целевой программы для обеспечения направленного фаззинга.

В первом случае, вставляются инструкции по сбору покрытия только на путях, ведущих к целевым фрагментам. Это позволяет инструменту фаззинга считать сгенерированные или измененные входные данные нужными, если выполняется переход в рамках путей, ведущих к целевым фрагментам. Все остальные входные данные, которые приводят к увеличению покрытия кода "*неинтересных*" нам фрагментов, далее не рассматриваются фаззером. Это позволяет фаззеру сосредоточить мутации на увеличении покрытия нужных фрагментов кода – путей, ведущих к целевым фрагментам.

Во втором случае, дополнительно вставляются инструкции "*exit (0)*" в те базовые блоки, из которых целевые инструкции/фрагменты недостижимы. Это позволяет многократно увеличить скорость фаззинга за счет невыполнения "*неинтересных*" фрагментов кода. В данном случае существует недостаток: не всегда можно корректно определить статическим анализом, что инструкции после точки вставки "*exit (0)*" влияют/не влияют на корректность работы программы. В результате,

инструмент фаззинга может найти ложные падения программы. Для фильтрации таких случаев надо запустить не инструментированную программу на найденных входных данных и верифицировать результаты.

Разработанная система использует динамическую инструментацию во время фаззинга для получения покрытия целевой программ, а также для восстановления неявных переходов. Далее восстановленные неявные переходы используются в статическом анализаторе для улучшения результатов. Получается, что во время фаззинга периодически запускается статический анализатор с учетом восстановленных неявных переходов. А за счет корректировки найденных путей возникает необходимость в ходе фаззинга подправить инструментацию. Таким образом, процесс фаззинга и статического анализа итеративно взаимно улучшаются.

В таблице 7 приводится сравнение разработанного инструмента с оригинальной версией *ISP-Fuzzer*.

Таблица 7. Сравнение результатов ISP-Fuzzer с его модифицированной версией.

Имя теста	***Найденные дефекты с методом инструментаций 1***	***Найденные дефекты с методом инструментаций 2***	***Найденные дефекты с ISP-Fuzzer***
Personal_Fitness_Manager	+	-	-
Humaninterface	+	-	-
H20Flowlnc	-	+	-
One_Vote	+	-	-
Middleout	+	-	-
Particle_Simulator	-	+	-
Single-Sign-On	+	-	-
Stream_vm2	-	-	+
Multipass3	-	+	-
3D_Image_Toolkit	+	-	-
ECM_TCM_Simulator	+	+	-
XStore	+	-	-
HackMan	+	+	-
SAuth	+	-	+
CGC_Board	+	+	-
Flash_File_System	+	-	+
ASL6parse	+	-	-
Network_Queuing_Simulator	-	-	+

Разработанный инструмент запущен в направленном режиме с адресами дефектов для всех целевых программ. В таблице приводятся результаты после десяти часов фаззинга. Как становится очевидно из таблицы, оба предложенных метода динамической инструментации позволяют найти больше ошибок в тестируемых программах. Первый метод показывает лучшие результаты по сравнению со вторым методом, но есть ошибки, которые сумел найти только второй метод.

В разделе 5.4 приводится описание разработанного автором метода интеграции динамического символьного выполнения и статического анализа с фаззингом. Динамическое символьное выполнение и статический анализ кода встраиваются в фаззинг, что позволяет итеративно улучшать результаты каждого компонента. Во время фаззинга восстанавливаются неявные вызовы функций, и данная информация передается статическому анализатору, что улучшает обнаружение некоторых путей в графе потока управления программы. Далее, динамическое символьное выполнение для полученных путей генерирует входные данные, которые будут обеспечивать их выполнение. Затем эти входные данные используются фаззингом для улучшения генерации/мутации входных данных и увеличения покрытия кода. Предлагаемый метод может быть использован для классического фаззинга, когда основным критерием является увеличение покрытия кода. Также, его можно использовать для анализа целевых путей и фрагментов кода. В последнем случае, инструмент фаззинга принимает набор адресов программ с потенциальными дефектами и передает их статическому анализатору. Статический анализ строит все пути, соединяющие точки входа программ с этими адресами. Затем инструмент динамического символьного исполнения строит набор входных данных, который будет обеспечивать выполнение построенных путей.

Разработанный инструмент, на основе предложенного метода состоит из четырех основных компонентов. Первый компонент – это инструмент фаззинга, который предоставляет набор мутаций и базовую инфраструктуру. Второй компонент – это клиентская библиотека на основе *DynamoRIO* для сбора покрытия кода. Третьим компонентом является инструмент динамического символьного исполнения *Anxiety*. Четвертый компонент – это механизм статического анализа двоичного кода программы. Дополнительно используется база данных для хранения информации о целевой программе, в частности, информация о базовых блоках.

