Государственное образовательное учреждение высшего профессионального образования Российско-Армянский (Славянский) университет

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

Диссертация

на соискание ученой степени доктора технических наук

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

Москва – 2024

Содержание

Введение

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 определение 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, содержащие 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 подходами.

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 теории компиляции и анализу программного кода. В том числе по материалам диссертации опубликовано 12 работ в изданиях, входящих в список изданий, рекомендованных ВАК РФ, 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.

В первой главе обосновывается актуальность исследований по обеспечению безопасности ПО. Рассматриваются области применения существующих методов и их ограничения, на основе чего определяются следующие основные направления исследования: создание платформы для интеграции различных инструментов анализа программ; создание средств поиска клонов в программных системах и базирующихся на них инструменты поиска копий известных уязвимостей, а также сопоставления

исходного и бинарного кода; создание средств поиска ошибок в программах, связанных с использованием форматных строк и динамической памяти, включая утечки памяти и использование освобожденной памяти; оптимизация методов фаззинга для различных сценариев использования. Далее, в данной главе анализируются требования и формулируется концепция платформы анализа программ и очерчиваются основные элементы архитектуры экспериментального образца платформы, обеспечивающей обработку большого объема открытого ПО, сбор артефактов из баз данных известных уязвимостей, единообразный подход комбинирования различных методов анализа кода – в зависимости от контекста конкретной задачи.

Во второй главе был осуществлен детальный обзор существующих методов статического и динамического анализа программ, которые являются ключом обеспечения безопасности. Проведен сравнительный анализ функциональных возможностей различных методов и инструментов. Были выявлены основные недостатки.

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

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

В пятой главе представлены описания разработанных методов фаззинга, включая их комбинацию с символьным выполнением и статическим анализом, а также методов фаззинга для программ, обрабатывающих структурированные данные.

Кроме того, рассматривается фаззинг интерфейсных функций, в том числе для платформы *интернета вещей*.

В шестой главе дается описание объединяющей платформы, которая интегрирует набор разработанных методов анализа кода, и с помощью разработанного программного интерфейса позволяет их комбинированное применение. Сервисы платформы распадаются на две группы. Первая предназначена для сбора и пополнения базы данных. В базе данных сохраняется информация о различных дистрибутивах ОС (*Debian*, *CentOS*, *FreeBSD*) и связанных с ними открытых пакетах, а также данные об известных уязвимостях и соответствующих исправлениях. Собранная информация включает в себя: исходный код, отладочную информацию и артефакты сборки дистрибутивов ОС и всех доступных пакетов; графы зависимостей программы (ГЗП) для исходного и бинарного кода дистрибутивов ОС и всех доступных пакетов; зависимости для сборки и запуска каждого пакета, а также команды сборки и компоновки. Вторая группа предназначена для анализа артефактов проектов, что, в частности, может выполняться в режиме непрерывной интеграции. Группа включает методы статического анализа для сопоставления исходных и бинарных файлов, поиска клонов кода, неисправленных уязвимостей, утечек динамической памяти и т.д. Также в группу входят методы динамического анализа, позволяющие комбинировать фаззинг с символьным выполнением и со статическим анализом, проводить эффективный фаззинг программ, работающих со структурированными данными.

В седьмой главе описывается практическая значимость 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 настолько широко

используемых проектах, что могло затронуть буквально всех, имеющих доступ к интернету.

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

1. Определение основных приоритетов исследования

В данной 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 автором 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. Полученные новые результаты в рамках исследований в данной главе выносятся на защиту в виде архитектуры и экспериментального образца платформы*.

ПО нарастающими темпами интегрируется в каждый аспект современной жизни – от персональных устройств до критических инфраструктур, функционирующих на базе внедренных программных систем. ПО постоянно развивается, и, вместе с ним, развиваются угрозы безопасности, связанные с ростом объема и сложности систем. Злоумышленники используют сложные методы для эксплуатации уязвимостей в программных системах, начиная от фишинговых атак до эксплуатации уязвимостей нулевого дня (*0-Day*). Изучив ежегодные отчеты *NIST* [32] можно заметить, что количество ежегодно найденных уязвимостей за последние десять лет выросло от 8,000 до более чем 28,000. И это с учетом того, что количество инструментов анализа кода увеличивается параллельно с качеством существующих инструментов.

Крупные компании индустрии понимают важность разработки безопасного ПО, и многие из них предлагают отдельные решения, а также платформы для анализа открытого ПО. Примером является платформа *CodeQL* [33] от компании *GitHub* [34],

которая позволяет выполнять различные запросы к коду и выявлять уязвимости. *CodeQL* обрабатывает код как данные, что позволяет моделировать уязвимости и ошибки в виде запросов к базам данных, извлеченным из кода. Этот подход позволяет использовать как стандартные запросы, созданные исследователями *GitHub* и сообществом, так и собственные запросы для специализированного анализа. Базы данных *CodeQL* содержат: абстрактное синтаксическое дерево, графы потока данных и управления. Они предоставляют полное представление кода и служат основой для проведения анализа. Компания *Google* предлагает проект *OSS-Fuzz* [35], который позволяет выполнять фаззинг на кластерных мощностях для открытого ПО. Проект *OSS-Fuzz* был запущен в 2016 году, после того, как уязвимость "*Heartbleed*" была обнаружена в *openssl* [36] – одном из самых популярных проектов с открытым исходным кодом для шифрования веб-трафика. Уязвимость могла затронуть потенциально каждого интернет-пользователя, и была вызвана относительно простой ошибкой переполнения буфера памяти, которую возможно было обнаружить с помощью фаззинга. Однако в то время фаззинг не был широко распространен среди разработчиков, так как не был достаточно автоматизирован. *Google* создал проект *OSS-Fuzz*, чтобы восполнить этот пробел. *OSS-Fuzz* запускает фаззеры для проектов с открытым исходным кодом и оповещает разработчиков об обнаруженных ошибках. С момента своего запуска *OSS-Fuzz* стал критически важным сервисом для сообщества открытого исходного кода, выйдя за рамки *Cu/Cu++* и обнаружив проблемы в языках безопасных для памяти – таких, как *Go*, *Rust* и *Python*. Более 22,000 [37] ошибок уже обнаружены благодаря этому проекту.

Помимо широкого спектра инструментов, необходимо также применять правильную методологию обеспечения безопасности ПО, начиная с этапа проектирования и разработки. Для этого применяются методы безопасного цикла разработки ПО [38]. В данном случае, одним из ключевых компонентов обеспечения комплексной безопасности является набор используемых качественных инструментов

анализа кода. Но, несмотря на большое количество доступных инструментов для различных задач анализа ПО, остается множество нерешенных вопросов, что напрямую влияет на безопасность. Методы статического анализа не всегда учитывают информацию об известных уязвимостях в открытом ПО. Это может привести к ситуациям, когда в проекте используется сторонний проект с известной уязвимостью. Ежегодно обнаруживаются десятки уязвимостей нулевого дня, которые становятся причинами миллионов хакерских атак. Согласно отчету *Google Project Zero* [39], в 2021 году были найдены более 50-ти таких уязвимостей, которые содержатся в разных ОС и браузерах, потенциально распространяясь на миллионы пользователей. Не существуют качественных и точных инструментов поиска клонов кода, включая и клонов известных уязвимостей. Такие инструменты имеют широкое применение во множестве задач, включая поиск вирусов, старых версий библиотек, клонов кода с ошибками, и т.д. Также отсутствуют высококачественные инструменты сопоставления исходного и бинарного кода, что могло бы решить множество задач обратной инженерии, возникающих при анализе кода. Не существуют качественных инструментов поиска утечек динамической памяти и ошибок форматной строки, доказательством чего является ежегодное обнаружение сотен таких уязвимостей [40], [41]. Методы фаззинга сталкиваются с серьезными ограничениями при генерации структурированных данных и цепочек вызовов системных/библиотечных функций. Это ограничивает получение высокого покрытия кода во время тестирования. Согласно анализу результатов [42] проекта *OSS-Fuzz*, в среднем, покрывается меньше 40% кода. Кроме этого, не существует платформы, в рамках которой можно было обеспечить взаимодействие нескольких инструментов анализа, а также обеспечить сбор, хранение и удобное использование артефактов большого объема открытого ПО. С учетом вышесказанного, были выбраны следующие основные направления исследования:

- Создание платформы для интеграции различных инструментов анализа программ;
- Создание средств поиска клонов в программных системах и базирующихся на них инструменты поиска копий известных уязвимостей, а также сопоставления исходного и бинарного кода;
- Создание средств поиска ошибок в программах, связанных с использованием форматных строк и динамической памяти, включая утечки памяти и использование освобожденной памяти;
- Оптимизация методов фаззинга для различных сценариев использования.

1.1. Платформа анализа

В рамках данного направления были проанализированы функциональные требования и проведены исследования, по результатам которых автором была 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, в отдельности, не удавалось найти каждым из методов. 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 является поиск копий известных уязвимостей в сочетании с направленным фаззингом. Этот подход привел к обнаружению ряда ошибок в пакетах операционной системы *Debian* (Раздел 5.4.2), что подтверждает эффективность комбинирования различных методов анализа.

1.1.1. Требования к платформе и архитектура платформы

Существует множество платформ для анализа кода, но обычно они сталкиваются с двумя основными ограничениями. Первое ограничение заключается в наличии ограниченного количества однотипных методов анализа (статического или динамического). Второе ограничение состоит в том, что доступные методы можно применять либо по отдельности, либо в рамках фиксированной схемы комбинаций. Таким образом, единообразная комбинация нескольких методов анализа, в зависимости от конкретной задачи, недоступна, что на практике ограничивает количество обнаруженных ошибок. Как уже отмечалось выше, платформа статического анализа *CodeQL* [33] от компании *GitHub* [34] обрабатывает код в качестве данных и позволяет выполнять различные запросы к коду для выявления уязвимостей. Несмотря на то, что можно написать собственные запросы для специализированного анализа, функциональные возможности платформы на этом ограничиваются, и, к примеру, нельзя применять методы динамического анализа. Другой платформой анализа кода, схожей с *CodeQL*, является *Joern* [43], который также ограничивается статическим анализом. Основное отличие *Joern* от *CodeQL* заключается в более высоком уровне абстракций запросов. Платформа *S2E* [44] нацелена на динамический анализ программ путем символьного выполнения и анализа свойств полученных путей. Не поддерживаются методы статического анализа и единообразная комбинация различных методов анализа. Кроме упомянутых, существует еще десятки аналогичных платформ с подобными ограничениями, включая как открытые, так и коммерческие решения. Из коммерческих стоит отметить *Coverity* [45] и *Svace* [46], которые являются индустриальными стандартами для статического анализа в жизненном цикле разработки безопасного ПО. В обоих инструментах доступен фиксированный набор детекторов для различных типов ошибок.

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 открыто-доступного 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 иные проекты пользователя.

Для выбора способа реализации единообразного комбинирования нескольких методов анализа необходимо учесть несколько важных особенностей. Во-первых, входные данные и результаты различных методов анализа могут в значительной мере отличаться как по структуре, так и по содержанию. Это значит, что при передаче результатов одного метода в качестве входных параметров другому может потребоваться предварительная обработка и конвертация данных. Во-вторых, количество и подходы существующих методов в предлагаемой платформе не ограничиваются только нашими разработками; они могут быть расширены за счет других инструментов анализа. 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. 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 посредством специально разработанных программных интерфейсов. *Python*-терминал обеспечивает единообразный и гибкий подход передачи данных, между разными инструментами анализа, средствами самого языка *Python*. С помощью *Python*-терминала можно написать скрипты/плагины для комбинированного запуска нескольких инструментов анализа. Возможно комбинировать любое количество инструментов анализа для конкретной задачи.

На рисунке 1 приводится схема работы предлагаемой платформы. Как показано на рисунке, 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 программы 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 созданных методов может быть запущен отдельно через пользовательский интерфейс. Во второй компонент 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 главе 6 приводится более детальное описание и реализация предложенной платформы.

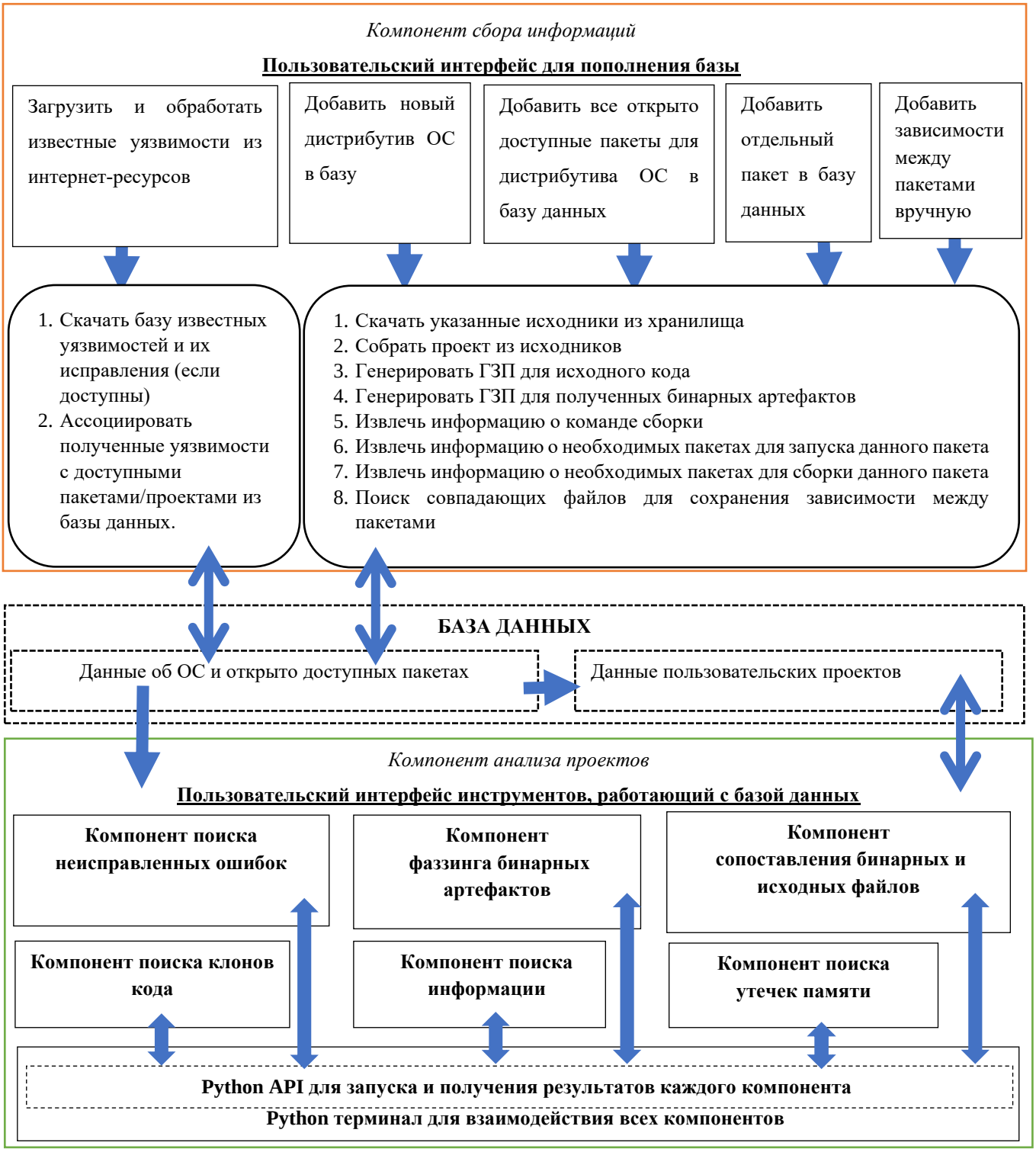

Рисунок 1. Схема работы предлагаемой платформы.

1.2. Направление поиска клонов кода и базирующееся на них методы

Поиск клонов (похожих фрагментов программного кода) выполняется с различными целями, например, для выявления случаев некорректного заимствования, что может привести к нарушению авторских или имущественных прав. Копирование фрагментов кода с уязвимостями может привести к многократному тиражированию одной ошибки. В контексте задач информационной безопасности поиск клонов может быть применен для поиска копий фрагментов кода, в которых содержатся известные уязвимости. Кроме того, он может быть использован для поиска использования старых версий открытого ПО в своих проектах.

В рамках этого направления автором разработаны методы поиска клонов исходного и бинарного кода. На базе этих методов разработаны инструменты поиска копий известных уязвимостей, сопоставления исходного и бинарного кода.

1.2.1. Методы решения

Для точного поиска клонов кода было использовано графовое представление программ, содержащих информацию о потоке данных и управления в виде ребер. Инструкции отображены на вершинах графа. Задача поиска клонов кода рассматривается как поиск максимально схожих подграфов в паре графов, соответствующих целевым фрагментам кода. В некоторых задачах (раздел 3.4) поиска клонов первого типа (T1 определение приводится в разделе 2.1) также использован лексический подход – для достижения максимальной производительности. Предложение данных методов обусловлен предварительным исследованием, проведенной в разделах 2.1 и 2.2. Эффективность предложенных методов доказывается экспериментальными результатами, приведенными в разделах 3.1 и 3.2. Инструменты поиска клонов кода, разработанные на основе предложенных методов, превосходят по качеству существующие инструменты.

Для поиска неисправленных уязвимостей регулярно отслеживаются интернет-ресурсы с целью сбора всей доступной информации о них, включая доступные исправления. Затем производится поиск точных клонов (T1) уязвимых фрагментов кода в целевом проекте. Предлагаемый метод решения был разработан с учетом предварительных исследований, проведенных в разделе 2.3. В разделе 3.4 представлены экспериментальные результаты разработанного инструмента, основанного на предложенном методе. С помощью последнего стало возможным найти множество реальных ошибок в открытом ПО, которые были подтверждены и исправлены сообществом разработчиков. Так, полученные результаты доказывают эффективность предложенного метода.

Проблема сопоставления исходного и бинарного кода решается следующим образом. Сначала компилируется исходный код в бинарные файлы. Далее, используется метод поиска клонов бинарного кода для полученных и целевых бинарных файлов. Это позволяет произвести сопоставления бинарных функций и инструкций. Затем сопоставляются инструкции бинарного кода со строками исходного кода на основе сопоставленных пар бинарных инструкций и отладочной информации. Предлагаемый метод решения был разработан с учетом предварительных исследований, описанных в разделе 2.4. Эффективность предложенного метода доказывается результатами реализованного инструмента, приведенными в разделе 3.3. Разработанный инструмент обеспечивает большее количество сопоставленных функций исходного и бинарного кода по сравнению с доступными аналогами.

1.3. Направление разработки методов поиска разных видов ошибок

В рамках этого направления автором разработаны методы поиска ошибок, связанные с использованием форматной строки и динамической памяти, включая утечки памяти и использование освобожденной памяти.

1.3.1. Методы решения

Для решения вышеуказанных проблем были разработаны, реализованы и протестированы три основных метода. Первый метод основан на статическом анализе и, как правило, использует графовое представление программы, содержащее поток управления и поток данных. Детекторы статического анализа проверяют разные свойства графа с целью определения ошибок. Второй метод базируется на динамическом символьном выполнении. Реализованный детектор проверяет некоторые свойства конкретно выполненных путей для поиска ошибок. Третий метод комбинирует статический анализ с символьным выполнением для достижения большей точности и масштабируемости одновременно. Предлагаемые методы решения были разработаны с учетом предварительных исследований проведенных в разделах 2.5 и 2.6. Результаты тестирования реализованных методов приводятся в разделах 4.1, 4.2 и 4.3. В ходе экспериментальных исследований комбинированный метод поиска утечек памяти показал наилучшие результаты, обойдя также существующие методы. С помощью него были найдены реальные ошибки в открытом ПО, которые были подтверждены и исправлены сообществом разработчиков.

1.4. Направление оптимизаций методов фаззинга

В рамках этого направления автором реализованы пять методов оптимизации фаззинга программ для разных сценариев анализа.

1.4.1. Методы решения

Предлагаемые методы были реализованы с учетом недостатков существующих инструментов фаззинга, которые детально исследованы в разделе 2.7.

Метод генерации сложно структурированных данных выполняет генерацию входных данных на основе формального описания БНФ (Бэкуса-Наура форма) грамматики. Это позволяет успешно пройти стандартные проверки корректности

входных данных и проанализировать более глубокие слои выполняемой программы. Реализация и тестирование предлагаемого метода описаны в разделе 5.1. Согласно результатам тестирования, приведенным в разделе 5.1.5, разработанный метод обеспечивает прирост покрытия кода на множестве известных компиляторах/интерпретаторах по сравнению с оригинальной версией инструмента фаззинга. Во время фаззинга *gcc-12* и *clang-14* (на момент тестирования последние версии) предложенный метод смог найти входные данные, приводящие оба компилятора к падению. В обоих случаях инструмент фаззинга эксплуатировал ограничения алгоритма синтаксического анализа рекурсивного спуска, превысив лимит стека, по умолчанию, для компилятора. Полученные результаты доказывают эффективность предложенного метода.

Метод фаззинга интерфейсных функций позволяет генерировать цепочки вызовов функций и использовать возвращаемые значения одних функций в качестве аргументов для других. Этот метод позволяет выявить ошибки в сложных сценариях, когда несколько функций должны участвовать для воспроизведения дефектов. Реализация и тестирование предлагаемого метода описаны в разделе 5.2. Согласно результатам тестирования, приведенным в разделе 5.2.9, предложенный метод на библиотеке *PluginBase*, входящей в состав платформы *SmartThings* (от "*Samsung Electronics Co. Ltd*"), нашел 15 уникальных ошибок, которые были подтверждены командой тестирования "*Samsung Electronics Co. Ltd*". Полученные результаты доказывают эффективность предложенного метода.

Метод направленного фаззинга предназначен для быстрой генерации входных данных с целью выполнения конкретных инструкций или фрагментов целевой программы. Данный метод полезен, когда из статического анализа получены фрагменты кода, которые могут содержать уязвимости/дефекты, и необходимы актуальные входные данные для покрытия этих фрагментов. Реализация и тестирование предлагаемого метода описаны в разделе 5.3. Согласно результатам

тестирования, приведенным в разделе 5.3.3, предложенный метод запущен в направленном режиме с известными адресами дефектов для всех целевых программ. В результате анализа, предложенный метод находит большее количество падений за тот же период фаззинга, что доказывает его эффективность.

Метод интеграции статического анализа и динамического символьного выполнения с фаззингом служит для улучшения покрытия кода. Динамическое символьное выполнение и статический анализ встроены в фаззинг, что позволяет итеративно улучшать результаты каждого компонента. Во время фаззинга восстанавливаются неявные вызовы функций, и данная информация используется статическим анализом, что улучшает обнаружение некоторых непокрытых путей в графе потока управления программы. А динамическое символьное выполнение, в свою очередь, генерирует входные данные для полученных путей, которые могут обеспечивать их выполнение. Сгенерированные данные используются фаззингом для увеличения покрытия кода. Реализация и тестирование предлагаемого метода описаны в разделе 5.4. Согласно результатам тестирования, приведенным в разделе 5.4.2, предложенный метод находит ошибки в реальных программах, которые не находятся оригинальным инструментом фаззинга за такой же период анализа.

Метод интеграции статического анализа с фаззингом предназначен для улучшения качества сгенерированных входных данных для интерфейсных функций. Статический анализ определяет все смещения входного буфера, с которыми и производится сравнение константных значений. В результате, статический анализ выдает множество пар, состоящих из смещения и соответствующего константного значения, с которыми и производится сравнение. В ходе фаззинга смещениям входного буфера присваиваются соответствующие константные значения, а также значения, которые больше и меньше. Это позволяет покрывать обе ветки инструкций сравнения в которых производятся сравнения со смещениями входного буфера. Реализация и тестирование предлагаемого метода описаны в разделе 5.5. Согласно

результатам тестирования, приведенным в разделе 5.5.4, предложенный метод обеспечивает большее покрытие кода по сравнению с оригинальным инструментом за одинаковые итерации фазинага.

1.5. Возможные направления будущих исследовании

Концепция платформы анализа программ, позволяющая собирать артефакты для больших объемов открытого ПО и информацию об известных уязвимостях, а также единообразным подходом комбинировать различные методы анализа кода, включая разработанные нами методы сопоставления исходных и бинарных файлов, поиска утечек памяти и фаззинга программ для разных сценариев, предоставляет новые возможности для анализа ПО и открывает потенциально новые направления для исследований. В частности:

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.

Необходимо также отметить, что направление анализа безопасности ПО и в дальнейшем будет активно развиваться, а его важность только будет возрастать с учетом внедрения ПО в каждый аспект нашей жизни. Новые технологии, включая машинное обучение и искусственный интеллект, откроют новый спектр задач безопасности ПО, решение которых потребует все новых и новых подходов.

2. Обзор целевых методов статического и динамического анализа

В данной главе осуществляется детальный обзор существующих методов статического и динамического анализа программ, которые являются ключом обеспечения безопасности. Производится сравнительный анализ функциональных возможностей различных методов и инструментов. Выявляются основные недостатки.

2.1. Методы поиска клонов исходного кода

Существуют четыре основных типа клонов исходного кода [47]:

Тип 1 (T1). Фрагменты кода, которые могут отличаться только пробелами, комментариями и форматированием кода;

Тип 2 (T2). Все клоны типа **T1**, а также фрагменты кода, которые могут различаться именами переменных, типами переменных, значениями переменных и констант.

Тип 3 (T3). Все клоны типа **T2**, а также фрагменты кода, где могут быть добавлены или удалены инструкции и переменные.

Тип 4 (T4). Неодинаковые фрагменты кода, которые выполняют одинаковые вычисления. Примером **T4** может быть рекурсивная и итеративная реализация одного и того же алгоритма. Некоторые авторы предполагают, что некое подмножество клонов типа **T3** может быть клоном типа **T4** (например, *a = b * 2* и *a <<= 1*).

В таблице 1 приводится пример клонов **T1-T3**.

2.1.1. Основные подходы поиска клонов исходного кода

Для поиска клонов типа **T1-T3** существуют пять основных подходов [48] (*текстовый*, *лексический*, *синтаксический*, *семантический* и *метрический*). Каждый подход имеет свои преимущества и ограничения. В зависимости от решаемой задачи и предъявляемых требований, выбирается конкретный подход поиска клонов кода, который позволяет оптимальным образом решить задачу. Существуют

комбинированные методы поиска клонов кода, состоящие из использования разных подходов на разных фазах поиска. В последние годы также становится популярным применение машинного обучения в разных фазах поиска клонов кода.

Таблица 1. Клоны типа T1, T2 и T3.

Оригинальный код	**Клон типа T1**

*void dumF (int n, float *F) {*
float sum = =0.0;
for (int i = 0; i < n; i++) {
sum = sum + F[i];
}
}
	*void dumF (int n, float *F) {*
	float sum = =0.0; // Комм.
	for (int i = 0; i < n; i++) {
	sum = sum + F[i];
	}
	}

Клон типа T2	**Клон типа T3**

*void sumI (int n, int *F) {*
int sum = =0; // Комм.
for (int i = 0; i < n; i++) {
sum = sum + F[i];
}
}
	*void prodI (int n, int *F) {*
	int prod = 1; // Комм.
	for (int i = 0; i < n; i++) {
	*prod = prod * F[i];*
	}
	}

В таблице 2 приводятся примеры клона типа **T4.**

Таблица 2. Клоны типа T4.

Рекурсивное вычисление факториала	**Итеративное вычисление факториала**
int factorial (int n) { *if (n > 1)* *return n * factorial (n - 1);* *else* *return 1;* *}*	*int factorial (int n) {* *int fact = 1;* *for (int i = 1; i <=n; ++i) {* *fact *= i;* *}* *return fact;* *}*

2.1.1.1. Текстовой подход поиска клонов кода

Методы на основе текстового подхода [49-64] рассматривают исходный код программы в качестве текста. Задача приводится к поиску совпадающих подстрок в

тексте. Существующие инструменты основаны на сравнении строк, подстрок, отпечаток кода и преобразований с помощью языка *TXL* [51] фрагментов кода. *sif* [64] находит файлы, имеющие совпадающие части. Сравниваются выбранные подмножества строк (отпечатки файлов) из этих файлов. Инструмент может найти точно скопированные файлы и файлы, у которых общая часть составляет всего 25%. *Johnson* [55-57] определяет понятие отпечатка (fingerprint) фрагмента кода. Инструмент, вместо построчного сравнения фрагментов кода, сравнивает их отпечатки.

Недостатки: клоны **T3** не могут быть найдены, клоны типа **T2** находятся частично. Переименованные и измененные типы (клоны **T2**) могут быть и не найдены. Методы не умеют определять границы функций, и клон может содержать фрагменты нескольких функций.

2.1.1.2. Лексический подход

Инструменты поиска клонов кода на основе лексического анализа сначала получают из кода последовательность токенов, после чего производится поиск совпадающих подпоследовательностей токенов.

Инструмент *CloneDetective* [65] производит поиск клонов кода с использованием суффиксного дерева. Вначале из кода получается последовательность токенов, которые нормализуются, и на их основе строится суффиксное дерево. Далее, для каждого суффикса клон определяется применением расстояния редактирования на суффиксном дереве. *CloneDetective* также позволяет определять скопированные фрагменты кода, содержащие ошибки копирования.

Инструменты на основе лексического анализа могут с высокой точностью находить все клоны типов **T1** и **T2**.

Недостатки: инструменты поиска клонов кода на основе лексического анализа находят клоны типа **T3** с низкой точностью (находятся не все клоны типа **T3**).

Например, в случае, когда была произведена перестановка некоторых инструкций в копированном фрагменте кода, лексический подход является неподходящим.

2.1.1.3. Синтаксический подход

Инструменты на основе синтаксического анализа используют дерево разбора (*parse tree*) или абстрактное синтаксическое дерево (АСД, *abstract syntax tree – AST*). Построение этих структур производится синтаксическим анализатором, который открыто доступен во многих компиляторах.

Falke [66] и *Tairas* [67] ищут изоморфные поддеревья АСД. *Falke* сначала находит клоны типа **T1** и **T2**. Далее он находит клоны типа **T3** путем объединения совпадающих поддеревьев АСД. [66] и [67] имеют высокую точность только при поиске клонов кода типов **T1** и **T2**. Они могут пропускать некоторые клоны типа **T3**, если структура АСД сильно изменяется. В обеих работах не производится тестирование на масштабных проектах (сотни тысяч или миллионов строк кода).

ClemanX [68-69] находит клоны кода путем сравнения характеристических векторов поддеревьев АСД. При каждом обновлении исходного кода производится обновление пар найденных клонов. Это делается следующим образом: для каждого добавленного фрагмента кода строится АСД и производится поиск изоморфных поддеревьев. При удалении конкретного фрагмента кода производится аналогичная операция. Строится АСД для удаленного фрагмента кода и убираются все пары клонов, которые содержат построенное АСД.

Chilowicz [70] вычисляет отпечатки для каждой вершины АСД таким образом, чтобы отпечаток вершины мог бы быть однозначно получен из отпечатков дочерних узлов. Из отпечатка можно восстановить поддерево. Все отпечатки хранятся в базе данных. Точно совпадающие поддеревья находятся с помощью запросов к базе отпечатков.

Недостатки: инструменты поиска клонов кода на основе синтаксического анализа находят клоны типа **Т3** с низкой точностью. Причина заключена в том, что добавленные и удаленные инструкции могут повлиять на структуру АСД значительным образом.

2.1.1.4. Семантический подход

Инструменты поиска клонов кода на основе семантического анализа в основном используют граф зависимостей программы (ГЗП). Вершинами ГЗП являются инструкции программы. Ребра между вершинами отображают поток управления и данных программы. Из вышесказанного следует, что ГЗП содержит всю информацию о семантике и структуре функций. Это позволяет производить более точный анализ программ. Методы, работающие на основе семантического подхода, осуществляют поиск схожих или изоморфных подграфов в каждой паре ГЗП.

Krinke [71] производит поиск схожих подграфов ГЗП, и клонами считаются соответствующие фрагменты кода. Вначале выбираются идентичные вершины ГЗП, которые представляют предикатные выражения программы. Эти вершины рассматриваются как начальные схожие подграфы. Далее, они расширяются путем добавления новых идентичных смежных вершин. Метод реализован для программ, написанных на ANSI-C. Предложенный подход обеспечивает высокую точность, но не масштабируется. Согласно результатам авторов, анализ 4000 строк кода может занять больше часа.

Komondoor [72] считает клонами фрагменты исходного кода программы, соответствующие изоморфным подграфам ГЗП. Для поиска изоморфных подграфов ГЗП используется обратный "*слайсинг*". Инструмент не масштабируется, согласно результатам авторов. Для анализа 12.000 строк кода понадобилось более 1.5 часа.

Higo [73] сначала хеширует вершины ГЗП на основе соответствующего исходного кода. После этого выбирается пара вершин с одинаковыми хешами и

применяется прямой/обратный "*слайсинг*". Если в ходе "*слайсинга*" встречаются вершины с одинаковыми хешами, то они добавляются в соответствующие подмножества изоморфных вершин. *Krinke*, *Komondoor*, *Higo* находят клоны типа **T1**, **T2**, **T3** с высокой точностью, но имеют большую вычислительную сложность: $O(N^3)$, где N – число вершин ГЗП. Это не позволяет использовать указанные инструменты для анализа программ, содержащих миллионы строк кода.

GPLAG [74] производит поиск плагиата в исходном коде программы. Он состоит из двух этапов. На первом этапе применяются алгоритмы с малой сложностью для фильтраций несхожих/неизоморфных пар ГЗП. На втором этапе производится поиск изоморфных ГЗП. *GPLAG* находит все три типа клонов кода. Он работает быстрее (обрабатывает 18.000 строк кода за ~20 минут), чем инструменты *Krinke*, *Komondoor* и *Higo*, благодаря применению фильтров. Несмотря на это, *GPLAG* не может анализировать большие проекты с миллионами строк исходного кода.

Gabel [75] преобразует ГЗП в дерево и применяет *DECKARD* [76-77] для поиска изоморфных поддеревьев. Инструмент масштабируется (может анализировать ядро Linux) за счет потери точности найденных клонов кода. Не все клоны **T3** могут быть найдены (при преобразовании ГЗП в АСД теряется информация). Также *Gabel* имеет высокий уровень ложных срабатываний. Кроме вышеуказанных недостатков, инструмент также долго генерирует ГЗП. Например, генерация ГЗП для ядра *Linux* занимает ~5 часов. Поиск клонов кода производится достаточно эффективно – занимает ~10 минут.

Недостатки: инструменты поиска клонов кода на основе семантического анализа работают сравнительно медленно по отношению к остальным методам (текстовый, лексический, синтаксический).

2.1.1.5. Метрический подход

Методы на основе метрического подхода вычисляют множество метрик для фрагментов кода и сравнивают полученные метрики. Как правило, метрики вычисляются для АСД или ГЗП соответствующих данному фрагменту кода. *Mayrand* [78] транслирует исходный код в АСД-представление и вычисляет четыре метрики на основе имен переменных программ, а также размещения исходного кода в файле, использованных инструкций и потока управления программы.

В работе *Kodhai* [79] сначала все файлы объединяются в один. После этого применяется стандартизация исходного кода, которая подразумевает удаление комментариев, пробелов и команд препроцессора. Вычисляются семь метрик для каждой функции, на основе которых производится поиск клонов кода в рамках функций. Метрики включают количество непустых строк исходного кода без комментариев, вызовов функций, аргументов функции, инструкции условий, локальных переменных, инструкции возвратов из функции, инструкции циклов. Как правило, инструменты на основе метрического подхода масштабируются и обладают высокой производительностью.

Недостатки: инструменты поиска клонов кода на основе метрического подхода обладают низкой точностью. Низкая точность в случае клонов типа **T3** более выражена, чем в случаях клонов типа **T1**, **T2**.

2.1.1.6. Комбинированные подходы

Chilowicz [80] производит поиск схожих функций. Для этого используются лексический анализ, граф вызовов функций и метрики. Сначала производится лексический анализ и получается последовательность лексем. На лексемах строится суффиксное дерево и определяются совпадающие участки разных функций. На основе полученных совпадающих последовательностей лексем каждая функция разделяется

на подфункции (факторизуется). Далее, строится граф вызовов программы, и на основе специальных метрик определяется схожесть функций.

Sheneamer [81] из АСД и ГЗП каждой функций получает множество свойств (32 – синтаксический, 28 – семантический), которые сохраняются в соответствующих векторах. Далее, применяется машинное обучение для классификации полученных векторов (схожих пар функций).

Oreo [82-83] сначала производит предобработку исходных файлов и строит вектора метрик для каждого метода. После этого применяется машинное обучение для классификации схожих методов на основе соответствующих векторов. Для обучения сети используется набор клонов, полученный инструментом *SourcererCC* [84-85].

2.1.2. Обзор доступных инструментов поиска клонов исходного кода

2.1.2.1. NiCad (текстовой подход)

NiCad [61-62] сначала нормализует исходный код программы. Далее, код разбивается на фрагменты, каждый из которых рассматривается как потенциальный клон. Все фрагменты сравниваются построчно парами для нахождения максимально общего множества совпадающих строк, которые и считаются клоном.

2.1.2.2. MOSS (текстовой подход)

MOSS [86-87] находит клоны с применением отпечатков. Исходный текст программы разбивается на k-граммы (подстроки с длиной k), которые хешируются. Далее, эти хеши используются как отпечатки, и поиск клонов производится на их основе. *MOSS* разработан в Стэнфордском университете и используется для поиска плагиата в работах студентов. ***MOSS доступен в виде онлайн-сервиса для некоммерческого использования***.

2.1.2.3. CCFinder (лексический подход)

CCFinderX [88-89] – один из широко используемых инструментов поиска клонов кода. На первом этапе производится разбиение исходного кода программы на последовательность токенов. Для этого используется набор некоторых правил (описаны в статье). Для поиска идентичных подпоследовательностей токенов максимальной длины строится суффиксное дерево.

CCFinderX также дает возможность нахождения фрагментов кода, которые копировались чаще всего. Инструмент работает для языков *Cu/Cu++, JAVA, COBOL*.

D-CCFinder [90] является распределенной реализацией *CCFinderX*. Он предназначен для анализа сверхбольших проектов с десятками миллионов строк исходного кода. На распределенной системе одна машина занимает роль мастера и разделяет все файлы на N-единицы. Далее, рассматриваются все неповторяющиеся пары единиц *(i, j)* и производится поиск клонов в этих парах с применением *CCFinderX*. Полученные результаты объединяются мастер-машиной. Распределение задач производится простым образом: если освобождается новая машина, ей дается новая пара единиц *(i, j)* для обработки.

Анализ ОС *FreeBSD* на предмет клонов, с минимальным размером 50 токенов, занял на кластере из 80 машин 51 часов (результаты авторов).

2.1.2.4. CPD (лексический подход)

CPD [91-92] является одним из популярных инструментов поиска клонов кода на основе лексического подхода. Он реализован на Java и поддерживает больше 20 языков (*Java, JSP, Cu/Cu++, C#, Fortran*). Для добавления нового языка необходимо дописать токенизатор (*tokenizer*) этого языка. *CPD* использует алгоритм *Karp-Rabin* [93] для поиска совпадающих последовательностей лексем. *CPD* дает возможность игнорировать литералы и имена переменных. Также при анализе проекта он может пропускать файлы, содержащие лексические ошибки. Инструмент легко расширяется

и масштабируется для анализа десятка миллионов строк исходного кода (в конце данной главы приводятся результаты).

2.1.2.5. SourcererCC (лексический подход)

SourcererCC [84-85] является известным инструментом поиска клонов кода на основе лексического подхода. Инструмент масштабируется и позволяет анализировать сотни миллионов строк исходного кода. *SourcererCC* сначала лексическим анализом получает последовательность лексем для каждого блока кода (блок лексем – БЛ). Далее, из каждого БЛ выбирает некоторое подмножество лексем, на основе которых он индексируется. При поиске клонов конкретного БЛ по индексу определяется множество БЛ, с которыми должно производиться сравнение. Для пары БЛ степень схожести определяется на основе пересечений элементов, соответствующих БЛ.

2.1.2.6. VUDDY (лексический подход)

VUDDY [94-95] – другой известный и масштабируемый инструмент поиска клонов кода, который использует лексический анализ, совместно с хешированием для определения клонированных функций. Сначала получается последовательность лексем каждой функции с использованием *ANTLR* [96]. Далее, производится нормализация полученной последовательности лексем путем унификации имен формальных параметров и локальных переменных функций, типов данных, вызовов функций. Из нормализованной последовательности лексем каждой функции выбирается некоторое подмножество отпечатков (*fingerprint*), на основе которых строится хеш. Таким образом, для каждой функции в базе хранится пара: длина лексем функций и созданный хеш. Поиск клонов производится в два этапа. Вначале определяются пары функций с одинаковой длиной лексем, после чего производится сравнение хешей. *VUDDY* способен находить только клоны **T1** и **T2** на уровне

функций. Если клоны содержатся в некоторых частях функций, то они не будут найдены. Инструмент масштабируется до миллиардов строк исходного кода.

VUDDY также имеет собранную базу (длина лексем и хеш) из более 1700 уязвимых функций. Он может производить поиск этих функций в произвольном проекте с открыто-доступным кодом. База может быть обогащена пользователем.

2.1.2.7. ReDeBug (лексический подход)

ReDeBug [97-98] масштабируется для анализа миллиардов строк исходного кода на предмет содержания уязвимого кода (фрагмент кода с ошибкой, который может быть эксплуатирован). Сначала из исходного кода удаляются избыточные пробелы и фигурные скобки. Далее, все символы преобразовываются в нижний регистр. После этого производится разбиение текста программы на лексемы. Это производится на основе некоторого набора регулярных выражений, которые также учитывают переходы на новые строки. Далее, алгоритм поиска клонов берет всевозможные пары последовательностей лексем, длиной N (определяется пользователем) и сравнивает на предмет схожести (формула $\frac{|F1 \cap F2|}{|F1 \cup F2|}$). Для поиска уязвимого кода сначала собирается база уязвимых фрагментов (на основе соответствующих исправлений из репозитории). Далее, эти фрагменты разбиваются на лексемы, и используется *фильтр Блума* для их хранения. При анализе нового ПО строится последовательность его лексем. Для каждой последовательности лексем длины N используется *фильтр Блума* и проверяется, содержит ли он уязвимый фрагмент кода из базы. Авторы собрали базу *diff*-фиксов для 1634 уязвимостей (до 2011 года, включительно). В таблице 3 приводятся результаты поиска этих уязвимостей в кодовой базе некоторых открыто-доступных проектов. Время анализа для каждого проекта на предмет содержания вышеуказанных 1634 уязвимостей не превысил 8 минут.

Таблица 3. Количество неисправленных ошибок

Имя теста	**Количество копий неисправленных ошибок**
Debian Lenny	1141
Ubuntu Maverick	697
Linux kernel 2.6.37.4	3
SourceForge (*Cu/Cu++*)	8711

2.1.2.8. DECKARD (синтаксический подход)

DECKARD [76-77] использует дерево разбора (ДР) для поиска клонов кода. Для поддеревьев ДР вычисляются характеристические векторы, после чего производится кластеризация векторов на основе Евклидова расстояния. Поддеревья ДР считаются клонами, если Евклидово расстояние соответствующих характеристических векторов достаточно мало (задается пользователем). Поддерживаются языки *Cu* и *Java*. *DECKARD* достаточно точно находит клоны типа **T3** благодаря тому, что, вместо поиска изоморфных поддеревьев, сравниваются характеристические векторы поддеревьев ДР. *DECKARD* использовался во множестве других работ. В *Tairas* [99] он используется для сравнения исходного кода проектов разных версий.

2.1.2.9. CloneDR (синтаксический подход)

CloneDR [100-101] является еще одним, широко используемым инструментом, разработанным компанией *Semantic Designs* [102], которая занимается разработкой инструментариев для проектирования и анализа ПО. Инструмент сначала находит изоморфные поддеревья (клоны типа **T1** и **T2**). Далее, объединяет найденные изоморфные поддеревья для получения клонов **T3**. ***Инструмент доступен в бинарной версии***.

2.1.3. Сравнение доступных инструментов

В данном разделе приводится сравнение наиболее распространенных и открыто доступных инструментов поиска клонов кода. Для этого используется набор тестов, описанный в таблице 4. В ней содержатся разные варианты клонов типа **T1**, **T2** и **T3**.

Таблица 4. Набор 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**: строки 9 и 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 таблице 5 приводится сравнение наиболее распространенных инструментов поиска клонов кода.

Таблица 5. Сравнение инструментов поиска клонов кода.

Имя теста	MOSS	CloneDR	CCFinder	PMD/CPD	SourcerCC	VUDDY	Deckard
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	+	+	+	+	+	-	+

2.1.3.1. Время работы доступных инструментов на большой кодовой базе

В таблице 6 приводится сравнение времени работы открыто-доступных инструментов. В ходе тестов на *Intel core i5* с 16Гб оперативной памятью инструменты использовали всю память и все 8 ядров.

Таблица 6. Сравнение времени работы.

Строки кода	SourcererCC	CCFinderX	DECKARD	NiCad	VUDDY	ReDeBug
1.000	2.3с.	3.с.	2с.	1с.	0.44с.	35.6с.
10.000	3.1с.	4.с.	9с.	4с.	0.81с.	35.6с.
100.000	50.7с.	21с.	1м. 34с.	21с.	5.17с.	42с.
1.000.000	1м. 44с.	2м. 18с.	1ч. 12м.	4м. 1с.	55с.	1м. 43с.
10.000.000	24м. 38с.	28м. 51с.	**Memory Error**	11ч. 32м.	12м. 43с.	18м. 32с.
100.000.000	9ч. 42м.	3д. 6ч.	-	**Internal Limit**	1ч. 32м.	2ч. 32м.
1.000.000.000	25д. 3ч.	**File I/O Error**	-	-	14ч. 17м.	1д. 3ч.

2.1.3.2. Заключение по инструментам поиска клонов исходного кода

Для поиска клонов типа **T1**, **T2** в пределах разных проектов лучше использовать *SourcererCC* или *CPD*. Первый лучше масштабируется и может быть применен для анализа сотен миллионов строк исходного кода. Второй поддерживает больше языков и легче расширяется. Оба инструмента находят клоны **T1**-**T2** с высокой точностью. *SourcererCC* кроме клонов **T1** и **T2** может найти некоторые (но не все) клоны **T3**. Клоны **T2**, в которых производится изменение типов переменных, в некоторых случаях могут быть пропущены. *CPD* находит все клоны **T1**, **T2**, но не может найти клоны типа **T3**.

Для инструментов *CloneDR* и *MOSS* исходный код недоступен. *DECKARD* работает медленнее по сравнению с инструментами на основе лексического подхода. Также он не масштабируется для анализа сотен миллионов строк исходного кода. *VUDDY* может найти клоны только **T1**, **T2** на уровне функций. Поиск клонов на уровне функций является большим ограничением. Кроме этого, в репозитории инструмента не ведется активная разработка (225 коммитов, последний раз был обновлен в 2017г.). В репозитории *CCFinderX* не ведется разработка с 2016г. (в результате просмотра всех зеркал на *GitHub* [103-105]). На официальной веб-странице последнее обновление было совершено в 2010г. [88]. В репозитории *NiCad* не ведется разработка с 2015г. (только 5 коммитов). *ReDeBug* используется для поиска конкретного фрагмента кода (уязвимого) в проекте. Таким образом, он неудобен для использования поиска всех клонов.

В заключение можно сказать, что разработка инструмента для эффективного поиска клонов типа **T3** является актуальной задачей и частично была рассмотрена в рамках диссертационной работы автора. ***В данной работе также будут представлены некоторые улучшения уже разработанного автором инструмента.***

2.2. Обзор методов поиска клонов и сравнении исполняемых файлов

Методы поиска клонов и сравнения исполняемых файлов широко используются для поиска уязвимых функций, анализа изменений между версиями ПО, поиска нарушений авторских прав и т.д.

Клоны исполняемого кода можно условно разделить на четыре типа. Первый тип – фрагменты кода, которые полностью совпадают. Второй – фрагменты кода, которые могут отличаться типами и значениями данных, а также названиями регистров. Третий тип – фрагменты кода, которые могут отличаться типами, значениями данных и названиями регистров, а также некоторыми инструкциями, которые могут присутствовать или отсутствовать в конкретном фрагменте. Четвертый тип – фрагменты кода, которые отличаются друг от друга, но имеют одинаковый функционал (например, рекурсивная и итеративная версия одного и того же алгоритма). В таблице 7 приводятся примеры первых трех типов клонов кода.

Таблица 7. Примеры типов клонов для архитектуры x86.

Фрагмент кода	Клон типа 1	Клон типа 2	Клон типа 3
public main *main proc near* *var_4= dword ptr -4* *argc= dword ptr 8* *argv= dword ptr 0Ch* *envp= dword ptr 10h* *push ebp* *mov ebp, esp* *mov [ebp+var_4], 5* *mov eax,[ebp+var_4]* *imul eax,[ebp+var_4]* *leave* *retn* *main endp*	*public main* *main proc near* *var_4= dword ptr -4* *argc= dword ptr 8* *argv= dword ptr 0Ch* *envp= dword ptr 10h* *push ebp* *mov ebp, esp* *mov [ebp+var_4], 5* *mov eax,[ebp+var_4]* *imul eax,[ebp+var_4]* *leave* *retn* *main endp*	*public main* *main proc near* *var_4= dword ptr -4* *argc= dword ptr 8* *argv= dword ptr 0Ch* *envp= dword ptr 10h* *push ebp* *mov ebp, esp* *mov [ebp+var_4],**10*** *mov **ecx**,[ebp+var_4]* *imul **ecx**,[ebp+var_4]* *leave* *retn* *main endp*	*public main* *main proc near* *var_1= dword ptr -4* *argc= dword ptr 8* *argv= dword ptr 0Ch* *envp= dword ptr 10h* *push ebp* *mov ebp, esp* *mov [ebp+var_1],**15*** *mov **ecx**,[ebp+var_1]* *-* *leave* *retn* *main endp*

2.2.1. Обзор доступных инструментов

В данном разделе приводится обзор доступных инструментов поиска клонов и сравнения исполняемых файлов.

2.2.1.1. Обзор инструмента Bitshred

Инструмент *Bitshred* [106-107] рассматривает фрагмент исполняемого кода как последовательность байтов или строк кода ассемблера. Авторы предлагают алгоритм сравнения отпечатков для поиска вредоносных участков кода. Для сравнения используется *фильтр Блума* [108]. Работа инструмента состоит из трех основных этапов. На первом этапе инструмент делит весь исполняемый код на куски с размером n-байт (задается пользователем). Затем для всех фрагментов создаются отпечатки с использованием *фильтра Блума*. После этого вычисляется сходство между двумя отпечатками. На последнем этапе схожие фрагменты группируются вместе.

2.2.1.2. Обзор инструмента BinDiff

BinDiff [109-111] является популярным инструментом для сравнения бинарных файлов. Для сопоставления функций реализованы разные метрики на основе информации об узлах и ребрах графов потока управления и графов вызовов. Для восстановления графов используется инструмент *IDA Pro* [112]. В работе авторы предлагают метод вычисления хеша для графов (*MD-index*). Ребрам графов (и потока управления, и вызовов) сопоставляется вектор с пятью элементами:

1. *z1* − номер ребра при топологическом обходе графа;
2. *z2* − количество входных ребер начальной вершины ребра;
3. *z3* − количество выходных ребер начальной вершины ребра;
4. *z4* − количество входных ребер конечной вершины ребра;

5. *z5* − количество выходных ребер конечной вершины ребра.

