На правах рукописи

Арутюнян Мариам Сероповна

Статический анализ исходного и исполняемого кода на основе поиска клонов кода

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

Автореферат
диссертации на соискание ученой степени
кандидата технических наук

Москва 2025

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

Научный руководитель:	**Саргсян Севак Сеникович,** кандидат физико-математических наук, доцент
Официальные оппоненты:	**Кознов Дмитрий Владимирович,** доктор технических наук, профессор Федерального государственного бюджетного образовательного учреждения высшего образования «Санкт-Петербургский государственный университет».
	Павленко Евгений Юрьевич, кандидат технических наук, доцент, доцент Высшей школы кибербезопасности федерального государственного автономного образовательного учреждения высшего образования «Санкт-Петербургский политехнический университет Петра Великого».
Ведущая организация:	Федеральное государственное бюджетное образовательное учреждение высшего образования «Московский государственный университет имени М.В.Ломоносова»

Защита состоится 17 апреля 2025 г. в 13:00 на заседании диссертационного совета 24.1.120.01 при Федеральном государственном бюджетном учреждении науки Институте системного программирования им. В. П. Иванникова Российской Академии Наук по адресу: 109004, г. Москва, ул. А. Солженицына, д. 25.

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

Автореферат разослан «___» _____2025 г.

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

Общая характеристика работы

Актуальность.

В современном обществе программное обеспечение (ПО) играет ключевую роль во всех сферах деятельности: в государственном управлении, экономике, в повседневной жизни и так далее. С постоянным расширением объемов и сложности ПО необходимость в обеспечении его качества, в том числе надежности, безопасности и эффективности, только увеличивается. Одной из техник, необходимых в процессах поддержки жизненного цикла ПО, в том числе нацеленных на повышение качества ПО, является техника выявления клонов программного кода – фрагментов программ, схожих по структуре или функциональности.

Клоны кода могут возникать как в исходном, так и в исполняемом коде. Существуют разные причины их возникновения, включая интеграцию готового кода, неиспользование механизмов абстракции (таких как функции, классы), а также компиляторные оптимизации и обфускацию программы. Различные исследования свидетельствуют о широком распространении клонов кода в программных пакетах.

Клонирование кода может привести к увеличению размера исходного и исполняемого кода программы, усложнению поддержки ПО, возникновению различных ошибок, и т. д. В некоторых случаях копирование фрагментов кода, содержащих уязвимости, может привести к их распространению в других системах. Например, криптографическая библиотека OpenSSL, широко используемая в различных проектах, часто копируется целиком или используется как статически связанная библиотека. Уязвимость HeartBleed (CVE-2014-0160) в этой библиотеке привела к утечке конфиденциальных данных с сотен тысяч веб-сайтов. Проблема остается актуальной, поскольку исправления, внесенные в оригинальную библиотеку, не всегда своевременно применяются к ее скопированным версиям, что создает риски даже спустя годы после обнаружения уязвимости.

Выявление клонов кода также может применяться для решения различных задач оптимизации программ путем идентификации вычислительных шаблонов и их автоматической замены на более эффективные реализации. Примером может служить циклическая проверка избыточности данных, используемых для сопоставления контрольных сумм, с помощью которых можно выявить ошибки в данных, произошедшие при передаче, хранении или копировании.

Уже несколько десятилетий продолжается разработка методов и подходов для поиска клонов кода с участием известных научных групп и специалистов, включая исследовательскую команду JetBrains в России, команду разработчиков CCFinder из Японии, группу под руководством Рейнера Кошке из Университета Бремена и т. д. Непосредственно сама автор со своими коллегами разработала несколько инструментов для поиска клонов кода, предназначенных для различных задач. Однако известные методы и инструменты в основном нацелены были на поиск клонов целых функций.

Клоны кода принято делить на четыре типа, отличающиеся сложностью их выявления. Детальное определение и объяснение которых приводится в обзорном разделе. Многие из существующих методов с высокой точностью находят клоны, отличающиеся только именами переменных и комментариями (типа-1 и типа-2). Однако для обнаружения более сложных клонов (типа-3) существует ряд ограничений, включая низкую точность и полноту. Эти ограничения затрудняют применение данных инструментов для анализа изменений между версиями программ, обнаружения уязвимых фрагментов кода и устаревших версий библиотек. Поиск клонов кода, отличающихся структурой, но с одинаковой семантикой (типа-4), в целях оптимизации не был рассмотрен ни в одной из известных в настоящее время работ.

Таким образом, актуальной является задача разработки новых методов поиска клонов фрагментов кода для исходного и исполняемого кода, а также методов их применения в процессах поддержки жизненного цикла программ, в частности для анализа изменений между версиями программ, идентификации статически связанных библиотек и поиска копий известных уязвимостей. При этом требуется обеспечить высокую точность (больше 90%), полноту (больше 90%) и масштабируемость (анализ десятков миллионов строк исходного кода или соответствующего исполняемого кода за несколько часов). Разработка таких методов не только повысит качество ПО, но и упростит процесс его поддержки и сопровождения, а также снизит риски, связанные с безопасностью и производительностью.

Целью диссертационной работы является разработка и реализация масштабируемых методов поиска клонов исходного и исполняемого кода, применимых в широком классе задач: для нахождения клонов произвольных фрагментов исходного и исполняемого кода, выявления изменений между версиями программ, идентификации статически связанных библиотек, обнаружения клонов известных уязвимостей, а также для оптимизации программ.