В случае направленного анализа вместо того, чтобы только увеличить общее покрытие кода, инструмент также периодически пытается генерировать входные данные, которые приведут к выполнению указанных пользователем адресов целевой программы. Для этого сначала применяется классический фаззинг, пока новые пути не будут обнаружены в течение некоторого времени (управляется пользователем). После чего статический анализатор строит все пути от точки входа целевой программы до указанных адресов. Затем, список базовых блоков, соответствующих этим путям, передается в *Anxiety* для построения соответствующих входных данных. *Anxiety* производит динамическое символьное выполнение только для указанных путей. После этого полученные входные данные передаются фаззингу. Если

переданные данные привели к выполнению целевых адресов, то направленный фаззинг прошел успешно. В противном случае, входные данные, которые увеличивают покрытие кода, добавляются в очередь фаззинга, и процесс повторяется.

В случае классического фаззинга, когда основным критерием является увеличение покрытия кода, процесс аналогичен направленному фаззингу. Единственное отличие заключается в списке базовых блоков, передаваемых в *Anxiety*. Статический анализ определяет список базовых блоков, обе ветви которых были выполнены, и передает их в *Anxiety* в виде "*черного*" списка. "*Черный*" список используется как метод оптимизации, во избежание обработки базовых блоков, обе ветви которых уже были выполнены.

В обоих случаях *DynamoRIO* сохраняет трассы выполнения целевой программы, чтобы использовать их для восстановления неявных переходов. Во время фаззинга периодически вызывается статический анализ, и база данных целевой программы обновляется на основе восстановленных адресов косвенных вызовов. Это позволяет взаимно улучшать статический и динамический анализ.

Статический анализ имеет две основные задачи: обнаружение путей в графе потока управления и анализ программных трасс. В первом случае, инструмент находит пути между двумя адресами целевой программы. Для обхода проблемы экспоненциального роста путей применяется ряд ограничений: максимальная длина пути, максимальное количество использований каждого базового блока или функции при построении одного пути и т.д. Построение путей состоит из двух основных этапов. Первый этап фильтрует некоторые функции на основе графа вызовов, которые не могут участвовать в построенных путях. Для функций, содержащих первый указанный адрес, производится прямой обход в ширину и строится множество всех доступных функций. Для функций, содержащих второй указанный адрес, производится обратный обход в ширину и строится множество всех доступных функций. Далее, производится пересечение полученных множеств, и поиск путей осуществляется только в их рамках. На втором этапе используется модифицированный *DFS* (*Depth First Search*) для обнаружения путей.

Во втором случае, статический анализ загружает набор трасс, сгенерированных инструментом фаззинга (*DynamoRIO*), и пытается найти все базовые блоки, обе ветви которых были выполнены. Для этого рассматриваются все базовые блоки из базы данных, содержащие условные инструкции перехода с явно указанными адресами. Далее, для каждого такого блока производится проверка в трассах программы на содержание обоих адресов перехода.

Экспериментальные результаты показали, что предложенный метод позволяет эффективно выявлять различные программные дефекты. В таблице 8 приводятся результаты фаззинга в направленном режиме. Статическим анализом определен набор адресов, который может содержать ошибку. Адреса переданы инструменту фаззинга с надеждой на то, что будут получены данные, приводящие к падению. В таблице 9 приводятся результаты фаззинга, где основная цель – увеличение покрытия кода. Все результаты вручную верифицированы.

Таблица 8. Найденные падения при направленном фаззинге.

Происхождения теста	***Имя теста***	***Падения***	***Покрыт***
Debian-6.0.10	faad	2	1/1
Debian-6.0.10	passwd	2	1/1
Debian-6.0.10	uuenview	13	1/1
DARPA	Flash_File_System	35	1/1
DARPA	3D_Image_Toolkit	30	1/1
DARPA	Charter	9	1/1
DARPA	Diary_Parser	9	1/1
DARPA	PRU	2	1/1
DARPA	Recipe_Database	23	1/1
DARPA	SCUBA_Dive_Logging	10	1/1
DARPA	SFTSCBSISS	1	1/1
DARPA	Simple_Stack_Machine	15	1/1
DARPA	CML	10	1/1
DARPA	Eddy	9	1/1
DARPA	FablesReport	3	7/15
DARPA	Multipass3	7	1/3
DARPA	Online_job_application	4	1/1
DARPA	Overflow_Parking	2	1/1
DARPA	PTassS	5	1/2
DARPA	Sample_Shipgame	5	2/2
DARPA	SAuth	1	1/3

Таблица 9. Найденные падения: критерия увеличение покрытия.

Происхождения теста	***Имя теста***	***Найденные падения***
Debian-6.0.10	blast2	3
Debian-6.0.10	faad	1
Debian-6.0.10	efax	1
Debian-6.0.10	wavpack	5
Debian-6.0.10	tic	4
Debian-6.0.10	ul	7
Debian-6.0.10	Bsd-form	6

В разделе 5.5 приводится описание разработанного автором *метода фаззинга интерфейсных функций. Сначала применяется статический анализ, который для интерфейсных функций определяет все смещения входного буфера, с которыми производится сравнение константных значений. Далле, полученная информация*

используется для оптимизации процесса фаззинга. В результате статического анализа выдается множество пар, состоящих из смещения и соответствующего константного значения, с которым производится сравнение. Статический анализ производится на базе промежуточного представления *LLVM* и графа вызовов. Инструмент фаззинга разработан на базе *libFuzzer*. Он на входе получает результаты статического анализа и использует их для мутаций данных. В ходе фаззинга смещениям входного буфера присваиваются соответствующие константные значения, а также значения, которые больше и меньше. В таблице 10 приводится сравнение разработанного инструмента с оригинальным *libFuzzer*. Во всех случаях, кроме "*dictionary_loader*" разработанный метод показывает лучшие результаты. Из таблицы также становится ясно, что эффективность инструмента гораздо выше при малых итерациях (10,000). Это дает большое преимущество при регулярном анализе больших проектов, где стоят временные ограничения.