После получения всех векторов считаются *MD-index* для графа *G* следующим образом:

$$emb(vector) = z1 + z2\sqrt{2} + z3\sqrt{3} + z4\sqrt{5} + z5\sqrt{7}$$

$$MD - index(G) = \sum_{\text{для всех } vector-\text{ов}} \frac{1}{\sqrt{emb(vector)}}$$

Далее, *MD-Index* используется для вычисления метрик на графах.

Работы [113] и [114] нацелены на усовершенствование инструмента *BinDiff*. В [113] авторы пытаются сопоставить функции из баз данных вредоносных программ. В работе [114] (инструмент *BinSlayer*) авторы используют венгерский алгоритм [115] для более оптимального сопоставления базовых блоков.

2.2.1.3. Обзор инструмента Diaphora

Diaphora [116] является плагином для *IDA Pro* [112], который принимает на вход два бинарных файла и производит сопоставление содержащихся функций. Инструмент позволяет определить схожие фрагменты сопоставленных функций. С помощью *Diaphora* также можно переносить символы (имена функций, комментарии, определения структур, прототип и т.д.) из одной версии программы в другую.

Сопоставление функций производится с помощью различных наборов эвристик ("*лучшие сопоставления*", "*частичные и ненадежные сопоставления*", "*ненадежные сопоставления*"), которые описаны ниже. Эвристики применяются поочередно. Если после применения текущей эвристики остаются не сопоставленные функции, применяется следующая по списку. *Diaphora* сохраняет информацию бинарных файлов в базе данных *SQLite* [117].

"**Лучшие сопоставления**" – эвристики этой группы сопоставляют те функции из сравниваемых бинарных файлов, которые совпадают на 100%:

- *Самая первая попытка. SQLite* базы данных совпадают для данной пары

бинарных файлов. Если это так, бинарные файлы считаются одинаковыми, и работа инструмента завершается.

- *Одинаковый псевдокод.* Псевдокод, сгенерированный декомпилятором *Hex-Rays* [118], одинаков для пары функций.
- *Одинаковый код ассемблера.* Инструкции ассемблера для пары функций одинаковые.
- *Хеши байтов и имена.* Первые байты соответствующих ассемблерных инструкций совпадают для пары функций. Кроме этого, проверяется совпадение исходных имен функций (если имена доступны).
- *Одинаковые адреса, узлы, ребра и мнемоники.* Количество базовых блоков, их адреса, ребра и мнемоники совпадают для пары функций.
- *Одинаковый OBA и хеш.* OBA (относительный виртуальный адрес) и хеш байтов одинаковы для пары функций.
- *Одинаковый порядок и хеш.* Пара функций имеет одинаковый хеш байтов и одинаковые позиции в базе *IDA Pro* (например, обе функции являются 100-ми функциями в базе *IDA Pro*).
- *Хеш функции. MD5* [119] хеши всех байтов пары функций одинаковы.
- *Хеш байтов.* Вычисляются *MD5* хеши конкатенации всех байтов всех инструкций данной функции кроме тех, которые могут отличаться по адресу относительного перехода (*relative calls and jumps*). Пара функций считается сопоставленной, если эти хеши совпадают.

"**Частичные сопоставления**" – эвристики из этой группы сопоставляют частично различающиеся функции:

- *Все или большинство атрибутов.* Пара функций имеют одинаковые атрибуты: количество базовых блоков, вызываемых функций, вызывающих функций.
- *Switch-структуры.* Случаи и значения всех операторов "*switch*" в паре функций

совпадают.

- *Одинаковые имена.* У пары функций одинаковые кодированное (*mangled*) или декодированное (*demangled*) имена.
- *Одинаковые адреса, узлы, ребра и цикломатическая сложность*. Пара функций имеет одинаковые адреса, число базовых блоков и ребер, совпадающие цикломатические сложности.
- *Хеши вызываемых функций.* Пара функций сопоставляется, если все вызываемые из них функции имеют одинаковые декодированные (*demangled*) имена.
- *Мнемоники и имена*. Инструкции функций имеют одинаковые мнемоники и одинаковые имена (если доступно) вызываемых функций.
- *Произведение маленьких простых чисел SPP (small-primes-product)*. Произведение небольших простых чисел, соответствующих мнемоникам, в обеих функциях равны [120].
- *Схожие имена.* Множества имен вызываемых функций для пары функций совпадает не менее на 50%.
- *Нечеткий (fuzzy) хеш псевдокода.* Для пары функций совпадают хеши генерированного алгоритмом "*Koret Default Hashing Algorithm*" библиотеки *DeepToad* [121].
- *Схожий псевдокод.* Псевдокоды, сгенерированные декомпилятором *Hex-Rays* для пары функций, схожи с коэффициентом – не меньше 0.6.
- *Нечеткий хеш АСД (абстрактное синтаксическое дерево).* Нечеткие хеши, вычисленные с помощью *SPP* для АСД декомпилированных функций, одинаковы для пары функций.
- *Произведение простых чисел (SPP) сильно связанных компонентов*. Произведение небольших простых чисел, соответствующих сильно связанным

компонентам графов потока управления каждой функции, одинаково для пары функций.

- *Количество циклов.* Пара функций имеет одинаковое количество циклов, которое больше 1, и количество базовых блоков в каждой функции больше 3.
- *Одинаковые узлы, ребра и сильно связанные компоненты.* Количество базовых блоков, ребер между ними, множество сильно связанных компонентов графа потока управления пар функций совпадает.

"**Ненадежные сопоставления**":

- *Сильно связанные компоненты.* Множества сильно связанных компонентов (должно быть минимум 2) графов потока управления каждой функцией одинаковы.
- *Количество циклов.* Пара функций имеет одинаковое количество циклов, которое больше 1.
- *Узлы, ребра, цикломатическая сложность и мнемоники*. Количество базовых блоков, ребер, цикломатическая сложность и мнемоники одинаковы.
- *Узлы, ребра, цикломатическая сложность и прототип.* Количество базовых блоков, ребер, цикломатическая сложность и прототип одинаковы.
- *Узлы, ребра, цикломатическая сложность, степень входящих и выходящих ребер.* Количество базовых блоков, ребер, цикломатическая сложность, количество вызывающих и вызываемых функций одинаковы.
- *Узлы, ребра и цикломатическая сложность.* Одинаковое количество узлов, ребер и значений цикломатической сложности графов потока управления пар функций.
- *Высокая сложность.* Пара функций имеет одинаковую высокую цикломатическую сложность, составляющую не менее 50.

2.2.1.4. Обзор инструмента Kam1n0

Kam1n0 [122] – масштабируемая платформа анализа бинарного кода, которая поддерживает поиск клонов кода ассемблера. Сначала собирается база бинарных файлов в хранилище. Инструмент получает на вход бинарный файл или фрагмент ассемблерного кода и выполняет поиск клонов в хранилище. Для поиска разработаны 3 алгоритма: *Asm-Clone* [123], *Sym1n0* [122] и *Asm2Vec* [124].

Алгоритм Asm-Clone пытается найти клоны базовых блоков на основе хеширования ассемблерных инструкций (предварительно производится нормализация кода – обобщение ссылок на память, регистров и константных значений). Для хеширования и сравнения используются вариации алгоритма *LSH* [125]. Далее, на основе сопоставленных базовых блоков производится поиск изоморфных подграфов для функции входного файла и функций из хранилища. Для поиска изоморфизма используется граф потока управления функций. Для масштабируемости используется технология *Map-Reduce* [126]. Во время процедуры *Map* создаются пары схожих базовых блоков (из входного файла и из хранилища). Во время процедуры *Reduce* эти пары объединяются, если они принадлежат одним и тем же функциям. В качестве результата, инструмент выдает сопоставленные подграфы графа потока управления.

Алгоритм *Sym1n0* выполняет семантический поиск клонов с помощью фаззинга и символьного выполнения.

Алгоритм *Asm2Vec* [124] вначале производит разбор ассемблерных инструкций каждой функций на токены (инструкций и его операндов). Далее, из графа потока управления получается набор путей (не пересекающихся), соответствующих потенциальным трассам выполнения программы. Каждая последовательность содержит набор токенов, соответствующий инструкциям. Порядок получения путей рандомизирован. Таким образом, из каждой функции получается последовательность токенов "*слов*", где также учитывается структура программы. Тренировка нейронной

сети производится с использованием полученного "*текстового*" (токены) представления начальных бинарных функций, в результате чего генерируются векторы для заданных функций. Вектор каждой функции содержит информацию о том, чем она отличается от других (несхожих функций). Используя векторное представление для данной бинарной функции, выводится список схожих функций.

2.2.1.5. Обзор инструментов BinHash

Инструмент *BinHash* [127] для сравнения функций вычисляет семантический хеш графа потока управления. Семантический хеш включает в себя несколько характеристик каждого базового блока функций: эффект выполнения базового блока на регистры, эффект выполнения базового блока на память, условие перехода, аргументы функции, которая вызывается из базового блока.

2.2.2. Обзор статьей и методов, для которых нет доступных реализаций.

В этом разделе описаны статьи, для которых нет доступных инструментов.

2.2.2.1. Обзор инструмента BinSign

BinSign [128] создает отпечатки (*fingerprint*) функций и производит поиск схожих функций на основе этих отпечатков. Для создания отпечатка из функций извлекаются некоторые характеристики, которые содержат информацию о:

1. Строковых константах и константных переменных;
2. Возвращаемых типах функций;
3. Типах и количестве аргументов функций;
4. Системных вызовах;
5. Вызовах функций;
6. Количестве вызовов функций по регистру и по адресу;

7. Количестве базовых блоков;
8. Количестве каждого типа инструкций;
9. Свойствах ребер графа потока управления.

Далее, извлеченные характеристики функций хешируются специальным образом для получения отпечатка функций. Отпечатки индексируются для быстрого поиска.

2.2.2.2. Обзор инструмента αDiff

αDiff [129] использует методы машинного обучения для определения схожих функций в разных версиях бинарных файлов (бинарные файлы могут быть кроссплатформенные). Инструмент основывается на том, что бинарные файлы состоят из функций и их можно получить достаточно точно с применением существующих инструментов (например, *IDA Pro*). Для классификации схожих функций используется 3 свойства на основе:

1. Необработанного бинарного кода функций;
2. Вызывающих и вызванных функций;
3. Вызовов системных и библиотечных функций.

Свойства на основе необработанного бинарного кода функций: бинарный код функций преобразуется в вектор размерностью *N*. Для этого используется сверточная нейронная сеть (*convolutional neural network – CNN*). Цель сети – преобразовать бинарные функции в вектор таким образом, что *Евклидовое расстояние* между соответствующими векторами было маленьким для схожих функций. Авторы создали базу для 66.823 пар бинарных файлов с более чем 2.5 миллионами размеченных пар функций. Согласно результатам авторов, инструмент превосходит показатели *BinDiff* и *BinGo* [130] на кроссплатформенных (*ARM*, *IA-32*, *IA-64*) бинарных файлах с разными опциями компиляций. αDiff может найти известные уязвимости в бинарных

файлах путем поиска клонов уязвимой функции. Авторы продемонстрировали поиск следующих уязвимостей: *CVE-2014-0160* (*Heartbleed*) в *openssl-1.0.1f*, *CVE-2014-6271* (*Shellshock*) в *bash-4.3*, *CVE-2014-4877* в *wget-1.15*, *CVE-2014-9295* (*Clobberin Time*) в *ntpd-4.27p10*.

2.2.3. Сравнение инструментов и итоги исследования

Был проведен сравнительный анализ двух наиболее популярных доступных инструментов: *Diaphora* и *Bindiff*. Инструменты были протестированы на проектах *openssl* и *php*. Для поиска клонов были использованы разные версии этих программ, компилированные разными компиляторами и разными уровнями оптимизаций. Результаты запусков приведены в таблице 8. Как видно из таблиц, оба инструмента имеют больший процент правильных срабатываний при сравнении (либо поиска клонов) в двух версиях одного и того же проекта, даже если они были скомпилированы разными компиляторами. Однако при применении "*тяжелых*" оптимизаций (флаг *-02*), количество правильных срабатываний уменьшается. Кроме этого, из таблицы 8 становится очевидно, что инструмент *Bindiff* показывает наилучшие результаты.

Таким образом, становится очевидно, что создание инструмента поиска клонов бинарного кода в сильно отличающихся (например, полученной при "*тяжелых*" оптимизациях) исполняемых файлах является актуальной задачей.

Таблица 8. Сравнение инструментов Diaphora и Bindiff.

Бинарный файл 1	**Компилятор, оптимизации 1**	**Бинарный файл 2**	**Компилятор, оптимизации 2**	**Количество сопоставленных функций Diaphora**	**Количество сопоставленных функций BinDiff**
openssl-1.0.1j	gcc O0	openssl-1.0.2r	gcc O0	**5469**	5359
openssl-1.0.1j	gcc O0	openssl-1.0.1j	gcc O2	2193	**4339**
openssl-1.0.2r	gcc O0	openssl-1.0.2r	gcc O2	2381	**4582**
openssl-1.0.1j	clang O0	openssl-1.0.2r	clang O0	176	**4128**
openssl-1.0.1j	clang O0	openssl-1.0.1j	clang O2	4102	**4797**
openssl-1.0.2r	clang O0	openssl-1.0.2r	clang O2	343	**4797**
openssl-1.0.1j	gcc O0	openssl-1.0.1j	clang O0	2181	**4339**
openssl-1.0.2r	gcc O0	openssl-1.0.2r	clang O0	682	**6499**
openssl-1.0.1j	gcc O0	openssl-1.0.2r	clang O0	176	**5978**
php 7.2.18	gcc O0	php 7.2.19	gcc O0	13022	**13035**
php 7.2.18	gcc O2	php 7.2.19	gcc O2	**9510**	9383
php 7.2.18	clang O0	php 7.2.19	clang O0	11636	**12198**
php 7.2.18	clang O2	php 7.2.19	clang O2	**8749**	7949
php 7.2.18	gcc O0	php 7.2.18	clang O0	5308	**11663**
php 7.2.19	gcc O0	php 7.2.19	gcc O2	3646	**8262**
php 7.2.19	gcc O0	php 7.2.19	clang O0	5321	**11667**

2.3. Обзор методов и технологий анализа изменений между версиями ПО

Данная глава посвящена исследованию методов анализа изменений между двумя версиями ПО.

2.3.1. Обзор статьей, посвященных методам анализа исправленных ошибок и уязвимостей

В работе [131] (**инструмент недоступен**) авторы пытаются определить, какое конкретное изменение привело к ошибке в программе. Для этого авторы сканируют ошибки в системе управления ошибок (*Bugzilla* [132], *Jira* [133] и т.д.). Далее, для каждой ошибки авторы стараются найти, какое именно изменение является исправлением данной ошибки. Для этого сканируется история изменений в системе управления версиями ПО. Чтобы найти исправление ошибки, анализируется лог коммитов. В работе описывается два метода анализа: синтаксический и семантический. Эти методы анализа применяются последовательно. Синтаксический анализ пытается найти шаблоны (номер ошибки из системы управления ошибок), описывающие ошибку, используя регулярные выражения. Семантический анализ пытается найти информацию об исправленной ошибке в логе коммита, используя следующие факторы:

- Ошибка (номер которой указана в логе коммита) имеет статус *Resolved/Close* с атрибутом *FIXED*;
- Описание ошибки содержится в логе коммита;
- Автор коммита также является ответственным за исправление ошибки (*assignee*);
- Прикрепленные к задаче файлы также присутствуют в коммите.

После нахождения коммита исправления, авторы пытаются найти изменения, которые привели к ошибке в проекте. Явным недостатком данного метода является то, что авторы полагаются на полноценное описание всех изменений разработчиками проектов, что не всегда является правдой. Существует много других работ [134], [135], [136], [137] для решения данной проблемы, но все они используют близкие к описанным методам метрики и алгоритмы. Методы и алгоритмы, представленные в описанных работах, имеют один большой недостаток – все производятся на комментариях и логах, написанных разработчиками проекта. Разработчики часто могут и не включать информацию об ошибке в логе коммита или даже не описывать ошибку в системе управления ошибками. Все это приводит к множественным пропущенным связям между ошибкой и изменениями. В работе [138] подробно описывается эта проблема на примере проекта *Apache* [139]. Авторы утверждают, что, в среднем, 54% исправленных ошибок не могут быть связаны с соответствующими исправлениями в коде. Такие же проблемы присутствуют в других известных работах. Например, инструмент *BagCache* [140], тоже использует похожие метрики, как в [141], [131], и имеет аналогичные проблемы.

Другая группа работ посвящена поиску известных уязвимостей в исходном коде. Работа [142] извлекает известные *CVE* из базы уязвимостей [143] и ищет похожие ошибки в заданном проекте. Для этого предлагается три алгоритма, основанных на: сравнении текста, анализе лексем, сравнении синтаксических абстрактных деревьев. В работе [144] описывается метод поиска уязвимостей с использованием технологии глубокого обучения. В работе поиск уязвимостей рассматривается как задача классификации текста. В качестве базы для обучения используется тестовый набор *SARD* [145]. Из тестового набора извлекаются необходимые данные об ошибке. Далее, используется глубокое обучение для классификации поиска ошибок. В работе [146] представлен инструмент *VCCFinder*

[147], который находит уязвимости, используя *SVM*-классификаторы. В качестве данных для обучения используется результаты анализа изменений в репозиториях – в более, чем 60 проектах. Для поиска изменений, которые могли исправить уязвимость, реализовано два алгоритма. Первый алгоритм ищет в базах данных уязвимостей [143] [148] такие *CVE*, у которых есть ссылка на соответствующий коммит исправления. Второй алгоритм анализирует лог коммитов во всех проектах и ищет указания идентификаторов *CVE*. На основе полученных коммитов исправлений реализован алгоритм для поиска схожих ошибок.

2.3.2. Обзор инструментов отслеживания изменений между версиями ПО с доступным исходным кодом

Отслеживание изменений в ПО имеет несколько практических применений. Проанализировав изменения, можно понять, не содержат ли они новые ошибки. Изменения программ с доступным исходным кодом можно получить простым синтаксическим сравнением двух версий файлов. Одним из известных инструментов для сравнения файлов является утилита *diff* операционной системы *Linux*. Синтаксическое сравнение файлов также доступно во всех системах управления версиями ПО. Такое сравнение выдает информацию только о добавленных/удаленных строках в файлах, и не учитывает структурные (семантические) изменения. Существует множество работ, посвященных более продвинутым методам сравнения версий программ. Инструмент *change distler* [149] основан на сравнении синтаксического абстрактного дерева программ. Работа основывается на предыдущих исследованиях авторов [150], где изменения в программе представляются в качестве изменений в абстрактном дереве. Представленный алгоритм находит такие изменения в дереве, как *добавление*, *удаление*, *перемещение* и *обновление*. Еще одним известным инструментом для удобного анализа изменений является *LSDIFF* [151] . Инструмент

представлен в работе [152]. Основное преимущество инструмента – способность распознавать систематические изменения и группировать их в отдельные категории. Группирование систематических изменений позволяет удобно просматривать их, и основывается на исследованиях, утверждающих, что участки кода с похожими структурными характеристиками подвергаются схожим изменениям [153]. В работе [154] представлен инструмент сравнения версий программ *ClDiff* [155]. Этот инструмент, как и *LSDIFF*, пытается сгруппировать и составить сжатое представление изменений.

2.3.3. Заключение по инструментам поиска изменений между версиями ПО

Исследования показали, что для решения задач анализа исправленных *CVE* большинство методов основан на анализе логов коммитов и системах отслеживания ошибок. Однако такой подход не позволяет найти все возможные исправления уязвимостей в коде. Таким образом, остается актуальной задачей разработки качественной системы анализа изменений между версиями ПО, в том числе с учетом исправлений *CVE*.

2.4. Обзор методов сопоставления исходного и бинарного кода

Данная глава посвящена исследованию методов сопоставления исходного и бинарного кода.

2.4.1. Описание инструмента Pigaios

Pigaios [156] - инструмент для сопоставления символов исходного кода символам соответствующего бинарного кода. Нужно отметить, что инструмент не требует, чтобы исходный код компилировался и компоновался. Для сопоставления выполняются следующие шаги:

1. Разбор исходного и бинарного кода;
2. Извлечение артефактов для каждой функции исходного и бинарного кода;
3. Сопоставление функций исходного кода функциям бинарного кода на основе артефактов и определение коэффициента схожести для каждого сопоставления.

Для разбора исходного кода используется *clang* [157]. Артефакты получаются из абстрактного синтаксического дерева и включают:

- Имя функции;
- Константные строки;
- Количество циклов, условий, вызовов функций, внешних и глобальных переменных;
- Количество "*switch*" операторов и их случаев;
- Является ли функция рекурсивной;
- Список вызываемых функций.

Pigaios сначала сопоставляет функции исходного кода к функциям бинарного кода на основе полученных артефактов. Если не все функции были сопоставлены, применяются следующие эвристики:

1. **Сопоставление вызываемых и вызывающих функций**. Для пары

сопоставленных функций сравниваются их вызываемые и вызывающие функции;

2. **Сопоставление ближайших функций**. Если некоторые функции *F1*, *F2*, *F3* находятся в одном файле исходного кода, и функции *F1* и *F3* сопоставляются, то *F2* сравниваются с функциями, адреса которых находятся между этими функциями;
3. **Сопоставление на основе редко встречающихся констант**. Для редко встречающихся строковых констант (менее 3-х раз) находятся функции в исходном и бинарном коде, использующие их. Далее производится сравнение и сопоставление этих функции.

Определение коэффициента схожести для каждого сопоставления основано на машинном обучении. База для обучения собрана ручным образом на основе анализа срабатываний инструмента.

В таблицах 9 и 10 приводится экспериментальные результаты *Pigaios*. Бинарные файлы получены компиляцией исходного кода. В некоторых тестах из бинарных файлов удалялись символы (с помощью инструмента *strip*).

Таблица 9. Результаты инструмента Pigaios без использования машинного обучения на бинарных файлах архитектуры x64.

Бинарный файл	**Компилятор, оптимизации**	**Размер бинарного файла**	**Исходный код**	**Количество строк исходного кода**	**Количество функций в бинарном файле**	**Количество функций в исходном коде**	**Количество сопоставленных функций**	**Ошибки первого рода**	**Ошибки второго рода**
libpng 1.2.29	gcc O2	648 КБ	libpng 1.2.29	68818	476	458	164	0	290
libpng 1.2.29 (stripped)	gcc O2	648 КБ	libpng 1.2.29	68818	476	458	116	0	340
libxml2.so (stripped)	gcc O2	1.8 МБ	libxml2	459871	2719	4582	586	0	2133
openssl-1.0.1j	gcc O3	3.2 МБ	openssl-1.0.1j	408512	5678	8880	2863	0	2790
openssl-1.0.1j (stripped)	gcc O3	3.2 МБ	openssl-1.1.0j	408512	5678	8880	1240	656	4400
php-7.2.19	gcc O2	38 МБ	php-7.2.19	1492357	10133	13639	1399	0	8700
php-7.2.19 (stripped)	gcc O2	38 МБ	php-7.2.19	1492357	10133	13639	1138	362	8950
openssl-1.0.2r	gcc O3	3.4 МБ	openssl-1.0.2r	518268	6011	9556	3019	0	2992
openssl-1.0.2r (stripped)	gcc O3	3.4 МБ	openssl-1.0.2r	518268	6011	9556	1256	621	4755
busybox-1.28.2 (stripped)	gcc O2	2.8 МБ	busybox-1.28.2	230222	4025	4500	1172	0	2853

Таблица 10. Результаты инструмента Pigaios с использованием машинного обучения на бинарных файлах архитектуры x64.

Бинарный файл	Компилятор, оптимизации	Размер бинарного файла	Исходный код	Количество строк исходного кода	Количество функций в бинарном файле	Количество функций в исходном коде	Количество сопоставленных функций	Ошибки первого рода	Ошибки второго рода
openssl-1.0.2r	Gcc O3	3.4 МБ	openssl-1.0.2r	518268	6011	9556	3228	0	2750
openssl-1.0.2r	Gcc O3	3.4 МБ	openssl-1.1.0j	444408	6011	9487	2514	0	2490
openssl-1.0.2r (stripped)	Gcc O3	3.4 МБ	openssl-1.0.2r	518268	6011	9556	834	638	5150
busybox-1.28.2 (stripped)	Gcc O2	2.8 МБ	busybox-1.28.2	230222	4025	4500	983	0	3020
libpng 1.2.29	Gcc O2	648 КБ	libpng 1.2.29	68818	476	458	163	0	313
libpng 1.2.29 (stripped)	Gcc O2	648 КБ	libpng 1.2.29	68818	476	458	99	0	377
libxml2.so (stripped)	Gcc O2	1.8 МБ	libxml2	459871	2719	4582	514	0	2205
openssl-1.0.1j	Gcc O3	3.2 МБ	openssl-1.0.1j	408512	5678	8880	2970	0	2708
openssl-1.0.1j	Gcc O3	3.2 МБ	openssl-1.1.0j	408512	5678	8880	913	594	4765
php-7.2.19	Gcc O2	38 МБ	php-7.2.19	1492357	10133	13639	630	0	9503
php-7.2.19 (stripped)	Gcc O2	38 МБ	php-7.2.19	1492357	10133	13639	269	127	9864
openssl-1.0.2r	Gcc O0	3.9 МБ	openssl-1.0.2r	518268	6804	9556	3715	0	3089
openssl-1.0.2r	Clang O0	3.9 МБ	openssl-1.0.2r	518268	6798	9556	3799	0	2999
openssl-1.0.2r (stripped)	Gcc O0	3.6 МБ	openssl-1.0.2r	518268	6739	9556	2111	870	4628
openssl-1.0.2r (stripped)	Clang O0	3.5 МБ	openssl-1.0.2r	518268	6734	9556	2284	1095	5639
openssl-1.0.2r	Gcc O2	3.4 МБ	openssl-1.0.2r	518268	6228	9556	3173	0	3055
openssl-1.0.2r	Clang 02	3.0 МБ	openssl-1.0.2r	518268	6000	9556	3023	0	2977
openssl-1.0.2r (stripped)	Gcc O2	3.0 МБ	openssl-1.0.2r	518268	5091	9556	1502	632	3589
openssl-1.0.2r (stripped)	Clang O2	2.7 МБ	openssl-1.0.2r	518268	5496	9556	1365	783	4131
php-7.2.19	Gcc O0	29.5 МБ	php-7.2.19	1492357	13544	13639	1764	0	11780
php-7.2.19	Clang O0	27.5 МБ	php-7.2.19	1492357	12198	13639	1779	0	10419
php-7.2.19 (stripped)	Gcc O0	16.6 МБ	php-7.2.19	1492357	13544	13639	1337	342	12207
php-7.2.19 (stripped)	Clang O0	16.0 МБ	php-7.2.19	1492357	12198	13639	1409	417	10789
php-7.2.19	Gcc O2	39.2 МБ	php-7.2.19	1492357	10133	13639	1399	0	8734
php-7.2.19	Clang O2	32.9 МБ	php-7.2.19	1492357	9341	13639	1195	0	8146
php-7.2.19 (stripped)	Gcc O2	12.5 МБ	php-7.2.19	1492357	10133	13639	1138	381	8995
php-7.2.19 (stripped)	Clang O2	12.0 МБ	php-7.2.19	1492357	7953	13639	913	259	7040

Обобщая результаты, можно сказать, что средний процент ошибок первого рода составляет 17.9%, средний процент ошибок второго рода составляет 72.8%. Также нужно отметить, что когда имена функций известны в бинарном файле (т.е. не использовали команду "strip"), средний процент правильно сопоставленных функций представляется намного больше. Средний процент ошибок первого рода при входных бинарных файлах, без информации, об именах функций составляет 32%, средний процент ошибок второго рода составляет 80.2%.

2.4.2. Обзор инструмента RESource

RE-Source [158] производит сопоставление бинарного кода с исходным кодом. На входе инструмент принимает только бинарный файл, выполняет поиск в онлайн-хранилище (список приведен далее), и возвращает информацию о сопоставлениях. Основные этапы онлайн-анализа заключаются в следующем:

1. Извлечение *интересных* особенностей бинарного файла: константные значения операндов, импортированные библиотеки и вызовы функций, значения константных строк;
2. Создание запроса на основе полученных особенностей и отправка поисковых систем кода: *Antepedia*, *Google*, *Koders*;
3. Обновление комментариев в ассемблере (рисунок 2).

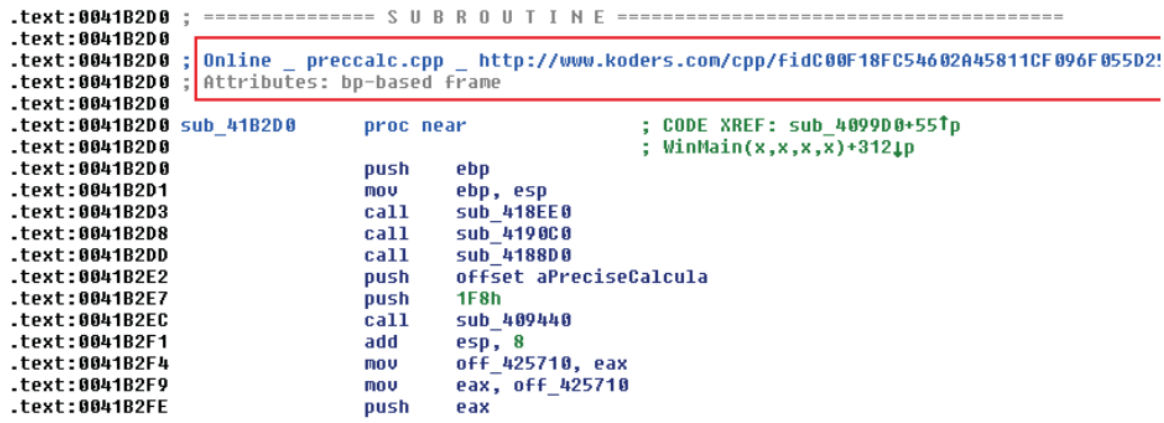

Рисунок 2. Результат онлайн анализа.

2.4.3. Обзор инструмента CodeBin

В *CodeBin* [159] представлен новый подход для идентификации бинарных функций путем поиска в хранилище предварительно обработанного исходного кода. Разработанный инструмент основан на извлечении характеристик исходного кода (список приведен ниже), которые сохраняются в процессе компиляции и сборки и, как правило, не зависят от платформы, компилятора или уровня оптимизации. Извлечение характеристик происходит без компиляции исходного кода. Такие же характеристики извлекаются из бинарных файлов. Характеристики используются для сопоставления функций исходного кода с функциями бинарного кода.

Для сопоставления функций используются отпечатки, генерирующиеся из характеристик кода. Эти характеристики заключаются в следующем:

1. Вызовы функций исследуемого кода;
2. Вызовы *API*-функций и стандартных библиотек;
3. Количество аргументов функций;

4. Сложность потока управления, основанного на цикломатической сложности $C = E - N + 2P$, где E – количество ребер, N – количество вершин, и P – количество связанных компонентов;
5. Целочисленные и строковые константы.

Для сопоставления бинарного кода с исходным кодом используется аннотированный граф вызовов функций. Каждая вершина в графе вызовов функций аннотируется, используя перечисленные свойства. Для сопоставления проводятся следующие шаги:

1. Извлекаются характеристики исходного кода. Исходный код разбирается для получения характеристик, и создается аннотированный граф вызовов функций (АГВФ). АГВФ сохраняется в базе данных для быстрого поиска и извлечения информации;
2. Извлекаются характеристики бинарного кода. Для каждого бинарного файла производится анализ для извлечения тех же характеристик, что и у исходного кода. В результате анализа создается бинарный АГВФ;
3. Поиск бинарного АГВФ в базе исходного АГВФ.

Таблица 11. Результаты инструмента CodeBin.

Проект	**Повторно используемые функции**	**Корректное сопоставление**	**Ошибки первого рода**	**Ошибки второго рода**
Miniz	114	65.8%	2.6%	16.7%
Sqlite	1391	74.5%	1.2%	13.8%
Redis	2329	65.2%	2.1%	12.1%
Coreutils	1856	53.4%	2.2%	19.5%
PCRE	342	31.2%	3.5%	38.9%
OpenSSL	3982	9.4%	1.4%	83.3%

В таблице 11 приводятся результаты сопоставления исходного кода с соответствующими бинарными файлами (компиляция производится опциями компилятора по умолчанию). Как видно из таблицы, в среднем, инструмент корректно сопоставляет 49.9% функций, неправильно сопоставляет 2.2% функций, и не сопоставляет 30.7% функций.

2.4.4. Обзор инструмента KARTA

Karta [160] является плагином для *IDA Pro*, который на входе получает бинарный файл и сопоставляет символы исходных файлов библиотек (список приведен ниже) к символам бинарного файла. При разработке метода основное внимание уделено тому, чтобы процесс сопоставления происходил быстро. Плагин является архитектурно-независимым и, по умолчанию, поддерживает сопоставление только с ниже указанными библиотеками: *libpng [161], zlib [162], openssl [36], openssh [163], net-snmp [164], gSOAP [165], libxml2 [166], libtiff [167], mDNSResponder [168], MAC-Telnet [169], libjpeg-turbo [170], libjpeg [171], icu [172], libvpx [173], treck [174]*.

Сопоставление основано на построении "*канонической формы*", которая получается из числовых констант, константных строк, количества инструкций ассемблера, размера фрейма стека, порядка вызовов функций. Но сопоставление только на основе "*канонической формы*" не масштабируется. Если в бинарном файле есть функция, которая полностью совпадает с некоторой функцией исходного кода и не похожа ни на одну другую функцию, то она считается "*якорем*". Используются следующие предположения (которые не всегда правильные):

1. Библиотека находится в непрерывном фрагменте в бинарном файле;
2. Все функции между якорями, у которых имеются самый маленький адрес и самый большой адрес, рассматриваются как функции библиотеки.

"*Якорные*" функции сопоставляются до получения "*канонического представления*". Данный факт ограничивает количество функций, для которых нужно построить "*каноническое представление*". "*Якоря*" создаются таким образом, чтобы однозначно идентифицировать библиотеку. Для каждой библиотеки рассматриваются все функции, которые имеют уникальные числовые константы и уникальные строки. Если константы достаточно сложны (числа с высокой энтропией или достаточно длинными строками) или могут быть сгруппированы так, чтобы быть достаточно уникальными (например, 3 уникальных строки средней длины в одной и той же функции), то функция помечается как якорь. Сопоставление происходит следующим образом:

1. Каждая поддерживаемая библиотека имеет идентификатор, с помощью которого определяется, находится ли она в бинарном файле, или нет. Идентификаторы основаны на уникальных строках в библиотеке (иногда с информацией о версии). Поскольку поддерживаемые библиотеки содержат уникальные строки, часто – с полной информацией о версии, большинство идентификаторов основано на строках и настроено для библиотеки, которую они пытаются идентифицировать. Как только библиотека найдена, инструмент пытается извлечь информацию о точной версии, которая используется бинарным файлом;
2. Вторым шагом является поиск якорных функций на основе уникальных числовых констант и уникальных строк, которые соответствуют функциям "*якорей*" библиотеки;
3. Используя диапазон сопоставленных функций (которые были определены соответствующими "*якорями*"), начинается сопоставление функций между ними. Для этого определяются диапазоны для каждого файла и делается поиск локально уникальных функций;

4. Последним шагом является сопоставление на основе "*канонического представления*".

В таблице 12 приводятся результаты инструмента.

Таблица 12. Результаты инструмента Karta.

Бинарный файл	**Архитектура**	**Компилятор, оптимизации**	**Количество функций**	**Найденная библиотека в бинарном файле**	**Количество сопоставленных функций**	**Количество ошибок первого рода**	**Количество ошибок второго рода**
openssl 1.0.1j	x64	gcc O3	5678	openssl 1.0.1j	1788	0	3850
openssl 1.0.1j (stripped)	x64	gcc O3	5678	openssl 1.0.1j	44	0	5600
tiffmedian 4.0.8 (часть libtiff)	x64	gcc O3	58	-	0	0	40
tiff2pdf 4.0.8 (часть libtiff)	x64	gcc O3	202	-	0	0	180
libtiff.so 4.0.8	x64	gcc O3	711	libtiff 4.0.8	201	0	480
libz.so 1.2.3 (часть zlib)	x64	gcc O3	159	zlib 1.2.11	62	29	100
libpng12.so 1.2.29	x64	gcc O2	484	libpng 1.2.29	130	3	320
libxml2.so	x64	gcc O2	2719	-	0	0	2680
openssl 1.0.1j	x86	gcc O3	5678	openssl 1.0.1j	1436	0	4200
libpng12.so 1.2.29	x86	gcc O2	494	libpng 1.2.29	86	0	390

Средний процент ошибок первого рода составляет 4.2%, ошибок второго рода – 79.3%. На основе проведенных исследований можно подчеркнуть следующие недостатки инструмента:

1. Поддерживается только несколько библиотек с фиксированными версиями;
2. Для поддержки библиотеки, или конкретной версии, нужно вручную добавлять уникальные идентификаторы, описанные выше;
3. Определение версии основано только на специальной константной строке, в

которой написана версия; иногда такой строки может и не быть;

4. При сопоставлении не учитывается поток данных;
5. Метод не справляется при встраивании функций.

2.4.5. Сравнение инструментов и заключение

В ходе исследования был проведен сравнительный анализ инструментов сопоставления исходного и бинарного кода: *Pigaios*, *RESource*, *CodeBin* и *Karta*. Исходный код *CodeBin* недоступен. Исходный код инструмента *RESource* доступен, но не поддерживается. *RESource* использует онлайн-анализ и требует соединения сети. Также необходимо подчеркнуть, что ряд онлайн кодовых баз, которые они поддерживали, на сегодняшний день недоступны. Карта предоставляет возможность сопоставления бинарного кода с кодами ряда библиотек. Исходный код инструмента доступен. Также восстанавливаются версии библиотек, которые статически слинкованы в бинарном файле. В среднем, ошибки первого рода составляют 4.2%, а ошибки второго рода – 79.3%. Однако инструмент имеет несколько серьезных недостатков (описаны выше). *Pigaios* принимает на входе исходный и бинарный коды и возвращает сопоставленные функции. Исходный код инструмента доступен. В среднем ошибки, первого рода составляют 17.9%, а ошибки второго рода – 72.8%. Инструмент не учитывает встраивание функций, поток данных, но активно разрабатывается и является самым предпочитаемым методом среди всех аналогов.

С учетом результатов существующих инструментов, задача качественного сопоставления исходных и бинарных файлов является актуальной.

2.5. Методы поиска утечек динамической памяти

В данном разделе приводится описание и сравнение существующих методов поиска утечек памяти.

2.5.1. Обзор инструмента SMOKE

SMOKE [175] использует два этапа для достижения высокой точности и масштабируемости. На первом этапе используется легкий, но неточный анализ для обнаружения всех потенциальных путей утечек памяти (отфильтровываются пути, которые не могут привести к утечке памяти). Для этого строится новое программное представление под названием граф потока использования ГПИ (use-flow graph). ГПИ содержит всю необходимую информацию о потоке управления для объектов кучи, включая порядок их использования в потоке управления. Каждое ребро ГПИ аннотировано условиями, при которых данный переход может произойти. ГПИ явно моделирует жизненный цикл указателей динамической памяти, создавая специальный узел "*out-of-scope*" (узел вне области видимости), который указывает, что объект кучи, на котором есть указатель, больше не используется.

На втором этапе *SMOKE* использует более точный анализ. Сначала он находит все пути, без операции освобождения памяти, ведущие от места выделения памяти к узлу – вне области видимости. Затем, для каждого обнаруженного пути он применяет решатель ограничений *Z3* [176], чтобы проверить его выполнимость. Это необходимо для фильтрации ложных срабатываний невыполнимых путей выделения/освобождения памяти.

Инструмент обеспечивает чувствительность к потоку, путям и контексту программ. Но имеет следующие недостатки:

1. Не чувствителен к полям;
2. Неточность анализа указателей в контексте ГПИ;

3. Не проверяет арифметические операция над указателями в результате чего не отличает *free(p + y)* от *free(p)*;
4. Не может учитывать библиотечные функции. Например, если есть библиотечные функции "*my_malloc*", "*my_free*" то созданные и освобожденные указатели этими функциями не будут проанализированы.

2.5.2. Обзор инструмента PCA

PCA [177] построен на базе *LLVM* [178]. На первом этапе инструмент компилирует исходные файлы в промежуточное представление *LLVM*. Затем, используя компоновщик *LLVM gold*, объединяет все файлы промежуточного представления в один и выполняет анализ указателей *Андерсена* [179]. С использованием полученной информации строится граф вызовов, который также содержит переходы, соответствующие косвенным вызовам. На последнем этапе строится межпроцедурный граф зависимостей данных (ГЗД) с использованием графа потока управления каждой функции, графа вызовов и информации указателей (каждый указатель на какое множество указывает).

Для обнаружения утечек памяти *PCA* собирает все узлы (обозначаем *N*) для заданной инструкции выделения памяти (обозначаем *A*), которые доступны на ГЗД из *A*. Если в *N* не содержится вызов функции "*free*", тогда происходит утечка памяти.

Инструмент обеспечивает только частичную чувствительность к потоку.

2.5.3. Обзор инструмента SVF

SVF [180] построен на базе *LLVM*. На первом этапе он компилирует исходные файлы в промежуточное представление *LLVM*. Затем, используя компоновщик *LLVM gold*, объединяет в один все файлы промежуточного представления. После этого вызывается модуль анализа указателей (доступно несколько вариантов, включая

анализ указателей Андерсена). На базе промежуточного представления *LLVM* и информации об указателях строится граф потока значений (ГПЗ). Узлы ГПЗ — это все переменные программы. Ребра строятся на основе информации об указателях и *use-def* [181] анализа. ГПЗ является ориентированным графом.

Инструмент *SVF* позволяет переключаться между модулями анализа указателей и построения ГПЗ для уточнения результатов обоих компонентов (анализ разреженных указателей выполняется на основе ГПЗ). *SVF* также обеспечивает разбиение памяти на области, что позволяет анализировать большие программы и рассматривать конкретные фрагменты. *SVF* позволяет реализовать разные детекторы на основе ГПЗ. Примером является детектор утечек памяти, который рассматривает задачу как проблему источника-приемника ("*source-sink*" проблема; каждое выделение памяти на любом пути должно достигать своего освобождения).

Инструмент обеспечивает чувствительность к потоку и контексту программ, но имеет следующие недостатки:

1. Нечувствителен к полям и путям программ;
2. Не масштабируется для анализа больших программ.

2.5.4. Обзор инструмента Fastcheck

Fastcheck [182] использует межпроцедурный алгоритм обнаружения утечек памяти в программах на *Cu*. Производится отслеживание потока значений от точек выделения динамической памяти к точкам их освобождения с использованием разреженного представления программы (разреженный граф потока значений). Ребра графа аннотированы условиями ветвления, при которых происходит присвоение значений. Ребра из инструкций вызовов и возврата помечены информацией о вызванной функции, что обеспечивает контекстно-чувствительность анализа. Обнаружение утечек памяти сводится к проблеме достижимости через разреженный

граф потока значений (если выделенная динамическая память не доходит до инструкций освобождения, тогда происходит утечка памяти). На первом этапе *Fastcheck* производит разбор исходного кода и строит граф потока управления. На втором этапе выполняется анализ *use-def* [181] цепочек для восстановления основного потока данных. На основе этой информации и регионального анализа указателей строится граф потока значений (ГПЗ). На данном этапе ребра ГПЗ пока не аннотированы условиями ветвления. *Fastcheck* для каждого выделения памяти пытается найти пути в ГПЗ с инструкциями его освобождения. Если таких путей нет, то сообщается об утечке памяти. В противном случае, извлекается подграф, относящийся к инструкции выделения памяти, и ребра этого подграфа аннотируются соответствующими условиями ветвления. Если существует хоть один выполнимый путь (проверяется *SMT*-решателем) без инструкции освобождения, тогда выдается сообщение об утечке памяти.

2.5.5. Обзор инструмента Clang Static Analyzer

Clang Static Analyzer (CSA) [183] производит поиск разных ошибок (в том числе поиск утечек памяти) путем символьного выполнения программ. Инструмент обеспечивает межпроцедурный анализ и чувствительность к путям выполнения. Для анализа используется специальная структура данных под названием разобранный граф (РГ). РГ строится на базе абстрактного синтаксического дерева и потока управления программы. Ядро анализа производит символьное выполнение, и во время обхода потока управления программы строится РГ. Все входные данные программы получают символьные значения. Все вызовы функций на путях выполнения встраиваются (если возможно). Это делается для того, чтобы контекст вызванной функции и все его пути выполнения стали доступными для анализа. Таким образом, рассматриваются все возможные пути выполнения программы. Вершины РГ содержат в себе следующую пару:

1. *ProgramPoint* – точка между двумя инструкциями в потоке управления программы;
2. *ProgramState* – абстрактное состояние программы (значения символьных переменных, стек вызовов функций и т.д.).

Ребром между двумя вершинами (*Point-1, State-1*), (*Point-2, State-2*) в РГ является инструкция программы. Инструкция находится между точками *Point-1, Point-2,* и ее выполнения обеспечивает переход программы из состояния *State-1* в *State-2*.

Для поиска конкретных типов ошибок (в том числе, утечек памяти) существуют отдельные детекторы, которым доступен РГ (пути потока управления и состояние программы после выполнения каждой инструкций). Утечки памяти находятся путем прослеживания указателей выделенных участков динамической памяти. Если существуют пути выполнения, на которых освобождение не производится, детектор выдает ошибки и соответствующие пути.

Несмотря на хорошие результаты, *CSA* имеет два важных ограничения:

1. Циклы выполняются ограниченное количество раз (по умолчанию 4) и после того, как лимит достигнут, РГ дальше не строится, и последующие инструкции и пути не анализируются;
2. Количество путей растет экспоненциально от количества инструкций ветвлений, и анализ достаточно больших программ невозможен.

2.5.6. Обзор инструмента Infer

Infer [184] позволяет найти утечки памяти и использование нулевых указателей для *Java* и *Cu/Cu++*. Во время анализа каждая функция представляется в виде:

{Precondition (P)} Command(C) {Postcondition(Q)} = Hoare3, где *P* и *Q* модели памяти по сепарационной логике, а *C* – инструкция или функция, которая

выполняется. Значение *Hoare3* истинно, если выполнение *C* приводит память из состояния *P* в *Q*. Для каждого пути графа вызовов последовательно выполняются соответствующие функциям тройки *Hoare3* и на основе состояния модели памяти *Q* обнаруживается утечка. Инструмент обеспечивает чувствительность к потоку, путям, полям и контексту программы.

2.5.7. Обзор инструмента PML Checker

PML Checker [185] получает на входе исходный код программы, после чего производя лексический и синтаксический анализы, получает абстрактное синтаксическое дерево (АСД). Из АСД строится граф потока управления и производится его оптимизация. Удаляются все базовые блоки и ребра, выполнения которых не влияют на выделение и освобождение динамической памяти. На базе графа потока управления строится поток данных для инструкций, производящий выделение динамической памяти. Инструмент рассматривает все пути выполнения программы. Если существует путь, на котором производится выделение памяти, и поток данных этой инструкций не доходит до инструкций освобождения, тогда считается, что существует потенциальная утечка памяти. Для окончательной проверки применяется символьный решатель. Данный инструмент обеспечивает чувствительность к потоку, путям и полям программы.

2.5.8. Сравнение инструментов и заключение

В таблице 13 приводится 6 примеров, полученных из реальных проектов, на которых были запущены доступные инструменты поиска утечек памяти.

Таблица 13. Примеры утечек памяти из реальных проектов.