Таблица 10. Сравнение разработанного метода с оригинальным libFuzzer.

Имя теста	**Количество итераций и прирост количества путей**			
	10k	**100k**	**500k**	**1m**
block_round_trip (zstd)	***+95%***	***+28%***	***+3%***	***0%***
decompress_dstSize_tooSmall (zstd)	***+51%***	***+17%***	***+0%***	***+1%***
dicionary_decompress (zstd)	***+44%***	***+25%***	***+28%***	***+3%***
dictionary_loader (zstd)	***+32%***	***+1%***	***+0%***	***-1%***
dictionary_round_trip (zstd)	***+51%***	***+46%***	***+2%***	***+1%***
dictionary_stream_round_trip (zstd)	***+1%***	***+0%***	***+1%***	***+1%***
huf_decompress (zstd)	***+18%***	***+5%***	***+3%***	***+3%***
raw_dictionary_round_trip (zstd)	***+42%***	***+18%***	***+13%***	***+1%***
simple_compress (zstd)	***+36%***	***+19%***	***+1%***	***+0%***
simple_decompress (zstd)	***+390%***	***+5%***	***+3%***	***+4%***
simple_round_trip (zstd)	***+36%***	***+21%***	***+1%***	***+1%***
zstd_frame_info (zstd)	***+16%***	***+0%***	***+0%***	***+0%***
fuzz_hpack_decode (haproxy)	***+0%***	***+1%***	***+4%***	***+8%***
ap-mgmt (hostap)	***+12%***	***+10%***	***+5%***	***+2%***
json (hostap)	***+7%***	***+5%***	***+5%***	***+3%***
x509 (hostap)	***+5%***	***+5%***	***+2%***	***+1%***
fuzz_parser (http-parser)	***+12%***	***+4%***	***+149%***	***+164%***
fuzz_url (http-parser)	***+1%***	***+0%***	***+0%***	***+0%***

В шестой главе приводится описание разработанной автором *архитектуры предлагаемой платформы, которая интегрирует множество разработанных методов анализа кода и позволяет их комбинированное применение с помощью доступного программного интерфейса*. Функциональные требования к платформе и ее архитектура разрабатывались с целью преодоления ограничений отдельных технологий анализа путем их единообразной комбинации в зависимости от

конкретной задачи. Также учитывалась возможность применения огромной базы доступного открытого ПО и информации об известных уязвимостях во время анализа. Для обеспечения функциональных требований, сервисы предлагаемой платформы разделены на две группы. Первая группа предназначена для сбора и пополнения базы данных артефактов ПО. В базе данных артефактов хранится информация о разных дистрибутивах ОС (*Debian*, *CentOS*, *FreeBSD*) и соответствующих им доступных пакетах. В последующем списке приводится набор хранимой информации:

1. Исходный код дистрибутивов ОС и всех доступных пакетов/проектов;
2. Артефакты сборки дистрибутивов и всех доступных пакетов/проектов;
3. Список доступных пакетов для всех версий дистрибутивов ОС;
4. ГЗП для исходного/бинарного кода дистрибутивов ОС и для всех доступных пакетов/проектов;
5. Отладочная информация для всех пакетов/проектов и дистрибутивов ОС;
6. Для каждого пакета зависимости для его сборки/запуска;
7. Для всех пакетов/проектов зависимости по коду – какие фрагменты/файлы совпадают 100%;
8. Команды сборки и компоновки для каждого доступного пакета/дистрибутива;
9. Список всех известных уязвимостей и набор фиксирующих исправлений.

Вторая группа сервисов предназначена для анализа проектов с комбинированным применением всех разработанных методов в рамках данной работы. Проектами могут являться добавленные в базу данных дистрибутивы ОС и их пакеты, а также другие проекты пользователя. Доступны следующие основные методы анализы:

1. Поиск информации о заданном файле/пакете/дистрибутиве ОС (можно получить любую информацию из базы данных, включая ГЗП);
2. Поиск клонов исходного и бинарного кода;
3. Сопоставления бинарных и исходных файлов;
4. Поиск неисправленных ошибок;
5. Поиск утечек памяти;
6. Фаззинг для бинарных артефактов собранных проектов (доступно множество вариантов запуска, в том числе с использованием данных от статических анализаторов).

Одним из ключевых моментов является *Python*-терминал, который предоставляет доступ ко всем базовым анализам через специально разработанный программный интерфейс. Он обеспечивает единообразный и гибкий подход передачи

данных, между разними инструментами анализа, средствами самого языка *Python*. С помощью *Python*-терминала возможно написать скрипты/плагины для комбинированного запуска нескольких анализов. Возможно комбинировать любое количество анализов под конкретную задачу.

В **разделе 6.1** приводится описание разработанного автором механизма сбора артефактов для дистрибутивов ОС, их пакетов и открыто доступного ПО. Сбор артефактов организован по модели сервер-клиент. Платформа позволяет использовать несколько серверов для параллельной генерации артефактов (только ОС *Debian* содержит более 50,000 пакетов, поддержка параллелизма необходима для повышения производительности). Каждый сервер содержит образы *QEMU* для поддерживаемых ОС. Процесс организован следующим образом:

1. Клиент отправляет список пакетов на главный сервер для сбора артефактов;
2. Главный сервер распределяет полученный список по доступным серверам;
3. На серверах запускается образ *QEMU* с запрашиваемой ОС;
4. На каждом сервере образ *QEMU* запускает сбор артефактов для полученного списка пакетов;
5. Каждый образ *QEMU* отправляет обратно сгенерированные артефакты;
6. Основной сервер сохраняет сгенерированные артефакты в базу данных.

Генерация артефактов для данного пакета на образе *QEMU* организована следующим образом:

1. Загрузка исходного кода пакета;
2. Установка всех зависимостей, необходимых для сборки пакета;
3. Генерация ГЗП для исходного кода пакета/проекта на основе промежуточного представления LLVM;
4. Перехват и сохранение команд сборки;
5. Сохранение отладочной информации при сборке;
6. Генерация ГЗП для полученных бинарных файлов пакета;
7. Сжатие всей информации и отправка обратно, на главный сервер.

Возможны случаи, когда некоторые пакеты из списка сборки несовместимы. Для решений подобных проблем запускаются чистые образы *QEMU*, на которых отдельно собираются конфликтующие пакеты. Для проектов пользователя и открыто доступного ПО должны быть представлены команды сборки, которые будут использованы для сбора артефактов. В настоящее время поддерживается генерация артефактов для архитектур *x86/64* и *ARM32/64*. Двоичные файлы *ARM32/64* создаются на образах *QEMU x86/64* с использованием кросс-компиляции.

В **разделе 6.2** описывается общая схема работы предлагаемой платформы. На рисунке 1 приведена схема взаимодействий компонентов. Первый пользовательский интерфейс предназначен для пополнения базы данных. Далее, доступны наборы анализов, которые можно запустить из второго пользовательского интерфейса или с соответствующим *Python API*. Пользователям доступен *Python*-терминал, из которого можно вызвать каждый компонент (соответствующие API функции) и, далее, обработать полученные результаты или передать их другим компонентам. *Python*-терминал – полнофункциональная среда *Python* с возможностью прямых запросов к базе данных.

В **разделе 6.3** приводится описание структуры базы данных. Используется мультимодальная система управления базами данных (СУБД), которая комбинирует документный и графовый подходы к хранению данных. На рисунке 4 приводится подробное описание структуры базы данных.

"**Коллекция дистрибутивов ОС**" представляет собой набор документов содержащих информацию о доступных дистрибутивах ОС. Каждый из этих документов одновременно является вершиной направленного графа. Из нее следует три ребра, ссылающихся на корневые узлы дерева: исходные файлы ОС, бинарные файлы ОС и пакеты, содержащихся в данном дистрибутиве ОС. Эти корневые узлы содержатся в коллекции "Коллекция подграфов ОС".

"**Коллекция пакетов ПО**" содержит набор пакетов ПО для всех дистрибутивов ОС. Эти документы содержат информацию о данном пакете, и одновременно, являются вершинами направленного графа, где из каждой вершины следует два ребра на корневые узлы дерева исходных файлов и бинарных файлов ПО. Эти корневые узлы содержатся в коллекции "Коллекция подграфов пакетов".

"**Коллекция файлов и директорий исходного/бинарного кода**" содержит набор документов, соответствующий директориям или исходным/бинарным файлам. Для каждого исходного/бинарного файла содержится список функций и строк/адресов кода, входящих в него. Вершины, соответствующие исходным/бинарным файлам или пустым директориям, являются листьями дерева. Вершины, соответствующие непустым директориям, содержат исходящие ребра на другие документы, которые, в свою очередь, соответствуют файлам и директориям, входящим в эту директорию.

Документы, соответствующие дистрибутивам ОС (из коллекции "Коллекция дистрибутивов ОС"), содержат следующую информацию:

- Путь к корневой директории дистрибутива ОС (содержит все файлы касательно данного дистрибутива);
- Путь к логу сборки ядра ОС для данного дистрибутива.

Документы, соответствующие пакету ПО (из коллекции "Коллекция пакетов ПО"), содержат следующую информацию:

- Путь к корневой директории пакета ПО (содержит все файлы, связанные данным пакетом ПО);
- Путь к логу сборки данного пакета ПО.