Утечка-1	**Утечка-2**
#include <stdio.h> #include <stdlib.h>	#include <stdio.h> #include <stdlib.h>

```c
#include <malloc.h>

typedef struct node {
  struct node *next;
  int *value;
} node_type;
static node_type *create_node(int value) {
  node_type *p = (node_type *)malloc(sizeof(node_type));
  p->value = (int *)malloc(sizeof(int));
  p->next = NULL;
  return p;
}
static void delete_node_leak(node_type **list_pptr, int value) {
  node_type *list = *list_pptr, *prev = NULL, *t;
  while (list != NULL && *(list->value) != value) {
    prev = list;
    list = list->next;
  }
  if (list == NULL)
    return;
  if (*(list->value) == value) {
    if (prev == NULL) {
      t = list;
      *list_pptr = list->next;
      free(t); // t-value is not freed
    } else {
      t = list;
      prev->next = list->next;
      free(t); // t-value is not freed
    }
  }
}
static void free_nodes_correct(node_type *list) {
  node_type *tlist;
  while (list != NULL) {
    tlist = list->next;
    free(list->value);
    free(list);
    list = tlist;
  }
}
int main(void) {
  node_type *list = create_node(1);
  list->next = create_node(2);
  delete_node_leak(&list, 2);
  free_nodes_correct(list);
  return 0;
}
```

```c
#include <malloc.h>

typedef struct node {
  struct node *next;
  int value;
} node_type;
static void delete_node(node_type **list, int value) {
  node_type *l = *list, *p = NULL;
  while (l != NULL && l->value != value) {
    p = l;
    l = l->next;
  }
  if (list == NULL)
    return;
  if (l->value == value) {
    if (p == NULL) {
      node_type *t = l;
      *list = l->next;
      free(t);
    } else {
      node_type *t = l;
      p->next = l->next;
      free(t);
    }
  }
}
static void free_nodes(node_type *list) {
  node_type *t;
  while (!list) {
    t = list->next;
    free(list);
    list = t;
  }
}
int main(void) {
  int nodes = 0;
  node_type *head = NULL;
  while (nodes < 1000) {
    // two nodes added each time
    node_type *plist = (node_type *)malloc(sizeof(node_type));
    plist->value = 1;
    node_type *tmp = (node_type *)malloc(sizeof(node_type));
    tmp->value = 2;
    tmp->next = head;
    plist->next = tmp;
    head = plist;
    // only delete node with value=2
    delete_node(&head, 2);
    nodes++;
  }
  free_nodes(head);
  return 0;
}
```

Утечка-3

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
```

Утечка-4

```c
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
```

```
typedef struct node {
  struct node *next;
  int value;
} node_type;
static void free_nodes_leak(node_type *list) {
  node_type *t;
  while (list != NULL) {
    t = list->next;
    if (list->value == 2)
      free(list); // leak if list->value != 2
    list = t;
  }
}
int main(void) {
  node_type *list = (node_type *)malloc(sizeof(node_type));
  list->value = 1;
  node_type *tmp = (node_type *)malloc(sizeof(node_type));
  tmp->value = 2;
  tmp->next = NULL;
  list->next = tmp;
  free_nodes_leak(list);
  return 0;
}
```

```
typedef struct node {
  struct node *next;
  int value;
} node_type;
static node_type *list = NULL;
static void free_nodes_leak(void) {
  node_type *l = list, *t;
  while (l != NULL) {
    t = l->next;
    if (l->value == 2)
      free(l); // leak if l->value != 2
    l = t;
  }
}
void insert_node(node_type *x) {
  x->next = list;
  list = x;
}
int main(void) {
  node_type *node = (node_type *)malloc(sizeof(node_type));
  node->value = 1;
  node->next = NULL;
  insert_node(node);
  node = (node_type *)malloc(sizeof(node_type));
  node->value = 2;
  node->next = NULL;
  insert_node(node);
  free_nodes_leak();
  return 0;
}
```

Утечка-5

```
#include <stdlib.h>

typedef struct node {
  struct node *next;
  int *data;
} node_type;
node_type *alloc_node(void) {
  node_type *p;
  p = (node_type *)malloc(sizeof(node_type));
  p->data = (int *)malloc(sizeof(int));
  return p;
}
void free_node(node_type *p) {
  if (p->data != NULL)
    free(p->data);
  free(p);
}
void insert_node(node_type **head, node_type *node) {
  node->next = *head;
  *head = node;
}
void free_nodes(node_type *head) {
  node_type *node;
  node = head;
  while (node != NULL) {
    head = node->next;
    free(node); // p->data not freed
```

Утечка-6

```
#include <stdlib.h>

typedef struct node {
  struct node *next;
  int *data;
} node_type;
node_type *alloc_node(void) {
  node_type *p;
  p = (node_type *)malloc(sizeof(node_type));
  p->data = (int *)malloc(sizeof(int));
  return p;
}
void free_node(node_type *p) {
  if (p->data != NULL)
    free(p->data);
  free(p);
}
int always_false(void){
  return 0;
}
void insert_node(node_type **head, node_type *node) {
  if (always_false()) {
    node->next = *head;
    *head = node;
  }
}
void free_nodes(node_type *head) {
```

```
    node = head;
  }
}
int main(void) {
  node_type *p, *head = NULL;
  p = alloc_node();
  insert_node(&head, p);
  p = alloc_node();
  insert_node(&head, p);
  free_nodes(head);
  return 0;
}
```

```
  node_type *node;
  node = head;
  while (node != NULL) {
    head = node->next;
    free(node->data);
    free(node);
    node = head;
  }
}
int main(void) {
  node_type *p, *head = NULL;
  p = alloc_node();
  insert_node(&head, p); // p is not inserted into head
  p = alloc_node();
  insert_node(&head, p); // p is not inserted into head
  free_nodes(head);
  return 0;
}
```

Требуется обратить отдельное внимание на ***Утечку-2***. Фактически, там не происходит утечек памяти, а можно было бы в цикле производить освобождение, тем самым оптимизируя потребление памяти.

В таблице 14 приводятся результаты инструментов. Из полученных результатов становится очевидно, что доступные инструменты ***не могут*** находить утечки памяти с высоким качеством. ***Это делает разработку инструмента поиска утечек актуальной задачей***.

Таблица 14. Результаты доступных инструментов поиска утечек памяти.

Имя инструмента	**Утечка-1**	**Утечка-2**	**Утечка-3**	**Утечка-4**	**Утечка-5**	**Утечка-6**
CSA	Не находит	Не находит	Находит	Не находит	Не находит	Находит
Infer	Не находит	Не находит	Находит	Не находит	Не находит	Находит
SMOKE	Parse error	Parse error	Parse error	Parse error	Parse error	Parse error
PCA	Не находит	Не находит	Не находит	Не находит	Не находит	Не находит
Fastcheck	Не находит	Не находит	Находит	Parse error	Parse error	Parse error
SVF	Не находит	Не находит	Не находит	Не находит	Не находит	Не находит
PML Checker	Ошибка запуска	Ошибка запуска	Ошибка запуска	Ошибка запуска	Ошибка запуска	Ошибка запуска

В таблице 15 приведено сравнение чувствительности инструментов.

Таблица 15. Сравнение чувствительности инструментов.

Имя инструмента	**Чувствительность к потоку**	**Чувствительность к путям**	**Чувствительность к контексту**	**Чувствительность к полям**
CSA	+	+	+	**частично**
Infer	+	+	+	**частично**
SMOKE	+	+	+	-
PCA	-	-	-	-
Fastcheck	+	+	**частично**	-
SVF	+	-	+	-
PML Checker	+	+	+	+

2.6. Инструменты поиска ошибок использования освобожденной памяти

AddressSanitizer (ASan) [186] добавляет дополнительную информацию (вставляет дополнительные инструкций) об используемой памяти в приложение во время компиляции. Ошибки обнаруживаются при выполнении программы. *ASan* добавляет в каждый блок памяти метаданные, которые используются для проверки границ блока памяти во время выполнения. Во время выполнения программы производится проверка добавленных метаданных при каждом доступе к памяти. При обнаружении ошибки создается детальный отчет о проблеме, включающий стек вызовов, который привел к ошибке. *ASan* может выявить ошибки чтения из неинициализированных участков памяти, чтение или запись из освобожденной памяти, выход за пределы выделенного блока памяти, двойное освобождение памяти и т.д. *ASan* интегрирован в компилятор *clang/LLVM* и может быть использован вместе с *gcc*.

Valgrind [187] производит запуск целевой программы в специальной виртуальной машине, которая выполняет инструментацию кода и собирает информацию о его выполнении. Инструментация позволяет отслеживать все выделения и освобождения памяти, все обращения к файловой системе и другие системные вызовы. Кроме этого, *Valgrind* отслеживает изменения содержимого памяти и регистров процессора во время выполнения программы. После завершения программы выводится информация о ее работе, включая обнаруженные ошибки, утечки памяти, неправильные обращения к файловой системе и т.д. В *Valgrind* имеются несколько отдельных инструментов, предназначенных для анализа определенного аспекта выполнения программы. Например, *Memcheck* используется для поиска ошибок использования памяти, а *Cachegrind* для анализа кэш-памяти процессора и оптимизации производительности программы.

Механизм работы *Dr.Memory* [188] схож с *Valgrind*. Вместо запуска в виртуальной машине, программа запускается с помощью *DynamoRIO* [189], который

является системой манипулирования кодом во время выполнения. Системные вызовы целевой программы перехватываются и заменяются соответствующими из *Dr.Memory*, что позволяет собирать нужную информацию.

Кроме вышеуказанных инструментов *Clang Static Analyzer (CSA)* [183], *SVF* [180] также могут находить ошибки использования освобожденной памяти. Принцип работы этих инструментов описан в предыдущем разделе.

2.6.1. Сравнение инструментов и заключение

Для сравнения описанных доступных инструментов мы использовали набор реальных проектов из статьи [190], где приводится полный обзор и сравнение существующих инструментов поиска ошибок использования освобожденной памяти. В таблице 16 приводятся полученные результаты.

Таблица 16. Сравнение результатов поиска ошибок использования освобожденной памяти

		ASan	**Valgrind**	**Dr.Memory**	**CSA**
Имя теста	**Номер уязвимости**				
gohttp	CVE-2019-12160	+	+	+	+
libheif	CVE-2019-11471	+	-	-	-
lrzip	CVE-2018-11496	+	-	-	-
lua	CVE-2019-6706	+	+	+	-
nasm	CVE-2019-8343	+	+	+	-
nasm	CVE-2018-20535	+	+	+	-
binaryen	CVE-2019-7703	+	-	-	-
jasper	CVE-2015-5221	+	+	+	-
binutils	CVE-2018-20623	+	-	-	-

Как видно из результатов, наилучший результат показал Asan, который является динамическим методом выявления ошибок. Несмотря на хорошие результаты в данном случае, такой подход имеет серьезный недостаток. Если выходные данные не приводят к пути выполнения ошибки, тогда ошибка не будет найдена. А другие подходы не обеспечивают нужный уровень точности. Таким образом, становится очевидно, что задача поиска ошибок использования освобожденной памяти остается актуальной.

2.7. Методы динамического анализа программ

В данном разделе приводится описание существующих методов и инструментов фазз-тестирования (фаззинга).

2.7.1. Инструмент AFL

AFL (American Fuzzy Lop) [191] является эффективным инструментом фаззинга, использующим информацию о покрытии кода целевой программы. Он широко используется для обнаружения ошибок безопасности в различных приложениях. *AFL* применяет генетический алгоритм для генераций/мутаций входных данных. Для получения покрытия кода целевой программы производится его инструментация. В базовые блоки программы добавляются инструкции, которые позволяют определить, какие входные данные обеспечивают выполнение новых путей кода. *AFL* производит приоритезацию входных данных на основе покрытия кода, чтобы как можно скорее достичь большего покрытия кода. С использованием *AFL* были найдено множество уязвимостей:

1. Ряд уязвимостей в криптографической библиотеке *openssl* [36]. Некоторые из найденных уязвимостей включают: *CVE-2015-0291*, *CVE-2015-0292* и *CVE-2015-0289* – разыменование нулевого указателя; *CVE-2015-0207*, *CVE-2015-0208* и *CVE-2015-0286* – падение программы;

2. Ряд уязвимостей в инструменте *ImageMagick* [192] работающим с изображениями. Некоторые из найденных уязвимостей включают: *CVE-2016-5687* – чтение за пределами массива; *CVE-2016-5688* – переполнение буфера; *CVE-2016-5689* – разыменование нулевого указателя;
3. Ряд уязвимостей в веб-браузере *Mozilla Firefox* [193]. Некоторые из найденных уязвимостей включают: *CVE-2014-1564*, *CVE-2014-8637* и *CVE-2014-1580*, неинициализированную память, которая позволяет получать конфиденциальную информацию из памяти процесса.

2.7.2. Инструмент LibFuzzer

LibFuzzer [194] – специальная библиотека фаззинга в компиляторной инфраструктуре *LLVM*. Она предназначена для фаззинга отдельных функций, написанных на *Cu/Cu++*. *LibFuzzer* использует покрытие кода, чтобы определить входные данные, обеспечивающие выполнение новых путей в коде. Входные данные, которые обеспечивают прирост покрытия, используются в дальнейших мутациях. *LibFuzzer* может использовать доступные санитайзеры во время сборки проекта/теста. С использованием *LibFuzzer* было найдено множество уязвимостей в ряде известных проектов:

1. В криптографической библиотеке *openssl* найдена утечка памяти *CVE-2016-0798*;
2. В библиотеке парсера *XML* формата *libxml2* [195]. Некоторые из найденных уязвимостей включают: *CVE-2015-7500* – падение программы; *CVE-2015-7942* – чтение за границы буфера;
3. Множество падений в таких известных проектах как *radare2* [196], *tensorflow* [197], *ffmpeg* [198], *wireshark* [199], *QEMU* [200], *SQLite* [117]. и т.д.

2.7.3. Инструмент Peach

Peach [201] поддерживается на нескольких платформах и предназначен для фаззинга протоколов и разных форматов файлов. Он имеет модульную архитектуру, которая позволяет легко расширять и настраивать его для конкретных нужд тестирования. Инструмент производит фаззинг на основе *XML*-конфигурационного файла под названием: *Peach pit*. В нем можно описать протокольные автоматы, контрольные суммы от конкретных полей и т.д., что позволяет генерировать валидные пакеты протокола. С инструментом предоставляется набор конфигурационных файлов, включая такие известные проколы, как: *HTTP*, *SSL* и т.д.

2.7.4. Другие инструменты фаззинга

Syzkaller [202] предназначен для фаззинга ядер разных ОС. Поддерживаются *Linux* и *Windows*. Инструмент содержит описания системных функций, на основе которых для каждой системной функции генерируется маленькая программа с ее вызовом. Далее, сгенерированная программа запускается, и *Syzkaller* определяет, произошла ли ошибка в системе.

GramFuzz [203] позволяет определять правила грамматик, на основе которых генерирует данные для фаззинга. Инструмент используется для фаззинга браузеров. Сначала рассматриваются множества примеров программ на *HTML*, *CSS* и *JavaScript*. Для этих примеров строятся абстрактные синтаксические деревья (АСД), и для каждого типа узла получается множество возможных элементов, которые он может принимать. Далее, для фаззинга выбирается исходный пример программы и для него строится АСД. После этого, путем обхода в глубину, вершины АСД заменяются возможными значениями, полученными на предыдущих шагах. Возможны также следующие операции: замена текущей вершины на вершину произвольного типа и копирование вершины в другое произвольное место.

Кроме вышеуказанных инструментов, существует ряд других инструментов фаззинга, в том числе:

1. *Sulley* [204] нацелен на фаззинг разных протоколов и форматов файлов;
2. *Honggfuzz* [205] – инструмент фаззинга общего назначения, который может работать в параллельном и распределенном режиме;
3. *JFuzz* [206] – платформа фаззинга на базе Java, которая может быть интегрирована в процесс разработки;
4. *Radamsa* [207] обеспечивает механизмы гибкого контроля генерируемых входных данных, и позволяет генерировать разные форматы.

2.7.5. Проект OSS-Fuzz

OSS-Fuzz [35] позволяет производить продолжительное фазз-тестирование с разными доступными инструментами на больших распределенных мощностях. Проект нацелен на улучшение безопасности открытого программного обеспечения. В проекте доступны следующие инструменты фаззинга: *libFuzzer*, *AFL++* и *Honggfuzz*. *OSS-Fuzz* нашел более 8,900 уязвимостей и более 28,000 ошибок в более чем 850 открыто-доступных проектах.

2.7.6. Заключение по существующим инструментам фаззинга

Несмотря на большое количество доступных инструментов фаззинга, нацеленные на разные задачи, остается множество сценариев тестирования, которые либо не поддерживают существующие инструменты, либо производят их недостаточно эффективно. Например:

1. Направленный фаззинг;
2. Эффективный фаззинг программ, принимающий сложно структурированные (БНФ – Бэкуса-Наура форма) данные;

3. Фаззинг интерфейсных функций, в том числе для платформ интернета вещей, с возможностью формирования цепочек вызовов;
4. Интеграция динамического символьного выполнения и статического анализа с фаззингом.

В рамках данной работы будут разработаны методы дополняющие существующие недостатки.

2.8. Заключение по анализу существующих технологий

Во второй главе представлен детальный обзор и сравнительный анализ существующих методов поиска клонов в исходных и бинарных файлах, сопоставления исходных и бинарных файлов, анализа изменений между версиями ПО, поиска утечек динамической памяти и ошибки использования освобожденной памяти, фаззинга программ и интерфейсных функций. Несмотря на большое количество доступных методов, существуют серьезные ограничения на класс задач, для которых они могут быть эффективно применены. Необходимо, с одной стороны, заниматься снятием этих ограничений, развивая предложенные методы, а, с другой стороны, при работе над ними целесообразно рассматривать их не по отдельности, а в совокупности, учитывая вклад, который по отдельности привносят методы, в общей картине обеспечения безопасности программ. Представляется, что такое совместное развитие и использование методов позволит получить новое качество анализа.

3. Методы поиска клонов кода и их применение

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

Основная часть инструмента поиска клонов исходного кода была разработана в рамках диссертационной работы автора [208]. В разделе 3.1 представлено краткое описание разработанного метода. В разделе 3.1.1 приведено сравнение реализованного метода с существующими методами, для того, чтобы продемонстрировать эффективность подхода. С учетом успешного опыта, схожий подход был использован для поиска клонов бинарного кода. Функционал, описанный в разделе 3.1.2, уже реализован в рамках данной работы и необходим для интеграции в общую платформу разработанной технологии. Инструмент поиска клонов бинарного кода и другие технологии, базирующиеся на поиске клонов, полностью разработаны в рамках данной работы. *Полученные новые результаты в рамках исследований в данной главе выносятся на защиту в виде масштабируемых и точных методов нахождения клонов кода, а также базирующегося на них метода сопоставления исходных и бинарных файлов*.

3.1. Методы поиска клонов исходного кода

Данный метод был разработан в рамках диссертационной работы автора [208]. Метод основан на семантическом анализе программ и может проанализировать проекты с сотнями миллионов строк исходного кода. Он состоит из двух основных частей. Первая часть создает ГЗП из промежуточного представления *LLVM* в ходе компиляции проекта (рисунок 3). Вторая часть инструмента отвечает за анализ ГЗП в целях нахождения клонов. Поиск клонов кода производится в четыре этапа:

разделение ГЗП на ЕС, фильтрация несхожих пар ЕС, поиск максимально схожих подграфов, фильтрация ложных срабатываний.

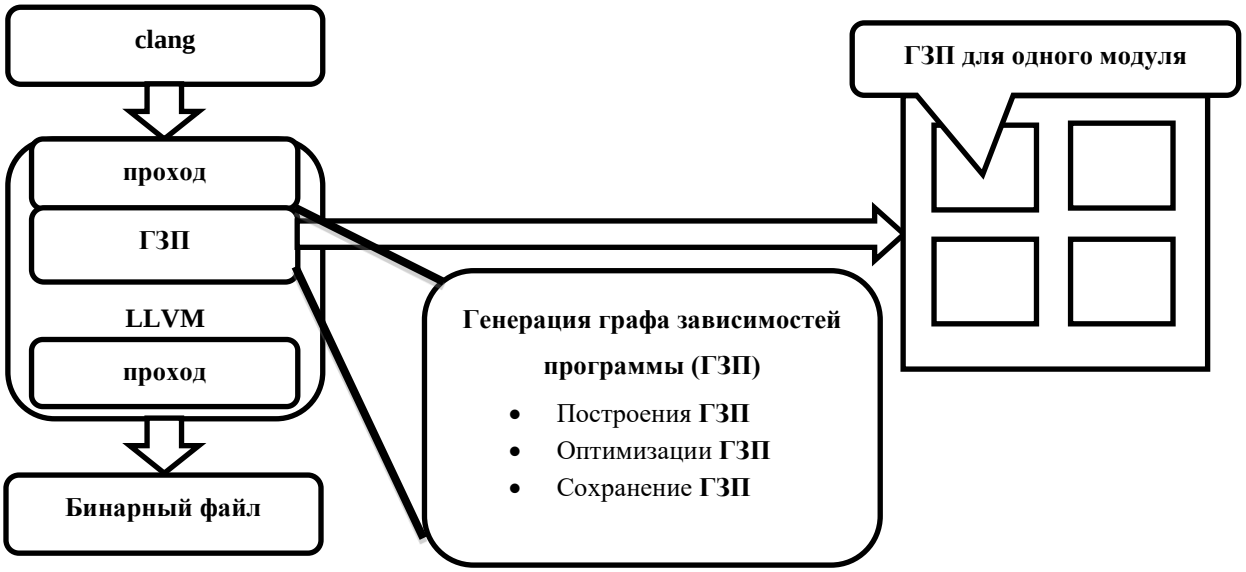

Рисунок 3. Архитектура инструмента, генерация ГЗП.

На рисунке 4 приводится пример ГЗП для простого фрагмента кода.

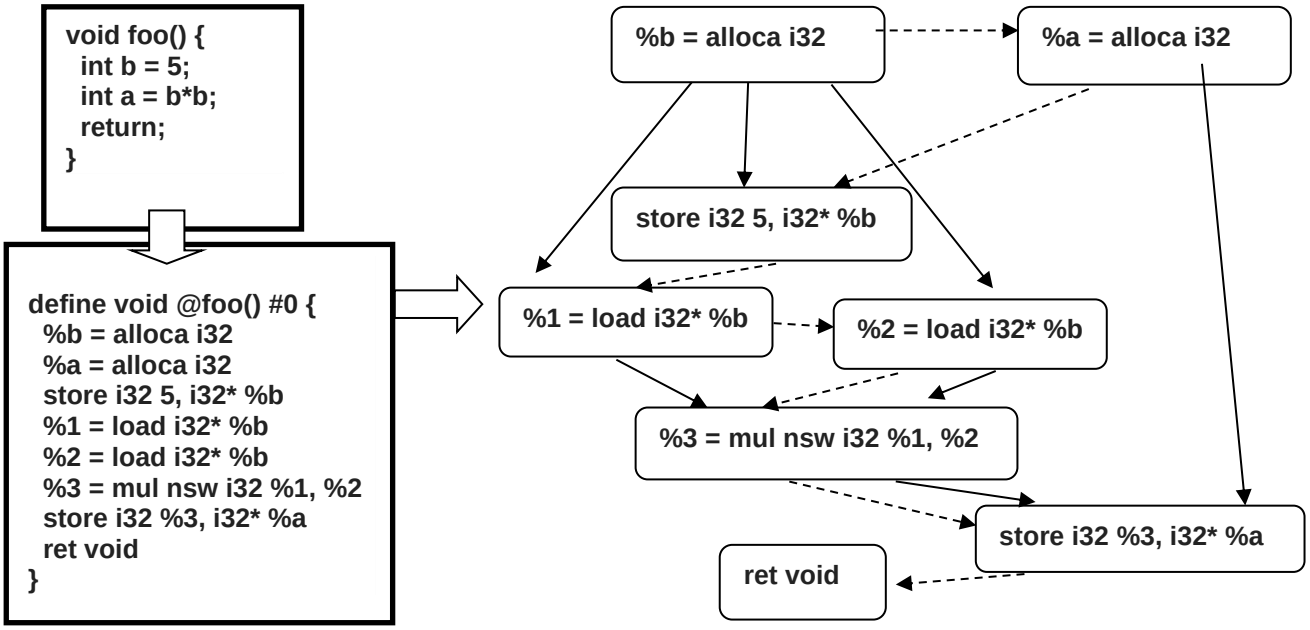

Рисунок 4. Пример ГЗП.

Для разделения ГЗП на ЕС были разработаны два основных метода. Первый метод разделяет ГЗП на слабо связанные компоненты, каждый из которых считается отдельной ЕС. Второй метод разделяет граф на такие множество подграфов, где каждому соответствует фрагмент кода с последовательными строками кода. Более подробное описание метода приводится в диссертационной работе автора [208].

После разделения ГЗП на ЕС начинается поиск клонов кода. Сначала применяются алгоритмы линейной сложности для отсева несхожих пар. Примером может служить простая проверка в паре ЕС достаточного количества одинаковых инструкций. Если в одном ЕС большинство инструкций не имеет схожих во втором ЕС, тогда схожие подграфы нужного размера точно не смогут быть найдены, так как не будут сопоставлены вершины графов. Как показывает практика большинство пар ЕС не являются клонами и быстро фильтруются. После фильтраций применяются разработанные алгоритмы поиска схожих подграфов. Более подробное описание всех алгоритмов приводится в диссертационной работе автора [208].

Необходимо также отметить, что на базе разработанной технологии поиска клонов исходного кода [208] автором был реализован инструмент поиска семантических ошибок в копированных фрагментах кода. Этот инструмент входит в состав *Svace* [46], являющимся индустриальным стандартом для жизненного цикла разработки безопасного ПО, и внедренного в таких компаниях, как "*Базальт СПО*" и ООО "*РусБИТех-Астра*".

3.1.1. Экспериментальные результаты

В таблице 17 приводится сравнение существующих инструментов с разработанным (*CCD*). Данный эксперимент проведен в диссертационной работе автора [208]. В качестве тестового набора берутся примеры, описанные в таблице 4. Как видно из таблицы только *CCD* умеет находить все клоны кода. В таблице 18 приводится сравнение времени работы разработанного и существующих открыто-

доступных инструментов. Тестирование было произведено на *Intel core i5* с 16Гб памятью. Как видно из таблицы, в среднем, разработанный автором инструмент работает в 3-4 раза медленнее, но может найти все три типа клонов кода. Причиной медленной работы является семантический подход поиска клонов кода. Но надо отметить, что *CCD* может эффективно распараллеливаться и, при достаточных вычислительных ресурсах, обеспечить нужную скорость работы.

Таблица 17. Сравнение доступных инструментов с разработанным

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	+	+	+	+	+	-	+	+

Таблица 18. Сравнение времени работы.

Имя теста	CCFinder	PMD/CPD	SourcerCC	VUDDY	Deckard	NiCad	CCD
Linux kernel	600с. **"Не находит клоны"**	3519с.	1340с.	4150с.	После 2451с. ошибка: **"Error: problem in clustering step"**	2869с.	13960с.
Firefox Mozilla	**"Error: failure to parse file"**	3203с.	560с.	2307с.	389с.	После 405с. ошибка: **"Clone analysis failed, code 139"**	11340с.

3.1.2. Улучшения, произведенные в инструменте с целью интеграции в общую платформу

Первое улучшение касается применения инструмента для двух разных версий одного проекта. Такое применение инструмента может быть важным, когда стоит задача анализа изменений между версиями проектов. Для решений этой задачи в инструмент были добавлены:

1. Поддержка сохранения графа вызовов функций;
2. Возможность предварительного сопоставления функций на основе графа вызовов.

Был разработан программный интерфейс (*API*) на базе *Python*, который позволяет иметь доступ к ГЗП и графу вызовов. Интерфейс поддерживает все базовые функциональности для работы с графами: доступ ко всем вершинам, для данной вершины – все входящие и выходящие ребра и т.д. Разработанный интерфейс позволяет писать плагины, которые будут реализовывать множество других функционалов. Примером может служить анализ характера изменений кода и т.д.

3.2. Методы поиска клонов бинарного кода

В этом разделе приводится описание разработанных автором методов поиска клонов бинарного кода. Предлагаемый метод является платформенно-независимым и может проанализировать исполняемые файлы архитектур: *x86*, *x86-64*, *ARM*. Работа метода состоит из трех основных этапов: генерация ГЗП для бинарных функций, разделение ГЗП на ЕС и поиск клонов кода.

Для генерации ГЗП сначала дизассемблируются исполняемые файлы с помощью *IDA Pro* и транслируются в *REIL* (*Reverse Engineering Intermediate Language*) представление платформы обратной инженерии *Binnavi* [209]. 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 происходит разделение ГЗП на ЕС, которые представляют собой подграфы ГЗП. Для пар ЕС эвристическим алгоритмом считается максимальный общий подграф, и, если он соответствует нужным критериям (количество инструкций, степень схожести), то соответствующие фрагменты кода выдаются как клоны.

3.2.1. Генерация ГЗП

Для генерации ГЗП используются средства и возможности платформы *Binnavi*. Последний предоставляет интерфейс для генерации и использования различных промежуточных представлений программы, основанных на языке *REIL*, в том числе генерацию графа потока управления, генерацию графа вызовов функций и графа зависимостей по данным. В рамках разработки инструмента в платформу *Binnavi* был добавлен новый функционал, который позволяет для каждой функции автоматически генерировать граф потока управления и граф зависимостей по данным и объединять их в единый ГЗП (рисунок 5). После этого созданные графы сохраняются для дальнейшего анализа.

3.2.2. Разделение ГЗП на подграфы

Если размер ГЗП большой (в одном ГЗП может быть более одного клона желаемого размера), то производится разделение на единицы сравнения (ЕС). ЕС-ы представляют собой подграфы ГЗП и рассматриваются как потенциальные клоны друг друга. Так как графы изначально генерируются для функций, перед сравнением производится разделение графов на ЕС (содержит, как минимум, одну вершину). Разработаны два метода для решения этой задачи: разделение с учетом базовых

блоков кода и разделение по слабо связанным компонентам графа. Разделение графов также позволяет увеличить эффективность алгоритмов анализа ГЗП (алгоритмы поиска максимальных изоморфных подграфов).

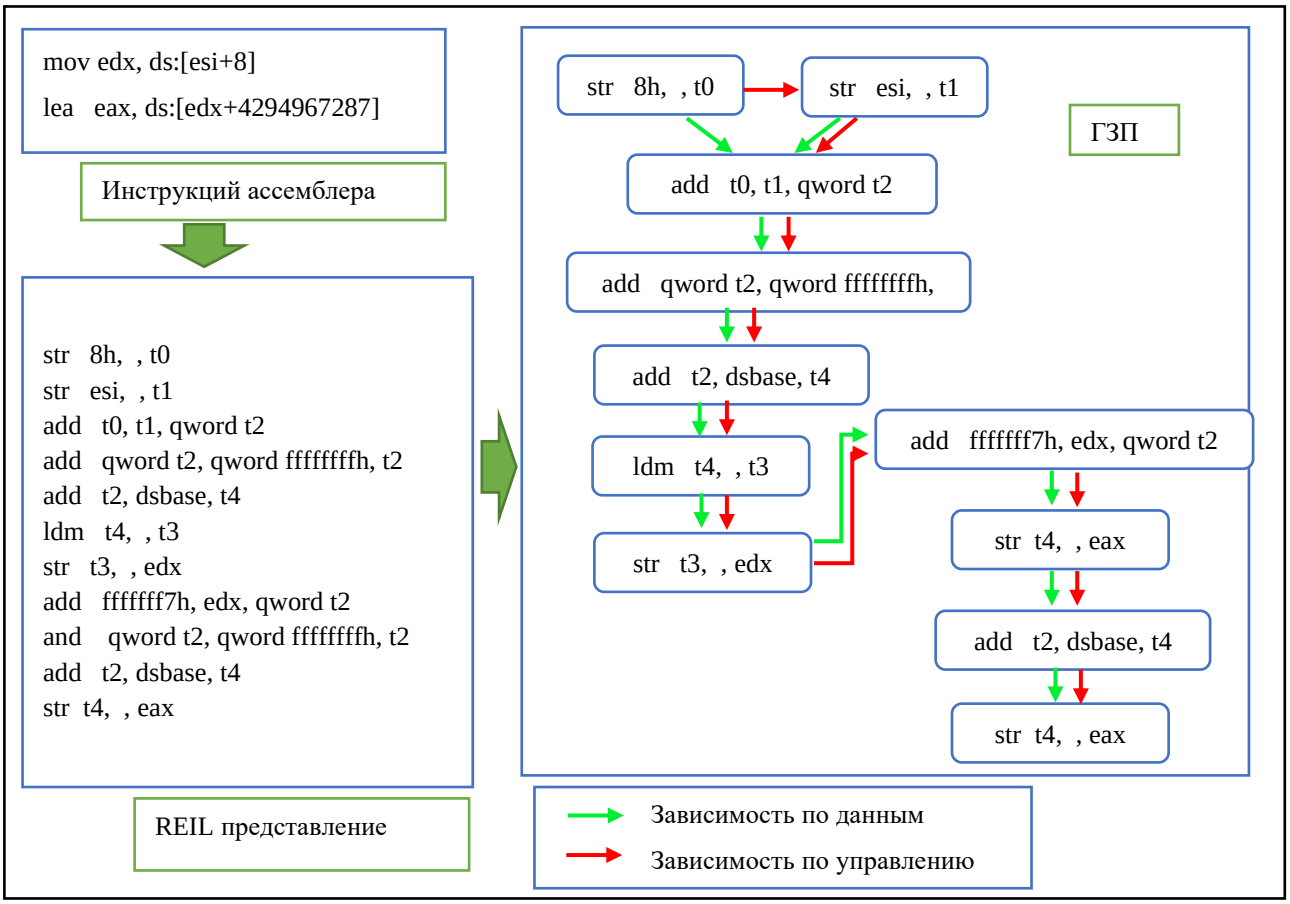

Рисунок 5. Пример графа зависимостей программы на базе REIL.

3.2.3. Анализ пар ГЗП

На этом этапе для каждой пары ГЗП строится наибольший общий подграф. Как известно, эта проблема в общем случае *NP*-полная, и для решения проблемы используется приближенный алгоритм. В большинстве случаев созданные графы разреженные, что и используется для более эффективного нахождения наибольшего общего подграфа. На первом шаге алгоритма производится фильтрация пар ГЗП на основе минимального числа инструкций для конечных клонов, задаваемой пользователем. Если пара ЕС не содержит заданное количество совпадающих

инструкций с одинаковыми кодами операций, тогда поиск наибольшего общего подграфа не производится. Большинство пар ЕС обрабатываются именно этим алгоритмом. Ниже приводится формальное описание алгоритма:

Алгоритм 1. *Алгоритм фильтрации пар ЕС*

```
1.  filterOfComparationUnites(EC1, EC2, minLength) {
2.    MAP1 = {}
3.    MAP2 = {}
4.    for (v in EC1) {
5.     if (v.opcode() in MAP1) {
6.       MAP1[v.opcode()]++
7.     } else {
8.       MAP1[v.opcode()] = 1
9.     }
10.   }
11.   for (v in EC2) {
12.    if (v.opcode()  in MAP2) {
13.      MAP2[v.opcode()]++
14.    } else {
15.      MAP2[v.opcode()] = 1
16.    }
17.   }
18.   commonNodes = 0
19.   for (vOpcode in MAP1) {
20.    if (vOpcode in MAP2) {
21.      commonNodes += min(MAP1[vOpcode], MAP2[vOpcode])
22.    }
23.   }
24.   if (commonNodes > minLength) {
25.    return true
26.   }
27.   return false
28. }
```

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

Доказательство:

Предположим, *EC1* и *EC2* содержат *n1* и *n2* вершин (*n = n1 + n2*). В алгоритме структура данных, соответствующая переменным *MAP1* и *MAP2*, является красно-черным деревом [210]. Из этого следует, что сложность операций доступа к конкретному элементу для *MAP1* – $O(\log_2 n1)$, *MAP2* – $O(\log_2 n2)$. Из этого следует, что существуют положительные константы *C1* и *C2* – такие, что удовлетворяются условиям: $|MAP1[vOpcode]| \leq C1 * \log_2 n1$ и $|MAP2[vOpcode]| \leq C2 * \log_2 n2$. В алгоритме в общей сумме производит не более:

$n1 * C1 * \log_2 n1 + n2 * C2 * \log_2 n2 + n1 * C2 * \log_2 n2$ операций. Из этого следует, что $n1 * C1 * \log_2 n1 + n2 * C2 * \log_2 n2 + n1 * C2 * \log_2 n2 <$

$3 * (C1 + C2) * (n1 + n2) * \log_2(n1 + n2) =$

$C * n * \log_2 n$, где *C = C1 + C2* а *n = n1 + n2*.

Из этого следует, что сложность алгоритма $O(n * \log_2 n)$. ***Теорема доказана***.

Вторая часть алгоритма находит наибольший общий подграф. Сначала сопоставляются те вершины в графах, которые не имеют входных ребер. В следующих итерациях рекурсивно рассматриваются и сопоставляются вершины, связанные ребром с вершинами, которые сопоставлялись в предыдущих итерациях. После нахождения наибольшего общего подграфа восстанавливаются соответствующие фрагменты бинарного кода.

3.2.4. Фильтрация результатов

Так как ГЗП строится из представления *REIL*, в некоторых случаях полученные машинные инструкции, соответствующие изоморфным подграфам, могут быть сильно разбросаны и не являться клонами. По этой причине возникает дополнительная необходимость фильтрации полученных клонов. Основной критерий фильтраций — это дозволенный процент разбросанности найденных фрагментов.

3.2.5. Результаты и заключение

В таблицах 19 и 20 приводятся результаты сопоставления функций двух бинарных файлов. Берутся разные версии одного и того же проекта и компилируются с разными опциями/компиляторами для получения различных бинарных версий функций. Далее, запускается инструмент поиска клонов кода и проверяются найденные клоны. Пара функций считается сопоставленной, если найдено более 90% сопоставления (от количества инструкций меньшего по размеру). Необходимо отметить, что результаты инструмента ухудшаются, когда производится сравнение бинарных файлов, полученных с оптимизациями *-O0* и *-O3*, поскольку компилятор агрессивным образом изменяет код. А в случае сравнения бинарных файлов, полученных с оптимизациями *-O2* и *-O3*, результаты достаточно хорошие. В таблице 21 показаны результаты сравнения разработанного инструмента *binCCD* с доступным, наилучшим (раздел *2.2.3*, обзор инструментов) инструментом *BinDiff*. Из результатов становится очевидно, что разработанный инструмент в большинстве случаев охватывает и превосходит результаты BinDiff.

Таблица 19. Результаты разработанного инструмента binCCD.

Бинарный файл 1	**Компилятор, оптимизации 1**	**Бинарный файл 2**	**Компилятор, оптимизации 2**	**Количество правильных сопоставлений**	**Количество неправильных сопоставлений**
python-3.5.1	gcc-7.2 O2	python-3.5.2	gcc-7.2 O2	3936	8
python-3.5.2	gcc-7.2 O2	python-3.6.3	gcc-7.2 O2	3456	489
openssl-1.0.1f	gcc-5.4 O2	openssl-1.0.1f	gcc-4.8 O2	4538	809
openssl-1.0.1f	clang-5.0 O2	openssl-1.0.1f	gcc-4.8 O2	3496	1551
postgreSQL-9.5.3	gcc-5.4 O2	postgreSQL-9.5.3	gcc-4.8 O2	9370	753
postgreSQL-9.5.3	clang-5.0 O2	postgreSQL-9.5.3	gcc-5.4 O2	9358	765
php-7.1.10	gcc-5.4 O2	php-7.1.10	gcc-4.8 O2	7917	839
php-7.1.10	clang-5.0 O2	php-7.1.10	gcc-4.8 O2	8849	15
openssl-1.0.1f	**gcc-5.4 O0**	**openssl-1.0.1f**	**gcc-5.4 O3**	**888**	**4051**
openssl-1.0.1f	**gcc-5.4 O2**	**openssl-1.0.1f**	**gcc-5.4 O3**	**4695**	**736**

Таблица 20. Количество функций в бинарных файлах.

Бинарный файл	**Компилятор, оптимизации**	**Количество функций в бинарном файле**
python-3.5.1	gcc-7.2 O2	3944
python-3.5.2	gcc-7.2 O2	3944
openssl-1.0.1f	gcc-4.8 O2	5436
openssl-1.0.1f	gcc-5.4 O0	6134
openssl-1.0.1f	gcc-5.4 O2	5429
openssl-1.0.1f	gcc-5.4 O3	5429
openssl-1.0.1f	clang-5.0 O2	5045
postgreSQL-9.5.3	gcc-4.8 O2	10299
postgreSQL-9.5.3	gcc-5.4 O2	10124
postgreSQL-9.5.3	clang-5.0 O2	10297
php-7.1.10	gcc-4.8 O2	8866
php-7.1.10	gcc-5.4 O2	8856
php-7.1.10	clang-5.0 O2	8864

Таблица 21. 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

3.3. Сопоставление исходных и бинарных файлов

Приводится описание разработанного автором метода сопоставления исходных и бинарных файлов. Для этого исходный код компилируется в бинарные файлы с использованием разных оптимизаций компилятора (*-O0*, *-O1*, *-O2*, *-O3*). При компиляции также сохраняется отладочная информация. Затем производится сопоставление всех полученных и входных бинарных файлов (рисунок 6). Для этого используется разработанный инструмент поиска клонов бинарного кода (см. *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. Одним из ключевых моментов этого подхода является перебор параметров компиляции для исходного кода и получение 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 различные сценарии встраивания функций).

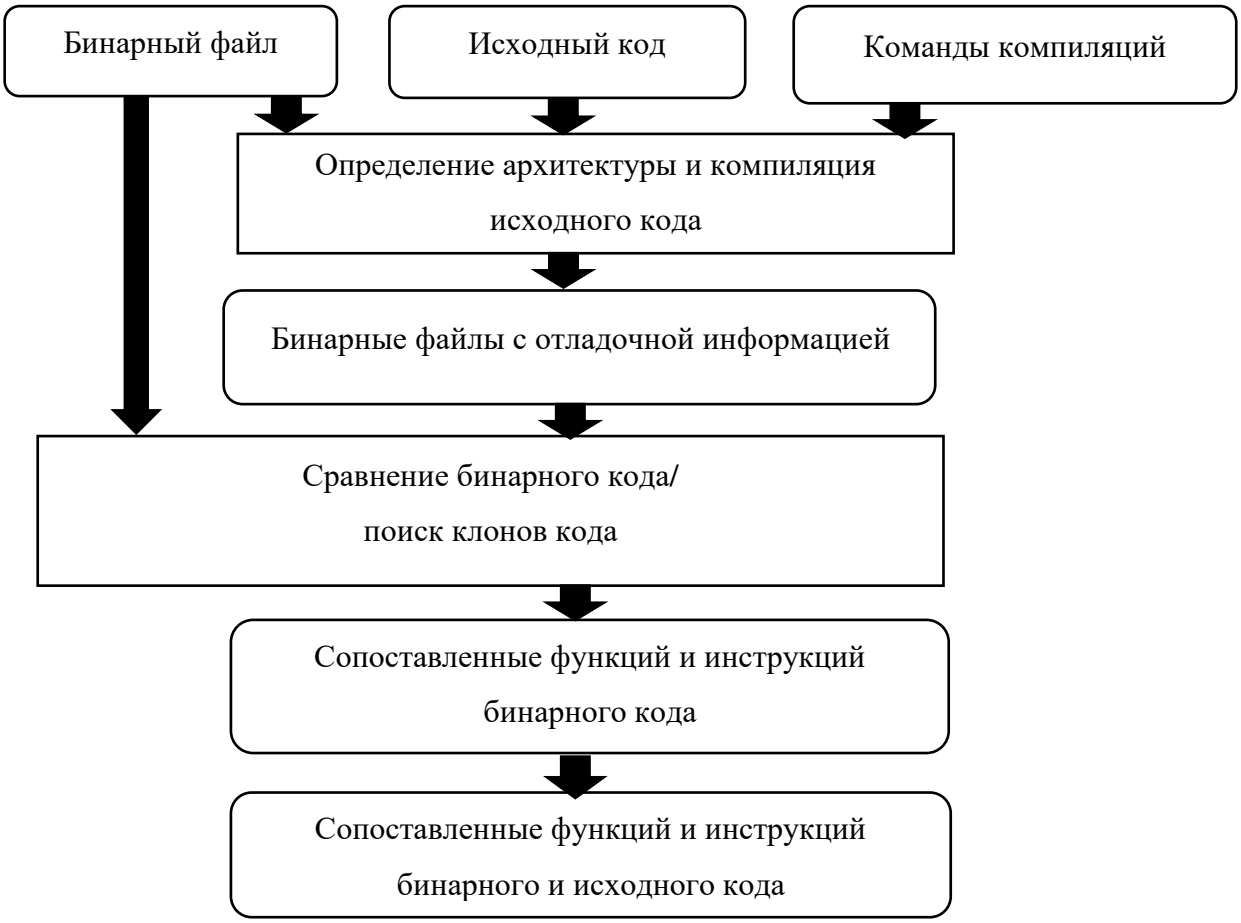

Рисунок 6. Схема работы инструмента (Bin2Source) сопоставления исходного и бинарного кода.

3.3.1. Компиляция исходного файла в бинарный код целевой архитектуры

Разработанный инструмент (*Bin2Source*) компилирует исходный код в бинарный код соответствующей архитектуры входного бинарного файла. Для определения архитектуры входного бинарного файла используется инструмент *Linux file* [211]. Также предоставляется возможность указания архитектуры пользователем. Далее, полученная или указанная архитектура входного бинарного файла используется для создания соответствующей виртуальной машины (*QEMU* [200]) для компиляции входного исходного кода. Созданной виртуальной машине передаются команды сборки/компиляции и производится сборка исходного кода с сохранением

отладочной информации. Отладочная информация используется для восстановления соответствий строк исходного кода с инструкциями бинарного кода. Инструмент поддерживает несколько целевых архитектур: *x86*, *x86_64*, *ARM32* и *MIPS*.

3.3.2. Сопоставление бинарных инструкций

Для сопоставления используется разработанный инструмент поиска клонов бинарного кода *binCCD* (см. *раздел 3.2*). В инструмент добавлена новая эвристика для улучшения результатов сопоставления бинарных функций. Вместо того, чтобы сразу рассматривать все пары функций из двух бинарных файлов и производить поиск клонов кода, выполняется начальное сопоставление функций на базе содержащихся уникальных константных строк. Для найденных пар функций запускается поиск клонов кода, и если *binCCD* сопоставляет функции больше, чем на 90%, тогда данное сопоставление подтверждается. Такой подход устойчив к встраиванию функций благодаря тому, что рассматриваются только уникальные константные строки. При дальнейшем сопоставлении пар функций могут также быть использованы эвристики на базе графов вызовов. Поскольку уже имеется некоторое количество сопоставленных функций, то можно рассматривать вызывающие и вызванные функции для сопоставленных пар. Если в результате остаются несопоставленные пары, тогда процесс может быть повторен, но уже с требованием меньшей процентности совпадения.

3.3.3. Сопоставление бинарных инструкций со строками исходного кода

Сопоставление инструкций бинарного кода со строками исходного кода получается из сопоставленных пар бинарных инструкций и отладочной информации. Как видно из рисунка 7, несколько инструкций бинарного кода сопоставлены с одной строкой исходного кода.

Рисунок 7. Пример сопоставления исходного и бинарного кода.

3.3.4. Тестовая система

Для проверки качества инструмента разработана специальная тестовая система (рисунок 8). 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.

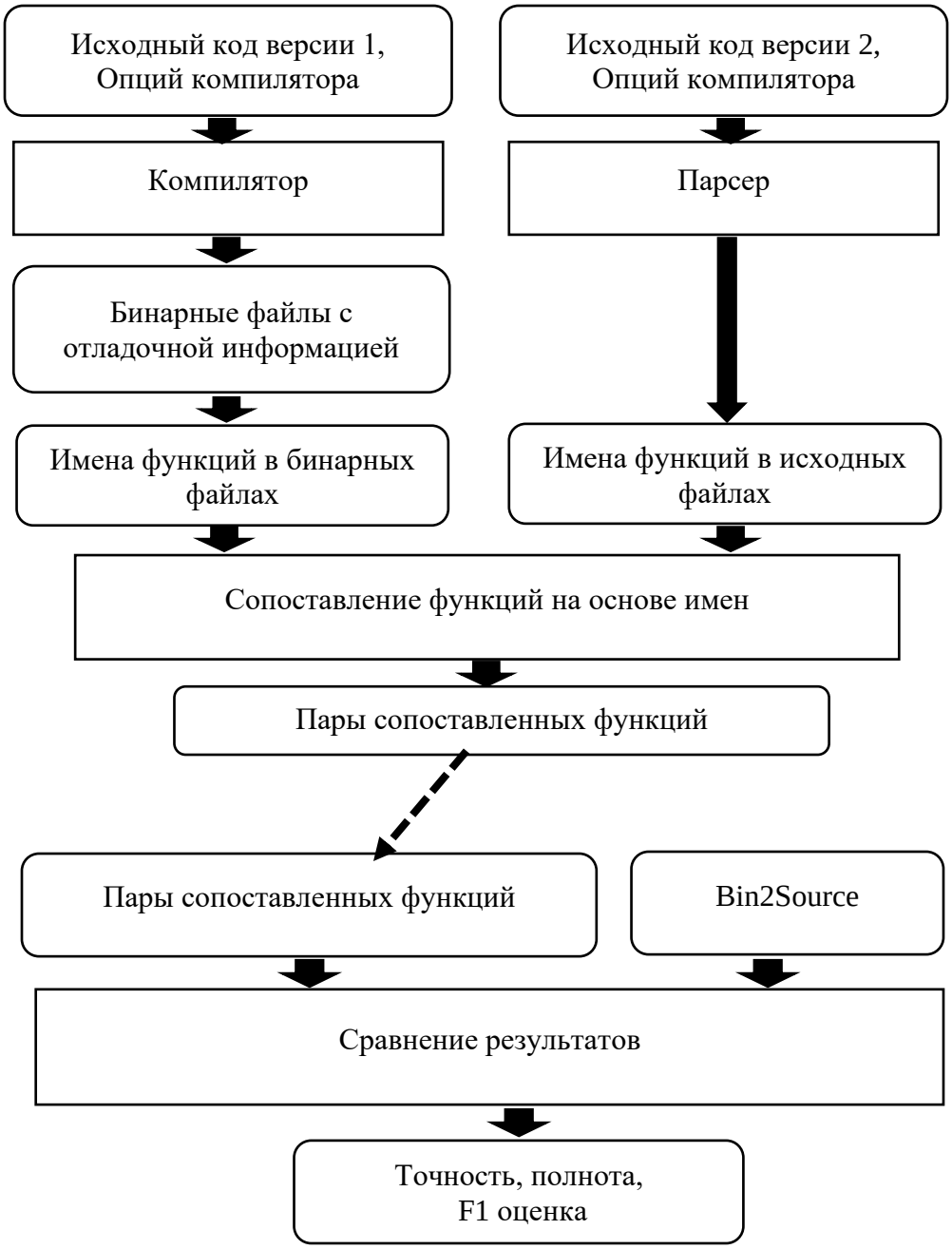

Рисунок 8. Схема тестирование Bin2Source.

3.3.5. Результаты

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 превышали 96%. В таблице 22 приводятся результаты *Bin2Source*.

Таблица 22. Результаты Bin2Source.

Исходный код (уровень оптимизаций)	**Бинарный код (уровень оптимизаций)**	**Архитектура**	**Точность (%)**	**Полнота (%)**	**F_1 оценка (%)**
coreutils-8.18 (O0)	coreutils-8.18 (O0)	x86	97.5	97.5	97.5
coreutils-8.18 (O0)	coreutils-8.18 (O0)	x64	98.3	98.3	98.3
coreutils-8.18 (O2)	coreutils-8.18 (O2)	x86	96.6	96.6	96.6
coreutils-8.18 (O2)	coreutils-8.18 (O2)	x64	97.7	97.7	97.7
coreutils-8.18 (O3)	coreutils-8.18 (O3)	x86	96.5	96.5	96.5
coreutils-8.18 (O3)	coreutils-8.18 (O3)	x64	97.6	97.6	97.6
coreutils-8.18 (O0)	coreutils-8.30 (O0)	x86	86.9	84.1	85.5
coreutils-8.18 (O0)	coreutils-8.30 (O0)	x64	86.9	84.0	85.4
coreutils-8.18 (O2)	coreutils-8.30 (O2)	x86	87.8	90.6	89.2
coreutils-8.18 (O2)	coreutils-8.30 (O2)	x64	86.8	90.3	88.5
coreutils-8.18 (O2)	coreutils-8.18 (O3)	x86	80.5	73.4	76.8
coreutils-8.18 (O2)	coreutils-8.18 (O3)	x64	79.3	74.8	77
coreutils-8.18 (O0)	coreutils-8.18 (O3)	x86	52.9	42.9	47.4
coreutils-8.18 (O0)	coreutils-8.18 (O3)	x64	54.0	41.4	46.9
В среднем			**85.66**	**83.26**	**84.35**

В качестве заключения можно сказать, что разработанный инструмент по своим результатам существенно превосходит доступные аналоги (см. раздел 2.4).

3.4. Поиск неисправленных ошибок

В этом разделе представляется разработанный автором метод, обнаруживающий копии известных уязвимостей в проектах, и используемых сторонних библиотеках. Метод регулярно отслеживает интернет-ресурсы, для сбора всех доступных уязвимостей (*CVE - Common Vulnerabilities and Exposures*) и соответствующих исправлений. Затем он производит поиск клонов уязвимых фрагментов кода в целевом проекте. Разработанный метод может быть также использован для обнаружения использования сторонних библиотек, содержащих известные неисправленные уязвимости. Это представляется важной задачей, поскольку на практике часто встречаются случаи, когда используется старая версия стороннего ПО, который содержит известные уязвимости. В то время, как в самом репозитории стороннего ПО уязвимости уже исправлены.

3.4.1. Высокоуровневое описание метода

На рисунке 9 представляется структура разработанного инструмента. Он состоит из четырех основных компонентов. Первый компонент собирает доступные уязвимости и соответствующие исправления. Для этого регулярно отслеживаются несколько интернет-ресурсов – таких как *GitHub* [34], *NVD* [212], *Mitre* [213]. Далее для каждой уязвимости извлекается соответствующее имя проекта. Затем инструмент пытается найти репозиторий исходного кода этого проекта на *GitHub* (используется *GitHub API*). Если проект существует на *GitHub*, инструмент проверяет историю изменений кода и извлекает коммиты, исправляющие эту конкретную уязвимость. Из этих коммитов извлекается исправление, которое сохраняется в нашей базе данных. Дополнительно извлекаются исправления для каждого пакета *Debian*.

Второй компонент собирает базу проектов с открытым исходным кодом, для этого рассматриваются: проекты *GitHub*, дистрибутивы ОС *Debian*, пакеты *Debian*.

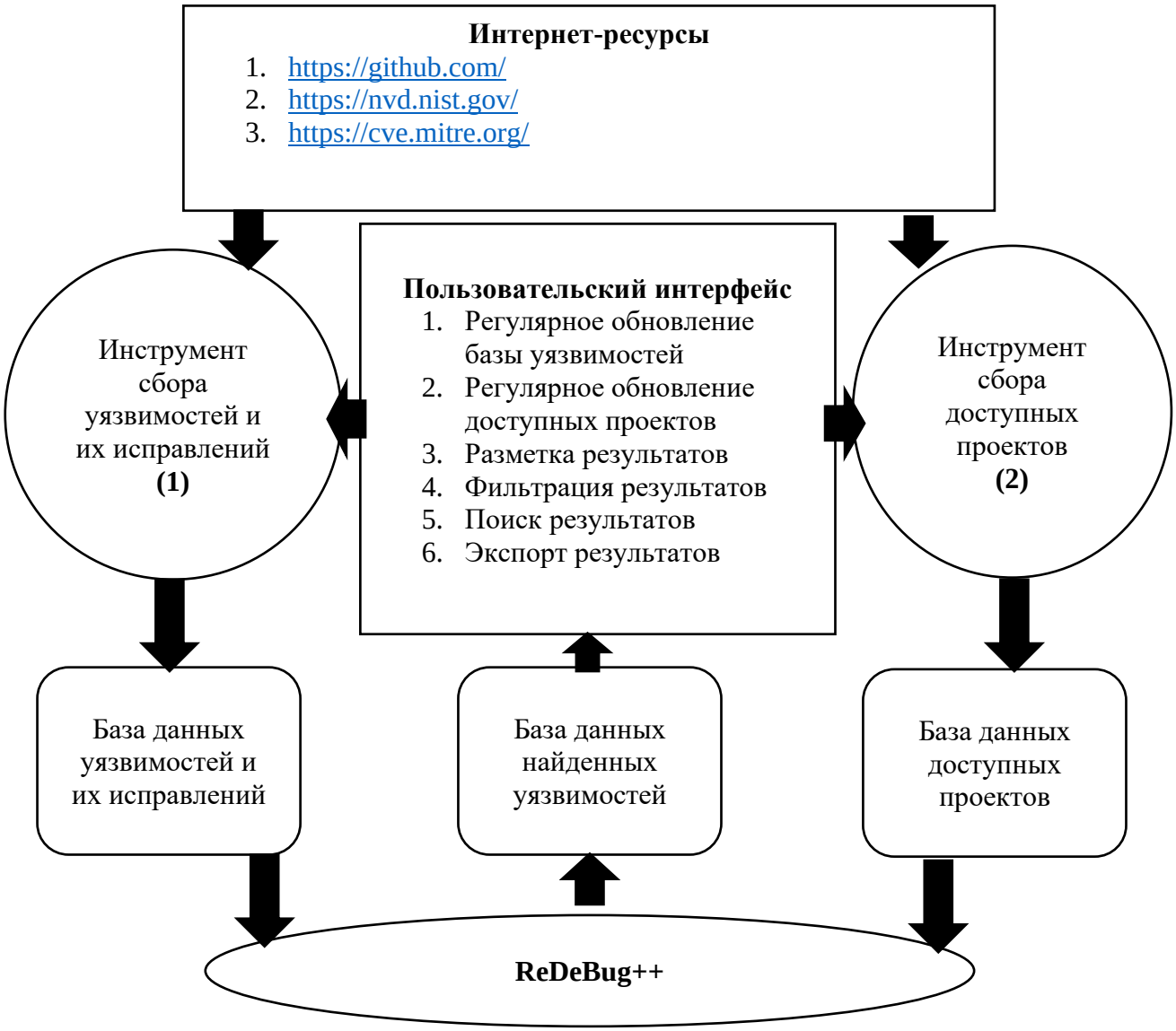

Рисунок 9. Схема работы инструмента поиска неисправленных уязвимостей.

Третий компонент – это *ReDeBug++*, который является улучшенной версией *ReDeBug* [97-98]. Он используется для поиска клонов неисправленных фрагментов кода. В качестве входных данных используются собранные базы исправленных уязвимостей и проекты с открытым исходным кодом. Из исправлений *ReDeBug++* извлекает ошибочные фрагменты кода и ищет их клоны в базе собранных проектов. Обнаруженные клоны сохраняются в базе данных, и представляется пользовательский

интерфейс, который имеет несколько функций:

1. Отображение, поиск и фильтрация обнаруженных ошибок;
2. Разметка (истина, ложь, неизвестно) обнаруженных результатов;
3. Экспорт результатов в разные форматы;
4. Обновление существующих баз данных известных уязвимостей и соответствующих исправлений;
5. Обновление существующих проектов в базе данных;
6. Добавление новых проектов в базу данных.

3.4.2. ReDeBug++

ReDeBug++ представляет собой улучшенную версию *ReDeBug*, которая реализована как однопоточная программа. В первую очередь, была добавлена

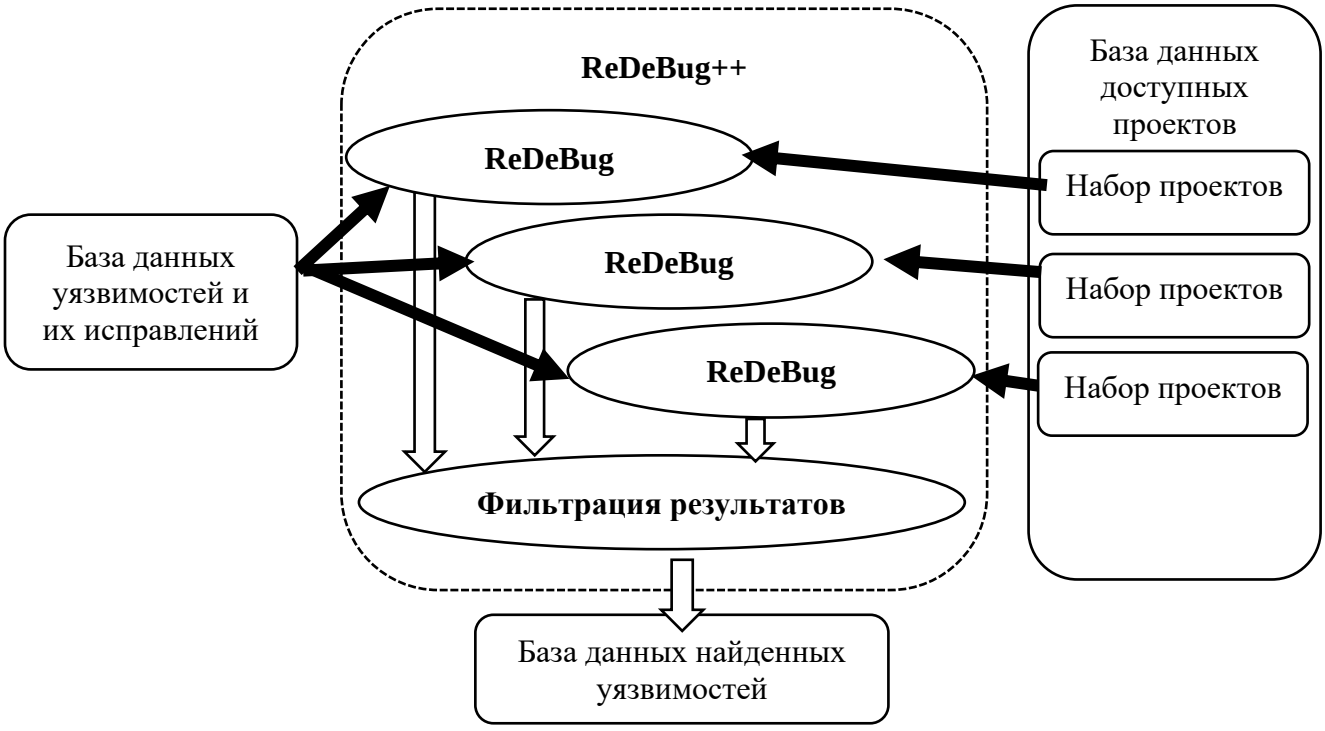

Рисунок 10. Схема работы ReDeBug++.

возможность работы инструмента в многопоточном режиме, поскольку регулярно должно анализироваться большое количество (больше 50,000) проектов. Схема реализации многопоточной обработки приведена на рисунке 10. Список проектов с открытым исходным кодом разбивается на наборы – в зависимости от конфигурации целевого оборудования. Затем для каждого набора пакетов и базы исправленных уязвимостей выполняется экземпляр *ReDeBug*. Когда все экземпляры *ReDeBug* завершаются, результаты фильтруются и сохраняются в базе данных.

В *ReDeBug* встречается несколько основных типов ложных срабатываний, для которых разработаны специальные фильтры, которые применяются перед сохранением результатов в базу данных. Ниже приводится описание ложных срабатываний:

1. Обнаружение копии команд препроцессора для Языков *Cu/Cu++* (например, "*#include*" и т.д.);
2. Обнаружение копии пробелов;
3. Обнаружение копии команд импорта библиотек (для *Python*);

В *ReDeBug* также было выполнено несколько других улучшений:

1. Код был обновлен для поддержки *python-3*, изначально инструмент был написан на *python-2.7*;
2. Исправлена ошибка, приводящая к зависанию инструмента;
3. Добавлена система тестирования для проверки корректности работы инструмента.

3.4.3. Экспериментальное тестирование и анализ результатов

Экспериментальное тестирование было произведено на *Intel(R) Xeon(R)*

Platinum 8160M, с 240Гб оперативной памятью и с постоянным использованием 20 ядер. В течение 48 часов удалось собрать более 42 тысяч пакетов *Debian* (размером 1,5 ТБ) и 2973 исправлений уязвимостей (некоторые из них исправляют более одной ошибки). Затем был запущен *ReDeBug++*, чтобы найти клоны неисправленных уязвимостей во всех пакетах. Инструменту потребовалось двенадцать часов, чтобы найти все клоны. В результате была обнаружена 401 неисправленных уязвимостей. Семь из них были подтверждены и исправлены, в частности, было обнаружено, что:

- *4Pane* [214] – менеджер файлов в ОС *Linux* использует старую версию архиватора *bzip2* [215], содержащую известную уязвимость *CVE-2019-12900*. Схожая ошибка была найдена в проекте *doublecmd* [216];
- *AdAway* [217] – блокировщик рекламы использует старую версию *tcpdump*, где содержатся известные уязвимости *CVE-2018-16452*, *CVE-2017-13030*, *CVE-2018-14879* и *CVE-2017-5486*;
- *Praat* [218] – инструмент анализа речи использует библиотеку аудиокодеков без потерь *flac* [219], содержащую известную уязвимость *CVE-2014-9028*;
- В проектах *LuaJIT* [220], *Grub2* [221] и *MoarVM* [222] метод сумел найти копию уязвимого фрагмента кода из проекта *minilua* [223], которому соответствует *CVE-2014-5461*.

Более детальный анализ результатов показал, что менее 10% исправлений (из общего числа 2973) встречаются в более 30% случаев. Один из часто встречающихся случаев – это использование старой/уязвимой версии *minilua*, который содержит *CVE-2014-5461*. Далее, приводится список пакетов, содержащих данную уязвимость: *moarvm-dev* [224], *grub-pc* [225], *enigma* [226], *enigma-data* [227], *grub2* [152], *grub-pc-bin* [228], *medit* [229]. На рисунке 11 приводится соответствующее исправление данной уязвимости.