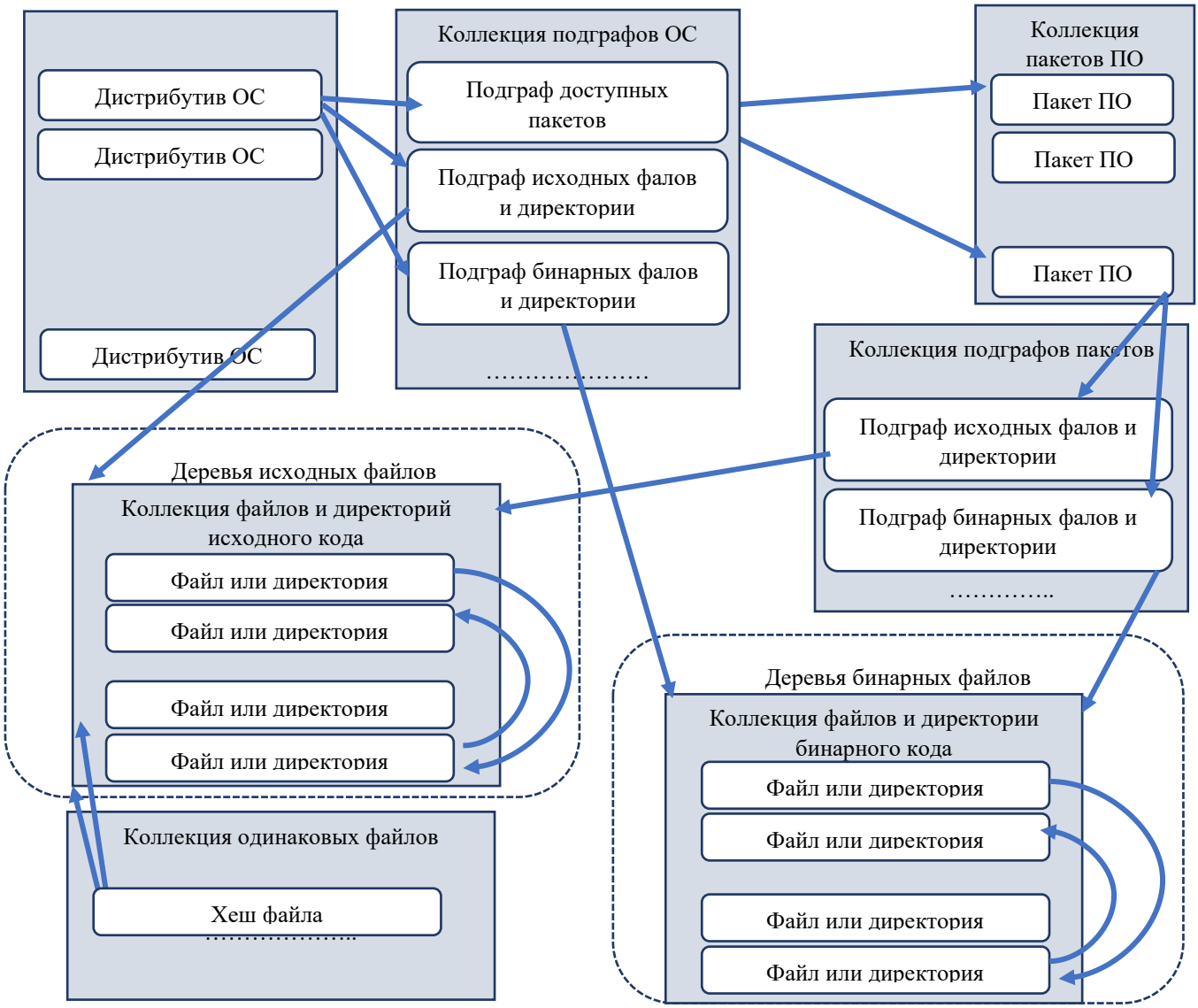

Рисунок 4. Структура базы данных

Документы из коллекции файлов и директории исходного кода содержат следующую информацию:

- Тип документа: исходный файл или директория;
- Имя файла или директории;
- Пути к соответствующим ГЗП файлам.

Документы из коллекции файлов и директории бинарного кода содержат следующую информацию:

- Тип документа: бинарный файл или директория;
- Имя файла или директории;
- Пути к соответствующим ГЗП файлам;
- Пути к файлам, содержащим отладочную информацию.

Документы из коллекции подграфов ОС содержат следующую информацию:

- Документы, соответствующие корневым вершинам "**Подграфов исходных/бинарных файлов и директории**", содержат абсолютный путь корневой директории дистрибутива ОС или пакета ПО.

В базу данных входит еще одна коллекция ("Коллекция одинаковых файлов") для сохранения информации о совпадающих файлах исходного кода в дистрибутивах ОС и пакетах ПО. Вершины коллекции "Коллекция одинаковых файлов" будут указывать на соответствующие документы из "Коллекция файлов и директорий исходного кода" (файлы, соответствующие этим документам, одинаковы).