```
+++ b/src/ldo.c
@@ -274,7 +274,7 @@ int luaD_precall (lua_State *L, StkId func, int nresults) {
     CallInfo *ci;
     StkId st, base;
     Proto *p = cl->p;
-    luaD_checkstack(L, p->maxstacksize);
+    luaD_checkstack(L, p->maxstacksize + p->numparams);
     func = restorestack(L, funcr);
       base = func + 1;
```

Рисунок 11. Исправление уязвимости CVE-2014-5461.

Другой, часто встречающийся случай – это использование старой/уязвимой версии файла *slirp/tcp_subr.c* из проекта *QEMU* в следующих пакетах ОС *Debian*: *bochs* [230], *bochs-wx* [231], *bochs-x* [232], *bochs-sdl* [233], *bochsbios* [234], *bximage* [235]. Использованная версия файла содержит уязвимости *CVE-2019-6778* и *CVE-2017-11434*. На рисунках 12 и 13 приводятся соответствующие исправления.

```
+++ b/slirp/tcp_subr.c
@@ -634,6 +634,11 @@ tcp_emu(struct socket *so, struct mbuf *m)
                        socklen_t addrlen = sizeof(struct sockaddr_in);
                        struct sbuf *so_rcv = &so->so_rcv;

+                       if (m->m_len > so_rcv->sb_datalen
+                                       - (so_rcv->sb_wptr - so_rcv->sb_data)) {
+                           return 1;
+                       }
+
                        memcpy(so_rcv->sb_wptr, m->m_data, m->m_len);
                        so_rcv->sb_wptr += m->m_len;
                        so_rcv->sb_rptr += m->m_len;
```

Рисунок 12. Исправление уязвимости CVE-2019-6778.

```
+++ b/slirp/bootp.c
@@ -123,6 +123,9 @@ static void dhcp_decode(const struct bootp_t *bp, int
*pmsg_type,
             if (p >= p_end)
                 break;
             len = *p++;
+            if (p + len > p_end) {
+                break;
+            }
             DPRINTF("dhcp: tag=%d len=%d\n", tag, len);

             switch(tag) {
```

Рисунок 13. Исправление уязвимости CVE-2017-11434.

3.4.4. Причины ложных срабатываний

Анализ результатов показал три основных типа ложных срабатываний. В первом случае, в базе данных были сохранены неправильные исправления, в результате чего были обнаружены клоны ложные, не содержащие уязвимостей. Причина этого заключается в том, что исправления уязвимостей собираются на основе сообщений, соответствующих коммитам в истории *git*. Коммиты, имеющие сообщения со схожим текстом "*This is similar to CVE-2019-13454*", воспринимаются как исправления уязвимости, что не корректно.

Во втором случае, для некоторых пакетов ОС *Debian* исходный код, загруженный командой "*apt-get source*" (данной командой собирается база пакетов *Debian*) включает старые версии файлов, а в основном репозитории разработки уязвимость уже исправлена. Таким образом разработанный инструмент обнаруживает копии неисправленных уязвимостей, которые в основном репозитории уже зафиксированы.

В последнем случае, для больших исправлений (более 50 строк) могут быть обнаружены клоны их фрагментов в других файлах, не содержащих дефектов. Это происходит в результате того, что инструмент *ReDeBug*++ принимает минимальную длину токенов, которые необходимо обнаружить. Большие исправления могут внести и другие изменения, не связанные с дефектом.

3.4.5. Категории найденных ошибок

Обнаруженные дефекты в пакетах *Debian* можно разделить на две основные категории. В первой категории репозиторий программного обеспечения, соответствующий конкретному пакету, обновлен и содержит исправления всех уязвимостей, но пакет исходного кода для *Debian* был собран из более старой версии и содержит неисправленные уязвимости. Порядка 65% обнаруженных дефектов

относятся к этой категории. Во вторую категорию входят проекты, которые используют старую версию стороннего ПО, содержащую уязвимости. Как правило, либо произведено копирование старой версии целиком, либо в репозитории *git* используется старый хеш для подмодуля (*git submodule*). Порядка 28% обнаруженных дефектов относятся к этой категории. Остальные 7% являются ложноположительными, причины которых приведены в предыдущем разделе.

3.5. Заключение

В этом разделе представлено описание методов поиска клонов кода в исходных и бинарных файлах. Поиск осуществляется на основе графов зависимостей программ, которые получаются из промежуточного представления исходного или бинарного кода, в зависимости от задачи. Клоны определяются как максимальные схожие подграфы в паре ГЗП. Экспериментальные результаты показывают, что разработанный метод превосходит существующие до этого методы. С применением метода поиска клонов кода был реализован инструмент сопоставления исходного и бинарного кода. Для этого исходный код транслируется в бинарный код, и, далее, производится сопоставление двух бинарных файлов на уровне функций и инструкций. После сопоставления бинарных инструкций производится сопоставление бинарного кода с исходным кодом с использованием отладочной информации. Разработанный инструмент показал, в среднем, 85%-ую точность и 83%-ую полноту, что существенно превосходит доступные аналоги. Далее, приводится описание метода поиска копий известных уязвимостей в исходном коде, с применением которого стало возможным найти множество копий известных уязвимостей, включая ошибки в *grub2*, являющемся загрузчиком ОС *Linux*.

4. Методы поиска ошибок использования динамической памяти и анализа помеченных данных

Приводится описание разработанных автором методов поиска утечек памяти, поиска проблем, связанных с некорректным использованием динамической памяти и анализа помеченных данных. *Полученные новые результаты в рамках исследования в данной главе выносятся на защиту в виде метода поиска утечек памяти для языков Си/Си++, который на первом этапе производит поиск утечек на специальном представлении программы, а на втором этапе производит проверку выполнимости путей ошибок методом направленного символьного выполнения*.

4.1. Анализ помеченных данных и поиск ошибок форматной строки

В данном разделе приводится описание разработанного автором метода анализа помеченных данных, на основе которого реализуется поиск ошибок форматной строки. А также приводится описание метода поиска ошибок использования освобожденной динамической памяти. Оба метода базируются на ГЗП всей программы, который содержит в себе 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).

4.1.1. Построение ГЗП всей системы

ГЗП всей системы строится на базе промежуточного представления компиляторной инфраструктуры *LLVM*. На рисунке 14 приводится пример кода на *C* и соответствующее промежуточное представление. На рисунке 15 приводится соответствующий граф.

```
#include <iostream>
int f(int *p) {
  return *p;
}
int main() {
  int *b;
  f(b);
  return 0;
}
```

```
define i32 @_Z1fPi(i32* %p) #2 {
  %1 = alloca i32*, align 8
  store i32* %p, i32** %1, align 8
  %2 = load i32** %1, align 8, !dbg !21
  %3 = load i32* %2, align 4, !dbg !21
  ret i32 %3, !dbg !21
}
define i32 @main() #2 {
  %1 = alloca i32, align 4
  %b = alloca i32*, align 8
  store i32 0, i32* %1
  %2 = load i32** %b, align 8, !dbg !22
  %3 = call i32 @_Z1fPi(i32* %2), !dbg !22
  ret i32 0, !dbg !23
}
```

Рисунок 14. Пример промежуточного представления LLVM.

Как видно из рисунка 15, передачу аргументов в графе можно проследить по потоку данных. Для этого создается отдельная вершина ("*[Z1fPi]i32* %p*") в графе, через которую производится поток данных из функции "*main*" в "*f*". Еще одним важным свойством ГЗП является то, что граф вызовов также отображается в нем. Если в некоторой точке программы происходит вызов функции "*f*", тогда все инструкции выхода из "*f*" соединяются ребром потока управления следующей инструкции его

вызова. Как видно из примера, инструкция "*ret*" функции "*f*" имеет ребро по потоку управления к инструкции "*ret*" функции "*main*", поскольку эта инструкция следует за вызовом "*f*".

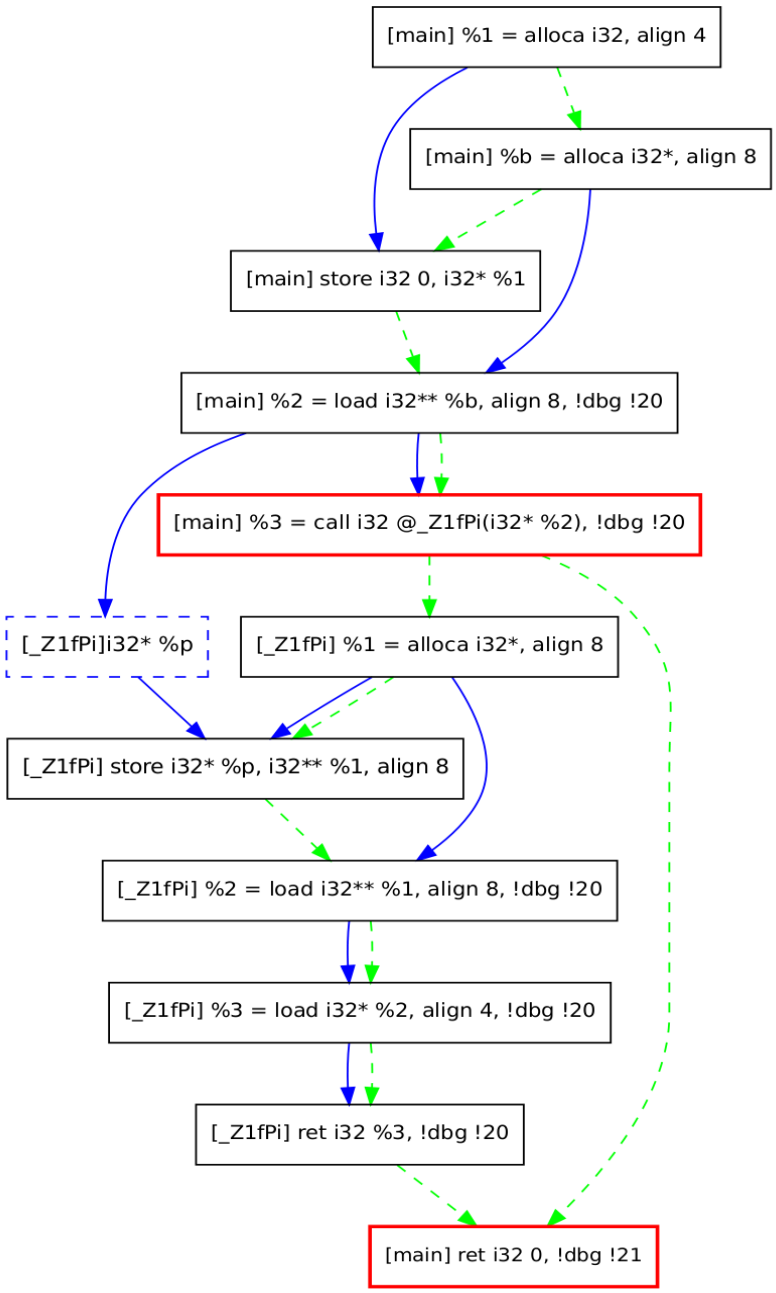

Рисунок 15. ГЗП промежуточного представления LVMM.

4.1.2. Анализ помеченных данных

Вышеописанная реализация ГЗП для всего проекта позволяет производить анализ помеченных данных достаточно простым способом. Вначале производится отметка помеченных вершин в графе. Обычно это – инструкции программы, которые производят вызовы системных функций, читающих из стандартного ввода, переменных окружения, файлов и т.д. Также пользователь сам может указать такие вершины. Далее, когда есть набор помеченных вершин, для них производится прямой обход ГЗП в ширину по ребрам соответствующих зависимостей по данным.

4.1.3. Поиск ошибок использования форматной строки

Детектор производит поиск ошибок форматной строки. Алгоритм достаточно прост, если:

1. Существует инструкция (помеченная инструкция), производящая ввод помеченных данных;
2. Существует путь по потоку управления от помеченной инструкции до вызова одной из функций $F = \{printf, fprintf, asprintf, sprintf, snprintf, vsprintf, vfprintf, vasprintf\}$;
3. Функция F использует помеченные данные напрямую (без форматной строки), либо количество или тип форматных символов не соответствуют аргументам;
4. Помеченные данные передаются как аргументы F.

Тогда программа может содержать ошибку форматной строки. Ниже приводится формальное описание алгоритма:

Алгоритм поиска ошибок форматной строки.

```
1. formatStringErrorDetection(programDependenceGraph) {
2.   for (callNode in programDependenceGraph.callNodes()) {
3.     if (callNode->providesTaintedInput() ) {
4.       formatFunctions = getForwardReachedFormatFunctionsBFS(callNode)
```

```
5.      for (formatFunctionCall in formatFunctions) {
6.        if ((not formatFunctionCall.usesFormatSymboles() or
7.            formatFunctionCall.formatSymbolesCount() != formatFunctionCall.argsCount()) and
8.            isDataFlow(programDependenceGraph, callNode, formatFunctionCall)) {
9.            reportFormatStringError(programDependenceGraph, callNode, formatFunctionCall)
10.       }
11.     }
12.   }
13.  }
14. }
```

Процедура *getForwardReachedFormatFunctionsBFS* производит поиск в ширину по ребрам потока управления и собирает все вершины графа, соответствующие инструкциям вызовов функций, принимающих как аргумент форматную строку. Процедура *isDataFlow* производит поиск в ширину по ребрам потока данных и проверяет достижимость одной из функций множества *formatFunctionCall*. Сложность последних двух процедур составляет $O(n + e)$, где n – количество вершин в ГЗП, а e – количество ребер [236], [237]. Функция *callNodes* возвращает множество вершин ГЗП, соответствующих инструкциям вызовов функций. Функция *providesTaintedInput* возвращает значение "*истина*", если соответствующая инструкция производит ввод помеченных данных. Последние две процедуры обладают константной сложностью из-за специфики реализаций ГЗП.

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

Доказательство: из определения сложности следует, что существуют положительные константы $C1$ и $C2$ – такие, что во время вызова процедур *getForwardReachedFormatFunctionsBFS* и *isDataFlow* будет выполнено не более $C1 * (n + e)$ и $C2 * (n + e)$ инструкций, соответственно. Алгоритм поиска в ширину может вернуть не более n-вершин. Из вышесказанного следует, что предложенный алгоритм в общей сумме производит не более:

$$n * \left(C1 * (n + e) + n * C2 * (n + e)\right) = n * (n + e) * (C1 + C2 * n) \leq$$
$$(C1 + C2) * n^2 * (n + e) = C * n^2 * (n + e)$$

Где $C1 + C2 = C$. Из этого следует, что сложность предложенного алгоритма $O(n^2 * (n + e))$. ***Теорема доказана***.

Сложность $O(n^2 * (n + e))$ достаточно большая, поскольку мы заменяем:

1. Количество вызовов функций, вводящих помеченные данные на *n*;
2. Количество вызовов функций, работающих с форматной строкой на *n*.

А на практике вышеуказанных функций (вводящих помеченные данные и функций, работающих с форматной строкой) немного, и ребра графа сопоставимы с количеством вершин, из чего следует, что алгоритм будет выполняться за линейное время от количества вершин ГЗП.

4.1.4. Поиск ошибок использования памяти после освобождения

Детектор производит поиск ошибок использования освобожденной памяти. Алгоритм детектора достаточно простой. Рассматриваются все вершины ГЗП, соответствующие указателям (например, "*[main] %b = alloca i32*, align 8*" на рисунке 15). Если присутствует такая вершина *P*, для которой:

1. Поток данных от *P* достигает до инструкции удаления памяти *D*;
2. Поток данных от *P* достигает до инструкции использования памяти *U*;
3. Существует путь потока управления от инструкций *D* к *U*, на котором не производится присваивание указателя *P*.

В таком случае программа содержит ошибку использования памяти после освобождения. Необходимо обратить внимание, что, если во втором пункте инструкция *U* является инструкцией освобождения, то тогда программа содержит ошибку двойного освобождения. Ниже приводится формальное описание алгоритма:

Алгоритм поиска ошибок использования памяти после освобождения.

```
1.  freeFunctions = {free, delete}
2.
3.  useAfterFreeErrorDetection(programDependenceGraph) {
4.    for (pointerInstruction in programDependenceGraph.pointerNodes()) {
5.      freeFunctions =  getForwardReachedFreeFunctionsBFS(pointerInstruction)
6.      useInstructions =  getForwardReachedUseInstructionsBFS(pointerInstruction)
7.      for (freeFunction in freeFunctions) {
8.        for (useInstruction in useInstructions) {
9.          if (isControlFlowWithoutPointerRewrite(programDependenceGraph,
10.                                                 pointerInstruction,
11.                                                 freeFunction, useInstruction) {
12.           reportUseAfterFreeError(programDependenceGraph, callNode, formatFunctionCall)
13.         }
14.       }
15.     }
16.   }
17. }
```

Процедура *getForwardReachedFreeFunctionsBFS* производит поиск в ширину по ребрам потока данных и собирает все вершины графа, соответствующие инструкциям вызовов функций освобождения указанной динамической памяти. Процедура *getForwardReachedUseInstructionsBFS* производит поиск в ширину по ребрам потока данных и собирает все вершины графа, соответствующие инструкциям использования указателя динамической памяти. Процедура *isControlFlowWithoutPointerRewrite* производит поиск в глубину по ребрам потока управления и строится путь от вершины *freeFunction* до *useInstruction*. Для всех инструкций найденного пути дополнительно проверяется присвоение нового значения указателю *pointerInstruction*. Функция возвращает значение "*истина*", если существует путь, на котором указатель не изменяется. Сложность указанных выше процедур составляет $O(n + e)$, где n – количество вершин в ГЗП, а e – количество ребер. Процедура *pointerNodes* возвращает множество вершин ГЗП, соответствующих указателям на динамическую память. Его сложность – константная из-за специфики реализаций ГЗП.

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

Доказательство: из определения сложности следует, что существуют положительные константы $C1$, $C2$ и $C3$ – такие, что во время вызова процедур *getForwardReachedFreeFunctionsBFS*, *getForwardReachedUseInstructionsBFS* и *isControlFlowWithoutPointerRewrite* будет выполнено не более $C1 * (n + e)$, $C2 * (n + e)$ и $C3 * (n + e)$ инструкций, соответственно. Стандартный алгоритм в ширину может вернуть не более n-вершин. Из этого следует, что предложенный алгоритм в общей сумме производит не более:

$$n * \big(C1 * (n + e) + C2 * (n + e) + n * n * C3 * (n + e)\big) =$$
$$n * (n + e) * (C1 + C2 + C3 * n^2) \leq$$
$$(C1 + C2 + C3) * n^3 * (n + e) = C * n^3 * (n + e)$$

Где $C1, C2, C3$ – положительные числа и $C1 + C2 + C3 = C$. Из этого следует что сложность алгоритма $O(n^3 * (n + e))$. ***Теорема доказана***.

Сложность $O(n^3 * (n + e))$ достаточно большая, поскольку мы заменяем:

1. Количество указателей на динамическую память на n;
2. Количество вызовов функций, освобождающих динамическую память на n;
3. Количество инструкций, использующих указатели динамической памяти на n.

А на практике количество вышеуказанных указателей, вызовов функций и инструкций можно заменить на некоторые константы. А также ребра графа сопоставимы с количеством вершин, из чего следует, что алгоритм будет выполняться за линейное время от количества вершин ГЗП.

4.1.5. Экспериментальное тестирование и выводы

В ходе экспериментального запуска разработанный инструмент смог найти известные уязвимости *CVE-2016-0705* и *CVE-2016-0799* в старой версии проекта *openssl*. На рисунке 16 приводится время работы инструмента на реальных проектах.

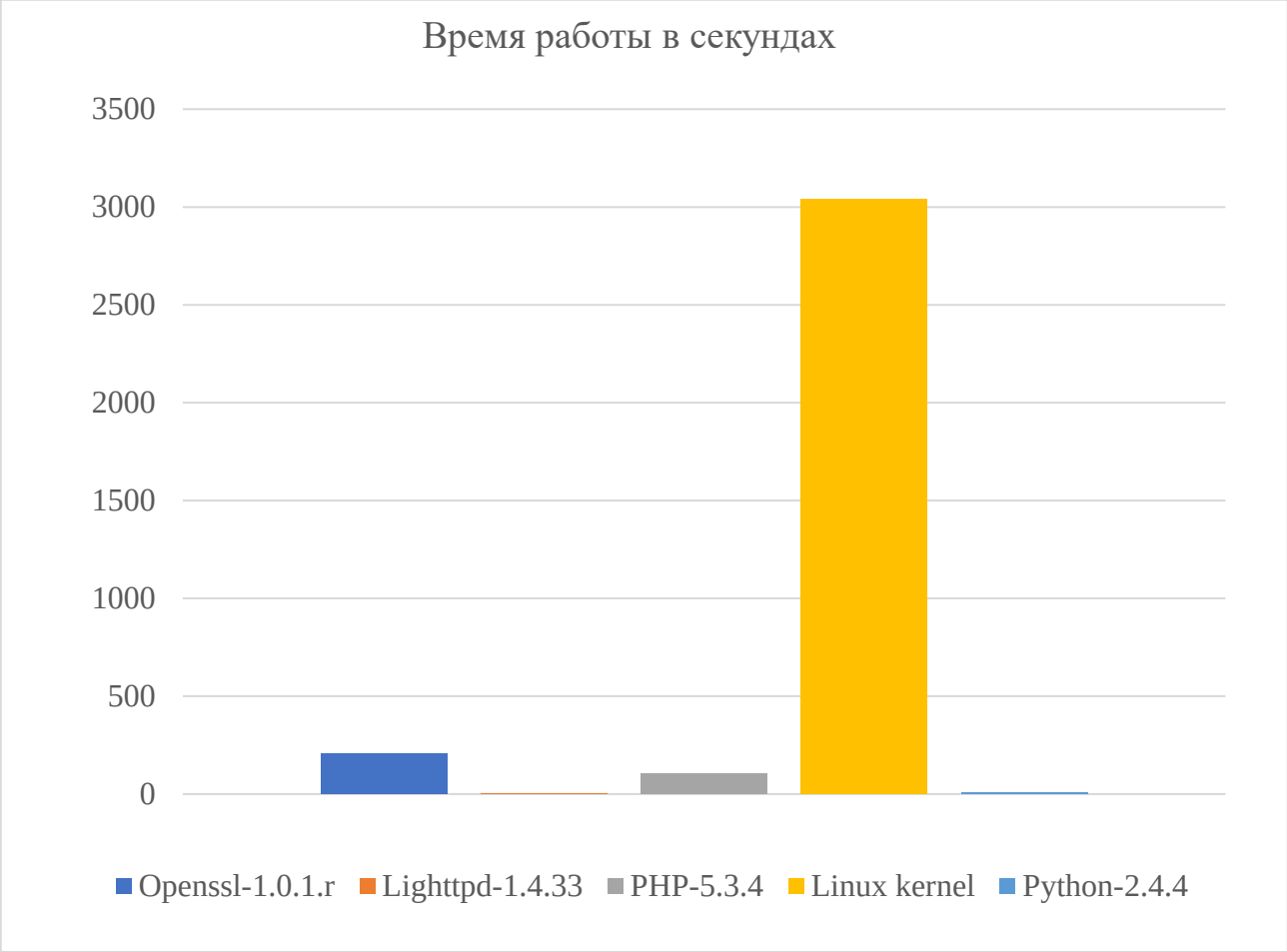

Рисунок 16. Время работы разработанного инструмента.

На рисунках 17 и 18, соответственно, приводится количество найденных ошибок использования форматной строки и освобожденной динамической памяти.

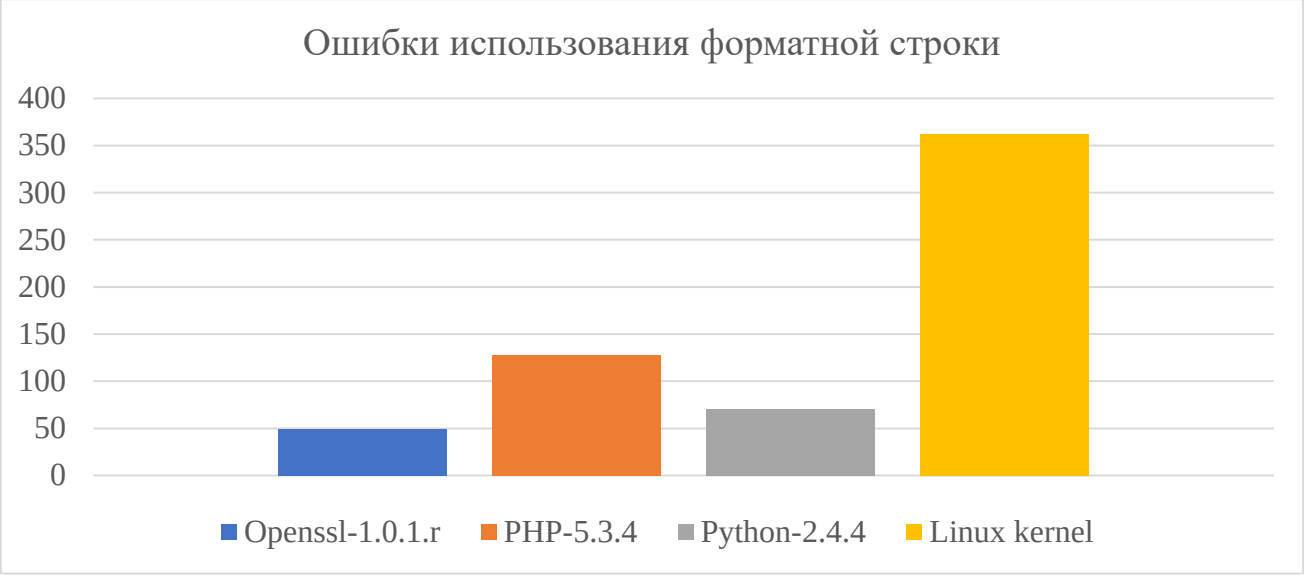

Рисунок 17. Найденные ошибки использования форматной строки.

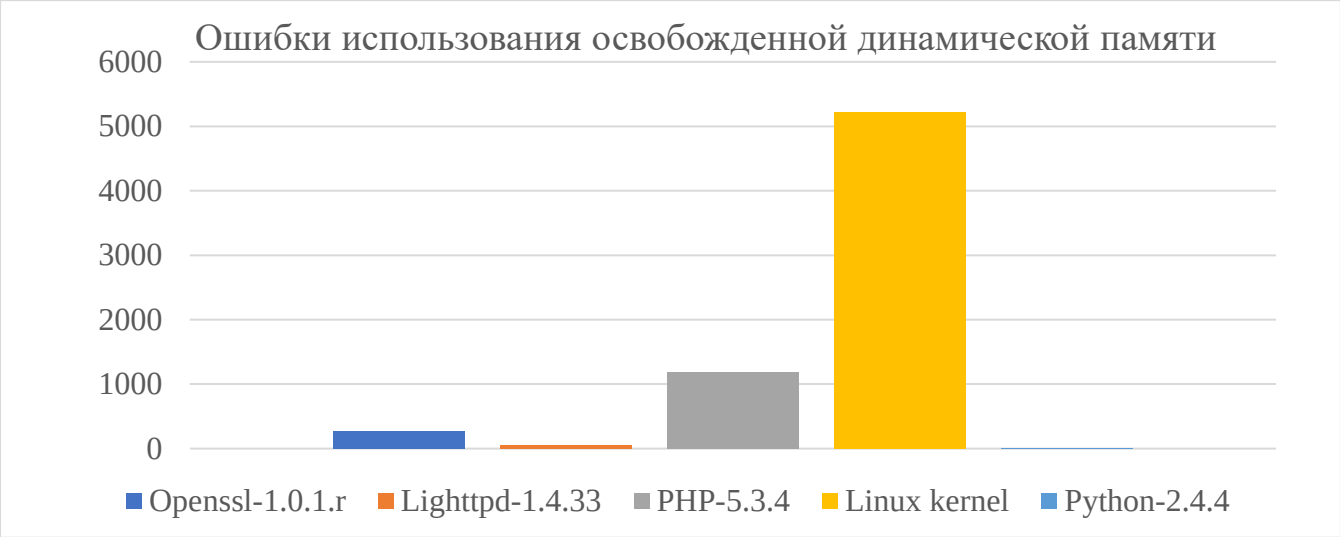

Рисунок 18. Найденные ошибки использования освобожденной динамической памяти

Как видно из результатов, применение данного подхода дает высокий уровень ложных срабатываний. На ядре *Linux* было найдено более 5000 ошибок использования освобожденной динамической памяти и 300 ошибок форматной строки. Дальнейшее улучшение метода приведет к снижению уровня ложных срабатываний, но достичь нужного 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 уже внедрен в процесс безопасной разработки.

4.2. Метод динамического поиска ошибок использования освобожденной памяти

В данном разделе приводится описание разработанного автором метода динамического поиска ошибок использования освобожденной 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. Предлагаемый инструмент реализован на базе *Triton* [238].

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 обеспечивали большее покрытие кода, получают высокий приоритет, что обеспечивает их использование при следующей итерации анализа. Данный подход позволяет ускорить рост покрытия кода.

4.2.1. Принцип работы инструмента Triton

В *Triton* используется алгоритм увеличения покрытия кода целевой программы, который состоит из трех основных этапов:

1. Выбор начальных входных данных;

2. Сбор ограничений (например, *input[0] == “b”*) для каждого пути выполнения программы;
3. Получение новых входных данных с помощью решения символьных выражений, полученных из собранных ограничений.

Рассмотрим пример программы на рисунке 19.

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == 'b') cnt++;
  if (input[1] == 'a') cnt++;
  if (input[2] == 'd') cnt++;
  if (input[3] == '!') cnt++;
  if (cnt >= 3) abort(); // error
}
```

Рисунок 19. Пример программы с аварийным завершением.

Алгоритм начинает свою работу, запустив программу на начальной входной строке, поместив ее в список входных данных. После первого запуска программы получается следующий набор ограничений: *<i0≠b, i1≠a, i2≠d, i3≠!>* (*предполагается, что входная строка была выбрана произвольным образом, и ни одно из условий не удовлетворяется*), где *i0, i1, i2, i3* представляют ячейки памяти, соответственно, *input[0], input[1], input[2]* и *input[3]*. В ходе работы алгоритма, по очереди решаются полученные ограничения, и для каждого элемента из входных данных генерируются "*дочерние данные*", удовлетворяющие соответствующим ограничениям, которые помещаются в список входных данных для дальнейшей обработки. Для каждого элемента из этого списка программа запускается заново, и работа алгоритма повторяется. Этот процесс продолжается до тех пор, пока все элементы из списка входных данных не будут поочередно рассмотрены. Применив алгоритм для

программы на рисунке 19 с начальной входной строкой "*good*", получится набор входных данных, приведенный на рисунке 20.

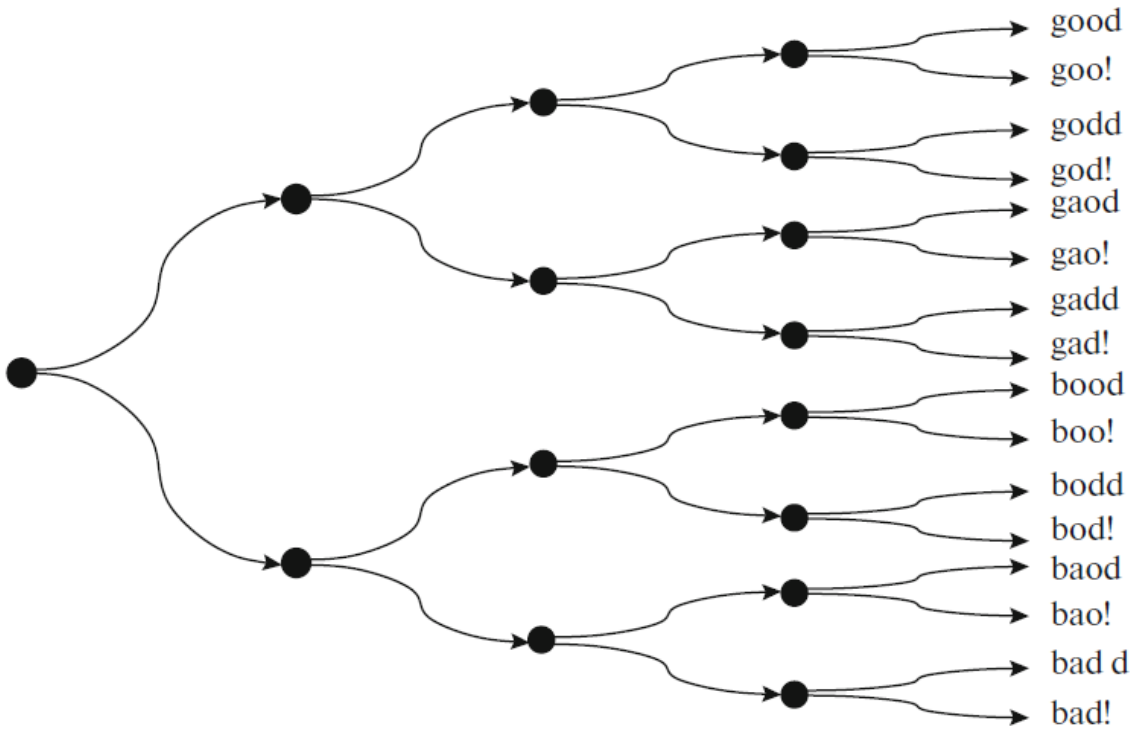

Рисунок 20. Входные данные после каждой итерации алгоритма.

Поскольку работа данного алгоритма требует неоднократного запуска программы, в *Triton* реализована возможность сохранения состояния программы. Использование этой функциональности позволяет значительным образом улучшить производительность выполнения.

4.2.2. Поиск ошибок использования памяти после освобождения

Разработанный алгоритм отслеживает функции выделения ("*malloc*") и освобождения ("*free*") памяти с применением инструментаций. В начале работы алгоритма создаются два множества (*allocSET* и *freeSET*), элементы которых являются парами, имеющими вид (*адрес*, *размер*), с помощью которых отслеживаются участки выделенной/освобожденной памяти. При операциях выделения памяти в *allocSET*

добавляются элементы. При освобождениях памяти соответствующие элементы переходят из *allocSET* в *freeSET*.

Во время выполнения инструкций, осуществляющих доступ к памяти, проверяется наличие соответствующего адреса в множестве *freeSET* и при наличии выдается ошибка. На рисунке 21 приведен пример работы алгоритма.

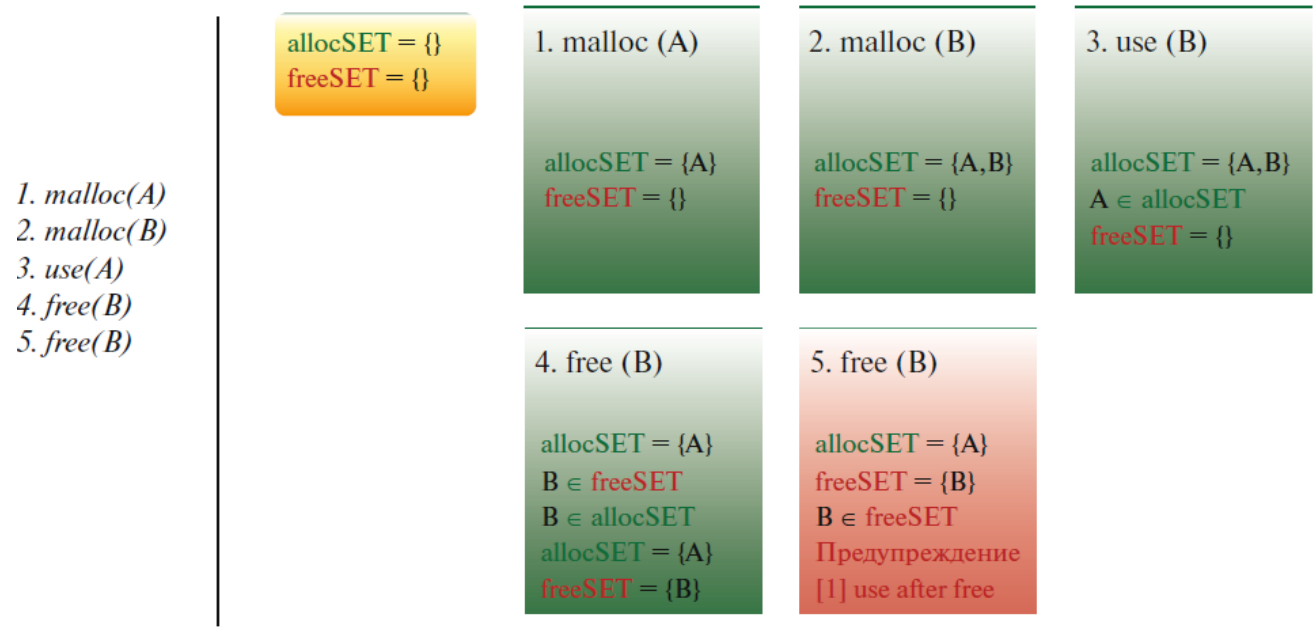

Рисунок 21. Пример работы алгоритма.

4.2.3. Другие улучшения инструмента Triton

В *Triton* было произведено несколько изменений для улучшения производительности и поддержки других типов входных данных. Ниже приводится подробное описание сделанных изменений.

4.2.3.1. Модификация покрытия кода в Triton

В оригинальной реализации *Triton* после каждой итерации программа всегда получала на входе последний элемент из списка входных данных, не учитывая, при

этом, количество открытых базовых блоков программы с помощью данного элемента. Это приводило к тому, что, вместе с обработкой входных данных, которые имели воздействие на покрытие кода программы, рассматривались и те входные данные, с помощью которых не было открыто никаких новых путей в программе. В ходе работы алгоритма для каждого входного элемента генерируются ее дочерние данные, что приводит к быстрому увеличению количества элементов в списке входных данных. Для более эффективного выполнения алгоритма можно определить приоритет выполнения сгенерированных входных данных.

В предлагаемой модификации каждому элементу из списка входных данных присваивается вес, который представляет из себя количество новых базовых блоков программы, которые были открыты с его помощью. В начале работы входным данным присваиваются нулевые веса. Во время первой итерации подсчитываются веса начальных входных данных, и после каждой итерации обновляются. Веса уже рассмотренных элементов передаются их дочерним элементам (входные данные, которые получились с помощью решения соответствующих ограничений), и к ним прибавляется количество новых открытых блоков.

Перед очередным запуском программы из списка выбирается элемент с наибольшим весом. На рисунке 22 видно, что после добавления весовых значений количество рассматриваемых входных данных уменьшается почти вдвое, что, в свою очередь, приводит к существенному увеличению производительности работы алгоритма (на некоторых тестах производительность возросла почти на 90%).

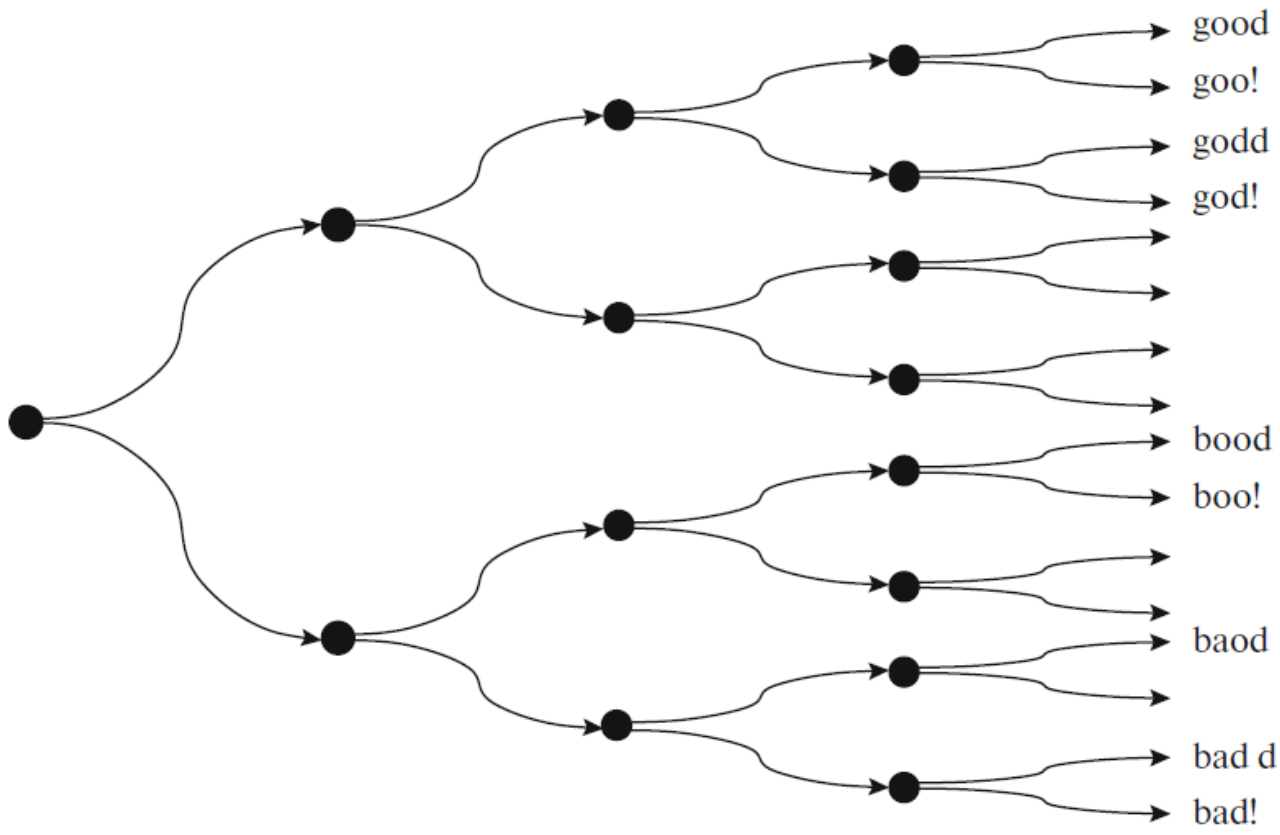

Рисунок 22. Входные данные после каждой итерации алгоритма, после добавления весовых значений.

4.2.3.2. Поддержка файлового входа

Еще одним недостатком Triton была поддержка программ, принимающих на входе только аргументы командной строки. Для расширения набора анализируемых программ была добавлена поддержка программ, использующих файлы в качестве источника входных данных. А также была добавлена возможность определения конкретных диапазонов входных данных, которые в ходе анализа будут помечены как символьные.

4.2.4. Экспериментальное тестирование и выводы

Для тестирования метода создано несколько примеров (рисунок 23), которые приводятся ниже. Для обоих случаев обнаружение повторного освобождения памяти невозможно без исследований всех путей программ. Для приведенных примеров ошибка воспроизводится, если выполнение программ достигнет строки 21 в первом примере, и строки 23 во втором примере. Первый пример основан на реальной ошибке повторного использования освобожденной памяти из проекта *libssh* [239]. Предлагаемый метод позволяет найти входные данные для достижения нужных точек в коде (строки 21, 26 и 23, 27), после выполнения которых будет воспроизведена ошибка.

Несмотря на то, что предложенный метод практически 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, нам удалось разработать комбинированный метод анализа утечек памяти (раздел 4.3), который может производить качественный анализ реальных проектов.

```
1.  struct readFile {
2.    char*  buffer;
3.    int    status;
4.  };
5.
6.  int search_key_in_text(char* str, char* key) {
7.    if (key != NULL && !strstr(str, key))
8.      return -1;
9.    else
10.     return 0;
11. }
12.
13. int main(int argc, char* argv[]) {
14.   struct readFile rf;
15.   rf.buffer = (char*)malloc(sizeof(char));
16.   int fd = open(argv[1], O_RDONLY | O_CREAT);
17.   read(fd, rf.buffer, 50);
18.   printf("File cont : %s\n", rf.buffer);
19.   if (argv[2] == NULL) {
20.     rf.status = 11;
21.     free(rf.buffer);
22.   }
23.   if (!search_key_in_text(rf.buffer, argv[2])) {
24.     // do some stuff
25.   }
26.   free(rf.buffer); //Use after free
27. }
```

```
1.  int myatoi(char *p) {
2.  int i = 0;
3.  while (*p != '\x00' && *p >= '0' && *p <= '9') {
4.      i *= 10;
5.      i += *(unsigned char *)p - '0';
6.      p++;
7.    }
8.    return i;
9.  }
10.
11. int main(int argc, char *argv[]) {
12. char* str = (char*) malloc (SIZE);
13. int num;
14. if (argc != 2) {
15.     printf("Usage: %s <string>\n", argv[0]);
15.     exit(-1);
17. }
18. strcpy(str, argv[1]);
19.
20. num = myatoi(str);
21. if (num >= 33 && ! (num % 2)) {
22.     printf("ok\n");
23.     free(str)
24. }
25. if (num > 1024 ) {
26.     printf("Number is out of range\n");
27.     free(str); //Use after free
28. }
29. return 0;
30. }
```

Рисунок 23. Примеры ошибки повторного освобождения памяти.

4.3. Комбинированный метод поиска утечек памяти

В данном разделе приводится описание разработанного автором метода поиска утечек памяти, который масштабируется для анализа десятков миллионов строк исходного кода и имеет приемлемую точность. Неудачный опыт (разделы 4.1 и 4.2) создания масштабируемых и точных инструментов для схожих задач только на базе статического или динамического анализов нам помог в решении данной задачи.

Предлагаемый метод использует граф потока данных и управления с последующей верификацей результатов на основе символьного выполнения. Для этого используется специальная структура данных под названием *ProcedureGraph*,

которая строится на базе промежуточного представления *LLVM*. В *ProcedureGraph* содержится поток данных и управления, одновременно.

Метод обнаружения ошибок состоит из двух основных этапов. На первом этапе из графов вызовов функций удаляются циклы, после чего 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, которая 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 ускорить анализ. Пользователь может также предоставить аннотации использованных библиотечных функций для улучшения результатов анализа. Разработано несколько детекторов утечек памяти для разных сценариев. Для каждой найденной ошибки детекторы также сохраняют поток управления, выполнение которого будет воспроизводить ошибку.

На втором этапе происходит верификация найденных ошибок. Для этого разработан движок символьного выполнения на базе *KLEE* [240]. На вход движок получает:

1. Поток управления (множество базовых блоков промежуточного представления *LLVM*);
2. Базовый блок, содержащий инструкцию выделения памяти;
3. Базовый блок, выполнение которого приведет к утечкам памяти.

Далее, движок производит символьное выполнение только в рамках указанного потока управления, что обеспечивает быстродействие анализа.

4.3.1. Описание общего процесса анализа

Процесс анализа состоит из двух основных этапов. Сначала производится сбор артефактов анализируемого проекта, после чего сам анализ. Ниже приводятся соответствующие команды ("*ml_hunter*" – имя разработанного инструмента):

1. *ml_hunter build "команда сборки: gcc, make, cmake"*;
2. *ml_hunter analyze*

Во время сборки проекта команды вызова компилятора *Cu/Cu++* заменяются специальным скриптом, который параллельно генерирует *LLVM*-представление программы. На полученном представлении запускается нами разработанный проход *LLVM*, который генерирует структуру данных *ProcedureGraph* для анализа.

Во время второго этапа производится восходящий анализ графа вызовов функций, начиная с вершин, не имеющих исходящих ребер (рисунок 24, уровень 1). Во время анализа каждой функции строятся соответствующие аннотации. Эти аннотации используются для анализа инструкций вызовов соответствующих функций.

4.3.2. Построение *ProcedureGraph*

ProcedureGraph содержит в себе граф потока управления, где вершинам соответствуют базовые блоки программы, а ребрам соответствуют переходы по управлению между этими блоками. Процедурный граф также содержит в себе граф потока данных со смещениями доступов 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 ключевым для разработанного метода.

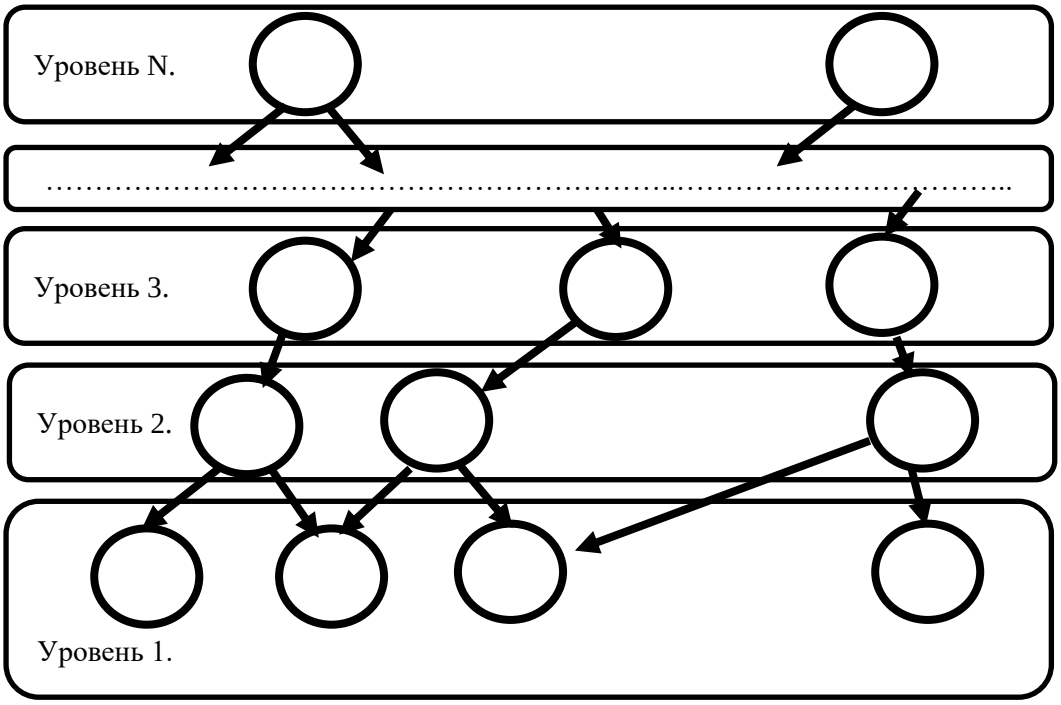

Рисунок 24. Уровни функций в графе вызовов.

Построение *ProcedureGraph* производится на базе промежуточного представления компиляторной инфраструктуры *LLVM* с применением доступного программного интерфейса. В частности, программный интерфейс предоставляет возможность получить:

1. Набор базовых блоков и содержащихся в них инструкций;
2. Для каждого базового блока набор входящих/исходящих ребер потока управления;
3. Для каждой инструкции множество use-def/def-use цепочек;
4. Для каждой функции множество вызванных/вызывающих функций.

4.3.3. Создание аннотаций

Разработанный инструмент поддерживает девять основных аннотаций:

1. ALLOC_RETURNED – функцией возвращается указатель на выделенную память;
2. INNER_ALLOC_RETURNED – функцией возвращается указатель, относительно которого происходит выделение памяти (выделяются поля структур или ячейки буфера);
3. DE_ALLOC_ARGUMENT – функция производит освобождение памяти указателя, переданного через аргумент;
4. INNER_DE_ALLOC_ARGUMENT – функция производит освобождение памяти относительно указателя, переданного через аргумент (освобождаются поля структур или ячейки буфера);
5. ALLOC_IN_GLOBAL – выделенная память в функции записывается в глобальную переменную;
6. DE_ALLOC_GLOBAL – производится освобождение глобальной переменной в функции;
7. ARGUMENT_COPY_TO_ARGUMENT – функция производит копирование одного аргумента в другой;
8. ARGUMENT_COPY_TO_GLOBAL – функция производит копирование аргумента в глобальную переменную;
9. GLOBAL_RETURNED – возвращается указатель на глобальную переменную.

Аннотации создаются параллельно анализу каждой функции (анализ соответствующей функции *ProcedureGraph*). Анализ производится восходящим образом на графе вызовов, что означает: все вызванные функции либо уже проанализированы и имеют соответствующие аннотации, либо являются библиотечными функциями. Перед началом анализа создаются аннотации по умолчанию для стандартных функций ("*malloc*", "*free*" и т.д.) и тех библиотечных функций, для которых пользователь указал аннотации вручную.

4.3.3.1. Создание аннотаций ALLOC_RETURNED

Рассматриваются все пути потока данных от инструкций выделения (вызовы функций, имеющие аннотацию ALLOC_RETURNED) памяти до инструкций возврата. Если существует путь, на котором не производится вызов функций, имеющих аннотацию DE_ALLOC_ARGUMENT, тогда для соответствующей функции создается и присваивается аннотация ALLOC_RETURNED. В аннотацию добавляется информация о потоке управления, приводящая к выполнению условий в случае которых она будет создана.

4.3.3.2. Создание аннотаций INNER_ALLOC_RETURNED

Рассматриваются все пути потока данных от инструкций выделения (вызовы функций, имеющие аннотацию ALLOC_RETURNED или INNER_ALLOC_RETURNED) памяти до инструкций возврата. Возможны два случая:

1) память выделяется функцией, имеющей аннотацию ALLOC_RETURNED, и существует путь, на котором не производится вызов функции, имеющей аннотацию DE_ALLOC_ARGUMENT. Если на рассмотренном пути указатель выделенной памяти присваивается полю/индексу другого указателя, который возвращается функцией, тогда создается аннотация INNER_ALLOC_RETURNED.

2) память выделяется функцией, имеющей аннотацию INNER_ALLOC_RETURNED, и существует путь, на котором не производится вызов функции, имеющей аннотацию INNER_DE_ALLOC_ARGUMENT, тогда для соответствующей функции создается и присваивается аннотация INNER_ALLOC_RETURNED.

В обоих случаях, в аннотациях добавляется информация о потоке управления, приводящая к выполнению условий, в случае которых она будет создана.

4.3.3.3. Создание аннотаций DE_ALLOC_ARGUMENT

Рассматриваются все пути потока данных – от аргументов функций до всех доступных инструкций вызовов функций. На каждом пути вычисляется количество инструкций, выделяющих (вызовы функций, имеющие аннотацию ALLOC_RETURNED) и освобождающих (вызовы функций, имеющие аннотацию DE_ALLOC_ARGUMENT) динамическую память. Если количество освобождений больше, чем выделений, для соответствующей функции создается и присваивается аннотация DE_ALLOC_ARGUMENT. В аннотацию добавляется информация о потоке управления, приводящая к выполнению условий, в случае которых она будет создана.

4.3.3.4. Создание аннотаций INNER_DE_ALLOC_ARGUMENT

Рассматриваются все пути потока данных – от аргументов функций до всех доступных инструкций вызовов функций. На каждом пути вычисляется количество инструкций, выделяющих (вызовы функций, имеющие аннотацию ALLOC_RETURNED по смещению, или INNER_ALLOC_RETURNED) и освобождающих (вызовы функций, имеющие аннотацию DE_ALLOC_ARGUMENT по смещению или INNER_DE_ALLOC_ARGUMENT) динамическую память по некоторому смещению от рассматриваемого аргумента. Если количество освобождений больше, чем выделений, для соответствующей функции создается и присваивается аннотация INNER_DE_ALLOC_ARGUMENT. В аннотацию добавляется информация о потоке управления, приводящая к выполнению условий, в случае которых она будет создана.

4.3.3.5. Создание аннотаций ALLOC_IN_GLOBAL

Функция получает аннотацию ALLOC_IN_GLOBAL, если в ней существует путь потока управления, на котором производится выделение памяти с присваиванием соответствующего указателя глобальной переменной. Дополнительно проверяется,

что в графе вызовов не существует пути из этой функции к такой, которая имеет аннотацию DE_ALLOC_GLOBAL для соответствующей глобальной переменной.

4.3.3.6. Создание аннотаций DE_ALLOC_GLOBAL

Функция получает аннотацию DE_ALLOC_GLOBAL, если в ней существует путь потока управления, на котором производится освобождение памяти и указатель которого содержится в глобальной переменной. Имя глобальной переменной также хранится в соответствующей аннотаций.

4.3.3.7. Создание аннотаций ARGUMENT_COPY_TO_ARGUMENT

Рассматриваются все пути потока данных между аргументами функций. Если существуют такие пути, то для них создаются аннотации ARGUMENT_COPY_TO_ARGUMENT. В аннотациях добавляется информация о номерах аргументов и потоке управления, приводящая к выполнению условий, в случае которых она будет создана.

4.3.3.8. Создание аннотаций ARGUMENT_COPY_TO_GLOBAL

Рассматриваются все пути потока данных – от аргументов функций до всех доступных глобальных переменных. Если существуют такие пути, то для них создаются аннотации ARGUMENT_COPY_TO_GLOBAL. В аннотацию добавляется информация о номере аргумента, имени глобальной переменной и потоке управления, приводящей к выполнению условий, в случае которых она будет создана.

4.3.3.9. Создание аннотаций GLOBAL_RETURNED

Функция получает аннотацию GLOBAL_RETURNED, если содержит инструкцию возврата, из которого доступна, при обратном обходе, глобальная

переменная. Имя глобальной переменной также сохраняется в соответствующей аннотации.

4.3.4. Поиск утечек памяти

Для поиска утечек памяти разработано несколько детекторов. Ниже приводится подробное описание для каждого из них.

4.3.4.1. Детектор AllocFreeImbalance

Цель данного детектора – найти пути выполнения в функциях, на которых выделяется динамическая память, освобождение которой не производится. А также указатель выделенной памяти не возвращается и не присваивается аргументам или глобальным переменным.

Данный детектор поверх представления *ProcedureGraph* строит новое графовое представление программы под названием *MemoryOperationGraph*. Вершинами графа являются базовые блоки программы, в которых производится локальное (указатель выделенной памяти не возвращается и не присваивается аргументам функций или глобальным переменным) выделение и освобождение динамической памяти. Кроме этих вершин, в *MemoryOperationGraph* входят также базовые блоки, соответствующие входным и выходным точкам функций.

Алгоритм построения *MemoryOperationGraph*.

Алгоритм на входе получает *ProcedureGraph* представление целевой функции: *procedureGraph*. Возвращает *MemoryOperationGraph* представление программы в виде карты – (*вершина*, *множество соседей*).

```
1. constructMemoryOperationsGraph(procedureGraph) {
2.   memoryOperationGraph = memoryOperationsGraphNodesConstr(procedureGraph)
3.   for (node in memoryOperationGraph) {
4.     bb = node.first
5.     nodesFrom = backwardBFSWithSkips(memoryOperationGraph, bb)
```

```
6.      for (n in nodesFrom) {
7.        memoryOperationGraph[n].insert(bb)
8.      }
9.    }
10.
11.   entryBB = procedureGraph.getEntryBlockNode();
12.   memoryOperationGraph[entryBB] = {}
13.   forwardReachable = getForwardReachableBasicBlocks(entryBB)
14.   for (bb in forwardReachable) {
15.     if (bb in memoryOperationGraph) {
16.       memoryOperationGraph[entryBB].insert(bb)
17.     }
18.   }
19.
20.   for (exitNode in procedureGraph.getExitBasicBlockNodes()) {
21.     backwardReache = getBackwardReachableBasicBlocks(exitNode)
22.     for (brNode in backwardReache) {
23.       if (brNode in memoryOperationGraph) {
24.         memoryOperationGraph[brNode].insert(exitNode)
25.         memoryOperationGraph[exitNode] = {}
26.       }
27.     }
28.   }
29.   return memoryOperationGraph
30. }
31. memoryOperationsGraphNodesConstr(procedureGraph) {
32.   memoryOperationGraph = {}
33.   for (callNode in procedureGraph.getCallNodes()) {
34.     if (isLocalAllocationOnly(callNode)) {
35.       memoryOperationGraph.insert(callNode.getParentBB(), {})
36.     } else if (isFree(callNode)) {
37.       memoryOperationGraph.insert(callNode.getParentBB(), {})
38.     }
39.   return memoryOperationGraph
40.   }
41. }
42. backwardBFSWithSkips(memoryOperationGraph , basicBlockNode) {
43.   Visited = {}
44.   nodesFrom = {}
45.   bbQueue = {}
46.   visited.insert(basicBlockNode)
47.   bbQueue.push(basicBlockNode)
48.   while (not bbQueue.empty()) {
49.     current = bbQueue.front()
```

```
50.    for (bb in current.in_neighbors()) {
51.      if (bb not in visited) {
52.        visited.insert(bb)
53.        if (bb in memoryOperationGraph) {
54.          nodesFrom.insert(bb)
55.        } else {
56.          bbQueue.push(bb)
57.        }
58.      }
59.    }
60.    bbQueue.pop()
61.  }
62.  return nodesFrom
63. }
```

Процедура *getForwardReachableBasicBlocks* возвращает все доступные базовые блоки потока управления для заданного блока при прямом обходе в ширину. Процедура *isLocalAllocationOnly* проверяет, что заданная инструкция вызова производит выделение памяти, указатель которой не присваивается глобальным переменным, не возвращается и не присваивается аргументам функций. Процедура *isFree* проверяет, что заданная инструкция вызова производит освобождение памяти.

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.*

Доказательство: рассмотрим любой путь *P* из входной точки до выходной, состоящий из последовательности ребер *{E1, E2, … En}* в *MemoryOperationGraph* графе. Рассмотрим любое из ребер *Ei*, состоящее из пары вершин *(Vi, Vi+1)*. Из процедуры *constructMemoryOperationsGraph* очевидно, что ребро *Ei* между

вершинами – *Vi, Vi+1* добавляется, если *Vi* достигается при обратном обходе в ширину графа потока управления с вершины *Vi+1* (в случае входной точки производится прямой обход). Из этого следует, что любому ребру *Ei = (Vi, Vi+1)* соответствует, как минимум, один путь потока управления между вершинами *Vi, Vi+1*. Путь *P* можно также представить в виде последовательности вершин *{V1, V2, … Vn+1}*, где любому *Vi, Vi+1* соответствует, как минимум, один путь потока управления, а *V1* – входная точка, *Vn+1* – выходная. Из этого следует, что существует, как минимум, один путь от *V1* до *Vn+1* в потоке управления, поскольку любое *Vi+1* достижимо из *Vi*. ***Теорема доказана***.

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

Доказательство: сложности алгоритмов поиска в ширину составляют $O(n + e)$, где n – количество вершин в ГЗП, а e – количество ребер [236]. Сложность процедур *backwardBFSWithSkips* и *isLocalAllocationOnly* также $O(n + e)$, поскольку производится поиск в ширину с некоторыми модификациями. Алгоритм в ширину может вернуть не более *n*-вершин. Процедура *isFree* имеет постоянную сложность. Из этого следует, что в предложенном алгоритме в общей сумме производится не более:

$$n * C1 * (n + e) + n * C2 * (n + e) + n * n +$$
$$n * C3 * (n + e) + n + n * C4(n + e) + n * n \leq$$
$$n * C1 * (n + e) + n * C2 * (n + e) + n * C3 * (n + e) + n * C4(n + e) +$$
$$3 * n * (n + e) \leq C * n * (n + e)$$

Где $C1, C2, C3, C4$ – положительные числа и $C1 + C2 + C3 + C4 + 3 = C$. Из этого следует, что сложность алгоритма $O(n * (n + e))$. ***Теорема доказана***.

Ниже приводится описание детектора утечек памяти, происходящих в рамках функции. Надо обратить внимание на то, что алгоритм в случае экспоненциального роста количества путей потока управления рассматривает не более *PATHS_LIMIT=100000* путей для рассматриваемой пары входных и выходных точек.

Алгоритм детектора AllocFreeImbalance

```
1.  checkImbalanceForMallocNodes(procedureGraph) {
2.    memoryOperationGraph = constructMemoryOperationsGraph(procedureGraph)
3.    mallocPathsReporst = {}
4.    entryBB = procedureGraph.getEntryBlockNode()
5.    for (exitBB in procedureGraph.getExitBasicBlockNodes()) {
6.      targetPaths = getMemoryOpGraphPathsBetweenTwoNodes(memoryOperationGraph,
7.                                                          entryBB, exitBB)
8.      for (bbPath in targetPaths) {
9.        path = {}
10.       for (bbNode in bbPath) {
11.         for (node in bbNode->getNodes()) {
12.           path.emplace_back(node)
13.         }
14.       }
15.       for (node in path) {
16.         if (node->isCall() and  isLocalAllocationOnly(node)) {
17.           forwardReached = findAllForwardReachableNodes(instrNode)
18.           if (not flowsToFree(forwardReached, bbPath)) {
19.             if (node in mallocPathsReporst) {
20.               mallocPathsReporst[node].push_back(bbPath)
21.             } else {
22.               mallocPathsReporst[node] = { bbPath }
23.             }
24.           }
25.         }
26.       }
27.     }
28.   }
29.   for (report in mallocPathsReporst) {
30.     reportImbalanceBug(report)
31.   }
32. }
33. flowsToFree(forwardReached, bbPath) {
34.   for (reachedNode in forwardReached) {
35.     if (reachedNode.isCall() and isFree(reachedNode) and
36.         isInTrace(reachedNode, bbPath)) {
37.       return true
38.     }
39.   }
40.   return false
41. }
42. isInTrace(node, bbPath) {
```

```
43.  parentBB = node.getParentBB()
44.  for (bb in bbPath) {
45.    if (bb == parentBB) {
46.      return true
47.    }
48.  }
49.  return false
50. }
```

Процедура *reportImbalanceBug* производит детальное сообщение найденных ошибок.

Ниже приводится описание алгоритма поиска всех путей между двумя вершинами графа *MemoryOperationGraph*. Надо обратить внимание, что алгоритм имеет ограничения *PATHS_LIMIT=100000* для случаев экспоненциального роста количества путей. Надо также отметить, что, на практике, данное ограничение сработало ***менее десяти раз*** для таких сложных проектов как *openssl* [36] и *ffmpeg* [198].

Алгоритм построения путей между вершинами MemoryOperationGraph.

Алгоритм на входе получает две вершины *MemoryOperationGraph* и возвращает все нециклические пути между заданными вершинами. В алгоритме используется дополнительное ограничение (PATHS_LIMIT) для случаев экспоненциального роста количества путей.

```
1.  PATHS_LIMIT = 100000
2.  getMemoryOpGraphPathsBetweenTwoNodes(memoryOperationGraph,
3.                                       sourceBB, destinationBB) {
4.    targetPaths = {}
5.    pathsQueue = {}
6.    pathsQueue.push({sourceBB})
7.    while (not pathsQueue.empty()) {
8.      path = pathsQueue.front()
9.      last = path.back()
10.     if (last == destinationBB) {
11.       targetPaths.emplace_back(path)
12.       pathsQueue.pop()
13.       continue
14.     }
15.     for (destNode in memoryOperationGraph[last]) {
```

```
16.      if (destNode not in path) {
17.        newPath = path
18.        newPath.emplace_back(destNode)
19.        pathsQueue.push(newPath)
20.      }
21.    }
22.    /* Обработка случаев экспоненциального роста количества путей. */
23.    if (pathsQueue.size() > PATHS_LIMIT) {
24.      return targetPaths
25.    }
26.    pathsQueue.pop()
27.  }
28.  return targetPaths
29. }
```

4.3.4.2. Детектор AllocFreeImbalanceOfGlobalVariables

Цель данного детектора найти утечку памяти в случаях, когда выделение памяти производится для глобальных переменных. Алгоритм, кроме *ProcedureGraph* и аннотаций, также использует граф вызовов программы. Основная идея алгоритма заключается в том, что, если существует функция с аннотацией ALLOC_IN_GLOBAL и в программе не производится его вызов, тогда может произойти утечка памяти. Простота данного детектора получается из-за того, что анализ аннотаций производится восходящим образом, и информация передается вызывающим функциям. Например, если в программе следующая последовательность вызовов – "*f()->g()->k()*" и функция "*k*" производит выделение в глобальной памяти. Если ни в одной функции не производится освобождение глобальной памяти, тогда аннотацию о выделении глобальной памяти получают "*g*" и "*f*" также. Детектором обрабатываются только функции, не имеющие вызовов, поскольку, в противном случае, в вызывающей функции, может быть произведено освобождение глобальной переменной.

Алгоритм детектора AllocFreeImbalanceOfGlobalVariables.

```
18. checkAllocFreeImbalanceOfGlobalVariables(callGrap , procedureGraph) {
19.   for (cgNode : callGrap) {
```

```
20.    if (cgNode->getInEdgesCount() == 0 ) {
21.      procedureGraph = cgNode->getProcedureGraph()
22.      functionName = cgNode->getFunctionName()
23.      for (annotationInfo in getAnnotationsForFunction(functionName)) {
24.        if (annotationInfo is ALLOC_IN_GLOBAL) {
25.          reportAllocFreeInconsistency(procedureGraph, annotationInfo)
26.        }
27.      }
28.    }
29.  }
30. }
```

Процедура *getAnnotationsForFunction* возвращает список всех аннотаций для заданной функции. Процедура *reportAllocFreeInconsistency* производит детальное сообщение найденных ошибок.

4.3.4.3. Проверка выполнимости путей ошибок на базе KLEE

На втором этапе статический анализ находит множество ошибок и соответствующие потоки управления (множество путей), которые приводят к их выполнению. Далее, производится проверка выполнимости найденных путей ошибок. Для этого разработан новый инструмент (*PM_KLEE*) на базе *KLEE*, который решает данную задачу. На входе инструмент получает:

1. Промежуточное представление *LLVM*;
2. Имя анализируемой функции;
3. Потоки управления (набор базовых блоков), которые будут выполнены в ходе работы инструмента (все остальные базовые блоки целевой функции и других вызванных функций не будут выполнены);
4. Множество "*важных*" базовых блоков, которые должны обязательно быть выполнены.

Инструмент подтверждает выполнимость заданного потока управления, если он

смог найти входные данные, которые позволяют поочередно выполнить указанные "*важные*" базовые блоки, не выходя за рамки указанного потока управления.

Кроме вышеуказанной функциональности, в инструменте произведен ряд улучшений:

1. Устранено динамическое поведение. Вызовы динамических функций заменяются символьными значениями;
2. Добавлена возможность автоматического создания символьных значений для необходимых переменных;
3. Добавлена возможность ограничения количества итераций циклов.

4.3.5. Сравнение с существующими инструментами

В таблице 23 приводится сравнение разработанного инструмента с существующими аналогами. Тестирование производится на наборе тестов из таблицы 13 – взятых из реальных проектов.

Таблица 23. Сравнение разработанного инструмента с существующими аналогами.

Имя инструмента	**Утечка-1**	**Утечка-2**	**Утечка-3**	**Утечка-4**	**Утечка-5**	**Утечка-6**
CSA	Не находит	Не находит	Находит	Не находит	Не находит	Находит
Infer	Не находит	Не находит	Находит	Не находит	Не находит	Находит
SMOKE	Parse error	Parse error	Parse error	Parse error	Parse error	Parse error
PCA	Не находит	Не находит	Не находит	Не находит	Не находит	Не находит
Fastcheck	Не находит	Не находит	Находит	Parse error	Parse error	Parse error
SVF	Не находит	Не находит	Не находит	Не находит	Не находит	Не находит
PML Checker	Ошибка запуска	Ошибка запуска	Ошибка запуска	Ошибка запуска	Ошибка запуска	Ошибка запуска
ml-hunter	Находит	Находит	Находит	Находит	Находит	Находит

Как видно из результатов, разработанный инструмент показывает наилучшие результаты.

4.3.6. Тестирование на наборе тестов Juliet

Разработанный инструмент был протестирован на наборе *Juliet Test Suite* [241], предназначенном для оценки надежности и безопасности программного обеспечения. В нем содержится более чем 112 000 экземпляров для тестирования широкого спектра распространенных уязвимостей программного обеспечения, включая: переполнение буфера, утечку памяти, форматную строку и т.д. Существуют тестовые примеры, написанные на нескольких языках программирования, включая *Cu/Cu++, Java*. *ml-hunter* был протестирован на примерах утечек памяти с идентификатором *CWE401_Memory_Leak*. В этих примерах функции разделены на две основные категории: "*хорошие*" функции без утечек памяти, и "*плохие*" с утечкой памяти. В тестах содержатся различные сценарии выделения и использования памяти. Для выделения памяти используются такие функции, как "*malloc*", "*calloc*", "*realloc*", "*strdup*" и т.д. Могут быть выделены простые типы, массивы и сложные структуры. Функции, содержащие утечку памяти, имеют суффикс "*_bad*" в имени. Это соглашение об именах используется для определения общего количества случаев утечек памяти, при этом функции без суффикса "*_bad*" считаются "*хорошими*" функциями, не содержащими утечек памяти. В наборе *CWE401_Memory_Leak* содержится 868 утечек памяти. Разработанный инструмент находит все утечки памяти с дополнительными двадцатью ложными срабатываниями и обеспечивает:

1. Количество истинно-положительных (TP) – 868;
2. Количество истинно-отрицательных (TN) – 4586;
3. Количество ложно-положительных (FP) – 20;
4. Количество ложно-отрицательный (FN) – 0.

В таблице 24 приводится сравнение *ml-hunter* с существующими инструментами. Из результатов становится очевидно, что разработанный инструмент по всем критериям обходит доступные.

Таблица 24. Сравнение ml-hunter с существующими инструментами.

Имя инструмента	TP	TN	FP	FN
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**

4.3.7. Результаты анализа реальных проектов

Разработанный инструмент был протестирован на ряде реальных проектов и смог найти множество утечек памяти. В таблице 25 приводится список всех найденных и подтвержденных ошибок.

Таблица 25. Найденные утечки памяти в реальних 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

Далее приводятся некоторые ошибки и их исправления, а также утечка памяти в библиотеке *ffmpeg* [198] на пути аварийного завершения программы. Номер открытой ошибки в системе *10342* [242]. На рисунке 25 приводится исправление.

```
diff --git a/fftools/ffmpeg_opt.c b/fftools/ffmpeg_opt.c
index cf385c388e..002df84dcc 100644
--- a/fftools/ffmpeg_opt.c
+++ b/fftools/ffmpeg_opt.c
@@ -462,17 +462,12 @@ static int opt_map_channel(void *optctx, const char *opt, const char *arg)
     AVStream *st;
     AudioChannelMap *m;
     char *allow_unused;
-    char *mapchan;

     av_log(NULL, AV_LOG_WARNING,
            "The -%s option is deprecated and will be removed. "
            "It can be replaced by the 'pan' filter, or in some cases by "
            "combinations of 'channelsplit', 'channelmap', 'amerge' filters.\n", opt);

-    mapchan = av_strdup(arg);
-    if (!mapchan)
-        return AVERROR(ENOMEM);
-
     GROW_ARRAY(o->audio_channel_maps, o->nb_audio_channel_maps);
     m = &o->audio_channel_maps[o->nb_audio_channel_maps - 1];

@@ -482,7 +477,6 @@ static int opt_map_channel(void *optctx, const char *opt, const char *arg)
         m->file_idx = m->stream_idx = -1;
         if (n == 1)
             m->ofile_idx = m->ostream_idx = -1;
-        av_free(mapchan);
         return 0;
     }

@@ -519,8 +513,7 @@ static int opt_map_channel(void *optctx, const char *opt, const char *arg)
         exit_program(1);
     }
     /* allow trailing ? to map_channel */
-    if (allow_unused = strchr(mapchan, '?'))
-        *allow_unused = 0;
+    allow_unused = strchr(arg, '?');
     if (m->channel_idx < 0 || m->channel_idx >= st->codecpar->ch_layout.nb_channels ||
         input_files[m->file_idx]->streams[m->stream_idx]->user_set_discard == AVDISCARD_ALL) {
         if (allow_unused) {
@@ -534,7 +527,6 @@ static int opt_map_channel(void *optctx, const char *opt, const char *arg)
         }

     }
-    av_free(mapchan);
     return 0;
 }
 #endif
```

Рисунок 25. Исправление ошибки на пути аварийного завершения программы в ffmpeg.

Ниже приводится множество утечек памяти в *radar2* [196] (платформа анализа бинарного кода). Номер открытых ошибок в системе – *21703* [243], *21704* [244] и *21705*

[245]. Некоторые исправления приводятся на рисунках 26 и 27.

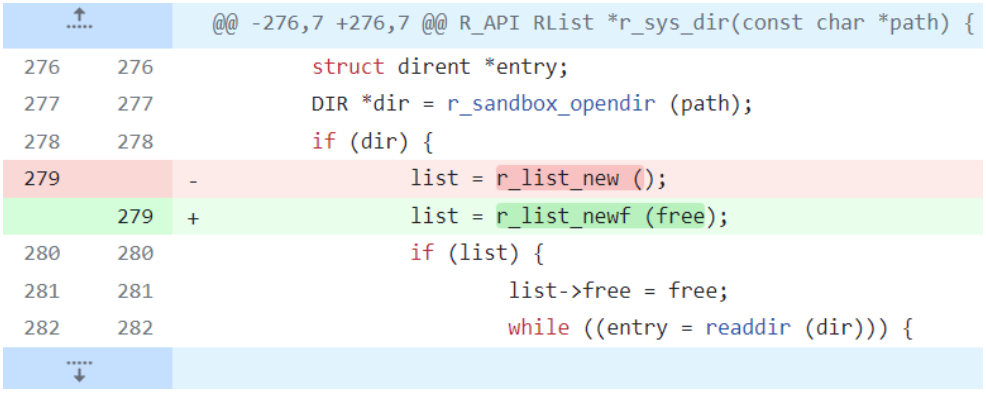

Рисунок 27. Исправление ошибки 21704 в Radar2.

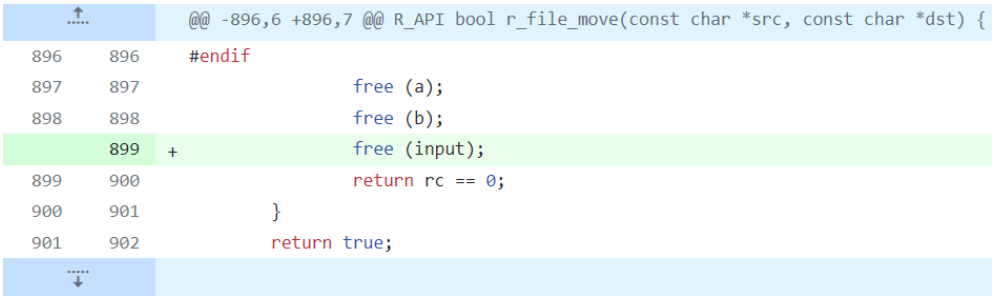

Рисунок 26. Исправление ошибки 21703 в Radar2.

Утечка памяти в библиотеке *openssl*. Номер открытой ошибки в системе *20870* [246]. На рисунке 28 приводится исправление.

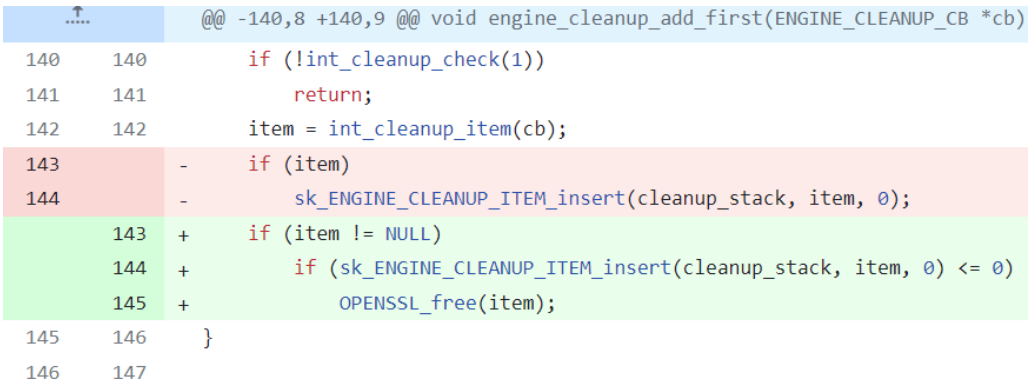

Рисунок 28. Утечка памяти в библиотеке OpenSSL.

4.4. Заключение по разработанным методам поиска ошибок

Были предприняты попытки разработать инструменты поиска таких ошибок, как утечка памяти, форматная строка и использование освобожденной памяти. В разделе 4.1 описывается метод, который на основе статического анализа находит вышеуказанные ошибки. Однако метод имеет множество ложных срабатываний и не может быть применен для анализа реальных проектов. В разделе 4.2 описывается метод динамического анализа для поиска ошибок использования освобожденной памяти. Несмотря на то, что разработанный метод практически не имеет ложных срабатываний, он также не применим для анализа реальных проектов. Основные причины – это не масштабируемость метода и ограниченное количество анализируемых путей (только тех путей, которые были открыты с помощью символьного выполнения). В разделе 4.3 описывается комбинированный метод поиска утечек памяти, который на первом этапе применяет статический анализ для нахождения всех возможных утечек памяти. А на втором этапе применяется символьное выполнение для проверки выполнимости путей. Разработанный метод масштабируется и обладает высокой точностью. Сравнение с существующими методами и найденные реальные ошибки доказывают его эффективность. В частности, разработанный инструмент *ml-hunter* превзошел по качеству такие широко используемые инструменты, как *CSA*, *Infer*, *SMOKE*, *PCA* и *SVF* на известном наборе *Juliet Test Suite*, *CWE401_Memory_Leak*. Кроме этого, *ml-hunter* смог найти десятки утечек памяти в известных открытых проектах, включая *openssl*. Найденные ошибки были нами опубликованы и исправлены при содействии сообщества разработчиков.

5. Методы фаззинга программ

В данной главе приводится описание разработанных автором методов фаззинга для разных задач. Разработан общий инструмент фаззинга под названием *ISP-Fuzzer* [3] [27], на базе которого реализуются все предлагаемые методы. 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.

На рисунке 29 приводится описание архитектуры разработанного инструмента. Ядро написано на *Cи/Cи++* для быстродействия. Поддерживается возможность прямого вызова *Python* кода из *Cи/Cи++*, на основе чего и реализован механизм драйверов. Такой подход делает инструмент легко расширяемым и эффективным для поддержки новых функциональных возможностей. *В рамках исследования в данной главе разработаны новые методы фаззинга, включающие генерацию сложно структурированных данных на базе БНФ автоматов, анализ интерфейсных функций, направленный анализ и интеграцию статического анализа, которые и выносятся на защиту*.

ISP-Fuzzer поддерживает множество входов для фаззинга: файлы, сетевые протоколы, командная строка, переменные окружения, стандартный вход и т.д. В инструмент интегрирован статический анализ кода, который обеспечивает различные функциональные возможности фаззера (направленный фаззинг, интеграция с символьным выполнением и т.д.). В состав *ISP-Fuzzer* также входит инструмент, который позволяет из трасс выполнения программы восстановить неявные вызовы функций. Такой подход позволяет итеративным образом взаимно улучшать фаззинг и статический анализ.

Ниже приводится краткое описание компонентов инструмента:

1. ***Контрольный модуль*** предназначен для определения последовательности применения плагинов. Например, на основе конфигурационного файла можно указать, какие мутации применить в процессе фаззинга.
2. ***Драйвер мутаций данных*** позволяют применить мутации с указанными ограничениями на входных данных.
3. ***Драйвер генераций БНФ данных*** взаимодействует с инструментом генерации БНФ структурированных данных и возвращает синтаксически валидные данные для указанного языка или формата данных.
4. ***Драйвер, вызывающий ДСР,*** взаимодействует с инструментом динамического символьного решателя и получает от него новые данные, которые могут увеличить покрытие кода. Полученные данные также могут положительно повлиять на мутирующие алгоритмы и обеспечить быстрый прирост покрытия кода.
5. ***Драйвер, вызывающий Radamsa***, взаимодействует с инструментом *Radamsa* [207] и получает от него данные, мутированные на основе указанных регулярных выражений.
6. ***Драйвер, генерирующий данные на основе Peach Pit***, позволяет генерировать данные на основе *Peach Pit* [201] и использовать их для фаззинга: сетевых протоколов, командной строки, разных форматов данных и т.д.
7. ***Драйвер предобработки данных*** предназначен для начальной обработки данных перед тем, как начать их мутировать. Например, если необходимо фаззить архивирующую программу, тогда целесообразно поменять данные внутри архива. С помощью этого плагина можно разархивировать текущий файл и мутировать его содержимое.

8. ***Драйвер, запускающий программу***, отвечает за запуск программы и его мониторинг. Он собирает информацию о статусе завершения программы и покрытии кода.

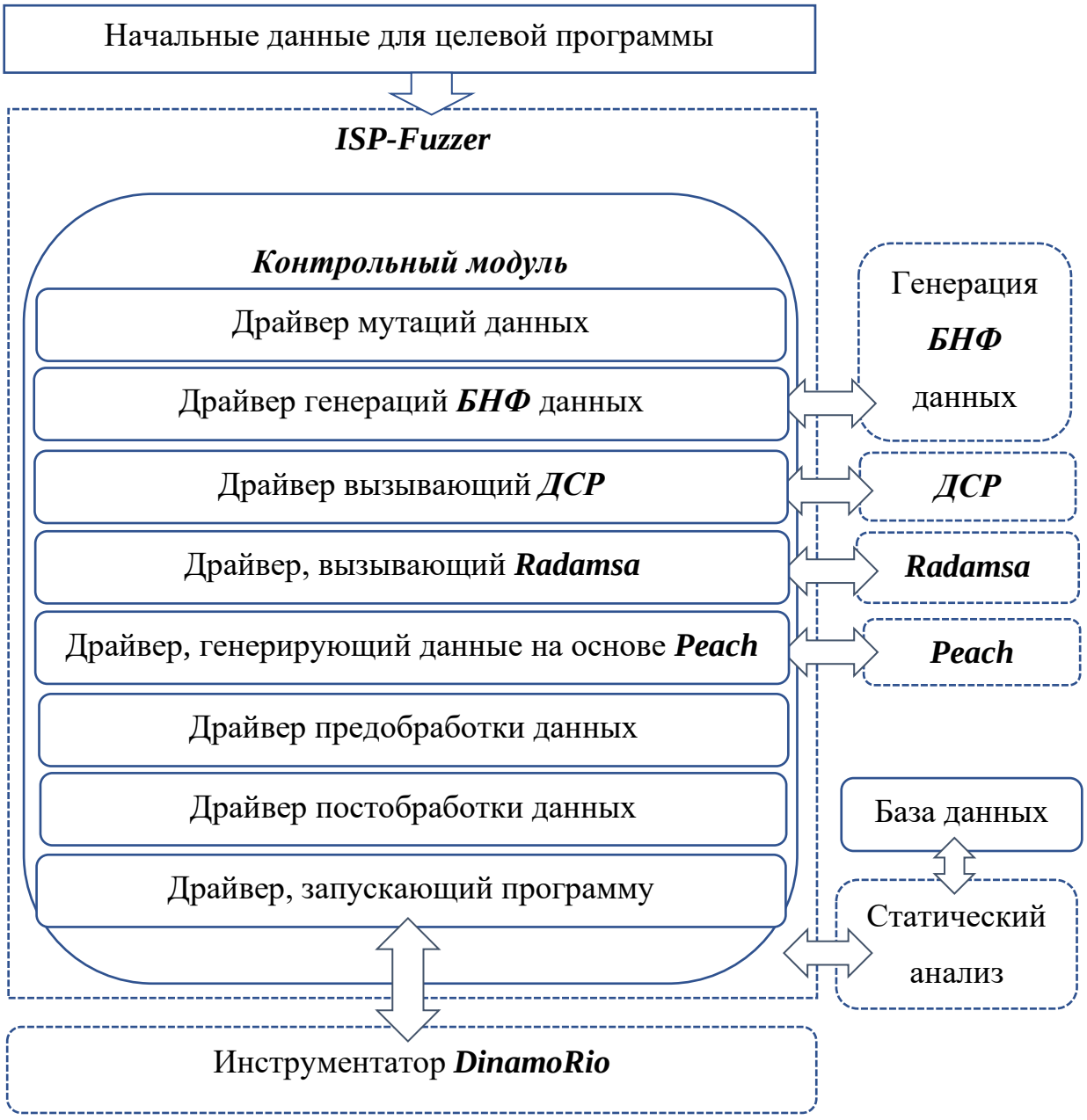

Рисунок 29. Архитектура ISP-Fuzzer.

Вместе с *ISP-Fuzzer* предоставляется инструментарий для обеспечения ряда дополнительных функциональных возможностей:

1. ***Инструментатор DinamoRio*** предназначен для запуска целевой программы в среде *DynamoRIO* [189], сбора покрытия кода, статусов заверений программы и трасс выполнения. Далее, эта информация передается фаззеру (конкретному плагину), на основе чего определяются входные данные, которые могли бы увеличить покрытие кода, привести к падению или зависанию программы. В данном инструменте реализована идея "*Fork Server*" [247], которая многократно ускоряет скорость запуска целевой программы и делает фаззинг эффективным. Инструмент также имеет возможность инструментировать конкретные базовые блоки кода для реализации направленного фаззинга.
2. ***Инструмент генерации БНФ данных*** дает возможность сгенерировать синтаксически корректные данные для конкретного языка программирования или формата данных.
3. ***Инструмент статического поиска путей*** позволяет найти целевые пути в потоке управления программы (граф вызовов функций тоже используется). Этот инструмент, в основном, используется для направленного фаззинга. С его помощью строится множество путей, которые приводят к целевым точкам программы. Далее, инструментируются только эти пути, что, в свою очередь, позволяет фаззеру сгенерировать данные, позволяющие приблизиться/дойти до целевых точек.
4. ***Инструмент восстановления вызовов функций*** позволяет из трасс программ восстановить неявные вызовы (виртуальные функции, указатели на функциях) функций. Далее, на основе этой информации обновляется база

данных целевой программы, что позволяет при следующей итерации более точно получать нужные пути из базы данных.

5. ***Инструмент, визуализирующий мутации файлов***, позволяет отобразить дерево мутаций, полученное в процессе фаззинга. Он показывает какие мутации для каких входных файлов были эффективными (открыли новые пути в целевой программе, рисунок 30).

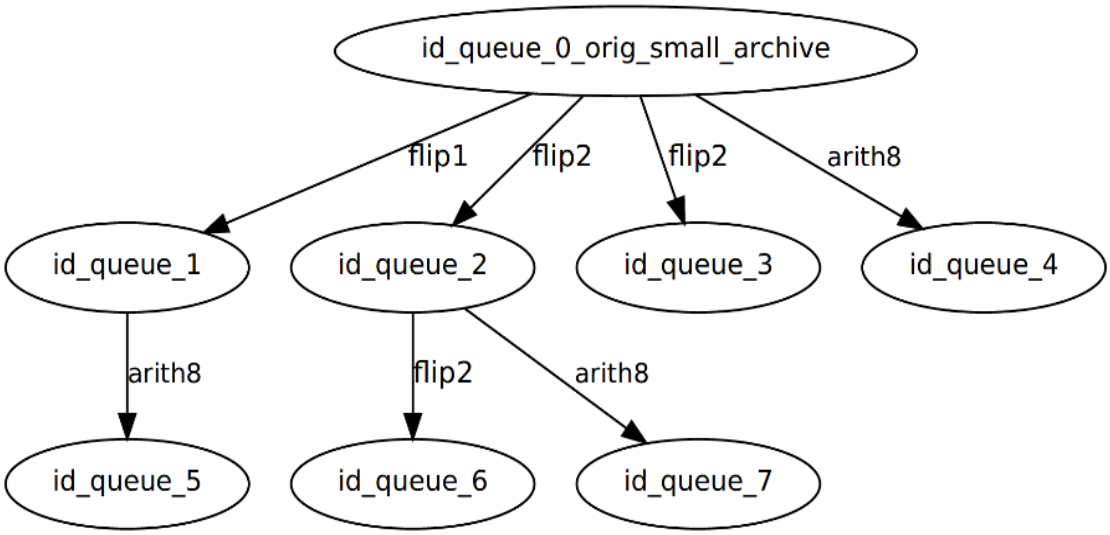

Рисунок 30. Пример дерева мутаций.