В **разделе 6.4** приводится обоснование выбора СУБД. Разработка реляционных баз данных ведется в течение нескольких десятилетий, и такие базы (например, *Postgresq, SQL Server, Oracle*) используются в многочисленных системах. Эти СУБД отличаются высокой стабильностью, производительностью и отказоустойчивостью. Несмотря на эти достоинства, в ходе анализа было решено отказаться от реляционных баз данных по следующим причинам:

1. ***Фиксированная схема***. Разрабатываемая система должна позволить написать плагины, которые будут работать с базами данных. Эти плагины должны иметь возможность добавлять новые атрибуты разным сущностям, т.е. обновлять схему таблиц. Частое обновление схем таблиц может привести к непредсказуемым последействиям, более того, разработчики плагинов обязаны будут знать подробности работы реляционных баз данных, чтобы проводить обновление схем.
2. ***Изменения в схеме могут привести к нарушениям нормальной формы базы***. Обработка таких ситуаций нетривиальна и может привести к огромным затратам ресурсов.

Учитывая эти проблемы, было решено выбрать СУБД без фиксированной схемы.

Существуют различные решения для хранения данных без фиксированной схемы. В ходе детального анализа (подробно приводится в тексте диссертации) кандидатом была выбрана мульти-модельная система *ArangoDB*, которая хорошо

документирована и имеет свой собственный интуитивный язык запросов *AQL* (*Arango Query Language*).

В **разделе 6.5** приводится описание разработанного автором *Python*-терминала и реализованного программного интерфейса для запуска и доступа к результатам всех инструментов. *Python*-терминал представляет из себя полноценную среду *Python*, в которой доступны все базовые функциональности этого языка. Приводятся примеры использования доступного программного интерфейса, включая выполнение базовых анализов и их комбинацию.

В **разделе 6.6** приводится примеры использования доступного программного интерфейса.

В **разделе 6.7** представлены экспериментальные результаты, касающиеся сбора артефактов для дистрибутивов операционной системы *Debian* и более чем 50,000 соответствующих пакетов. Кроме того, приведены примеры комбинированного использования различных инструментов анализа с целью обнаружения сложных дефектов. В частности, были проведены следующие эксперименты: 1) совместное применение фаззинга и символьного выполнения для всех бинарных файлов, исходный код которых содержит известную уязвимость; 2) направленный фаззинг на адреса бинарного кода, соответствующие исходному коду с известными уязвимостями. В результате комбинированного применения указанных методов стало возможным найти ряд падений в старых версиях пакетов *Debian*.

В среднем, сбор базы данных для всех пакетов одного дистрибутива ОС *Debian* занимает порядка 40 часов на 100-ядерной машине (частота одного ядра ~2.3GHz). В результате комбинированного использования различных инструментов анализа только на пакетах ОС *Debian* стало возможным найти десятки копий известных уязвимостей и сгенерировать данные с помощью фаззинга, приводящие к падению этих пакетов.

В седьмой главе приводится описание практической значимости диссертационной работы. Приводится обобщение найденных ошибок в открытом ПО благодаря разработанным качественно новым методам анализа кода и удобной объединяющей платформе. Описывается критичность найденных ошибок в открытом ПО, которые в некоторых случаях обнаруживались и исправлялись в таких широко используемых проектах, что могло затронуть буквально всех, имеющих доступ к интернету. Количество найденных ошибок разного типа превышает девяносто, включая ошибки в таких широко используемых проектах, как *openssl*, *grub2*, *clib*, *ffmpeg*, платформа управления умными устройствами от "*Samsung Electronics Co. Ltd*"

и других. Десятки из этих ошибок были исправлены и представлены нами сообществу разработчиков открытого ПО.

В заключение формулируются основные результаты диссертационной работы и предлагаются направления дальнейших исследований.

ОСНОВНЫЕ РЕЗУЛЬТАТЫ РАБОТЫ

В результате проведения теоретических и практических исследований получены следующие результаты:

1. Разработана архитектура и реализован экспериментальный образец платформы анализа программ, обеспечивающая: сбор артефактов для большого объема открытого ПО и информации об известных уязвимостях; единообразный подход комбинирования различных методов анализа кода в зависимости от конкретной задачи.
2. Разработаны и реализованы масштабируемые и точные методы нахождения клонов кода, основанные на поиске схожих подграфов максимального размера для графов зависимостей программ, построенных на основе промежуточных представлений исходного и бинарного кода.
3. Разработан и реализован метод сопоставления исходных и бинарных файлов, который из входного исходного кода получает множество бинарных файлов, скомпилированных с разными уровнями оптимизации и содержащих отладочную информацию, после чего производит сопоставление инструкций полученных и выходных бинарных файлов на основе разработанного инструмента поиска клонов бинарного кода, а в конце выполняет сопоставление исходного кода с инструкциями входных бинарных файлов на основе отладочной информации сопоставленных бинарных инструкций.
4. Разработан и реализован метод поиска утечек памяти для языков Си/Си++, который на первом этапе производит поиск утечек на специальном представлении программы, содержащей поток управления и данных со смещениями доступа к указателям и полям структур, а на втором этапе производит проверку выполнимости путей ошибок методом направленного символьного выполнения.
5. Разработан и реализован метод фаззинга программ, генерирующий структурированные данные на основе специализированных автоматов БНФ грамматик, где веса автоматов динамическим образом меняются в процессе фаззинга, что обеспечивает адаптацию шаблонов генерируемых программ в зависимости от их эффективности для увеличения покрытия кода.