6. ***Инструмент, определяющий, какие фрагменты входа влияют на какие базовые блоки кода.*** Данный инструмент позволяет из трасс программы и соответствующих входных данных определить, какие фрагменты данных могли повлиять на открытие конкретного базового блока. Далее, эта информация может быть использована в разных задачах. Например, мутировать только нужные фрагменты входа для достижения конкретного блока кода и т.д.

Для обеспечения эффективной работы инструмента на больших мощностях разработан отдельный скрипт (менеджер), позволяющий параллельное и распределенное выполнения *ISP-Fuzzer* (рисунок 31).

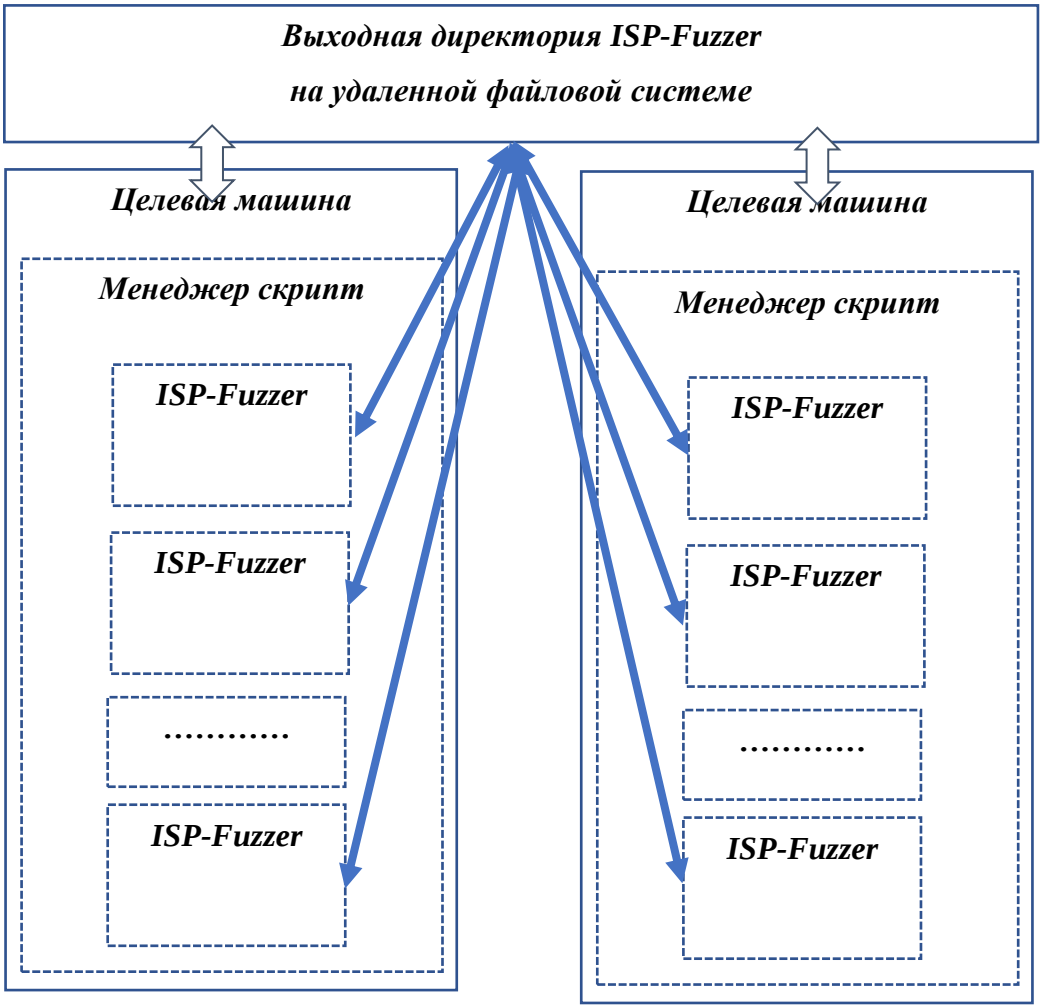

Рисунок 31. Схема параллельного и распределенног фаззинга.

Менеджер запускает множество экземпляров фаззера, которые периодически сканируют общую выходную директорию на "*интересные*" входные данные, найденные другими экземплярами фаззера. Если есть "*интересные*" входные данные, текущий экземпляр добавляет их в свою очередь, и продолжает работу. Таким образом, получается, что в каждом экземпляре фаззера содержатся все "*интересные*" входные данные, которые приводят к увеличению покрытия кода. Для

одновременного фаззинга на нескольких машинах используется распределенная файловая система. В качестве выходной директории указывается удаленная директория, что обеспечивает синхронизацию экземпляров фаззера на разных машинах.

5.1. Фаззинг программ, принимающих сложные структурированные данные

В данной секции приводится описание разработанного автором метода фаззинга программ, принимающих на входе 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 описания правил в платформе *ANTLR* (*ANother Tool for Language Recognition*) [248], что позволяет обеспечить автоматическую поддержку более 280 языков и форматов данных.

5.1.1. Внутренне представление ANTLR

Каждое БНФ правило в *ANTLR* представлено в виде универсального автомата, где каждое состояние для нетерминальных символов представляет собой такой же автомат (рисунок 32). *ANTLR* используется как генератор парсеров. После

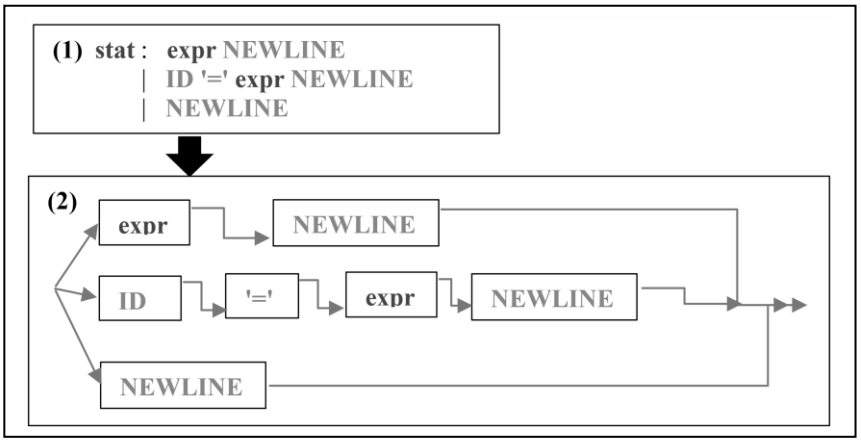

Рисунок 32. Пример автомата ANTLR.

лексического анализа, если полученная последовательность лексем позволяет из начального состояния автомата попасть в конечное, тогда входные данные считаются синтаксически корректными.

5.1.2. Механизм весов ребер автомата

В *ANTLR* была также добавлена поддержка весов для ребер автомата (рисунок 33), 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 автоматов. 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 $P(i) = \frac{w_i}{(w_1 + \cdots + w_N)}$ (обратите внимание, что, если имеется только одно исходящее ребро, то $P(1) = \frac{w_1}{w_1} = 1$.

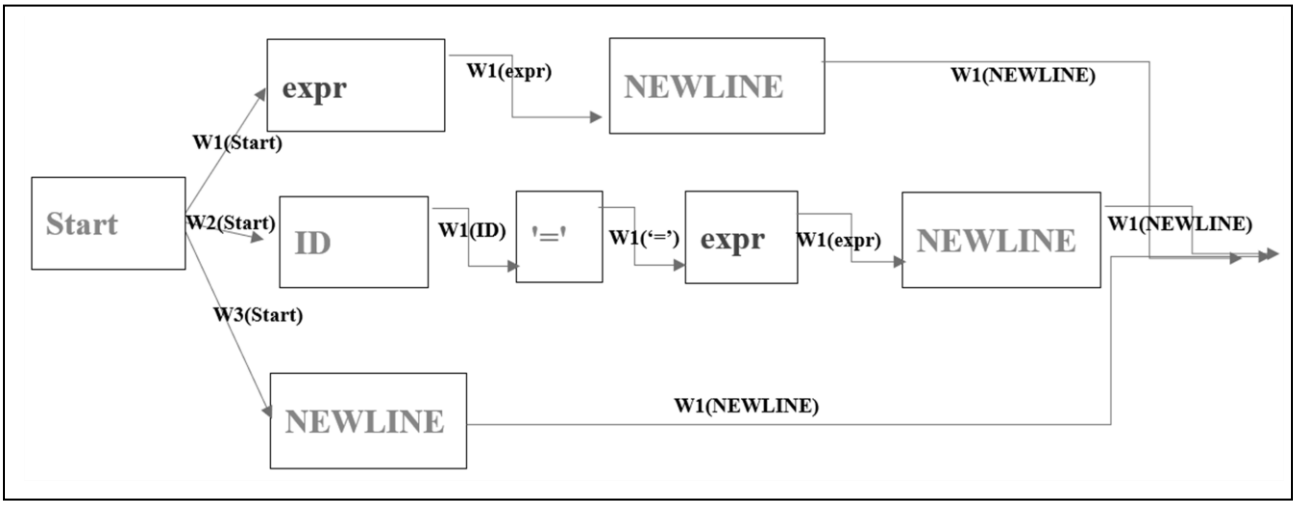

Рисунок 33. Пример автомата ANTLR с поддержкой весов.

5.1.3. Генерация БНФ структурированных данных

Основная идея генерации данных – это:

1. Произвольный проход от начального состояния автомата к конечному состоянию, что предоставляет некую последовательность состояний автомата – путь состояний автомата;
2. Для каждого терминального состояния пути – добавление в буфер соответствующие ему символы;
3. Для каждого нетерминального состояния рассмотрение соответствующего ему автомата и нахождение в нем пути от начального состояния до конечного, состоящего 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.

Были разработаны две версии алгоритма генераций БНФ структурированных данных под названием *SD-Gen* и *SD-Gen++*. Основная разница этих алгоритмов заключатся в том, что *SD-Gen* – выбор текущего ребра для построения пути делает произвольным образом. *SD-Gen++* при работе получает также веса всех ребер автомата, и при выборе текущего ребра учитывает его вес. Чем больше вес ребра, тем больше вероятность его выбора. Такой подход позволяет создать тесную интеграцию с инструментом фаззинга и улучшение результатов.

5.1.4. Интеграция с инструментом фаззинга

Интеграция *SD-Gen* с инструментом фаззинга происходит достаточно просто. Инструмент фаззинга периодически вызывает *SD-Gen* и запрашивает некоторое количество примеров целевого языка или формата данных.

В случае *SD-Gen++* интеграция с инструментом фаззинга происходит по-иному (рисунок 34). В *ISP-Fuzzer* 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.

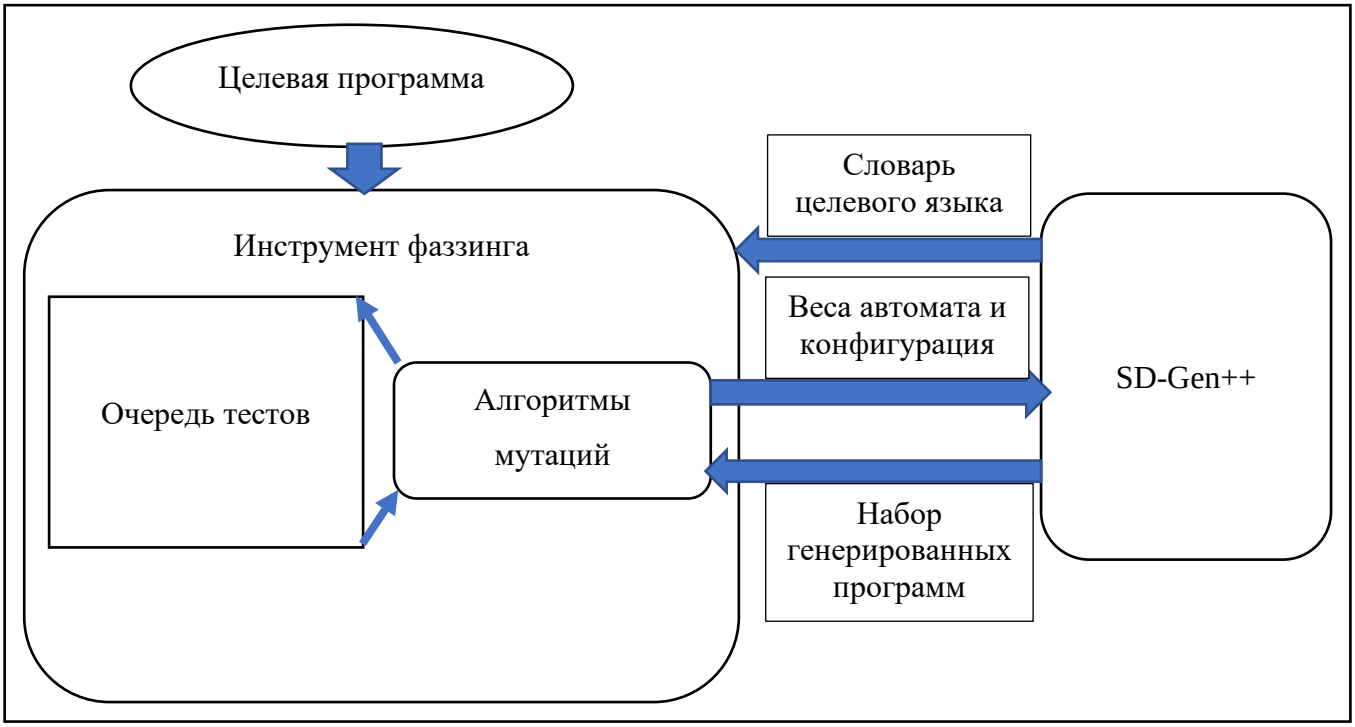

Рисунок 34. Схема интеграции SD-Gen++ с инструментом фаззинга.

5.1.5. Экспериментальные результаты

Сравнение инструментов фаззинга с применением разработанного инструмента *SD-Gen* приводится в таблице 26. Время выполнения тестов 90 минут. Как видно из таблицы, *SD-Gen* во всех случаях позволяет улучшить результаты. В случае фаззинга интерпретатора *Python* прирост количества базовых блоков составляет более 25%, что доказывает эффективность разработанного метода. Сравнение результатов фаззинга с применением инструментов *SD-Gen* и *SD-Gen++* приводится в таблице 27. Тестирование производилось в течение двух недель. Как видно из таблицы, во всех случаях скорость фаззинга ухудшается для *SD-Gen++*, поскольку в текущей реализации веса передаются через файл (*SD-Gen* работает без весов). Но, несмотря на

это, покрытие кода во всех случаях увеличивается, кроме *clang-11*. На *clang-11* видно незначительное ухудшение покрытия кода, обусловленное замедлением инструмента фаззинга, который может быть исправлен (вместо файла веса могут быть переданы через разделяемую память).

Таблица 26. Сравнение инструментов фаззинга при применении SD-Gen.

Имя тест	***Количество всех базовых блоков***	***Количество базовых блоков ISP-Fuzzer***	***Количество базовых блоков ISP-Fuzzer + SD-Gen***	***Прирост***
gcc-7.1	41329	6240	6356	**+116**
g++-7.1	41440	6378	6381	**+3**
python-2.7	123819	13315	16979	**+3664**
php-v7.1.7	342988	7497	8003	**+506**
luca-5.3.4	8266	2007	2515	**+508**

Эффективность *SD-Gen++* доказывается тем, что во всех остальных случаях, кроме *clang-11*, количество открытых путей увеличивается 2–3 раза.

Таблица 27. Сравнение результатов фаззинга с применением инструментов SD-Gen и SD-Gen++

	С SD-Gen	**С SD-Gen++**	**Прирост**
Имя теста	**Clang-11**	**Clang-11**	
Количество выполнений целевой программы	1,294,192	786,089	**-508,103**
Количество выполненных путей	935	874	**-59**
Bitmap-покрытие (%)	50.4	48.3	**-2.1**
Имя теста	**GCC-5.5**	**GCC-5.5**	
Количество выполнений целевой программы	1,009,786	963,028	**-46,758**
Количество выполненных путей	956	2347	**+1,391**
Bitmap-покрытие (%)	67.4	68.2	**+0.8**
Имя теста	**GCC-11**	**GCC-11**	
Количество выполнений целевой программы	787,195	370,175	**-417,020**
Количество выполненных путей	843	3,948	**+3,105**
Bitmap-покрытие (%)	70.6	78.2	**+7.6**
Имя теста	**Fortran**	**Fortran**	
Количество выполнений целевой программы	972,926	835,971	**-136,955**
Количество выполненных путей	734	2,353	**+1,619**
Bitmap-покрытие (%)	66	71.3	**+5.3**
Имя теста	**Python-2**	**Python-2**	
Количество выполнений целевой программы	1,636,220	189,489	**-1,446,731**
Количество выполненных путей	973	2,896	**+1,923**
Bitmap-покрытие (%)	35.4	36.2	**+0.8**
Имя теста	**Python-3**	**Python-3**	
Количество выполнений целевой программы	577,347	170,886	**-406,461**
Количество выполненных путей	1155	3302	**+2,147**
Bitmap покрытие (%)	41.7	46.7	**+1**

5.1.5.1. Найденные реальные ошибки

Во время фаззинга *gcc-12* и *clang-14* удалось найти входные данные (рисунок 35), приводящие оба компилятора к падению. В обоих случаях инструмент фаззинга эксплуатировал ограничения алгоритма синтаксического анализа рекурсивного спуска, превысив лимит стека по умолчанию для компилятора.

typedef (& const & const & & const & & & const & & & const & & & & const & & const & const & & const & const & & const & const & const & const & & const & & & const & & const & & const & const & & & const & & const & const & & & const & & const & const & const & & & const & & const & const & const & & & const & & const & const & const ***…… длинное повторение &***……. const & & const & const & & **const ... k_) ( )** -> **char = delete ; ; ;**

template < friend (& const & const & const & & & & & & & & const & const & & const & const & & const & const & const & & const & const & const & const & & & const & & const & const & & & & & const & & const & & & const & const & const & & & const & & const & const & & & & & & const & & const & & const & & const & & const & const & const & & ***…… длинное повторение &***……. const & & const & const & & **... _r) ( )** -> **char** > **;**

Рисунок 35. Данные, приводящие компиляторы gcc-12 и clang-14 к падению.

5.2. Метод фаззинга интерфейсных функций в том числе для интернета вещей

В данном разделе описывается разработанный автором метод фаззинга интерфейсных функций для языка *Java*. Предлагаемый метод имеет несколько преимуществ:

1. Автоматически присваиваются аннотации всем функциям и классам из *JAR* файла [250], которые используются во время фаззинга для генераций разных сценариев использования интерфейсных функций;

2. Генератор последовательности вызовов интерфейсных функций и их аргументов может получить сложные комбинации, где возвращаемые значения одной функции могут быть использованы в других;
3. Сгенерированные сценарии могут быть запущены на разных устройствах (например, на мобильных телефонах). Может быть произведен параллельный запуск на множестве устройств с синхронизацией результатов;
4. Поддержка шаблонов начальных сценариев использования интерфейсных функций (например, инициализация и очищение ресурсов до или после основных вызовов интерфейсных функций). Это позволяет эффективно увеличивать покрытие кода и снизить количество некорректных сценариев.

Предлагаемый метод был реализован в рамках инструмента *ISP-Fuzzer* [3] [27] и протестирован на библиотеке *PluginBase*, входящей в состав платформы *SmartThings* [251] от компании "*Samsung Electronics Co. Ltd*" [252]. В результате экспериментального запуска разработанный инструмент смог найти 15 уникальных дефектов, приводящих к отказу системы, что демонстрирует эффективность предлагаемого решения. Найденные результаты были подтверждены разработчиками компании "*Samsung Electronics Co. Ltd*".

5.2.1. Высокоуровневое описание метода

На рисунке 36 приводится описание общей схемы предлагаемого метода. Серверная часть инструмента производит генерацию данных (последовательность вызовов интерфейсных функций с соответствующими аргументами), которые будут выполнены на клиентской стороне. Серверная часть может одновременно работать с множеством клиентских приложений и синхронизовать полученные результаты, что позволяет многократно увеличить эффективность фаззинга.

На основе присвоенных функциям и классам аннотаций строятся последовательности (цепочки) вызовов интерфейсных функций. Разработан специальный протокол передачи данных между сервером и клиентами, который позволяет отправлять сгенерированные последовательности вызовов интерфейсных функций и получать соответствующие покрытия кода. В зависимости от того, какое покрытие обеспечивает сгенерированная последовательность вызовов, генетический алгоритм подбирает следующую последовательность вызовов и набор аргументов. Последовательности вызовов, которые привели к аварийному завершению или зависанию программы, сохраняются для последующего анализа экспертами.

С помощью конфигурационного файла можно указать шаблоны генерации новых последовательностей вызовов. Например, можно указать интерфейсные функции, которые будут вызваны перед и после каждой сгенерированной последовательностью вызовов для реализации специфического протокола использования программного интерфейса.

Рисунок 36. Общая схема предложенной платформы.

5.2.2. Фаззинг интерфейсных функций библиотеки PluginBase

На рисунке 37 приводится схема фаззинга интерфейсных функций из библиотеки *PluginBase*, входящей в состав *SmartThings*. *SmartThings* – платформа,

связывающая разные устройства интернета вещей, она осуществляет управление ими. Управление устройствами производится через интерфейсные функции, входящие в библиотеку *PluginBase*. Для выполнения фаззинга было разработано два инструмента. Первый инструмент генерирует аннотации для классов и методов входящий в *JAR* библиотеку. На основе этих аннотаций генерируются последовательности вызовов интерфейсных функций для исполнения на клиентской машине. Если функция получает как аргумент произвольный объект, производится генерация кода, создающего этот объект.

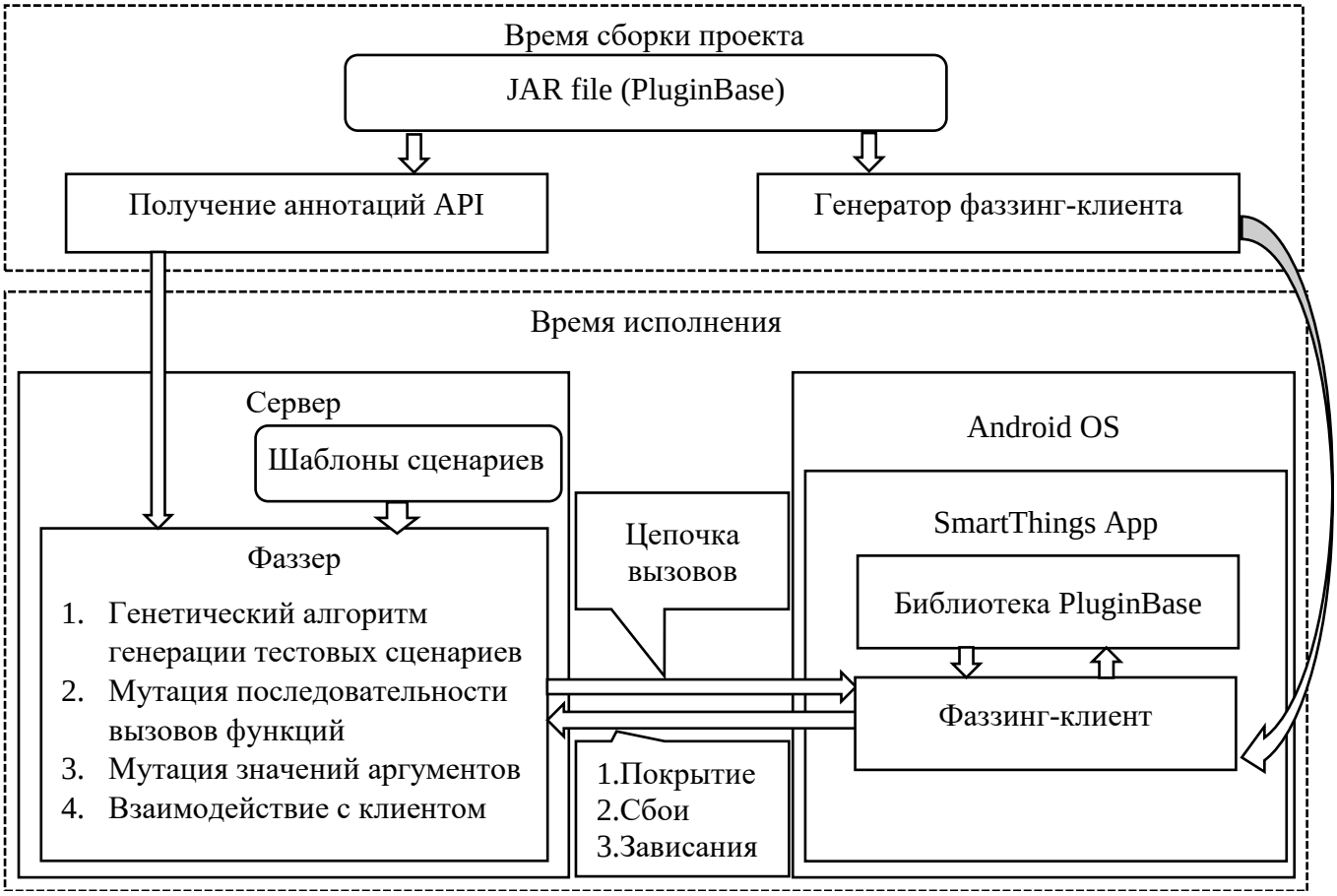

Рисунок 37. Схема фаззинга PluginBase/SmartThings.

Построение сложных объектов производится следующим образом:

1. Если есть интерфейсная функция, которая возвращает объект данного типа, производится вызов этой функции;

2. В противном случае, вызывается конструктор данного объекта. Если конструктор принимает, как аргумент, сложный объект, процесс повторяется рекурсивно, пока все аргументы создаваемого объекта не будут иметь примитивный тип или тип, сконструированный из примитивных типов.

Второй инструмент генерирует встраиваемый модуль (плагин) для приложений *SmartThings*. Плагин получает последовательность вызовов интерфейсных функций от сервера и выполняет их на клиенте (мобильный телефон). Плагин представляет собой конечный автомат, поддерживающий протокол общения с сервером (получает последовательность команд, отправляет покрытие кода и информацию о падениях/зависаниях приложения). Дополнительно инструментируется код самой библиотеки *PluginBase* для получения покрытия кода в процессе исполнения цепочек вызовов.

5.2.3. Генерация вызовов интерфейсных функций

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

На втором этапе (рисунок 38) 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. Производится мутация всех аргументов функций из цепочки, пока покрытие кода не перестанет увеличиваться. В этом случае, инструмент переходит к мутациям самой цепочки вызовов функций. В процессе мутации могут быть добавлены или удалены некоторые вызовы. Если сгенерированная цепочка вызовов увеличивает покрытые кода, то она сохраняется для дальнейшей обработки. Чем больше увеличивается покрытие кода, тем дольше будет обрабатываться соответствующая цепочка вызовов и аргументы функций. Наблюдаемый прирост покрытия на втором этапе, как правило, составляет еще 30-40%.

Рисунок 38. Схема основного цикла фаззинга на серверной машине.

Пользователь может управлять процессом мутации, задав в шаблонном файле следующие параметры:

1. Последовательности вызовов функций, которые сначала должны быть обработаны;
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 в другой.

После того, как цепочка вызовов функций и соответствующие аргументы сгенерированы, они передаются следующему компоненту (генератор последовательности команд выполнения – ГПКВ). ГПКВ создает соответствующие последовательности команд, выполнение которых на клиентской стороне обеспечит вызовы интерфейсных функций с соответствующими аргументами.

5.2.4. Мутация данных

Модуль мутации данных (рисунок 38) получает на вход начальные значения (могут быть пустыми) и длину требуемой последовательности данных. Возвращает мутированные/сгенерированные данные заданной длины. В инструменте поддерживается следующий набор мутаций данных:

1. Расширение данных: с одинаковым произвольным значением, произвольными значениями, с нулевыми байтами;
2. Сокращение данных путем удаления произвольного количества байтов;
3. Модификация данных: присвоить последовательности байтов произвольные значения, присвоить последовательности байтов одно и то же произвольное значение, присвоить произвольному интервалу байтов нулевое/произвольное значение;

4. Инверсия произвольного бита;
5. Генерация строковых данных: повторение строки, создание не валидного списка *UTF-8* строк, создание *BOM* (*byte order mark U+FEFF*);
6. Модификация строковых данных: перевод символов в нижний/верхний регистр.
7. Модификация числовых значений: присваивание максимального/минимального значения, применение арифметических операций.

5.2.5. Мутация цепочек вызовов интерфейсных функций

В ходе мутаций цепочек вызовов используется информация о зависимостях интерфейсных функций (автоматически восстановленная или указанная пользователем). Цель данного действия – получить семантически "*связанные*" программы, где возвращаемое значение одной из функций используется в другой. При создании новой цепочки вызовов произвольным образом выбирается некая функция "*f*" из списка доступных интерфейсных функций. Если существует функция, возвращаемое значение которой может быть использовано как аргумент функции "*f*", тогда перед вызовом "*f*" добавляется ее вызов. Далее, возвращаемое значение добавленной функций передается в качестве аргумента вызова "*f*". Процесс повторяется рекурсивно для добавленных функций, пока длина цепочки не превысит заданную границу.

В случае мутаций цепочки производится добавление/удаление некоторой последовательности вызовов. Максимальная длина таких последовательностей задается пользователем. На мутацию цепочек вызовов можно повлиять с помощью шаблонов конфигурационного файла. Предоставляется возможность:

1. Определить набор функций, которые будут вызваны перед/после каждой цепочки;
2. Определить конкретные значения или интервалы значений для аргументов функций;
3. Составить список функций, которые могут или не могут присутствовать в цепочках вызовов;
4. Задавать зависимости между функциями.

5.2.6. Генерация последовательности команд выполнения

Генератор последовательности команд производит две основные операции:

1. Вычисляет сколько данных потребуется для данной цепочки вызовов, включая все аргументы функций;
2. Генерирует последовательность байтов, которая будет выполнена/интерпретирована на клиентской машине.

Для выполнения пункта 1 инструменту необходима цепочка вызовов и аннотаций интерфейсных функций и классов. Для выполнения пункта 2 инструменту также необходимо будет получить конкретные данные для аргументов функций (данные от модуля мутации данных).

Вычисление размера необходимых данных (пункт 1) производится рекурсивным обходом составных типов сложного объекта. Размер данных вычисляется суммированием размера составных типов. На этом этапе собирается дополнительная информация о возможных значениях конкретных аргументов. Есть три основных класса: *MustBeNull* (всегда должен иметь значение *null*); *NotNull* (никогда не должен иметь значение null); *Any* (может иметь любое значение). Эта информация передается модулю мутаций для генерации корректных данных. Это обеспечивает семантическую корректность вызовов функций. Например, некоторые примитивные типы в *Java* не могут иметь значение *null*. Если есть функция с

аргументом такого типа, то вызов этой функций (вызов производит клиент фаззинга – интерпретатор последовательности команд) с *null*-аргументом приведет к некорректному поведению программы. А, на самом деле, такая ситуация не может возникнуть, поскольку компилятор *Java* не позволит компиляцию такого кода. Генерация последовательности команд выполнения производится на основе простейшей регистровой машины. Каждая операция (положить значение в переменную, вызов функций и т.д.) кодируется в протоколе (в таблице 28 приведены коды). Алгоритм проходит по цепочке вызовов функций и получает последовательность команд для регистровой машины.

Таблица 28. Коды операций 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

5.2.7. Интерпретатор команд

Интерпретатор команд 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 машины). В таблице 28 приводятся доступные коды операций. Каждый метод и класс (также поля класса) имеет уникальный идентификатор (*ID*), который позволяет определить нужный метод, класс и поля класса. На рисунке 39 приводится фрагмент кода и соответствующая последовательность команд регистровой машины.

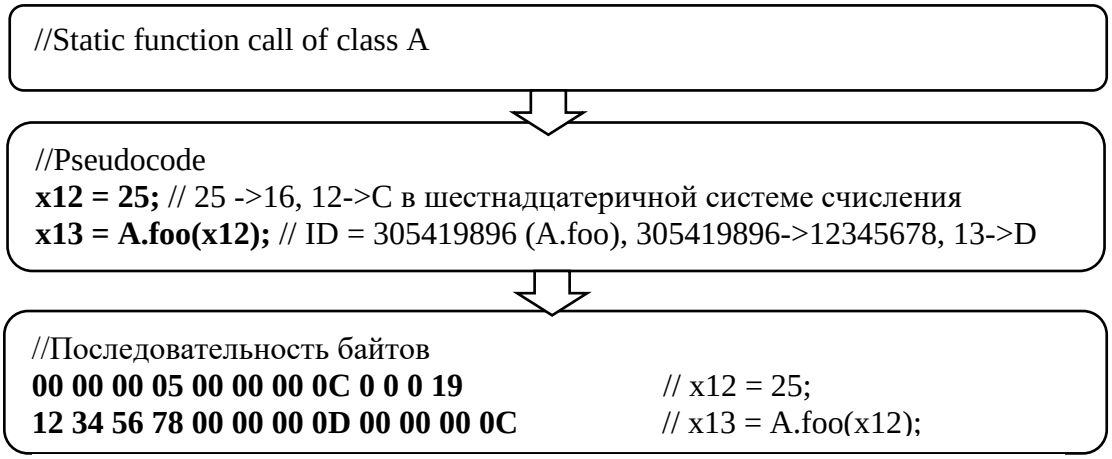

Рисунок 39. Пример последовательности команд выполнения.

Разработан отдельный инструмент, который из аннотаций *JAR* файла генерирует интерпретатор. Это позволяет при обновлении библиотеки автоматически генерировать новый интерпретатор, поддерживающий вызовы добавленных/модифицированных функций.

5.2.8. Получение покрытия кода

Для получения покрытия кода используется техника, реализованная в инструменте *AFL* [253]:

1. Каждому базовому блоку целевой библиотеки сопоставляется целое число из интервала [0, 65536). Для сбора покрытия используются две глобальные переменные: карта памяти (*map*) в размере 65536 элементов и переменная *prevLocation*. Обе переменные инициализированы нулями.
2. При выполнении базового блока *X*, которому было присвоено целое число *bbIndexX* на первом шаге, производится следующее изменение в карте памяти:

   ```
   id := bbIndexX ^ prevLocation;
   map[id] := map[id] + 1;
   prevLocation := bbIndexX >> 1;
   ```

   Таким образом, в карте памяти хранится информация о выполненных базовых блоках и переходах.

Такой подход позволяет выявить уникальные пути выполнения (хотя возможны коллизии). Инструментация кода для сбора покрытия выполняется на основе платформы *ASM* [254].

5.2.9. Результаты

В ходе экспериментального запуска на библиотеке *PluginBase*, входящей 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*". На рисунке 40 приводятся некоторые функции, вызовы которых с указанными параметрами приводят к возникновению исключительных ситуаций. Как видно из результатов (рисунок 41) инструмент может найти не только "*простые*" ошибки, когда передаются *null* объекты. Успешно были найдены исключения выхода индекса за пределы строки и другие типы исключений, определенные в самой платформе *SmartThings*.

В заключение можно сказать, что разработанная система показала свою гибкость и эффективность при тестировании такой сложной системы, как платформа интернет вещей *SmartThings*.

```
// com.samsung.android.plugins.PluginDataStorageException
JSONConverter.jsonToRcsRep ("{}");

// java.lang.StringIndexOutOfBoundsException
ShpConverter.vidToShpSSID ("nohyphen");

// java.lang.NullPointerException
QcPluginDevice.getQcPluginDevice ("foo");

// os.android.RemoteException
AutomationManager.getInstance ().getRegisteredAutomation ("1");
```

Рисунок 40. Найденные необработанные исключительные ситуации.

```
package com.samsung.android.plugin.tv;
import android.os.Bundle;
import android.view.Window;
import android.view.WindowManager;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.InetSocketAddress;

import com.samsung.android.oneconnect.utils.ShpConverter;
import com.samsung.android.plugins.QcPluginDevice;
import com.samsung.android.plugins.automation.AutomationManager;
import com.samsung.android.scclient.JSONConverter;

public class MainActivity extends PluginBaseActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      final Window window = getWindow();
      window.setSoftInputMode(
         WindowManager.LayoutParams.SOFT_INPUT_ADJUST_RESIZE |
            WindowManager.LayoutParams.SOFT_INPUT_STATE_HIDDEN);
      getTheme().applyStyle(R.style.AppTheme, true);
      setContentView(R.layout.activity_main);
      System.out.println(" JSONConverter.jsonToRcsRep");
      try {
         JSONConverter.jsonToRcsRep("{}");
      } catch (Throwable e) {
         e.printStackTrace();
      }

      System.out.println(" ShpConverter.vidToShpSSID");
      try {
         ShpConverter.vidToShpSSID("nohyphen");
      } catch (Throwable e) {
         e.printStackTrace();
      }
      System.out.println(" QcPluginDevice.getQcPluginDevice");
      try {
         QcPluginDevice.getQcPluginDevice("foo");
      } catch (Throwable e) {
         e.printStackTrace();
      }
      System.out.println(" AutomationManager.getRegisteredAutomation");
      try {
         AutomationManager mgr = AutomationManager.getInstance();
         mgr.getRegisteredAutomation("1");
      } catch (Throwable e) {
         e.printStackTrace();
      }
   }
}
```

Рисунок 41. Фрагмент кода демонстрирующий найденные исключения.

5.3. Направленный фаззинг путем динамической инструментации

В данном разделе 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. Вначале для достижения цели статическим анализом выявляются все пути в программе, которые соединяют точку входа программы с рассматриваемыми инструкциями. Затем применяются 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 неинструментированную программу на найденных входных данных и

верифицировать результаты. На рисунке 42 приводится общая архитектура предлагаемого решения.

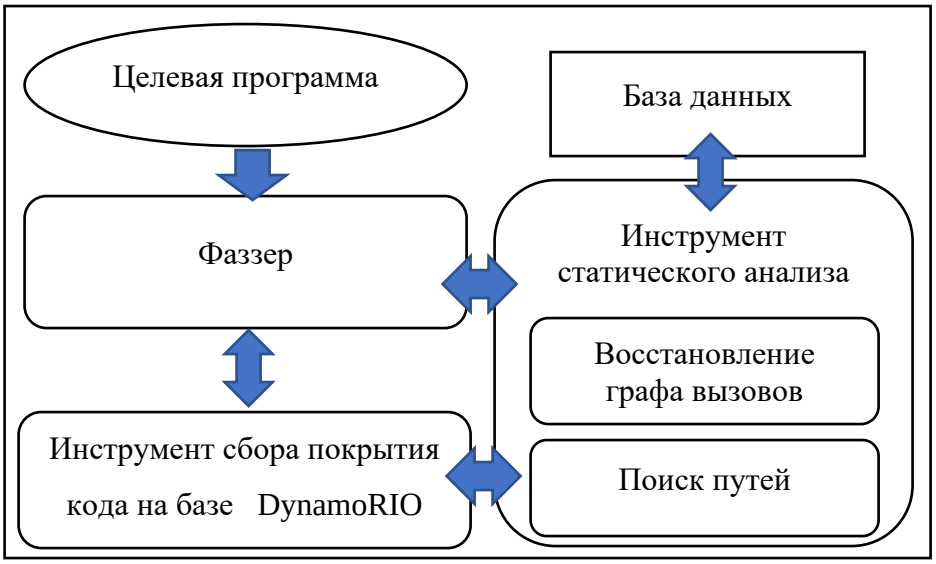

Рисунок 42. Архитектура направленного фаззинга.

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

5.3.1. Построение путей статическим анализом

Целью данного этапа является построение всех путей из входной точки программы к целевым точкам, содержащим потенциальные уязвимости/ошибки. Как результат (рисунок 43), инструмент возвращает два списка, первый – это набор всех

базовых блоков, принадлежащий к найденным путям ("*белый*" список), второй – это набор всех остальных базовых блоков ("*черный*" список). Инструмент построения путей состоит из трех основных этапов.

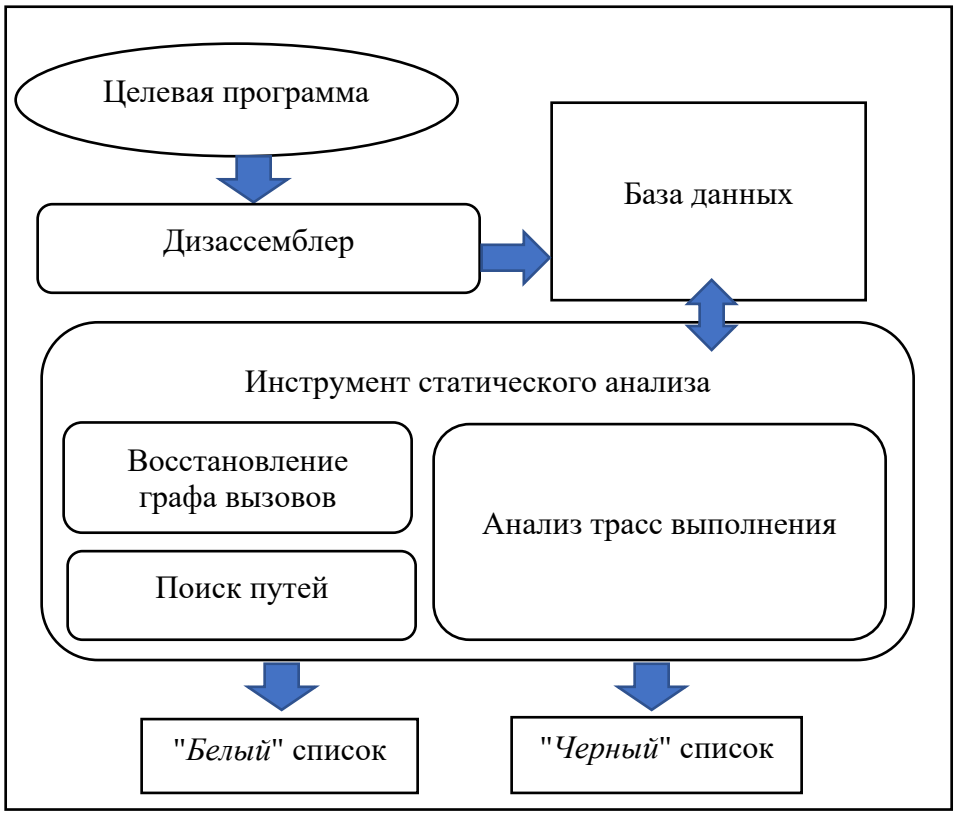

Рисунок 43. Схема статического анализа.

На первом этапе производится фильтрация некоторых функций на основе графов вызовов. Из функций, содержащих входную точку программы (обычно это функция "*main*"), производится прямой обход в ширину по графу вызовов, и строится множество доступных функций. Из функций, содержащих целевую инструкцию, производится обратный обход в ширину по графу вызовов, и строится множество доступных функций. Далее, производится пересечение полученных множеств и поиск путей выполняется только в рамках этих функций.

Второй этап состоит из двух шагов. Сначала используется модифицированный алгоритм поиска в глубину, на графе вызовов, для поиска всех путей из входной

функции (функция, содержащая входную инструкцию) к целевой (функция, содержащая потенциально уязвимую точку). Далее, производится поиск путей внутри каждой из этих функций. Модифицированный алгоритм поиска в глубину позволяет указать два ограничения: максимальная длина пути и максимальное количество использования одного базового блока или функций во время поиска. Эти ограничения необходимы для избежания случаев экспоненциального роста количества путей.

На последнем этапе происходит дополнительный анализ базовых блоков кода, которые не попали в построенные пути, но могут иметь влияние на их выполнение. Такие случаи могут возникнуть из-за использования глобальных переменных и передачи аргументов через указатели и ссылки (смотри пример на рисунке 44). Без этого этапа базовые блоки функций "*g*" и "*k*" не окажутся в "*белом*" списке (а на самом деле должны были быть).

```
……
void k (int &a) {
   a = 1;
}
void g (int &a) {
   k(a);
}
int main () { // Входная точка
   int a = 0;
   g(k);
   if (k > 0) {
     // Целевая точка
   }
}
……
// Граф вызовов: main()-->g()-->k()
```

Рисунок 44. Пример, где статический анализ должен учесть вызовы функций.

5.3.2. Восстановление неявных переходов в графе вызовов

Для восстановления значения регистровых операндов (адреса функций) в инструкциях вызовов реализована поддержка трассировки на базе *DynamoRIO*. Ребра графа вызовов восстанавливаются дополнительным статическим анализом полученных трасс выполнения. В трассе для каждого выполненного блока также сохраняется его предыдущий блок. Статический анализ рассматривает базовые блоки трасс и, если существует блок, принадлежащий какой-то функции "*f*", а ранее выполненный блок принадлежит какой-то функции "*g*", то в графе вызовов есть ребро между этими функциями. Обнаруженные новые ребра добавляются в базу данных анализируемой программы, что позволяет итеративно улучшать результаты статического поиска путей. Это, в свою очередь, улучшает результаты фаззинга. В таблице 29 приводится статистика найденных неявных переходов.

Таблица 29. Количество найденных неявных переходов.

Имя теста	**Начальное количество ребер в графе вызовов**	**Количество ребер в графе вызовов после восстановления**
mutool-1.12.0	35711	35944
jasper-1.900.0	2717	2871
faad2-2.7	2015	2087
ffmpeg-4.1	73881	74468
strings-2.26.1	166	184

5.3.3. Результаты

В таблице 30 приводится сравнение разработанного инструмента с оригинальной версией *ISP-Fuzzer* (оригинальная версия *ISP-Fuzzer* базируется на *AFL*). В качестве тестового набора используются уязвимые примеры из проекта *DARPA* [255]. Разработанный инструмент запущен в направленном режиме с адресами

дефектов для всех целевых программ. Количество итераций фаззинга одинаково во всех случаев. В таблице 30 приводятся результаты после десяти часов фаззинга.

Таблица 30. Сравнение оригинального 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	-	-	+

Как становится очевидным из результатов, оба предложенных метода динамической инструментации позволяют найти больше ошибок в тестируемых программах. Первый метод по результатам превосходит второй, несмотря на то, что второй метод ускоряет скорость фаззинга. Основная причина этого – то, что второй метод добавляет инструкции "*exit (0)*" в некорректных местах, и программа завершается раньше. Эту проблему можно частично обойти путем улучшения инструмента статического анализа. Другим интересным фактом является то, что второй метод умеет находить такие ошибки, который первый метод не успел найти в

течение заданного времени фаззинга. Причина этого в том, что второй метод в разы может ускорить фаззинг, производя большее количество запусков целевой программы и мутаций входных данных.

5.4. Интеграция динамического символьного выполнения и статического анализа с фаззингом

В этом разделе 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.

5.4.1. Схема разработанного инструмента

Разработанный инструмент состоит из четырех основных компонентов (рисунок 45). Первый компонент – это инструмент фаззинга, который предоставляет набор мутаций и базовую инфраструктуру. Второй компонент – это инструмент на

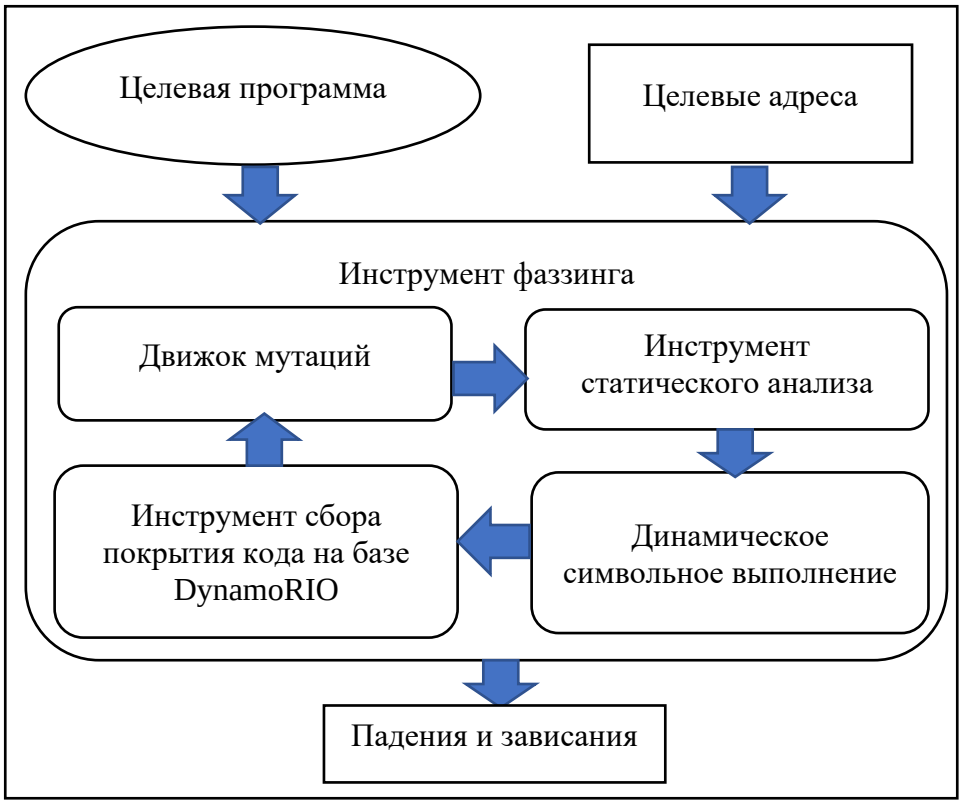

Рисунок 45. Схема фаззинга.

основе *DynamoRIO* для сбора покрытия кода. Третьим компонентом является инструмент динамического символьного исполнения *Anxiety* [256]. Четвертым компонентом является инструмент статического анализа бинарного кода программы. В нем используется база данных для хранения информации о целевой программе, в частности, информация о базовых блоках.

В случае анализа целевых путей инструмент периодически пытается генерировать входные данные, которые приведут к выполнению указанных пользователем адресов целевой программы. Для этого, сначала применяется

классический фаззинг до тех пор, 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 движку фаззинга. Если это приводит 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, что используется для восстановления неявных переходов. Во время фаззинга периодически вызывается статический анализ, и база данных целевой программы обновляется на основе восстановленных адресов неявных вызовов. Это обеспечивает взаимное улучшение статического и динамического анализа. Более подробное описание статического анализа приводится в разделе 5.3.1.

5.4.2. Результаты

В таблице 31 приводятся результаты фаззинга, где основная цель – увеличение покрытия кода. Все результаты вручную верифицированы. Тестирование проводится

на разных пакетах ОС *Debian*. Время фаззинга – 10 часов. Выбранные пакеты содержат копии известных уязвимостей. Выбор этих пакетов производился с помощью разработанной нами платформы для анализа проектов (см. раздел 6.6.2).

Таблица 31. Результаты интеграции символьного выполнения и статического анализа с фаззингом.

ОС пакета	***Имя пакета***	***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 таблице 32 приводятся результаты фаззинга в направленном режиме. Статическим анализом (см. раздел 6.6.2) определен набор адресов, которые могут содержать ошибку. Адреса переданы инструменту фаззинга с надеждой, что будут получены данные, приводящие к падению. Время фаззинга – 20 часов.

Таблица 32. Результаты фаззинга в направленном режиме.

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

В обоих случаях оригинальная версия *ISP-Fuzzer* базированная на *AFL*, не находит ни одной ошибки.

5.5. Повышение эффективности фаззинга интерфейсных функций на основе извлеченных константных значений

В этом разделе 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. Далее, полученная информация используется для оптимизации процесса фаззинга*. Как результат, статический анализ выдает 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* [194]. 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.

5.5.1. Статический анализ для получения смешений и соответствующих константных значений

В процессе сборки целевого проекта используется *gllvm* [257] для извлечения промежуточного представления *LLVM*. На основе полученного представления производится статический анализ. Для этого используются две основные структуры данных. Первая структура данных – это граф вызовов, который используется для обнаружения всех путей, начинающихся с функции *LLVMFuzzerTestOneInput* (это входная точка фаззинга в инструменте *libFuzzer*). Для каждого обнаруженного пути анализируются соответствующие функции. Для анализа этих функций используется вторая структура данных, которая представляет собой граф потока данных, где ребра маркированы со смещениями доступа (рисунок 46). Поскольку "*Data+8*" передается в функцию "*parser1*", соответствующее ребро помечается смещением "*8*". Метка "*"

означает, что операция загрузки памяти выполняется с заданного адреса. Метка "*fake_dd*" специально добавлена для обеспечения прямого потока данных.

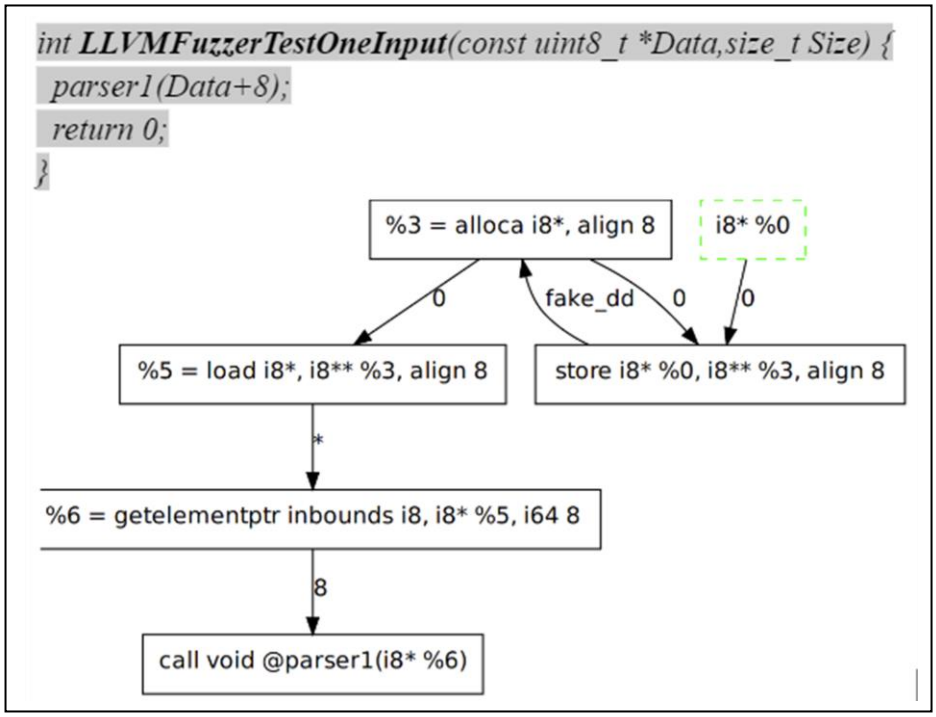

Рисунок 46. Пример графа потока управления со смещениями на ребрах.

5.5.1.1. Поиск путей на графе вызовов

Обнаружение всех путей на графе вызовов выполняется в два этапа. На первом этапе используется стандартный поиск в ширину для удаления циклов. На втором – обнаруживаются все пути от функций *LLVMFuzzerTestOneInput* к любой другой функции, не имеющей исходящих ребер.

Возможны случаи, когда граф вызовов достаточно сложный (много вызовов между функциями), что может привести к экспоненциальному росту путей. Для решения данной проблемы, в ходе построения, вводятся ограничения на рассматриваемое количество незавершенных путей. Если алгоритм переходит ограничение, используется поиск в глубину для нахождения одного пути.

5.5.1.2. Извлечение константных значений и соответствующих смещений

Для извлечения константных значений и соответствующих смещений входного буфера необходимо учитывать два факта:

1. Входной буфер, полученный от функции *LLVMFuzzerTestOneInput*, может быть передан последующим функциям с некоторым смещением или частично. Для решения этой проблемы при анализе путей графа вызовов необходимо рассматривать смещения потока данных от аргумента к вызываемой функции. Например, на рисунке 46 видно, что первый аргумент функции *LLVMFuzzerTestOneInput* передается в функцию "*parser1*" со смещением "*8*". Это означает, что, если в функции "*parser1*" будет извлечено константное значение с некоторым смещением *X*, необходимо к его смещению добавить "*8*", чтобы иметь правильное значение.
2. Для извлечения константных значений и их смещений необходимо рассматривать все пути от аргументов функции до любой инструкции ("*cmp*", "*switch*"), выполняющей сравнение с константным операндом. Затем, для каждого пути вычислить его смещение и извлечь фактическое значение константы, с которым производится сравнивание. На этом этапе также может произойти экспоненциальный рост путей, что решается аналогично разделу 5.5.1.1.

5.5.2. Аннотаций системных и библиотечных функций

Вызовы некоторых системных/библиотечных функций могут прервать поток данных. Примером может служить вызов системной функции "*memcpy*". Бывают случаи, когда целевой буфер копируется в новый, и затем смещения этого нового буфера сравниваются с константами. Для поддержки таких случаев был добавлен специальный файл, описывающий поведение таких функций (рисунок 47). На основе

информации из этого файла в граф потока данных добавляются дополнительные ребра для обеспечения непрерывности потока данных. Информация этого файла обозначает, что вызов функции "*memcpy*" обеспечивает прямой поток данных от второго аргумента к первому. Вызов "*memcpy_s*" обеспечивает прямой поток данных от третьего аргумента к первому.

```
[
        {
          "function_name": "memcpy",
          "src_arg_no": 1,
          "target_arg_no": 0
        },
        {
          "function_name": "memcpy_s",
          "src_arg_no": 2,
          "target_arg_no": 0
        }
]
```

Рисунок 47. Пример аннотаций системных функций.

5.5.3. Модификация libFuzzer

Было произведено изменение стратегии мутации *libFuzzer* для использования извлеченных константных значений. Если в ходе *X=10000* (определено экспериментально, может быть изменено пользователем) итераций не обнаружены новые пути, включается добавленная мутация *ConstantWarmUp*. Данная мутация применяет константные значения на входном буфере. Это производится двумя этапами.

На первом этапе перебираются все ранее найденные эффективные тестовые случаи и, по очереди, применяются на них константные значения. Если в результате удается покрыть новый путь, соответствующие входные данные и константное значение с его смещением сохраняются в специальную карту "*effectives*".

На втором этапе, когда все константные значения уже применены по отдельности, алгоритм перебирает все возможные подмножества констант для каждого теста из "*effectives*" и применяет их снова. Формальное описание алгоритма приводится ниже:

Алгоритм мутаций ConstantWarmUp

```
1.  constantWarmUp(ConstatnsInformation, RecentNewTestCases) {
2.   effectives = []
3.   for (test in RecentNewTestCases) {
4.    for (constant_info in ConstatnsInformation) {
5.     new_test = ApplyNormal(test, constant_info)
6.     if (run_target(new_test) is PROVIDES_NEW_COVERAGE) {
7.      effectives[test].push_back(constant_info)
8.     }
9.     new_test = ApplyIncrement(test, constant_info)
10.    if (run_ target(new_test) is PROVIDES_NEW_COVERAGE) {
11.     effectives[test].push_back(constant_info)
12.    }
13.    new_test = ApplyDecrement(test, constant_info)
14.    if (run_ target(new_test) is PROVIDES_NEW_COVERAGE) {
15.     effectives[test].push_back(constant_info)
16.    }
17.   }
18.  }
19.  SetOfSubsets = []
20.  for (test, constants in effectives) {
21.   SetOfSubsets[test] = make_subsets(constants)
22.  }
23. }
24. for (test, subsets in SetOfSubsets) {
25.  for (subset in subsets) {
26.   new_test = ApplySubset(test, subset)
27.   run_ target(new_test)
28.  }
29. }
```

Функция *ApplyNormal* применяет константу с соответствующим смещением на входные данные (*test*). Функция *ApplyIncrement* увеличивает константу на единицу, после чего применяет с соответствующим смещением на входные данные. Функция *ApplyDecrement* уменьшает константу на единицу, после чего применяет с

соответствующим смещением на входные данные. Функция *make_subsets* создает множество всех подмножеств заданных констант. Функция *ApplySubset* применяет множество констант с соответствующими смещениями на входные данные. Функция *run_target* запускает целевую программу на входных данных и считает покрытие кода.

5.5.4. Экспериментальные результаты

Разработанный инструмент был протестирован на ряде проектов, интегрированных в платформе *OSS-Fuzz*. В таблице 33 приводится количество полученных константных значений статическим анализом.

Таблица 33. Константы, полученные статическим анализом.

Имя теста	**Количество полученных констант**
block_round_trip (zstd)	74
decompress_dstSize_tooSmall (zstd)	76
dicionary_decompress (zstd)	344
dictionary_loader (zstd)	66
dictionary_round_trip (zstd)	164
dictionary_stream_round_trip (zstd)	97
huf_decompress (zstd)	109
raw_dictionary_round_trip (zstd)	89
simple_compress (zstd)	76
simple_decompress (zstd)	344
simple_round_trip (zstd)	164
zstd_frame_info (zstd)	70
fuzz_hpack_decode (haproxy)	125
ap-mgmt (hostap)	245
json (hostap)	64
x509 (hostap)	122
fuzz_parser (http-parser)	218
fuzz_url (http-parser)	30

В таблице 34 приводится сравнение результатов разработанного метода фаззинга с оригиналом. Во всех случаях, кроме "*dictionary_loader*", разработанный метод показывает лучшие результаты. Из таблицы также становится ясно, что эффективность инструмента гораздо больше при малых итерациях (10k.). Это дает большое преимущество при регулярном анализе больших проектов, где стоят временные ограничения.

Таблица 34. 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%***

5.6. Заключение

В данной главе приводится описание разработанных методов фаззинга для разных задач:

- Метод генерации входных данных на базе БНФ-грамматик что доказал свою эффективность, обеспечив большее покрытие кода по сравнению с классическими подходами фаззинга. Во время фаззинга *gcc-12* и *clang-14* (на момент тестирования последние версии) удалось найти входные данные, приводящие к падению обоих компиляторов.
- Метод фаззинга интерфейсных функций, который доказал свою эффективность, найдя 15 уникальных аварийных завершений программы в библиотеке *PluginBase*, входящей в состав платформы *SmartThings* (версия 1.7.9). Результаты были подтверждены командой тестирования "*Samsung Electronics Co. Ltd*".
- Метод направленного фаззинга, который на примерах из проекта *DARPA* смог найти большее количество падений по сравнению с классическими методами фаззинга.
- Метод интеграции динамического символьного выполнения и статического анализа с фаззингом позволил найти десятки падений в пакетах ОС *Debian*, включая *faad*, *wavpack*, *password* и т.д.
- Метод, который сначала применяет статический анализ, определяющий все смещения входного буфера для интерфейсных функций, с которыми производится сравнение константных значений. Далее, эта информация используется в процессе фаззинга для улучшения эффективности сгенерированных входных данных. При экспериментальном тестировании стало очевидно, что предложенный метод позволяет при малых итерациях (до 10,000) фаззинга увеличить покрытие кода более, чем в три раза.

Таким образом, разработанные новые методы позволили улучшить эффективность фаззинга по сравнению с существующими подходами, что позволило обеспечить большое покрытие кода и найти десятки ошибок в широко используемых проектах.

6. Описание предлагаемой платформы

В данной 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.

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. 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 разными 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. Например, после того, как статический анализ нашел потенциальные точки ошибок, можно запустить фаззинг в направленном режиме с целью генерации данных для покрытия найденных точек. Написанные скрипты/плагины возможно запустить в режиме непрерывной интеграции.

6.1. Сбор артефактов

Сбор артефактов производится для дистрибутивов ОС, их пакетов, а также для 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 решения 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*. Бинарные файлы *ARM32/64* создаются на образах *QEMU x86/64* с использованием кросс-компиляции.

6.2. Схема работы предлагаемой платформы

На рисунке 1 приведена схема взаимодействий компонентов. Пользовательский интерфейс компонента сбора информаций предназначен для пополнения базы данных. Компонент анализа проектов предоставляет наборы методов, которые можно запустить из соответствующего пользовательского интерфейса или с помощью доступного *Python*-терминала. Пользователям доступен программный интерфейс в среде *Python*-терминала, который позволяет вызывать каждый компонент (соответствующие API функции) и далее обрабатывать полученные результаты или передавать их другим компонентам. *Python*-терминал – полнофункциональная среда *Python* с возможностью прямых запросов к базе данных.

6.2.1. Пользовательский интерфейс пополнения базы данных

Компонент пополнения базы данных содержит следующие функциональные возможности:

1. Добавление нового дистрибутива ОС в базу данных;
2. Добавление всех открыто-доступных пакетов для дистрибутива ОС в базу данных;
3. Добавление отдельного пакета в базу данных;
4. Добавление директории, содержащей бинарные файлы, в базу данных;
5. Добавление зависимостей между пакетами вручную;
6. Добавление известных уязвимостей и соответствующих исправлений в базу данных;
7. Возможность произвести вышеуказанные действия через веб и командный интерфейс.

6.2.2. Пользовательский интерфейс поиска клонов кода

Компонент поиска клонов кода предоставляет следующие функциональные возможности доступные пользователям:

1. Поиск клонов для заданных исходных файлов;
2. Поиск клонов для заданных бинарных файлов;
3. Поиск клонов для заданных фрагментов (исходных или бинарных);
4. Поиск клонов пакетов по зависимостям (от других пакетов);
5. Возможность задавания степени схожести;
6. Возможность задавания списка пакетов, в которых должен производиться поиск;
7. Возможность ранжирования результатов:
    7.1. степень схожести,
    7.2. частота использования,
    7.3. история копирования.

6.2.3. Пользовательский интерфейс фаззинга бинарных файлов

Компонент фаззинга бинарных файлов предоставляет следующие функциональные возможности, доступные пользователям:

1. Выбор бинарных файлов, на которых должен запускаться фаззинг;
2. Выбор опций/конфигураций запуска;
3. Выбор начальных входных данных;
4. Возможность задавания вычислительных ресурсов, на которых должно производиться тестирование (включая распределенный запуск на нескольких машинах).

6.2.4. Пользовательский интерфейс сопоставления исходных и бинарных файлов

Компонент сопоставления исходных и бинарных файлов предоставляет следующие функциональные возможности, доступные пользователям:

1. Возможность выбора бинарного файла (из базы данных или произвольный бинарный файл);
2. Выбор проектов из базы данных, с которыми должно производиться сопоставление;
3. Возможность задавания минимальной степени схожести, при которой считается, что исходный код сопоставлен с бинарным.

6.2.5. Пользовательский интерфейс поиска неисправленных ошибок

Компонент поиска неисправленных ошибок предоставляет следующие функциональные возможности доступные пользователям:

1. Возможность выбора известных уязвимостей, для которых будет произведен поиск клонов кода;
2. Возможность выбора набора проектов, в которых должен быть произведен поиск клонов известных уязвимостей;
3. Возможность указания уязвимого фрагмента кода, для которого должен быть произведен поиск клонов.

6.2.6. Пользовательский интерфейс поиска информации

Компонент поиска информаций позволяет производить поиск в базе данных по наименованиям проектов и соответствующих им ОС. Для каждого пакета есть возможность:

1. Просмотра файлов;
2. Добавления аннотаций;

3. Получения содержавшихся в нем уязвимостей;
4. Получения команд сборки;
5. Получения бинарных артефактов.

6.2.7. Пользовательский интерфейс поиска утечек памяти

Компонент поиска утечек памяти предоставляет возможность выбора набора проектов, которые должны быть проанализированы.

6.3. Описание структуры базы данных

В данной главе приводится описание используемой базы данных. Для реализации используется мультимодальная система управления базами данных (СУБД), комбинирующая реляционный, документный и графовый подходы хранения данных. Предлагаемая структура базы данных выбрана для: легкой реализаций поиска информации о пакетах и файлах, эффективного взаимодействия различных методов анализа кода.

6.3.1. Описание структуры базы данных для хранения артефактов

На рисунке 48 приводится подробное описание всех документов и связей базы данных.

"**Коллекция дистрибутивов ОС**" представляет из себя набор документов, содержащих информацию о доступных дистрибутивах ОС. Каждый из этих документов является одновременно вершиной направленного графа. Из вершины выходят три ребра, ссылающиеся 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 выходят 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.

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" указывают на соответствующие документы из "Коллекция файлов и директорий исходного кода" (файлы, соответствующие этим документам, одинаковы).

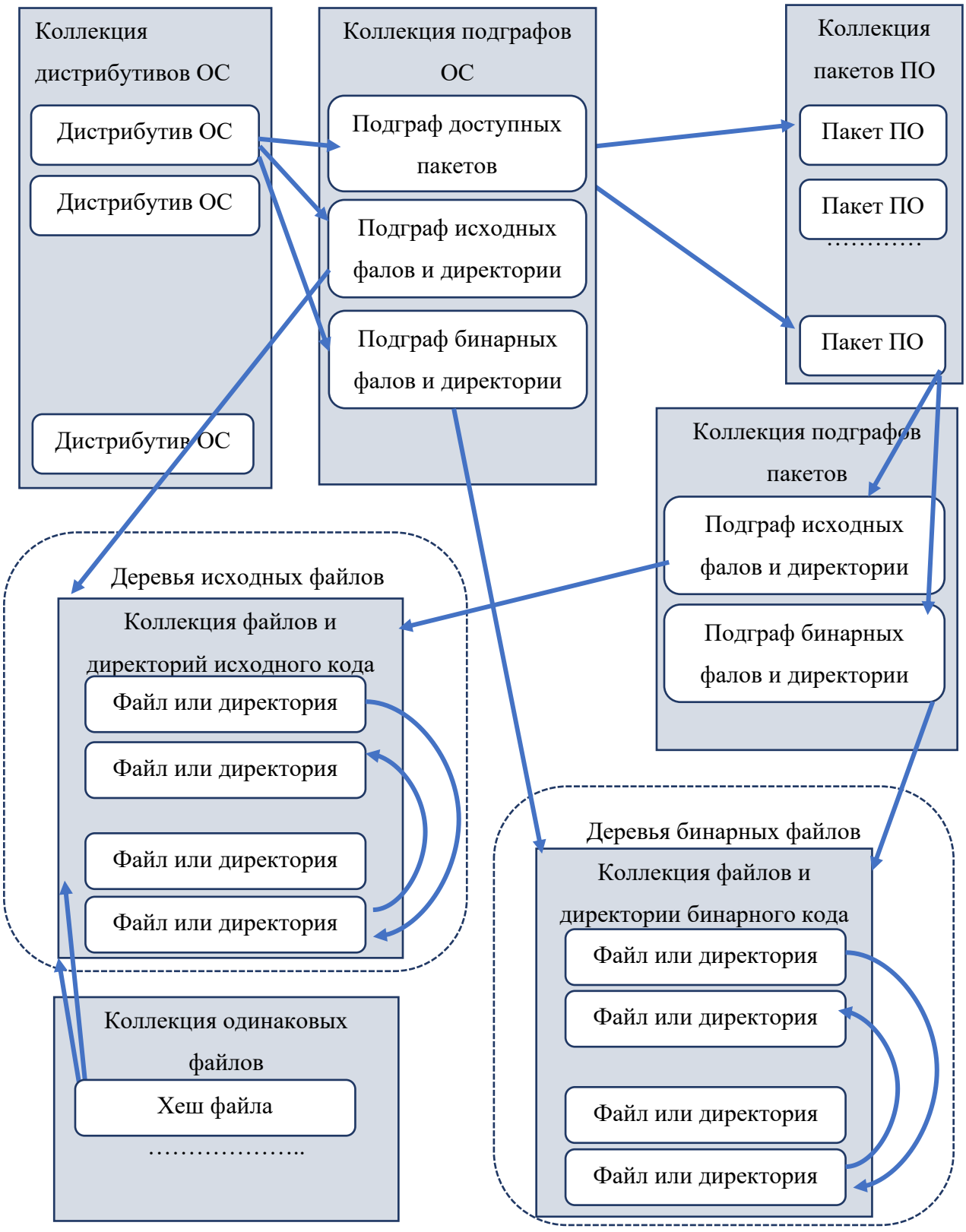

Рисунок 48. Схема базы данных.

6.3.2. Описание структуры базы данных для результатов инструментов

Результаты методов анализа для каждого пользователя также хранятся в базе данных, что позволяет удобно передавать их последующим методам. В этом разделе описывается структура сохраняемых результатов для основных методов анализа.

6.3.2.1. Хранение аннотированных фрагментов кода

Компонент поиска информации позволяет также сохранить аннотации для фрагментов кода. Это – достаточно полезный механизм для сохранения важной информации, которую пользователь может выявить во время анализа. На рисунке 49 приводится схема хранения аннотаций.

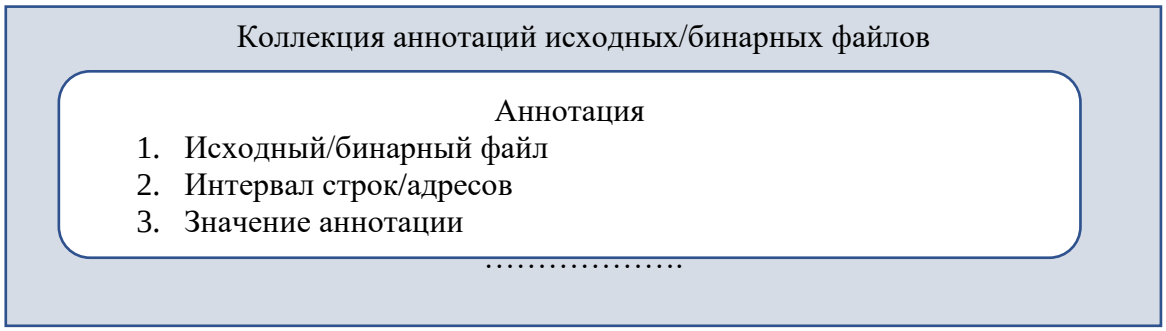

Рисунок 49. Схема хранения аннотаций.

6.3.2.2. Хранение найденных клонов кода

Для сохранения найденных клонов в базу данных используется схема на рисунке 50. Вершины графов (документы из коллекции клонов исходного или бинарного кода) содержат информацию о фрагменте файла, который был найден в качестве клона. Также содержится информация о дистрибутиве ОС или пакете ПО, который содержит клон. Ребра графа в базе данных хранятся в виде документов. В каждом ребре содержится степень схожести для пары фрагментов кода и карта строк или адресов. В карте информация о соответствии строк или адресов первого и второго фрагмента. Для каждого проекта клонов кода имеется отдельная коллекция ребер и вершин.

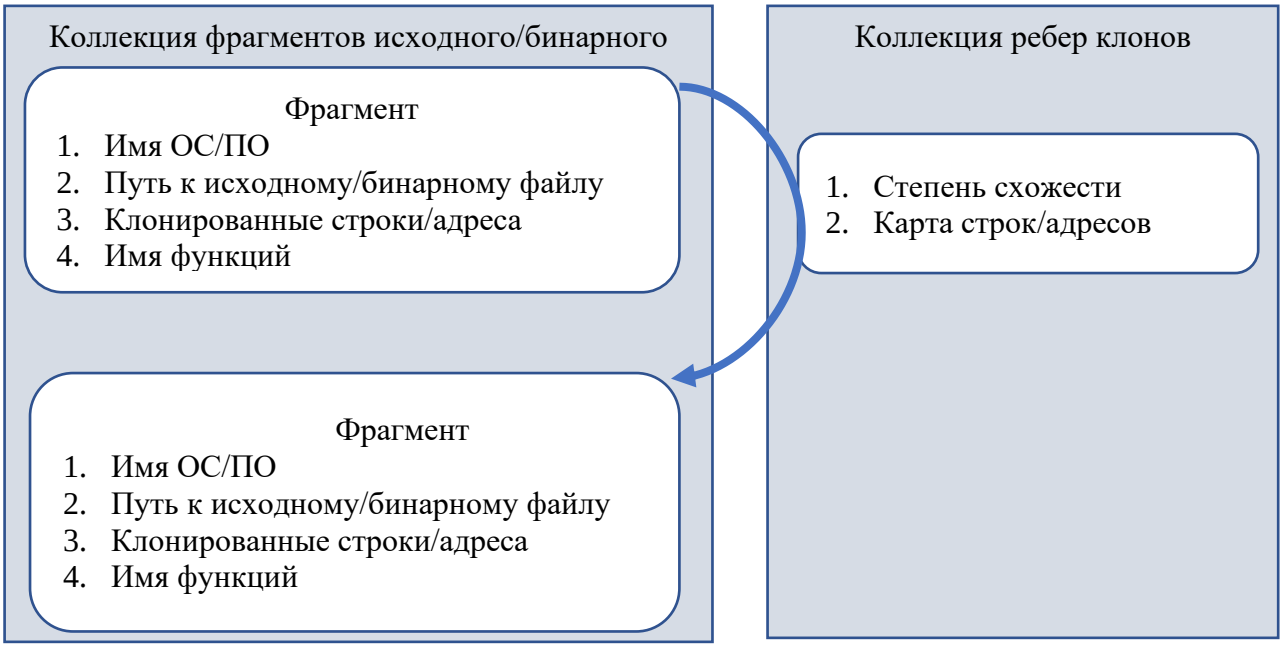

Рисунок 50. Схема хранения клонов кода в базе данных.

6.3.2.3. Хранение результатов сопоставления бинарных и исходных файлов

На рисунке 51 приводится схема хранения найденных сопоставленных функций. Для каждого проекта сопоставления бинарных и исходных файлов создается отдельный документ в коллекции "Коллекция проектов сопоставленных бинарных и исходных файлов". Каждый документ имеет набор исходящих ребер к бинарным функциям (документы коллекции "Коллекция функций бинарных файлов"), которые были сопоставлены. Из каждой бинарной функции идет исходящее ребро к соответствующей функции исходного кода (документы коллекции "Коллекция функций исходных файлов"). Эти ребра содержат карту бинарных адресов и строк исходного кода, которые были сопоставлены. И, в конце, из каждой исходной функции идет исходящее ребро к соответствующему файлу, в котором содержится данная функция. Такая структура позволяет иметь всю информацию о сопоставлении, и удобным образом производить просмотр соответствующих результатов в пользовательском интерфейсе.

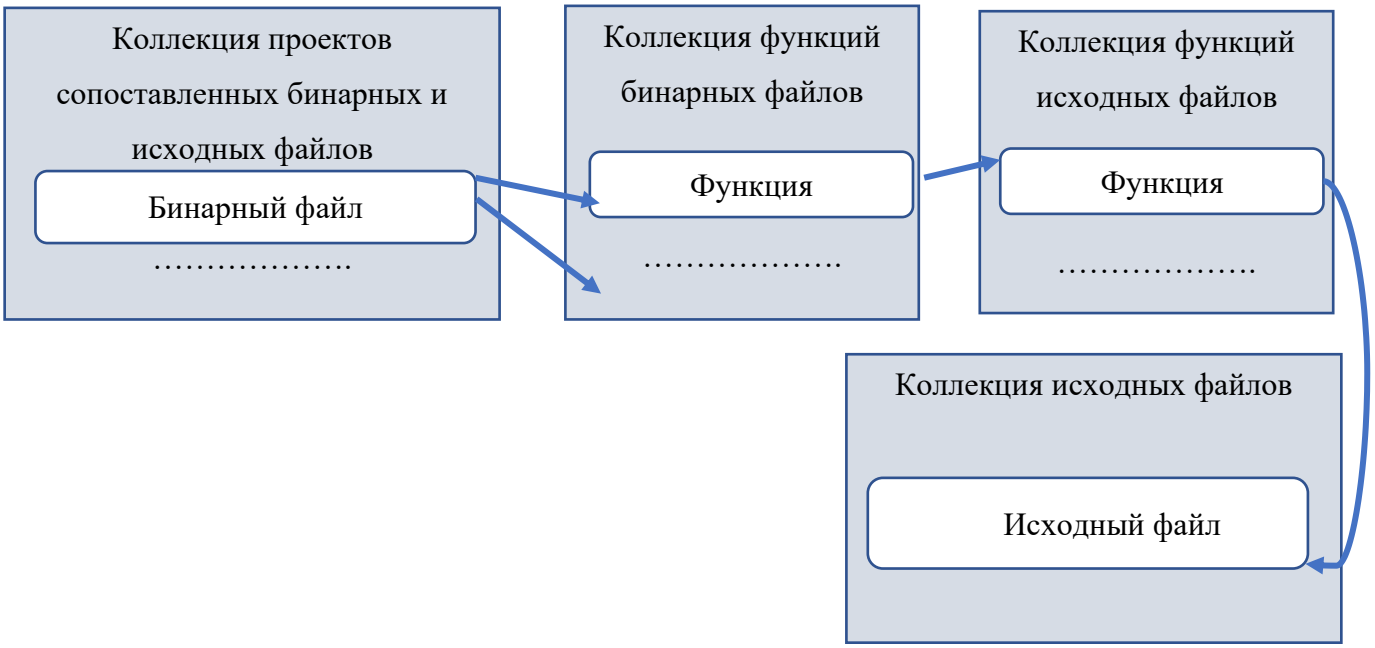

Рисунок 51. Схема хранения сопоставленных исходных и бинарных файлах.

6.3.2.4. Хранение результатов фаззинга бинарных файлов

На рисунке 52 приводится схема хранения результатов инструмента фаззинга бинарных файлов. Каждому документу из коллекции "Коллекция проектов фаззинга" сопоставляется документ из "Коллекция файлов и директории бинарного кода". Из этого, далее, однозначно опеределяется соответствующий проект и дистрибутив ОС.

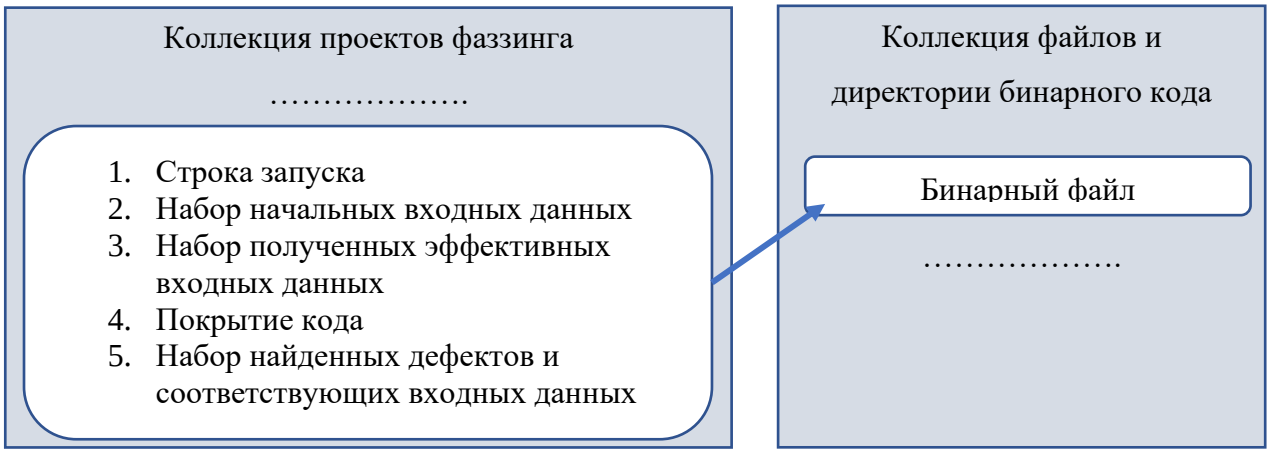

Рисунок 52. Схема хранения результатов инструмента фаззинга.

6.3.2.5. Хранение результатов поиска неисправленных ошибок

На рисунке 53 приводится схема хранения результатов поиска неисправленных ошибок. Каждому документу из коллекции "Коллекция проектов поиска неисправленных ошибок" сопоставляются документы из "Коллекция файлов и директории исходного кода" и "Коллекция известных уязвимостей и их исправлений". Ребра графа, соединяющиеся с коллекцией исходных файлов, отмечаются соответствующими строками, на которых найдена ошибка.

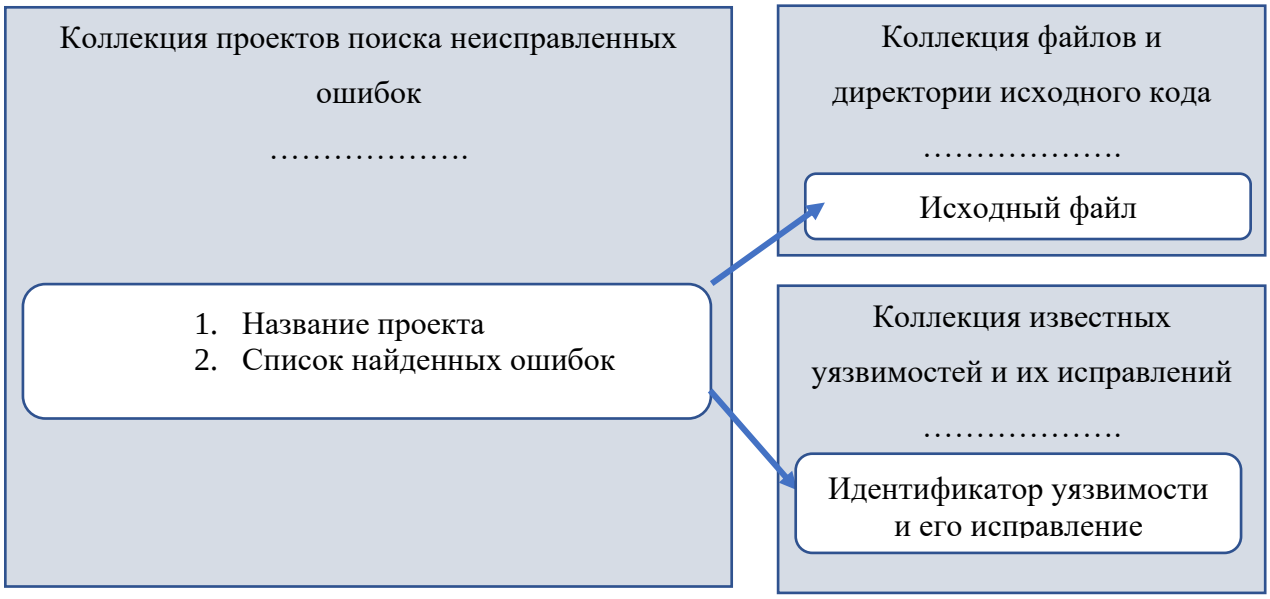

Рисунок 53. Схема хранения результатов поиска неисправленных ошибок.

6.3.2.6. Хранение результатов поиска утечек памяти

На рисунке 54 приводится схема хранения результатов поиска утечек памяти. В каждом проекте хранятся результаты поиска утечек памяти в формате *SARIF* [258]. Кроме этого, каждый документ из коллекции проектов утечек памяти связан ребром с соответствующим проектом, на котором был запущен анализ.

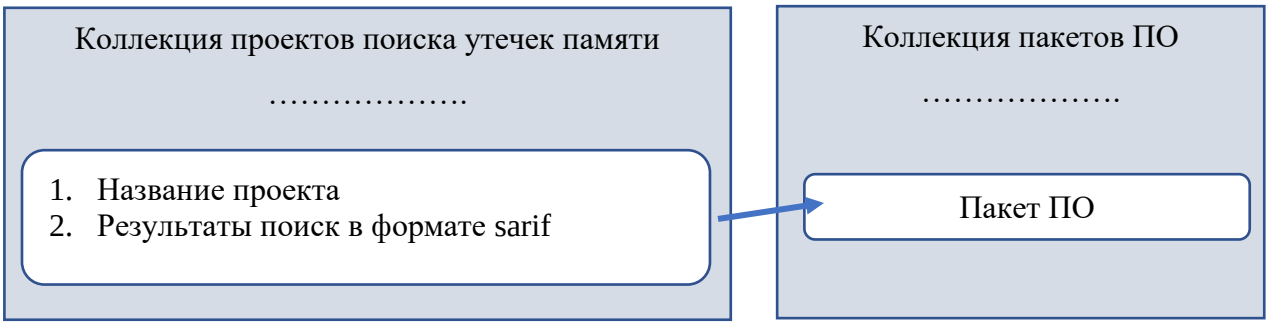

Рисунок 54. Схема хранения результатов поиска утечек памяти.

6.4. Обоснование выбора СУБД

В данном разделе приводится исследование, на базе которого была выбрана подходящая СУБД.

6.4.1. Выбор между реляционными и не реляционными базами данных

Разработка реляционных СУБД ведется в течение нескольких десятилетий и они (например, *Postgresql* [259], *SQL Server* [260], *Oracle* [261]) используются в многочисленных системах. Эти СУБД отличаются высокой стабильностью, производительностью и отказоустойчивостью. Несмотря на эти достоинства, в ходе анализа было решено отказаться от реляционных СУБД по следующим причинам:

1. ***Фиксированная схема***. Разрабатываемая система должна будет позволять пользователям писать свои собственные 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 СУБД, 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.

6.4.2. Выбор СУБД без фиксированной схемы

Существуют различные подходы для хранения данных; наиболее популярными из них являются: хранилища столбцов (*column-store*) [262], документный (*document*) [263], ключ-значение (*key-value*) [264] и графовый (*graph*) [265]. Все данные, которые будут генерироваться основным функционалом системы, можно хранить в виде графов (например, клоны кода, или сопоставления одной функции с несколькими другими функциями). Однако пользовательские плагины могут генерировать любые данные, которые необязательно будут логично представляться в виде графа. Поэтому мы решили использовать сравнительно новый подход хранения данных с несколькими моделями [266]. Такие СУБД позволяют хранить данные разными способами (например, данные могут храниться как в виде графа, так и в виде простого документа). Такая модель хорошо вписывается в логику хранения данных разрабатываемой системы. Во время исследований таких моделей мы рассмотрели *OrinetDB* [267]. Разработка системы ведется с 2010 года. По заявлению авторов проекта, *OrientDB* является стабильным и имеет высокую производительность. Более детально изучив этот проект, мы нашли несколько обсуждений [268], где пользователи жаловались на нестабильность системы, и на многочисленные ошибки. Пользователи также жаловались на отказ разработчиков исправлять те или иные ошибки. Кроме того, более детально изучив ошибки в *github* (*Issue list*) [269] и списки CVE, мы заметили серьезные ошибки безопасности в некоторых версиях платформы. В итоге было решено отказаться от использования *OrientDB*.

Другой популярной мульти-модельной СУБД является *ArangoDB* [270]. Система хорошо документирована, имеет свой собственный и интуитивный язык запросов (*AQL* [270]). Разработка ведется с 2011 года. По заявлению авторов, система стабильна и не уступает по производительности другим известным базам. Данные сравнения производительности можно увидеть на сайте [271], а также повторить результаты тестирования по описанной методике. Система также популярна в *github* и имеет больше 12000 "*звезд*".

6.4.3. Тестирования ArangoDB

Были проведены эксперименты, которые по функциональности близки к логике разрабатываемой платформы:

1. Загрузка около 25.000 пакетов ОС *Debian* в базу данных в виде дерева. Корень дерева представляет собой документ с описанием дистрибутива, его непосредственные приемники – корневые директории пакетов и т.д. (каждый пакет представляется в виде дерева). Для 25 000 пакетов в дереве создается более 8.000.000 вершин (количество всех файлов и директории). Загрузка всех 8 миллионов вершин и ребер в базу данных на одном ядре занимает чуть меньше 20 минут.
2. Обход графа с 2 миллионами вершин по ширине занимает около 16 секунд.
3. Обход графа с 4 миллионами вершин по ширине занимает около 50 секунд
4. Обход графа с 5.5 миллионами вершин по ширине занимает около 70 секунд
5. Обход графа с 7.5 миллионами вершин по ширине занимает около 105 секунд
6. Обход графа с 9 миллионами вершин по ширине занимает около 160 секунд

Как вывод, производительность ArangoDB приемлема для предлагаемой платформы.

6.5. Доступный Python API

В данной секции приводится набор доступных интерфейсных функций (*API*) в разработанной платформе.

6.5.1. Python API для поиска информации

Для компонента поиска информации доступен набор интерфейсных функций, обеспечивающий следующие функциональные возможности:

1. Получение списка всех доступных пакетов;
2. Получение списка всех доступных дистрибутивов ОС;
3. Получение документа, соответствующего заданному пакету;
4. Получение документа, соответствующего заданному дистрибутиву ОС;
5. Получение списка всех пакетов, содержащих заданный исходный файл;
6. Получение списка всех пакетов, содержащих заданный бинарный файл;
7. Получение документа содержащего пути всех исходных файлов заданного пакета;
8. Получение документа содержащего пути всех бинарных файлов заданного пакета;
9. Получение списка всех пакетов, для которых зависимости по сборке совпадают с зависимостями заданного пакета;
10. Получение списка всех пакетов, для которых зависимости по установке совпадают с зависимостями заданного пакета;
11. Получение документа, содержащего пути всех исходных файлов заданного дистрибутива ОС;
12. Получение документа, содержащего пути всех бинарных файлов заданного дистрибутива ОС;
13. Получение списка всех пакетов, входящих в заданный дистрибутив ОС;
14. Получение списка всех дистрибутивов ОС, содержащих заданный пакет;

15. Получение списка всех пакетов, необходимых для сборки заданного пакета;
16. Получение списка всех пакетов, необходимых для установки заданного пакета;
17. Получение пути к файлу, содержащему отладочную информацию для заданного бинарного файла;
18. Получение пути к файлу, содержащему ГЗП заданного исходного файла;
19. Получение списка всех пакетов, входящих в состав заданного пакета;
20. Получение списка всех пакетов, в состав которых входит заданный пакет;
21. Аннотация интервала строк в заданном исходном файле;
22. Аннотация интервала адресов в заданном бинарном файле;
23. Получение аннотаций для заданной строки исходного файла;
24. Получение аннотаций для заданного адреса бинарного файла;
25. Получение списка функций для заданного исходного файла;
26. Получение списка функций для заданного бинарного файла;
27. Получение пути к файлу, содержащему ГЗП заданного бинарного файла.

6.5.2. Python API для поиска клонов кода

Для компонента поиска клонов кода доступен набор интерфейсных функций, обеспечивающих следующие функциональные возможности:

1. Получение списка всех доступных проектов клона исходного кода;
2. Получение сортированного списка документов, соответствующих клонам исходного кода из заданного пользовательского проекта;
3. Получение имени функций, в котором содержится фрагмент заданного исходного кода;
4. Получение списка всех документов, соответствующих клонам заданного исходного кода;

5. Получение списка всех документов, соответствующих найденным клонам исходного кода в заданных пакетах ПО и дистрибутивах ОС;
6. Получение списка всех документов, соответствующих найденным клонам исходного кода в заданных файлах;
7. Получение списка всех документов, соответствующих найденным клонам исходного кода для заданного фрагмента;
8. Получение списка всех доступных проектов клона заданного бинарного кода;
9. Получение сортированного списка документов, соответствующих клонам бинарного кода из заданного пользовательского проекта;
10. Получение функции, в которой содержится фрагмент заданного бинарного кода;
11. Получение списка всех документов, соответствующих найденным клонам бинарного кода в заданных пакетах ПО и дистрибутивах ОС;
12. Получение списка всех документов, соответствующих найденным клонам бинарного кода в заданных файлах;
13. Получение списка документов, соответствующих найденным клонам бинарного кода для заданного фрагмента.

6.5.3. Python API для сопоставления исходных и бинарных файлов

Для компонента сопоставления исходных и бинарных файлов доступен набор интерфейсных функций, обеспечивающих следующие функциональные возможности:

1. Получение списка всех доступных проектов сопоставленных исходных и бинарных файлов;

2. Получение бинарного файла, для которого производится сопоставление заданного исходного кода;
3. Получение сопоставленных функций исходного кода для заданного бинарного кода;
4. Произведение сопоставления всех функций заданного бинарного файла со всеми функциями из заданных файлов исходного кода.

6.5.4. Python API для поиска неисправленных ошибок

Для компонента поиска неисправленных ошибок доступен набор интерфейсных функций, обеспечивающих следующие функциональные возможности:

1. Получение списка всех доступных проектов поиска неисправленных ошибок;
2. Получение списка найденных уязвимостей для заданного пользовательского проекта;
3. Получение списка уязвимостей для заданного исходного файла из пользовательского проекта;
4. Произведение поиска заданного набора неисправленных ошибок в заданном проекте.

6.5.5. Python API для фаззинга бинарных файлов

Для компонента фаззинга бинарных файлов доступен набор интерфейсных функций, обеспечивающий следующие функциональные возможности:

1. Получение списка всех доступных проектов фаззинга бинарных файлов;
2. Получение списка всех найденных входных данных, приводящих к падению заданного бинарного файла из пользовательского проекта;
3. Получение списка всех найденных входных данных, приводящих к зависанию заданного бинарного файла из пользовательского проекта;

4. Получение списка всех найденных входных данных, приводящих к увеличению покрытия кода для заданного бинарного файла из пользовательского проекта;
5. Получение покрытия кода заданного бинарного файла из пользовательского проекта;
6. Произведение фаззинга заданного бинарного файла с соответствующими настройками (ресурсы, начальные входные данные, режим запуска инструмента и т.д.).

6.5.6. Python API для поиска утечек памяти

Для компонента поиска утечек памяти доступен набор интерфейсных функций, обеспечивающих следующие функциональные возможности:

1. Получение списка всех доступных проектов поиска утечек памяти;
2. Получение списка найденных утечек памяти в формате *SARIF* для заданного пользовательского проекта;
3. Произведение поиска утечек памяти для заданного проекта.

6.6. Примеры использования Python API

В данном разделе приводится несколько примеров использования доступного программного интерфейса.

6.6.1. Пример запуска компонента поиска клонов исходного кода

В нижеприведенном примере (рисунок 55) производится поиск клонов исходного кода в рамках проекта "*openssl-1.0.2s*".

```
# Импортировать компонент поиска клонов исходного кода
import CCD
# Импортировать компонент поиска информаций
import IS
# Найти документы соответствующие пакету openssl-1.0.2s
pkg = IS.get_package("openssl-1.0.2s")
# Получить список исходных файлов openssl-1.0.2s
src_list = IS.get_package_source_tree(pkg)
# Произвести поиск клонов кода с длиной минимум 10 строк кода.
# Схожесть фрагментов минимум 90%. Инструмент поиска ccd.
# Найденные клоны будут сохранены в базе данных под названием "openssl_clones"
CCD.find_src_clones_within_files(src_list, 10, 0.9, "openssl_clones")
```

Рисунок 55. Пример поиска клонов исходного кода.

6.6.2. Пример комбинаций разных компонентов

В приведенном ниже примере (рисунок 56) всем клонам бинарного кода из файла "libssl.so", входящего в проект "*openssl-1.0.2s*", сопоставляются соответствующие функции исходного кода, которые затем выводятся на экран.

6.7. Экспериментальное тестирование платформы

В данной секции приводятся результаты экспериментального тестирования разработанной платформы.

6.7.1. Сбор базы данных для одного дистрибутива ОС Debian

Последние версии ОС *Debian* (начиная с *Debian-10*) имеют более 50,000 пакетов. При экспериментальном тестирование скорость сборки (с генерацией артефактов) одного пакет на одном ядре (*Intel 2.3GHz*) в среднем занимает ~4 минуты. Необходимая оперативная память на одно ядро в среднем ~500Мб. На 100 ядерном машине сбор базы данных для всех пакетов одного дистрибутива ОС *Debian* занимает порядка 40 часов.

```
# Импортировать компонент поиска клонов бинарного кода
import BCCD
import IS
# Импортировать компонент сопоставления исходных и бинарных файлов
import SBM
pkg = IS.get_package("openssl-1.0.2s")
# Получить список исходных и бинарных файлов "openssl-1.0.2s"
bin_list = IS.get_package_bin_tree(pkg)
src_list = IS.get_package_src_tree(pkg)
# Произвести поиск клонов бинарного кода
clone_project_name = "openssl_bin_clones"
BCCD.find_bin_clones_within_files(bin_list, 10, 0.9, clone_project_name)
# Получить список клонов для clone_project_name
bin_clones=CCD.get_bin_clones_within(clone_project_name)

# Для бинарного файла "libssl.so" произвести сопоставление с исходным кодом
libssl_src2bin_project_name = "libssl_src2bin"
for bin_file in bin_list:
 if "libssl.so" in bin_file:
   SBM.matched_binary_file_to_src(bin_file, src_list, 0.9, libssl_src2bin_project_name)
   break

# Для каждой бинарной функции напечатать сопоставленный исходный код при наличии
for bclone in bin_clones:
 if "libssl.so" in bclone.file_name
   print SBM.get_matched_source_function_for(bclone.function_name, libssl_src2bin_project_name)
```

Рисунок 56. Пример комбинированного применения программного интерфейса.

6.7.2. Сценарии комбинаций нескольких методов анализа кода

С использованием доступного *Python*-терминала и соответствующих программных интерфейсов было протестировано несколько сценариев гибкой комбинации разработанных методов анализа. База данных содержит несколько версий дистрибутивов ОС *Debian*, а также базу всех известных уязвимостей.

Сценарий 1:

1. Найти все пакеты ОС *Debian* исходный код которых содержит копии известных уязвимостей (находятся методом, описанным в разделе 3.4);
2. Для бинарных файлов соответствующих пакетов производить фаззинга с динамическим символьным выполнением (метод описан в главе 5.4.2).

В результате тестирования были найдены ряд падений в пакетах ОС *Debian*, которые приведены в таблице 31.

Сценарий 2:

1. Найти все пакеты ОС *Debian* исходный код которых содержит копии известных уязвимостей (находятся методом, описанной в главе 3.4);
2. Для исходного и бинарного кода найденных пакетов произвести сопоставление (делается методом, описанным в разделе 3.3) для получения адресов бинарного кода, содержащих копии известных уязвимостей;
3. Для бинарного кода найденных пакетов и соответствующих адресов из пункта 2, произвести направленный фаззинг (метод описан в разделе 5.4.2).

В результате тестирования были найдены ряд падений в пакетах ОС *Debian*, которые приведены в таблице 32.

6.8. Заключение по разработанной платформе

Экспериментальное тестирование разработанной нами платформы продемонстрировала возможность сбора большой базы ПО, а также эффективность гибкой комбинаций разработанных методов анализа кода. *Python*-терминал предоставляет возможность быстрой реализаций любых сценариев комбинированного применения разработанных методов, что позволяет находить новые ошибки. Более того, два сценария комбинации статического анализа с фаззингом позволили найти множество ошибок в пакетах ОС *Debian*, что доказывает эффективность разработанной платформы.

7. Практическое значимость диссертационной работы

В рамках диссертационной работы были решены ряд актуальных проблем, связанных с анализом безопасности ПО. В исследованиях по направлению поиска клонов кода и базирующихся на них технологий было создано множество инструментов анализа кода, включая инструмент поиска копий неисправленных уязвимостей. Применение данного инструмента позволило найти ряд ошибок в открытом ПО, имеющем широкое использование. В частности, стало возможно найти копию известной уязвимости в загрузчике ОС *Linux*, которая была принята и исправлена сообществом (описано в разделе 3.4.3).