6. Разработан и реализован метод фаззинга интерфейсных функций, который позволяет генерировать цепочки вызовов функций и использовать возвращаемые значения одних функций в качестве аргументов для других, что обеспечивает возможность подготовки необходимых ресурсов для тестирования сложных сценариев использования нескольких функций в среде выполнения.
7. Разработан и реализован метод направленного фаззинга для быстрой генерации входных данных с целью выполнения конкретных инструкций или фрагментов целевой программы, содержащие потенциальные уязвимости или дефекты.
8. Разработан и реализован метод интеграции статического анализа с фаззингом, который применяет статический анализ для получения константных значений, используемых в условных операторах, и затем использует эти константы для генерации входных данных, покрывающих соответствующие ветви кода.
9. Разработанными методами и реализованной платформой проанализированы десятки тысяч проектов, суммарный объем которых превышает сотни миллионов строк исходного и соответствующего бинарного кода, что позволило найти более девяносто ошибок разного типа в открытом и проприетарном ПО, включая ошибки в такие широко используемых проектах, как *openssl*, *grub2*, *clib*, *ffmpeg*, платформа управления умными устройствами от "*Samsung Electronics Co. Ltd*" и других. Десятки из этих ошибок были исправлены и представлены нами сообществу разработчиков открытого ПО.
10. По данному направлению исследований, при непосредственном участии и под руководством автора, выполнены три кандидатские диссертации и более пяти магистерских и дипломных работ.

ПУБЛИКАЦИИ ПО ТЕМЕ ДИССЕРТАЦИИ

Статьи в журналах, рекомендованных ВАК РФ, Scopus, Web of Science

1. S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, S. Asryan, M. Mishechkin, T. Ghukasyan, "Grammar-Based Fuzzing", *Proceedings of Ivannikov Memorial Workshop (IVMEM)*, 2018.
2. S. Sargsyan, S. Asryan, J. Hakobyan, S. Kurmangaleev, "Combining dynamic symbolic execution, code static analysis and fuzzing", *Proceedings of the Institute for System Programming*, vol. 30, pp. 25-38, 2018.
3. S. Sargsyan, J. Hakobyan, M. Mehrabyan, M. Mishechkin, V. Akozin, S. Kurmangaleev, "ISP-Fuzzer: Extendable Fuzzing Framework", *Proceedings of Ivannikov Memorial Workshop*, 2019.
4. S. Sargsyan, S. Kurmangaleev, J. Hakobyan, M. Mehrabyan, S. Asryan, H. Movsisyan, "Directed Fuzzing Based on Program Dynamic Instrumentation", *Proceedings of International Conference on Engineering Technologies and Computer Science (EnT)*, 2019.

5. S. Sargsyan, J. Hakobyan, H. Movsisyan, S. Kurmangaleev, V. Sirunyan, M. Mehrabyan, "Improving fuzzing performance by applying interval mutations", *Proceedings of the Institute for System Programming*, vol. 31, № 1, pp. 78-88, 2019.
6. S. Sargsyan, V. Vardanyan, H. Aslanyan, M. Harutunyan, M. Mehrabyan, K. Sargsyan, H. Hovahannisyan, H. Movsisyan, J. Hakobyan, S. Kurmangaleev, "GENES ISP: Code analysis platform", *Proceedings of Ivannikov Ispras Open Conference*, 2020.
7. С. Саргсян, В. Варданян, Д. Акопян, А. Агабалян, М. Меграбян, Ш. Курмангалеев, А. Герасимов, М. Ермаков, С. Вартанов, "Платформа автоматического фаззинга программного интерфейса приложений", *Труды Института системного программирования РАН*, т. 32, № 2, с. 161-173, 2020.
8. S. Sargsyan, M. Tovmasyan, J. Hakobyan, H. Aslanyan, S. Kurmangaleev, "A framework for a systematic survey of known software defects", *Proceedings of Ivannikov Memorial Workshop*, 2021.
9. S. Sargsyan, J. Hakobyan, M. Mehrabyan, R. Mkoyan, V. Sahakyan, V. Melkonyan, M. Arutunian, A. Fahradyan, A. Avetisyan, "Advanced Grammar-Based Fuzzing", *Proceedings of Ivannikov Memorial Workshop*, 2022.
10. S. Sargsyan, J. Hakobyan, L. Nersisyan, K. Sargsyan and V. Melkonyan, "Improving fuzzing efficiency based on extracted constant values", *Proceedings of Ivannikov Ispras Open Conference*, 2022.
11. S. Sargsyan, H. Aslanyan, S. Asryan, J. Hakobyan, S. Kurmangaleev, V. Vardanyan, "Multiplatform Static Analysis Framework for Program Defects Detection", *Proccidings of Computer Science and Information Technologies*, 2017.
12. S. Sargsyan, S. Kurmangaleev, A. Belevantsev, A. Avetisyan, "Scalable and accurate detection of code clones", *Programming and Computer Software*, vol. 42, pp. 27-33, 2016.
13. С. Саргсян, Ш. Курмангалеев, А. Белеванцев, А. Асланян, А. Балоян, "Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ", *Труды Института системного программирования РАН*, т. 27, № 1, с. 39-50, 2015.
14. С. Саргсян, "Поиск семантических ошибок, возникающих при некорректной адаптации скопированных участков кода", *Труды Института системного программирования РАН*, т. 27, № 2, с. 93-104, 2015.
15. S. Sargsyan, "Improving Fuzzing Using Input Data Offsets Comparison Information", *Programming and Computer Software*, vol. 49, pp. 122-127, 2023.
16. A. Avetisyan, S. Kurmangaleev, S. Sargsyan, M. Arutunian, A. Belevantsev, "LLVM-based code clone detection framework", *Proceedings of Computer Science and Information Technologies (CSIT)*, 2015.
17. А. Асланян, Ш. Курмангалеев, В. Варданян, М. Арутюнян, С. Саргсян, "Платформенно-независимый и масштабируемый инструмент поиска клонов кода в бинарных файлах", *Труды Института системного программирования РАН*, т. 28, № 5, с. 215-226, 2016.
18. С. Асрян, С. Гайсарян, Ш. Курмангалеев, А. Агабалян, Н. Овсепян, С. Саргсян, "Обнаружение ошибок, возникающих при использовании динамической памяти после освобождения", *Труды Института системного программирования РАН*, т. 30, № 3, с. 7-20, 2018.