В исследованиях по направлению поиска ошибок использования динамической памяти и анализа помеченных данных было разработано множество инструментов. Последний метод, комбинирующий статический анализ с символьным выполнением для поиска утечек памяти, показал *state-of-the-art* результаты. Применение данного метода позволило найти десятки ошибок в широко используемых открытых проектах, включая *openssl* (описано в разделе 4.3.7).

В исследованиях по направлению оптимизации методов фаззинга в разных сценариях были разработаны методы: генерации сложно структурированных данных на базе БНФ автоматов; анализа интерфейсных функций, направленного анализа; интеграции символьного выполнения и статического анализа с фаззингом. На практике, эти методы показали свою эффективность. Метод генерации структурированных данных позволил в некоторых случаях увеличить количество выполненных путей до трех раз для таких проектов, как *gcc*, *python* и *fortran* (описано в разделе 5.1.5). Метод анализа интерфейсных функций позволил найти в библиотеке *PluginBase*, входящей в состав платформы *SmartThings* (версия 1.7.9), 15 уникальных аварийных завершений программы (описано в разделе 5.2.9). Результаты были подтверждены командой тестирования "*Samsung Electronics Co. Ltd*" и исправлены. Необходимо отметить, что с помощью платформы *SmartThings* можно управлять

подключенными к сети умными устройствами, и эксплуатация указанных дефектов могло иметь серьезные последствия. Метод интеграции символьного выполнения и статического анализа с фаззингом позволил найти ряд ошибок в стандартных пакетах ОС *Debian* (описано в разделе 5.4.2).

Разработанная платформа путем комбинации нескольких методов анализа позволила найти ошибки в стандартных пакетах ОС *Debian* (описано в разделе 6.7.2). Кроме того, платформа активно применяется в жизненном цикле разработки десятков проектов, выполняемых в ИСП РАН и ЦППТ РАУ с 2021 года, что покрывает многие из требований ГОСТ Р 56939-2016 и "*Методики выявления уязвимостей и недекларированных возможностей в программном обеспечении*" ФСТЭК Российской Федерации. С помощью платформы и разработанных методов анализа кода стало возможно найти и исправить сотни ошибок во время разработки ПО, что существенно повлияло на расходы, безопасность и надежность разрабатываемых систем. Вдобавок надо отметить, что предложенная платформа может быть использована при создании отраслевых и корпоративных репозиториев безопасного ПО. Разработанная платформа анализа апробирована на пакетах ОС *Debian*, которые в том числе служат основой для построения дистрибутивов *Astra Linux* и *Alt Linux*. Найденные ошибки в пакетах подтверждают работоспособность платформы.

Отдельно разработанные методы реализованы в инструментах *Svace* и *ISP-Fuzzer*, входящих в состав *Crusher*. Эти инструменты являются индустриальным стандартом для жизненного цикла разработки безопасного ПО и покрывают многие из требований ГОСТ Р 56939-2016 и "*Методики выявления уязвимостей и недекларированных возможностей в программном обеспечении*" ФСТЭК Российской Федерации.

Суммируя, можно сказать следующее: кроме разработанных качественно новых методов анализа кода и удобной объединяющей платформы, в ходе экспериментального тестирования было найдено и исправлено множество

критических ошибок в открытом ПО. В некоторых случаях, ошибки находились и исправлялись в таких широко используемых открытых проектах, что это могло затронуть буквально всех, имеющих доступ к интернету. Таким образом, выполненная работа в рамках данной диссертации, кроме создания новых методов анализа и платформы, также позволила улучшить безопасность ряда существующих открытых проектов. В Таблица 35 суммируются найденные ошибки в реальных проектах в рамках диссертационной работы.

Таблица 35. Типы и количество ошибок, обнаруженные в промышленном ПО.

№	Объекты анализа	Название инструмента, который обнаружил	Тип Ошибки	Количество Ошибок	Статус
1.	4Pane	Инструмент поиска неисправленных уязвимостей	Копия известной уязвимости	1	Сообщено и исправлено
2.	doublecmd	Инструмент поиска неисправленных уязвимостей	Копия известной уязвимости	1	Сообщено и исправлено
3.	AdAway	Инструмент поиска неисправленных уязвимостей	Копия известной уязвимости	4	Сообщено и исправлено
4.	praat	Инструмент поиска неисправленных уязвимостей	Копия известной уязвимости	1	Сообщено и исправлено
5.	luaJIT	Инструмент поиска неисправленных уязвимостей	Копия известной уязвимости	1	Сообщено и исправлено
6.	grub2	Инструмент поиска неисправленных уязвимостей	Копия известной уязвимости	1	Сообщено и исправлено
7.	moarVM	Инструмент поиска неисправленных уязвимостей	Копия известной уязвимости	1	Сообщено и исправлено
8.	openssl	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
9.	ffmpeg	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
10.	radare2	Инструмент поиска утечек памяти	Утечка памяти	2	Сообщено и исправлено
11.	bind9	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
12.	clib	Инструмент поиска утечек памяти	Утечка памяти	3	Сообщено и исправлено
13.	coturn	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено

14.	cups	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
15.	cyclonedds	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
16.	gpac	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
17.	pupnp	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
18.	varnish-cache	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
19.	masscan	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
20.	FreeRDP	Инструмент поиска утечек памяти	Утечка памяти	2	Сообщено и исправлено
21.	libvips	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
22.	zstd	Инструмент поиска утечек памяти	Утечка памяти	1	Сообщено и исправлено
23.	gcc-12	Фаззинг с применением БНФ грамматик	Падение	1	Сообщено, но не исправлено
24.	clang-14	Фаззинг с применением БНФ грамматик	Падение	1	Сообщено, но не исправлено
25.	PluginBase, Samsung SmartThings	Фаззинг интерфейсных функций	*PluginDataStorageException*	1	Сообщено и исправлено
26.	PluginBase, Samsung SmartThings	Фаззинг интерфейсных функций	*StringIndexOutOfBoundsException*	1	Сообщено и исправлено
27.	PluginBase, Samsung SmartThings	Фаззинг интерфейсных функций	*NullPointerException*	12	Сообщено и исправлено
28.	PluginBase, Samsung SmartThings	Фаззинг интерфейсных функций	*RemoteException*	1	Сообщено и исправлено
29.	blast2	Платформа анализа с комбинацией нескольких методов	Падение	3	Не сообщено, но верифицировано
30.	faad	Платформа анализа с комбинацией нескольких методов	Падение	3	Не сообщено, но верифицировано
31.	efax	Платформа анализа с комбинацией нескольких методов	Падение	1	Не сообщено, но верифицировано
32.	wavpack	Платформа анализа с комбинацией нескольких методов	Падение	5	Не сообщено, но верифицировано
33.	tic	Платформа анализа с комбинацией нескольких методов	Падение	4	Не сообщено, но верифицировано
34.	ul	Платформа анализа с комбинацией нескольких методов	Падение	7	Не сообщено, но верифицировано
35.	Bsd-form	Платформа анализа с комбинацией нескольких методов	Падение	6	Не сообщено, но верифицировано
36.	passwd	Платформа анализа с комбинацией нескольких методов	Падение	2	Не сообщено, но верифицировано
37.	uuenview	Платформа анализа с комбинацией нескольких методов	Падение	13	Не сообщено, но верифицировано

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.

Заключение

Данная работа посвящена развитию и разработке методов статического и динамического анализа программ, их комбинации и интеграции, что позволяет дополнительно обнаружить нетривиальные уязвимости и ошибки. Кроме того, целью работы являлась разработка и реализация платформы, которая позволяет автоматически собирать большую базу проектов открытого ПО, а также доступные уязвимости, и использовать их во время анализа. Разработанная платформа обеспечивает программный интерфейс, что позволяет гибким образом объединить несколько анализов под конкретную задачу. Отдельно разработанные методы реализованы в инструментах *Svace* и *ISP-Fuzzer*, входящий в состав *Crusher*. Эти инструменты являются индустриальным стандартом для жизненного цикла разработки безопасного ПО и покрывают многие из требований ГОСТ Р 56939-2016 и "*Методики выявления уязвимостей и недекларированных возможностей в программном обеспечении*" ФСТЭК Российской Федерации. 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.

В дальнейших работах по совершенствованию предложенных методов анализа и разработанной платформы можно выделить следующие направления:

- Улучшение модели взаимодействия методов статического и динамического анализа программ: разработка новых метрик переключения между разными методами во время комбинированного анализа;
- Интеграция в разработанную платформу доступных инструментов статического и динамического анализа программ, а также поддержка соответствующих программных интерфейсов;
- Поддержка разработанных методов статического анализа кода для других языков программирования (*Java*, *Rust*, *Solidity* и т.д.);

- Автоматический фаззинг интерфейсных функций;
- Применение МО для генерации эффективных входных данных для фаззинга;
- Поддержка множества сценариев комбинаций различных методов анализа по умолчанию;
- Автоматическое определение наилучшего набора комбинаций методов анализов – в зависимости от целевой программы.

Список литературы

Статьи автора 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, pp. 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, 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,* vol. 27, № 1, pp. 39-50, 2015.

[14] С. Саргсян, "Поиск семантических ошибок, возникающих при некорректной адаптации скопированных участков кода", *Труды Института системного программирования РАН,* т. 27, № 2, pp. 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", *Proccedings 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,* vol. 28, № 5, pp. 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, pp. 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,* vol. 23, pp. 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] Ш. Курмангалеев, А. Асланян, М. Арутюнян, Р. Оганесян, В. Варданян и С. Саргсян, "LibraryIdentifier". РФ Патент ЭВМ № 2021665076, 17 09 2021.

[26] Ш. Курмангалеев, Г. Иванов, В. Варданян, А. Асланян, А. Федотов и С. Саргсян, «Инструмент анализа изменений между двумя версиями программы "patchAnalysis"». РФ Патент ЭВМ № 2019661049, 16 08 2019.

[27] Ш. Курмангалеев, М. Мишечкин, В. Акользин и С. Саргсян, «Инструмент фаззинга программ "ISP-Fuzzer"». РФ Патент ЭВМ № 2019661047, 16 08 2019.

[28] Ш. Курмангалеев, М. Мишечкин, В. Акользин и С. Саргсян, «Модуль направленного фаззинга программ для "ISP-Fuzzer"». РФ Патент ЭВМ № 2019660716, 12 08 2019.

[29] Ш. Курмангалеев, С. Саргсян, А. Асланян и И. Г.С., «Инструмент поиска клонов кода для бинарных файлов "BINCCD"». РФ Патент ЭВМ № 2019661048, 16 08 2019.

[30] Ш. Курмангалеев, С. Саргсян, В. Варданян, А. Асланян, Д. Акопян, М. Арутюнян, М. Меграбян, О. Мовсисян, К. Саргсян и Р. Оганесян, "ISP Genes". РФ Патент ЭВМ № 2020663670, 30 10 2020.

[31] Ш. Курмангалеев, С. Саргсян, В. Варданян и И. Г.С., «Инструмент поиска клонов кода для C/C++ программ "CCD"». РФ Патент ЭВМ № 2019660800, 13 08 2019.

Цитируемая литература

[32] "NIST CVE report". https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&search_type=all&isCpeNameSearch=false. Дата обращения 01.08.2024.

[33] "codeql". https://codeql.github.com/. Дата обращения 01.08.2024.

[34] "github". https://github.com/. Дата обращения 01.08.2024.

[35] "oss-fuzz". https://github.com/google/oss-fuzz. Дата обращения 01.08.2024.

[36] "openssl". https://www.openssl.org/. Дата обращения 01.08.2024.

[37] "cve_report_oss_fuzz". https://www.code-intelligence.com/blog/intro-to-oss-fuzz . Дата обращения 01.08.2024.

[38] "sdl". https://www.microsoft.com/en-us/securityengineering/sdl. Дата обращения 01.08.2024.

[39] "google-zero". https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html. Дата обращения 01.08.2024.

[40] "memory_leak_cve". https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=memory+leak. Дата обращения 01.08.2024.

[41] "format_string_cve". https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=format+string. Дата обращения 01.08.2024.

[42] "results_oss_fuzz". https://introspector.oss-fuzz.com/projects-overview. Дата обращения 01.08.2024.

[43] "Joern". https://joern.io/. Дата обращения 01.08.2024.

[44] "S2E". https://s2e.systems/. Дата обращения 01.08.2024.

[45] "coverity". https://scan.coverity.com/. Дата обращения 01.08.2024.

[46] "svace". https://www.ispras.ru/technologies/svace/. Дата обращения 01.08.2024.

[47] P. Gautam, H. Saini, "Various Code Clone Detection Techniques and Tools: A Comprehensive Survey", *SmartCom 2016: Smart Trends in Information Technology and Computer Communications* , pp. 655-667, 2016.

[48] Chanchal K. Roy, James R. Cordy, Rainer Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach", *Science of Computer Programming*, vol. 74, pp. 470-495, 2009.

[49] M. H. Alalfi, J. R. Cordy, T. R. Dean, "Models are code too: Near-miss clone detection for Simulink models", *28th IEEE International Conference on Software Maintenance*, 2012.

[50] C. Roy, "An empirical study of function clones in open source software systems", *15th Working Conference on Reverse Engineering*, 2008.

[51] J. Cordy, "The TXL source transformation language", *Science of Computer Programming*, vol. 61, № 3, pp. 190-210, 2006.

[52] J. R. Cordy, "SIMONE: Architecture-Sensitive Near-miss Clone Detection for Simulink Models", *First International Workshop on Automotive Software Architecture (WASA)*, 2015.

[53] J. R. Cordy, C. K. Roy, "Tuning Research Tools for Scalability and Performance: The NICAD Experience", *Experimental Software and Toolkits*, 2014.

[54] S. Ducasse, M. Rieger, S. Demeyer, "A language independent approach for detecting duplicated code", *15th International Conference on Software Maintenance (ICSM)*, pp. 109-119, 1999.

[55] J. Johnson, "Identifying redundancy in source code using fingerprints", *Centre for Advanced Studies on Collaborative Research*, 1993.

[56] J. Johnson, "Visualizing textual redundancy in legacy source", *Centre for Advanced Studies on Collaborative research*, 1994.

[57] J. Johnson, "Substring matching for clone detection and change tracking", *0th International Conference on Software Maintenance*, 1994.

[58] U. Manber, "Finding similar files in a large file system", *Usenix Technical Conference*, 1994.

[59] D. Martin, J. Cordy, "Analyzing web service similarities using contextual clones", *5th International Workshop on Software Clones*, 2011.

[60] R. Wettel, R. Marinescu, "Archeology of code duplication: Recovering duplication chains from small duplication fragments", *7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2005.

[61] C. Roy, "Detection and analysis of near miss software clones", *25th IEEE International Conference on Software Maintenance*, 2009.

[62] C. Roy, R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization", *16th IEEE International Conference on Program Comprehension (ICPC)*, 2008.

[63] C. Roy, J. Rody, "Are scripting languages really different", *4th ICSE International Workshop on Software Clones*, 2010.

[64] M. Stephan, M. H. Alafi, A. Stevenson, J. R. Cordy, "Towards Qualitative Comparison of Simulink Model Clone Detection Approaches", *6th International Workshop on Software Clones*, 2012.

[65] E. Juergens, F. Deissenboeck, B. Hummel, S. Wagner, "Do code clones matter?", *31st International Conference on Software Engineering (ICSE)*, 2009.

[66] R. Falke, P. Frenzel, R. Koschke, "Empirical evaluation of clone detection using syntax suffix trees", *Empirical Software Engineering*, vol. 13, pp. 601-643, 2008.

[67] R. Tairas, J. Gray, "Phoenix-based clone detection using suffix trees", *44th Annual Southeast Regional Conference*, pp. 679-684, 2006.

[68] Tung Thanh Nguyen, Hoan Anh Nguyen, Jafar M. Al-Kofahi, Nam H. Pham, Tien N. Nguyen, "Scalableand incremental clone detection for evolving software", *25th IEEE International Conference on Software Maintenance (ICSM)*, 2009.

[69] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, Tien N. Nguyen, "ClemanX: Incremental clone detection tool for evolving software", *31st International Conference on Software Engineering (ICSE)*, 2009.

[70] M. Chilowicz, E. Duris, G. Roussel, "Syntax tree fingerprinting for source code similarity detection", *17th IEEE International Conference on Program Comprehension (ICPC)*, 2009.

[71] J. Krinke, "Identifying similar code with program dependence graphs", *8th Working Conference on Reverse Engineering (WCRE)*, 2001.

[72] R. Komondoor, S. Horwitz, "Using slicing to identify duplication in source code", *8th International Symposium on Static Analysis (SAS)*, 2001.

[73] Y. Higo, S. Kusumoto, "Code clone detection on specialized PDG's with heuristics", *15th European Conference on Software Maintenance and Reengineering*, 2011.

[74] C. Liu, C. Chen, J. Han, P. Yu, "GPLAG: Detection of software plagiarism by program dependence graph analysis", *12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD).*, 2006.

[75] M. Gabel, L. Jiang, Z. Su, "Scalable detection of semantic clones", *30th International Conference on Software Engineering (ICSE)*, 2008.

[76] L. Jiang, G. Misherghi, Z. Su, S. Glondu, "DECKARD: Scalable and accurate treebased detection of code clones", *29th International Conference on Software Engineering* , 2007.

[77] "Deckard". https://github.com/skyhover/Deckard. Дата обращения 01.08.2024.

[78] Mayrand, Leblanc, Merlo, "Experiment on the automatic detection of function clones in a software system using metrics", *12th International Conference on Software Maintenance (ICSM)*, 1996.

[79] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika, V. Saranya, "Detection of Type-1 and Type-2 code clones using textual analysis and metrics", *International Conference on Recent Trends in Information, Telecommunication and Computing*, 2010.

[80] M. Chilowicz, E. Duris, G. Roussel, "Finding similarities in source code through factorization", Electronic Notes in Theoretical Computer Science, vol. 238, № 5, pp. 47-62, 2009.

[81] A. Sheneamer, J. Kalita, "Semantic Clone Detection Using Machine Learning", *15th IEEE International Conference on Machine Learning and Applications* , 2016.

[82] Vaibhav Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, C. Lopes, "Oreo: Detection of Clones in the Twilight Zone", *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[83] "OREO". https://github.com/Mondego/oreo. Дата обращения 01.08.2024.

[84] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes, "SourcererCC: Scaling Code Clone Detection to Big Code", *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[85] "SourcererCC". https://github.com/Mondego/SourcererCC. Дата обращения 01.08.2024.

[86] "A System for Detecting Software Plagiarism". http://theory.stanford.edu/~aiken/moss/. Дата обращения 01.08.2024.

[87] S. Schleimer, D. S. Wilkerson, A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting", *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[88] "CCFinder". URL: http://www.ccfinder.net. Дата обращения 01.08.2024.

[89] T. Kamiya, S. Kusumoto, K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code", *EEE Transactions on Software Engineering*, vol. 28, № 7, pp. 654-670, 2002.

[90] S. Livieri, Y. Higo, M. Matushita, K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder", *9th International Conference on Software Engineering (ICSE).*, 2007.

[91] "CPD". https://pmd.sourceforge.io/pmd-5.3.0/usage/cpd-usage.html. Дата обращения 01.08.2024.

[92] "PMD". https://github.com/pmd/pmd. Дата обращения 01.08.2024.

[93] "Karp-Rabin". https://xlinux.nist.gov/dads/HTML/karpRabin.html. Дата обращения 01.08.2024.

[94] "VUDDY". https://github.com/iotcube/hmark. Дата обращения 01.08.2024.

[95] S. Kim, S. Woo, H. Lee, H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery", *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[96] "ANTRL". https://www.antlr.org/. Дата обращения 01.08.2024.

[97] "ReDebug". https://github.com/dbrumley/redebug. Дата обращения 01.08.2024.

[98] J. Jang, A. Agrawal, D. Brumley, "ReDeBug: finding un- patched code clones in entire os distributions", *IEEE Symposium on Security and Privacy*, 2012.

[99] R. Tairas, J. Gray, "Sub-clone refactoring in open source software artifacts", *ACM Symposium on Applied Computing*, pp. 2373-2374, 2010.

[100] "Clone Doctor: Software Clone Detection and Reporting". http://www.semdesigns.com/products/clone/. Дата обращения 01.08.2024.

[101] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Biep, "Clone Detection Using Abstract Syntax Trees", *Proceedings of International Conference on Software Maintenance*, 1998.

[102] "Semantic Designs". http://www.semdesigns.com/. Дата обращения 01.08.2024.

[103] "gpoo/ccfinderx". https://github.com/gpoo/ccfinderx. Дата обращения 01.08.2024.

[104] "petersenna/ccfinderx-core". https://github.com/petersenna/ccfinderx-core. Дата обращения 01.08.2024.

[105] "radekg1000/ccfinderx". https://github.com/radekg1000/ccfinderx. Дата обращения 01.08.2024.

[106] "bitshred". https://github.com/dbrumley/bitshred. Дата обращения 01.08.2024.

[107] J. Jang, D. Brumley, "Bitshred: Fast, scalable code reuse detection in binary code", *CyLab*, 2009.

[108] B. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, vol. 13, isuue 7, pp. 422-426, 1970.

[109] T. Dullien, R. Rolles, "Graph-based comparison of executable objects", *Actes du SSTIC05*, 2005.

[110] T. Dullien, E. Carrera, S. Eppler, S. Porst, "Automated Attacker", *RTO-MP-IST091*, 2010.

[111] T. Dullien, E. Carrera, S. Eppler, S. Porst, "Automated attacker correlation for malicious code", *DTIC Document*, 2010.

[112] "IDA Pro disassembler". https://www.hex-rays.com/products/ida. Дата обращения 01.08.2024.

[113] I. Briones, A. Gomez, "Graphs, entropy and grid computing: Automatic comparison of malware", *Proceedings of Virus Bulletin International Conference*, 2008.

[114] M. Bourquin, A. King, E. Robbins, "BinSlayer: Accurate Comparison of Binary Executables", *ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[115] H. W. Kuhn, "The Hungarian Method for the assignment problem", *Naval Research Logistics Quarterl*, vol. 2, № 1-2, pp. 83-97 1955.

[116] "diaphora help". https://github.com/joxeankoret/diaphora/blob/master/doc/diaphora_help.pdf. Дата обращения 01.08.2024.

[117] "sqlite". https://sqlite.org/index.html. Дата обращения 01.08.2024.

[118] "Hex Rays". https://hex-rays.com/. Дата обращения 01.08.2024.

[119] R. Rivest, "The MD5 Message-Digest Algorithm", *RFC Editor*, 1992.

[120] "BinDiff". https://github.com/google/bindiff. Дата обращения 01.08.2024.

[121] "DeepToad". https://github.com/joxeankoret/deeptoad. Дата обращения 01.08.2024.

[122] "Kam1n0". https://github.com/McGill-DMaS/Kam1n0-Community. Дата обращения 01.08.2024.

[123] S. Ding, B. Fung, P. Charland, "Kam1n0: MapReduce-based Assembly Clone Search for Reverse Engineering", *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD)*, pp. 461-470, 2016.

[124] S. Ding, B. Fung, P. Charland, "Asm2Vec: boosting static representation robustness for binary clone search against code obfuscation and compiler optimization", *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[125] A. Andoni, P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions", *47th Annual IEEE Symposium*, 2006.

[126] "Map Reduce". https://www.databricks.com/glossary/mapreduce. Дата обращения 01.08.2024.

[127] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, P. Narasimhan, "Binary Function Clustering Using Semantic Hashes", *EEE ICMLA*, 2012.

[128] L. Nouh, A. Rahimian, D. Mouheb, M. Debbabi, A. Hanna, "BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables", *ICT Systems Security and Privacy Protection*, pp. 341-355, 2017.

[129] L.Bingchang, H. Wei, Z. Chao, L. Wenchao, L. Feng, P. Aihua, Z. Wei, "αDiff: Cross-Version Binary Code Similarity Detection with DNN", *Proceedings of the*

33rd ACM/IEEE International Conference on Automated Software Engineering, 2018.

[130] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, H. B. K. Tan, "Bingo: cross-architecture cross-os binary search", *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[131] J. Sliwerski, T. Zimmermann, A. Zeller, "When Do Changes Induce Fixes", *ACM SIGSOFT Software Engineering Notes*, vol. 3, № 4, 2005.

[132] "BugZilla". https://www.bugzilla.org. Дата обращения 01.08.2024.

[133] "JIRA". https://www.atlassian.com/software/jira. Дата обращения 01.08.2024.

[134] A. Bachmann, A. Bernstein, "Software process data quality and characteristics: a historical view on open and closed source projects", *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009.

[135] V. Dallmeier, T. Zimmermann, "Extraction of bug localization benchmarks from history", *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp.433-436, 2007.

[136] S. Kim, T. Zimmermann, E. J. Whitehead, A. Zeller, "Predicting faults from cached history", *International Conference on Software Engineering (ICSE)*, 2007.

[137] A. Mockus, L. G. Votta, "Identifying reasons for software changes using historic databases", *International Conference on Software Maintenance (ICSM)*, 2000.

[138] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, A. Bernstein, "The Missing Links: Bugs and Bug-fix Commits", *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, 2010.

[139] "Apache". https://httpd.apache.org/. Дата обращения 01.08.2024.

[140] F. Rahman, D. Posnett, A. Hindle, E. Barr, P. Devanbu, "BugCache for Inspections: Hit or Miss?", *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011.

[141] C. Corley, N. Kraft, L. Etzkorn, S. Lukins, "Recovering Traceability Links between Source Code and Fixed Bugs via Patch Analysis", *Proceedings of the 6th*

International Workshop on Traceability in Emerging Forms of Software Engineering, 2011.

[142] J. Yang, X. Song, Y. Xiong, Y. Meng, "An Open Source Software Defect Detection Technique Based on Homology Detection and Pre-identification Vulnerabilitys", *International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2018.

[143] "MITRE". https://www.mitre.org. Дата обращения 01.08.2024.

[144] A. XU, T. Dai, H. Chen, Z. Ming, W. Li, "Vulnerability Detection for Source Code Using Contextual LSTM", *5th International Conference on Systems and Informatics*, 2018.

[145] "SARD". https://samate.nist.gov/SARD/testsuite.php. Дата обращения 01.08.2024.

[146] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, Y. Acar, "VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits", *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications*, 2015.

[147] "VCCFinder". https://github.com/hperl/vccfinder. Дата обращения 01.08.2024.

[148] "NVD". https://nvd.nist.gov. Дата обращения 01.08.2024.

[149] "Change Distiller". https://bitbucket.org/sealuzh/tools-changedistiller/src/master/. Дата обращения 01.08.2024.

[150] B. Fluri, H. C. Gal, "Classifying Change Types for Qualifying Change Couplings", *IEEE Workshop on Program Comprehension*, 2006.

[151] "LSDIFF". https://linux.die.net/man/1/lsdiff. Дата обращения 01.08.2024.

[152] M. Kim, D. Notkin, "Discovering and representing systematic code changes", *International Conference on Software Engineering (ICSE)*, pp. 309-319, 2009.

[153] M. Fowler, "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 2000.

[154] Kaifeng Huang, Bihuan Chen, Xin Peng, Daihong Zhou, Ying Wang, Yang Liu, Wenyun Zhao, "ClDiff: Generating Concise Linked Code Differences",

Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, 2018.

[155] "ClDIFF". https://github.com/FudanSELab/CLDIFF. Дата обращения 01.08.2024.

[156] "Pigaios". https://github.com/joxeankoret/pigaios. Дата обращения 01.08.2024.

[157] "Clang: a C language family frontend for LLVM". https://clang.llvm.org/. Дата обращения 01.08.2024.

[158] A. Rahimian, P. Charland, S. Preda, M. Debbabi, "RESource: A Framework for Online Matching of Assembly with Open Source Code", *Foundations and Practice of Security*, vol. 7743, pp. 211–226, 2013.

[159] "ON MATCHING BINARY TO SOURCE CODE". https://users.encs.concordia.ca/~mmannan/student-resources/Thesis-MASc-Shahkar-2016.pdf. Дата обращения 01.08.2024.

[160] "Karta – Matching Open Sources in Binaries". https://research.checkpoint.com/karta-matching-open-sources-in-binaries/. Дата обращения 01.08.2024.

[161] "libpng". http://www.libpng.org/pub/png/libpng.html. Дата обращения 01.08.2024.

[162] "zlib". https://zlib.net/. Дата обращения 01.08.2024.

[163] "openssh". https://www.openssh.com/. Дата обращения 01.08.2024.

[164] "net-snmp". http://www.net-snmp.org/. Дата обращения 01.08.2024.

[165] "gSOAP". https://www.cs.fsu.edu/~engelen/soap.html. Дата обращения 01.08.2024.

[166] "libxml2". https://gitlab.gnome.org/GNOME/libxml2/-/wikis/home. Дата обращения 01.08.2024.

[167] "libtiff". http://www.libtiff.org/. Дата обращения 01.08.2024.

[168] "mdnsresponder". https://www.howtogeek.com/338914/what-is-mdnsresponder-and-why-is-it-running-on-my-mac/. Дата обращения 01.08.2024.

[169] "MAC-Telnet". https://github.com/haakonnessjoen/MAC-Telnet. Дата обращения 01.08.2024.

[170] "libjpeg-turbo". https://github.com/libjpeg-turbo/libjpeg-turbo. Дата обращения 01.08.2024.

[171] "libjpeg". https://libjpeg.sourceforge.net/. Дата обращения 01.08.2024.

[172] "icu". https://icu.unicode.org/home. Дата обращения 01.08.2024.

[173] "libvpx". https://www.webmproject.org/code/. Дата обращения 01.08.2024.

[174] "treck". https://treck.com. Дата обращения 01.08.2024.

[175] G. Fan, R. Wu, Q. Shi, X. Xiao, J. Zhou, C. Zhang, "SMOKE: Scalable Path-Sensitive Memory Leak Detection for Millions of Lines of Code", *International Conference on Software Engineering (ICSE)*, 2019.

[176] "Z3". https://github.com/Z3Prover/z3. Дата обращения 01.08.2024.

[177] W. Li, H. Cai, Y. Sui, D. Manz, "PCA: memory leak detection using partial call-path analysis", *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[178] "LLVM". https://llvm.org/. Дата обращения 01.08.2024.

[179] "Andersen pointer analysis". https://github.com/grievejia/andersen. Дата обращения 01.08.2024.

[180] "SVF". https://github.com/SVF-tools/SVF. Дата обращения 01.08.2024.

[181] "Use-def chain". https://en.wikipedia.org/wiki/Use-define_chain. Дата обращения 01.08.2024.

[182] S. Cherem, L. Princehouse, R. Rugina, "Practical memory leak detection using guarded value-flow analysis", *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[183] "CSA". https://clang-analyzer.llvm.org/. Дата обращения 01.08.2024.

[184] "fbinfer". Available: https://fbinfer.com/. Дата обращения 01.08.2024.

[185] X. Sun, S. Xu, C. Guo, J. Xu, N. Dong, X Ji, S. Zhang, "A Projection-Based Approach for Memory Leak Detection", *IEEE 42nd Annual Computer Software and Applications Conference*, 2018.

[186] "AddressSanitizer". https://github.com/google/sanitizers/wiki/AddressSanitizer. Дата обращения 01.08.2024.

[187] "Valgrind". https://valgrind.org/. Дата обращения 01.08.2024.

[188] D. Bruening, Q. Zhao, "Practical memory checking with Dr. Memory", *International Symposium on Code Generation and Optimization*, 2011.

[189] "DynamoRIO". https://dynamorio.org/. Дата обращения 01.08.2024.

[190] B. Gui, W. Song, H. Xiong, J. Huang, "Automated Use-After-Free Detection and Exploit Mitigation: How Far Have We Gone?", *IEEE Transactions on Software Engineering* , vol. 48, № 11, pp. 4569 - 4589, 2022.

[191] "AFL". https://lcamtuf.coredump.cx/afl/. Дата обращения 01.08.2024.

[192] "ImageMagic". https://imagemagick.org/index.php. Дата обращения 01.08.2024.

[193] "Firefox Mozilla". https://www.mozilla.org/hy-AM/firefox/new/. Дата обращения 01.08.2024.

[194] "LibFuzzer". https://llvm.org/docs/LibFuzzer.html. Дата обращения 01.08.2024.

[195] "libxml2". https://github.com/GNOME/libxml2. Дата обращения 01.08.2024.

[196] "Radar2". https://rada.re/n/. Дата обращения 01.08.2024.

[197] "Tensorflow". https://www.tensorflow.org/. Дата обращения 01.08.2024.

[198] "FFmpeg". Available: https://ffmpeg.org/. Дата обращения 01.08.2024.

[199] "Wireshark". https://www.wireshark.org/. Дата обращения 01.08.2024.

[200] "QEMU". https://www.qemu.org/. Дата обращения 01.08.2024.

[201] "Peach". https://peachtech.gitlab.io/peach-fuzzer-community/. Дата обращения 01.08.2024.

[202] "Syzkaller". https://github.com/google/syzkaller. Дата обращения 01.08.2024.

[203] "gramfuzz". https://pypi.org/project/gramfuzz/. Дата обращения 01.08.2024.

[204] "Sulley". https://github.com/OpenRCE/sulley. Дата обращения 01.08.2024.

[205] "Honggfuzz". https://github.com/google/honggfuzz. Дата обращения 01.08.2024.

[206] "jfuzz". https://people.csail.mit.edu/akiezun/jfuzz/. Дата обращения 01.08.2024.

[207] "radamsa". https://github.com/Hwangtaewon/radamsa. Дата обращения 01.08.2024.

[208] "Диссертация С.С. Саргсян". https://www.ispras.ru/dcouncil/docs/diss/2016/sargsyan/sargsyan.php. Дата обращения 01.08.2024.

[209] "binnavi". https://www.zynamics.com/binnavi.html. Дата обращения 01.08.2024.

[210] "red-black tree". https://www.geeksforgeeks.org/introduction-to-red-black-tree/. Дата обращения 01.08.2024.

[211] "linux file". https://phoenixnap.com/kb/linux-file-command. Дата обращения 01.08.2024.

[212] "nvd". Available: https://nvd.nist.gov/. Дата обращения 01.08.2024.

[213] "mitre". https://cve.mitre.org/. Дата обращения 01.08.2024.

[214] "4pane". https://www.4pane.co.uk/. Дата обращения 01.08.2024.

[215] "bzip2". https://sourceware.org/bzip2/. Дата обращения 01.08.2024.

[216] "doublecmd". https://doublecmd.sourceforge.io/. Дата обращения 01.08.2024.

[217] "adaway". https://adaway.org/. Дата обращения 01.08.2024.

[218] "praat". https://github.com/praat/praat. Дата обращения 01.08.2024.

[219] "flac". https://github.com/xiph/flac. Дата обращения 01.08.2024.

[220] "luajit". https://luajit.org/. Дата обращения 01.08.2024.

[221] "grub". https://www.gnu.org/software/grub/. Дата обращения 01.08.2024.

[222] "moarvm". https://www.moarvm.org/. Дата обращения 01.08.2024.

[223] "minilua". https://github.com/edubart/minilua. Дата обращения 01.08.2024.

[224] "moarvm-dev". https://packages.debian.org/sid/moarvm-dev. Дата обращения 01.08.2024.

[225] "grub-pc". https://packages.debian.org/sid/grub-pc. Дата обращения 01.08.2024.

[226] "enigma". https://packages.debian.org/search?keywords=enigma. Дата обращения 01.08.2024.

[227] "enigma-data". https://packages.debian.org/search?keywords=enigma-data. Дата обращения 01.08.2024.

[228] "grub-pc-bin". https://packages.debian.org/sid/grub-pc-bin. Дата обращения 01.08.2024.

[229] "medit". https://packages.debian.org/buster/medit. Дата обращения 01.08.2024.

[230] "bochs". https://packages.debian.org/unstable/bochs. Дата обращения 01.08.2024.

[231] "bochs-wx". https://packages.debian.org/stable/otherosfs/bochs-wx. Дата обращения 01.08.2024.

[232] "bochs-x". https://packages.debian.org/unstable/bochs-x. Дата обращения 01.08.2024.

[233] "bochs-sdl". https://packages.debian.org/unstable/bochs-sdl. Дата обращения 01.08.2024.

[234] "bochsbios". https://packages.debian.org/unstable/bochsbios. Дата обращения 01.08.2024.

[235] "bximage". https://packages.debian.org/unstable/bximage. Дата обращения 01.08.2024.

[236] "bfs complexity". https://algowiki-project.org/ru/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%B2_%D1%88%D0%B8%D1%80%D0%B8%D0%BD%D1%83_(BFS). Дата обращения 01.08.2024.

[237] "dfs_complexity". https://www.geeksforgeeks.org/time-and-space-complexity-of-depth-first-search-dfs/. Дата обращения 01.08.2024.

[238] "triton". https://triton-library.github.io/. Дата обращения 01.08.2024.

[239] "libssh". https://www.libssh.org/. Дата обращения 01.08.2024.

[240] "klee". https://github.com/klee/klee. Дата обращения 01.08.2024.

[241] "juliet". https://samate.nist.gov/SARD/test-suites/112. Дата обращения 01.08.2024.

[242] "ffmpeg-error-10342". https://trac.ffmpeg.org/ticket/. Дата обращения 01.08.2024.

[243] "radare2 issues 21703". https://github.com/radareorg/radare2/issues/21703. Дата обращения 01.08.2024.

[244] "radare2 issues 21704". https://github.com/radareorg/radare2/issues/21704. Дата обращения 01.08.2024.

[245] "radare2 issues 21705". https://github.com/radareorg/radare2/issues/21705. Дата обращения 01.08.2024.

[246] "openssl issues 20870". https://github.com/openssl/openssl/issues/20870. Дата обращения 01.08.2024.

[247] "fork server". https://github.com/mirrorer/afl/blob/master/docs/technical_details.txt. Дата обращения 01.08.2024.

[248] "antlr". https://www.antlr.org/. Дата обращения 01.08.2024.

[249] "crusher". https://www.ispras.ru/technologies/crusher/. Дата обращения 01.08.2024.

[250] "JAR". https://docs.oracle.com/javase/8/docs/technotes/guides/jar/jarGuide.html. Дата обращения 01.08.2024.

[251] "smartthings". https://www.samsung.com/global/galaxy/apps/smartthings/. Дата обращения 01.08.2024.

[252] "Samsung Electronics Co. Ltd ". https://www.samsung.com. Дата обращения 01.08.2024.

[253] "AFL". https://github.com/google/AFL. Дата обращения 01.08.2024.

[254] "ASM". https://asm.ow2.io/. Дата обращения 01.08.2024.

[255] "darpa". https://www.darpa.mil/. Дата обращения 01.08.2024.

[256] "anxiety". https://www.ispras.ru/projects/instrument_dinamicheskogo_simvolnogo_ispolneniya_programm_anxiety/. Дата обращения 01.08.2024.

[257] "gllvm". https://github.com/SRI-CSL/gllvm. Дата обращения 01.08.2024.

[258] "sarif". https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning/sarif-support-for-code-scanning. Дата обращения 01.08.2024.

[259] "Postgresql". https://www.postgresql.org/. Дата обращения 01.08.2024.

[260] "SQL Server". https://www.microsoft.com/en-us/sql-server. Дата обращения 01.08.2024.

[261] "Oracle". https://www.oracle.com/index.html. Дата обращения 01.08.2024.

[262] "c.-o. DBMS". https://en.wikipedia.org/wiki/Column-oriented_DBMS. Дата обращения 01.08.2024.

[263] "d.-o. DBMS". https://en.wikipedia.org/wiki/Document-oriented_database. Дата обращения 01.08.2024.

[264] "Key Value DBMS". https://redis.io/nosql/key-value-databases/. Дата обращения 01.08.2024.

[265] "Graph DB". https://www.oracle.com/autonomous-database/what-is-graph-database/. Дата обращения 01.08.2024.

[266] "Multi-model DB". https://www.chaossearch.io/blog/why-enterprises-need-a-true-multi-model-platform. Дата обращения 01.08.2024.

[267] "OrientDB". https://orientdb.org/. Дата обращения 01.08.2024.

[268] "orientdb issues". https://discourse.orientdb.org/t/running-into-numerous-stability-issues-on-2-2-x-suggestions/795. Дата обращения 01.08.2024.

[269] "orientdb github issues". https://github.com/orientechnologies/orientdb/issues.

[270] "ArangoDB". https://www.arangodb.com/. Дата обращения 01.08.2024.

[271] "ArangoDB performance". https://www.arangodb.com/performance/. Дата обращения 01.08.2024.

Приложение А

СВИДЕТЕЛЬСТВА О ГОСУДАРСТВЕННОЙ РЕГИСТРАЦИИ ПРОГРАММ ДЛЯ ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2019660800

«Инструмент поиска клонов кода для C/C++ программ «CCD»

Правообладатель: ***Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)***

Авторы: ***Курмангалеев Шамиль Фаимович (RU), Саргсян Севак Сеникович (AM), Варданян Ваагн Геворгович (AM), Иванов Григорий Сергеевич (RU)***

Заявка № **2019619620**
Дата поступления **01 августа 2019 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***13 августа 2019 г.***

Руководитель Федеральной службы по интеллектуальной собственности

Г.П. Ивлиев

РОССИЙСКАЯ ФЕДЕРАЦИЯ

RU 2019660716

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

(12) ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ

Номер регистрации (свидетельства):
2019660716

Дата регистрации: **12.08.2019**

Номер и дата поступления заявки:
2019619670 01.08.2019

Дата публикации: **12.08.2019**

Контактные реквизиты:
+7 (926) 751-81-48,
fireman@ispras.ru

Авторы:
Курмангалеев Шамиль Фаимович (RU),
Мишечкин Максим Владимирович (RU),
Акользин Виталий Владимирович (RU),
Саргсян Севак Сеникович (AM)

Правообладатель:
Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)

Название программы для ЭВМ:
«Модуль направленного фаззинга программ для «ISP-Fuzzer»

Реферат:
Программа предназначена для генерации входных данных, которые приводят к выполнению нужных фрагментов бинарного кода программы. В модуль внедрен механизм взаимодействия со статическим анализом программ, который позволяет находит все пути, приводящие к нужным фрагментам. На основе этих путей производится специальная инструментация целевой программы, которая позволяет модулю считать текущие мутированные данные эффективными, если они позволили приблизится к нужным фрагментам. Каждый раз рассматриваются и, далее, мутируются только те входные данные, которые приблизили к нужным фрагментам/точкам целевой программы.

Язык программирования: C++, Python

Объем программы для ЭВМ: 22,2 Кб

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2019661048

«Инструмент поиска клонов кода для бинарных файлов «BINCCD»

Правообладатель: ***Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)***

Авторы: ***Курмангалеев Шамиль Фаимович (RU), Саргсян Севак Сеникович (AM), Асланян Айк Каренович (AM), Иванов Григорий Сергеевич (RU)***

Заявка № **2019619667**

Дата поступления **01 августа 2019 г.**

Дата государственной регистрации в Реестре программ для ЭВМ ***16 августа 2019 г.***

Руководитель Федеральной службы по интеллектуальной собственности

Г.П. Ивлиев

РОССИЙСКАЯ ФЕДЕРАЦИЯ

RU 2019661047

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

(12) ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ

Номер регистрации (свидетельства):
2019661047

Дата регистрации: **16.08.2019**

Номер и дата поступления заявки:
2019619673 01.08.2019

Дата публикации: **16.08.2019**

Контактные реквизиты:
+7 (926) 751-81-48,
fireman@ispras.ru

Авторы:
Курмангалеев Шамиль Фаимович (RU),
Мишечкин Максим Владимирович (RU),
Акользин Виталий Владимирович (RU),
Саргсян Севак Сеникович (AM)

Правообладатель:
Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)

Название программы для ЭВМ:
«Инструмент фаззинга программ «ISP-Fuzzer»

Реферат:
Программа предназначена для динамического анализа программ в целях обнаружения дефектов. Программа периодически мутирует входные данные анализируемой программы и проверяет статус его завершений. Таким образом становится возможно находить такие входные данные, которые приводят программу к зависанию и падению. Инструмент поддерживает множество разных входов: переменные окружения, командную строку, сеть, стандартный вход и т.д. Программа спроектирована таким образом, что в ней легко интегрируются новые функциональные возможности. Благодаря этому в ней внедрены взаимодействие с инструментами: динамического символьного решателя, генераций БНФ грамматик и т.д. Инструмент эффективным образом распараллеливается и может работать на распределенных системах.

Язык программирования: C++, Python, Java

Объем программы для ЭВМ: 22,2 Кб

РОССИЙСКАЯ ФЕДЕРАЦИЯ

RU 2019661049

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

(12) **ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ**

Номер регистрации (свидетельства):
2019661049

Дата регистрации: **16.08.2019**

Номер и дата поступления заявки:
2019619669 01.08.2019

Дата публикации: **16.08.2019**

Контактные реквизиты:
+7 (926) 751-81-48,
fireman@ispras.ru

Авторы:
Курмангалеев Шамиль Фаимович (RU),
Иванов Григорий Сергеевич (RU),
Варданян Ваагн Геворгович (AM),
Асланян Айк Каренович (AM),
Федотов Андрей Николаевич (RU),
Саргсян Севак Сеникович (AM)

Правообладатель:
Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)

Название программы для ЭВМ:
«Инструмент анализа изменений между двумя версиями программы «patchAnalysis»

Реферат:
Инструмент patchAnalysis предназначен для анализа изменений между двумя версиями программы. Инструмент получает на вход два бинарных файла и вычислят изменения между ними. patchAnalysis позволяет находить: изменения в новой версии программы, которые могут содержать ошибки, исправленные ошибки в старой версии программы, вредоносный код и т.д. Инструмент основан на множестве разных эвристик и поиске изоморфных подграфов в паре графов зависимостей программы (ГЗП). На первом этапе из бинарного кода восстанавливается структура программы (граф вызов функций, графы потока управления для каждой функции и т.д.). Далее попарно сравниваются все функции в двух версиях программы. При сравнении используются разные эвристики, а также поиск изоморфных подграфов в представлении ГЗП. После сравнения производится анализ найденных изменений для поиска и локализации возможных ошибок. Инструмент масштабируется и применим для анализа бинарных файлов в несколько сотен мегабайт.

Язык программирования: C++

Объем программы для ЭВМ: 296 Кб

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2020663670

«ISP Genes»

Правообладатель: ***Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)***

Авторы: ***Курмангалеев Шамиль Фаимович (RU), Саргсян Севак Сеникович (AM), Варданян Ваагн Геворгович (AM), Асланян Айк Каренович (AM), Акопян Дживан Андраникович (AM), Арутюнян Мариам Сероповна (AM), Меграбян Матевос Саргисович (AM), Мовсисян Оганнес Мушегович (AM), Саргсян Карен Грачикович (AM), Оганесян Рипсиме Ашотовна (AM)***

Заявка № **2020662800**

Дата поступления **23 октября 2020 г.**

Дата государственной регистрации в Реестре программ для ЭВМ ***30 октября 2020 г.***

Руководитель Федеральной службы по интеллектуальной собственности

Г.П. Ивлиев

РОССИЙСКАЯ ФЕДЕРАЦИЯ

RU **2021665076**

ФЕДЕРАЛЬНАЯ СЛУЖБА
ПО ИНТЕЛЛЕКТУАЛЬНОЙ СОБСТВЕННОСТИ

(12) **ГОСУДАРСТВЕННАЯ РЕГИСТРАЦИЯ ПРОГРАММЫ ДЛЯ ЭВМ**

Номер регистрации (свидетельства):
2021665076

Дата регистрации: **17.09.2021**

Номер и дата поступления заявки:
2021663858 07.09.2021

Дата публикации: **17.09.2021**

Контактные реквизиты:
Тел.: +7(903)700-79-86; e-mail: m.kalugin@ispras.ru

Авторы:
Курмангалеев Шамиль Фаимович (RU),
Асланян Айк Каренович (AM),
Арутюнян Мариам Сероповна (AM),
Оганесян Рипсиме Ашотовна (AM),
Варданян Ваагн Геворгович (AM),
Саргсян Севак Сеникович (AM)

Правообладатель:
Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)

Название программы для ЭВМ:
«LibraryIdentifier»

Реферат:
Программа позволяет производить поиск статически скомпонованных программных компонентов (библиотек) в бинарных файлах ПО. Инструмент обладает возможностью сформировать базу интересующих пользователя библиотек, которая может быть использована для анализа. Такой поиск может производиться с целью выявления использования устаревших версий библиотек, включения кода с нарушением лицензии на использование, кода содержащего критические ошибки. Тип ЭВМ: IBM PC-совмест. ПК; ОС: Linux.

Язык программирования: C++, Python

Объем программы для ЭВМ: 63 КБ

Приложение Б

АКТЫ О ВНЕДРЕНИИ РЕЗУЛЬТАТОВ ДИССЕРТАЦИИ

АКТ

о внедрении результатов диссертационной работы Саргсяна Севака Сениковича «Методы оптимизации алгоритмов статического и динамического анализа программ», представленной на соискание ученой степени доктора технических наук по специальности 2.3.5 «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей»

Результаты диссертационной работы Саргсяна Севака Сениковича «Методы оптимизации алгоритмов статического и динамического анализа программ» внедрены в процесс разработки дистрибутива Alt Linux. Платформа анализа, предложенная автором диссертации, успешно применена для популярных пакетов операционной системы Debian Linux, которые также используются в Alt Linux. В частности, при использовании инструмента поиска неисправленных уязвимостей подтверждено, что найденные ошибки исправлены в кодовой базе соответствующих пакетов Alt Linux. Данный инструмент будет применяться при проверке актуальности версий пакетов дистрибутива и внесения необходимых исправлений, сделанных в сообществах открытого исходного кода. Другие инструменты платформы (инструменты фаззинга и инструменты поиска утечек памяти) применяются при дополнительных проверках базовых пакетов операционной системы в цикле разработки безопасного программного обеспечения.

Председатель совета директоров Смирнов А.В.

ООО «Базальт СПО

ASTRA LINUX®

ООО «РусБИТех-Астра»

Варшавское шоссе д. 26, стр. 11, г. Москва, 117105 • тел. +7 (495) 369 48 16 • E-mail: info@astralinux.ru • www.astralinux.ru • ОГРН 5167746207459

УТВЕРЖДАЮ

Научный руководитель, д.т.н., профессор

П.Н. Девянин

«05» 09 2024 г.

АКТ

о внедрении результатов диссертационной работы Саргсяна Севака Сениковича «Методы оптимизации алгоритмов статического и динамического анализа программ»

Результаты диссертационной работы Саргсяна Севака Сениковича, представленной на соискание ученой степени доктора технических наук по специальности 2.3.5 «Математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей», применяются в работе ГК Астра: разработанная в диссертации платформа анализа апробирована в масштабе всего дистрибутива операционной системы на пакетах Debian Linux, которые в том числе служат основой для построения дистрибутива ОС Astra Linux. Найденные при применении платформы ошибки в системных пакетах (grub2, zstd, passwd и др.) демонстрируют практическую значимость платформы и исправлены в соответствующих версиях пакетов ОС Astra Linux. Применены инструменты поиска исправленных уязвимостей, поиска утечек памяти, инструменты фаззинга, а также комбинированные в рамках платформы инструменты.

Директор департамента анализа безопасности В.Ю. Тележников