19. S. Asryan, S. Gaissaryan, S. Kurmangaleev, S. Sargsyan, A. Aghabalyan, N. Hovsepyan, "Dynamic Detection of Use-After-Free Bugs", *Programming and Computer Software*, vol. 45, № 7, pp. 365-371, 2019.
20. M. Arutunian, H. Hovhannisyan, V. Vardanyan, S. Sargsyan, S. Kurmangaleev, H. Aslanyan, "A Method to Evaluate Binary Code Comparison Tools", *Proceedings of Ivannikov Memorial Workshop*, 2021.
21. H. Aslanyan, H. Movsisyan, M. Arutunian, S. Sargsyan, "Bin2Source: Matching Binary to Source Code", *Proceedings of Ivannikov Ispras Open Conference*, 2021.
22. H. Aslanyan, Z. Gevorgyan, R. Mkoyan, H. Movsisyan, V. Sahakyan, S. Sargsyan, "Static Analysis Methods For Memory Leak Detection: A Survey", *Proceedings of Ivannikov Memorial Workshop*, 2022.
23. Ш. Курмангалеев, В. Корчагин, В. Савченко, С. Саргсян, "Построение обфусцирующего компилятора на основе инфраструктуры LLVM", *Труды Института системного программирования РАН*, т. 23, с. 77-92, 2012.
24. H. Aslanyan, H. Movsisyan, H. Hovhannisyan, Z. Gevorgyan, R. Mkoyan, A. Avetisyan, S. Sargsyan, "Combining Static Analysis with Directed Symbolic Execution for Scalable and Accurate Memory Leak Detection", *IEEE Access*, vol. 12, pp. 80128-80137, 2024.

Свидетельства о государственной регистрации программы для ЭВМ автора по теме диссертации

25. Курмангалеев Ш.Ф., Саргсян С.С., Варданян В.Г., Иванов Г.С., «Инструмент поиска клонов кода для C/C++ программ "CCD"». РФ Патент ЭВМ № 2019660800, 13.08.2019.
26. Курмангалеев Ш.Ф., Мишечкин М.В., Акользин В.В., Саргсян С.С., «Модуль направленного фаззинга программ для "ISP-Fuzzer"». РФ, свидетельства о государственной регистрации ЭВМ № 2019660716, 12.08.2019.
27. Курмангалеев Ш.Ф., Саргсян С.С., Асланян А.К., Иванов Г.С., «Инструмент поиска клонов кода для бинарных файлов "BINCCD"». РФ, свидетельства о государственной регистрации ЭВМ № 2019661048, 16.08.2019.
28. Курмангалеев Ш.Ф., Мишечкин М.В., Акользин В.В., Саргсян С.С., «Инструмент фаззинга программ "SP-Fuzzer"». РФ, свидетельства о государственной регистрации ЭВМ № 2019661047, 16.08.2019.
29. Курмангалеев Ш.Ф., Иванов Г.С., Варданян В.Г., Асланян А.К., Федотов А.Н., Саргсян С.С., «Инструмент анализа изменений между двумя версиями программы "patchAnalysis"». РФ, свидетельства о государственной регистрации ЭВМ № 2019661049, 16.08.2019.
30. Курмангалеев Ш.Ф., Саргсян С.С., Варданян В. Г., Асланян А. К., Акопян Д.А., Арутюнян М.С., Меграбян М.С., Мовсисян О.М., Саргсян К.Г., Оганесян Р.А., "ISP Genes". РФ, свидетельства о государственной регистрации ЭВМ № 2020663670, 30.10.2020.
31. Курмангалеев Ш.Ф., Асланян А.К., Арутюнян М.С., Оганесян Р.А., Варданян В.Г., Саргсян С.С, "LibraryIdentifier". РФ, свидетельства о государственной регистрации ЭВМ № 2021665076, 17.09.2021.