Для достижения поставленной цели были сформулированы и решены следующие задачи:

1. Разработать и реализовать унифицированный метод нахождения клонов фрагментов исходного и исполняемого кода.
2. Разработать и реализовать метод оптимизации фрагментов кода, вычисляющих циклическую проверку избыточности с последующей заменой на более эффективную реализацию с учетом целевой аппаратной архитектуры.
3. Разработать и реализовать двухэтапный метод выявления изменений между версиями программ, который сочетает метрический подход с разработанным методом поиска клонов кода на основе графов, содержащих зависимости управления и зависимости данных, для сопоставления функций в формате «многие ко многим» и отображения измененных инструкций.
4. На основе разработанного метода поиска клонов фрагментов кода разработать и реализовать методы идентификации статически связанных библиотек и поиска копий известных уязвимостей.

Методология и методы исследования.

Для решения задач, поставленных в диссертационной работе, использовались методы теории графов, теории компиляции и обратной инженерии.

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

1. Унифицированный метод поиска клонов произвольных фрагментов кода в исходном и исполняемом коде, основанный на графах, содержащих зависимости управления и зависимости данных программы, способного анализировать десятки миллионов строк исходного и соответствующего исполняемого кода.
2. Метод оптимизации программ, использующих вычисление циклической проверки избыточности (ЦПИ), при помощи поиска клонов и подстановки эффективных реализаций ЦПИ с учетом аппаратной платформы.
3. Двухэтапный метод выявления изменений между версиями программ, который сочетает метрический подход с разработанным методом поиска клонов кода на основе графов, содержащих зависимости управления и зависимости данных, для сопоставления функций в формате «многие ко многим» и отображения измененных инструкций.

4. Методы идентификации статически связанных библиотек и поиска копий известных уязвимостей с использованием разработанного метода поиска клонов фрагментов кода.

Теоретическая и практическая значимость.

Теоретическая значимость данной диссертационной работы заключается в разработанных методах и алгоритмах анализа кода, включая методы поиска клонов фрагментов кода, нахождения изменений между версиями программ и идентификации используемых библиотек, которые в ходе экспериментального тестирования показали свое превосходство по сравнению с существующими решениями.

Практическая значимость заключается в использовании разработанных методов в платформе анализа кода GenesISP, в статическом анализаторе BinSide и в компиляторе GCC. GenesISP внедрен в цикл разработки ПО в ИСП РАН и ЦППТ РАУ с 2021 года. Реализованные методы могут эффективно применяться в жизненном цикле разработки безопасного ПО, что покрывает многие из требований ГОСТ Р 56939-2024 «Разработка безопасного программного обеспечения. Общие требования» и «Методики выявления уязвимостей и недекларированных возможностей в программном обеспечении» ФСТЭК Российской Федерации.

Гранты и контракты.

Исследования по теме диссертации проводились в рамках научных проектов, поддержанных следующими грантами: совместным грантом КН Армении и РФФИ 20RF-033 «Разработка и реализация масштабируемых методов анализа современных операционных систем» и грантом КН Армении 21SCG-1B003 «Разработать и реализовать систему анализа безопасности и сертификации программного обеспечения». Результаты диссертационной работы также были использованы в рамках гранта РФФИ 18-07-01153, «Исследование и разработка методов поиска ошибок на основе метода поиска клонов кода».

Апробация работы.

Основные результаты работы докладывались на следующих конференциях:

1. Ежегодная научная сессия СНО ЕГУ 2016, Ереван, Армения, 2016 г.
2. XIII Годичная научная конференция Российско-Армянского университета, Ереван, Армения, 2018 г.
3. Международная конференция «Иванниковские чтения», Великий Новгород, Россия, 2019 г.

4. XIV Годичная научная конференция Российско-Армянского университета, Ереван, Армения, 2019 г.
5. XXVIII Международная конференция студентов, аспирантов и молодых ученых «Ломоносов» 2021 г.
6. Международная конференция «Иванниковские чтения», Нижний Новгород, Россия, 2021 г.
7. Международная конференция «GNU Tools Cauldron 2023», Кембридж, Великобритания, 2023 г.
8. Международная конференция «VALID 2024», Венеция, Италия, 2024 г.
9. Международная конференция «FOSDEM 2025», Брюссель, Бельгия, 2025 г.

Публикации.

Основные результаты по теме диссертации изложены в 12 печатных изданиях [1-12], в том числе, 4 научные статьи в рецензируемых журналах, входящих в перечень рекомендованных ВАК РФ. Работы [4], [5], [6] и [9] индексированы в Scopus. Статьи [11] и [12] опубликованы в журналах, входящих в первый квартиль SJR. Получено 2 свидетельства о регистрации программ для ЭВМ [13][14].

Личный вклад.

Полученные результаты, выносимые на защиту, являются личной работой автора. В опубликованных совместных трудах задачи формулировались и исследовались совместно с соавторами при активном участии соискателя. В работах [1] и [8] все результаты были получены лично автором. В статьях [2] и [4] автором разработаны алгоритм сопоставления функций, анализ характера изменений в новых версиях исполняемых файлов и структура инструмента. В статье [3] автором разработаны и написаны разделы 4 и 5, в [5] – 1, 2, 3; в [6] - 4.B, 4.C, 4.D, 4.E; в [7] - в [9] - 1, 3, 4, 5; в [10] – 1, 2, 3, 4.1, 5, 6, 7; в [11] – 1, 2, 4-10; в [12] – 1, 2, 4-10. Разработка зарегистрированных программных систем [13] и [14] проводилась при личном участии автора.

Объем и структура диссертации.

Диссертация состоит из введения, пяти глав, заключения. Полный объем диссертации составляет 133 страницы, включая 27 рисунков и 24 таблицы. Список литературы содержит 202 наименования.

Краткое содержание работы.

Во введении представлена цель работы, обоснована ее актуальность, определены понятия и сформулированы основные результаты работы, а также их научная новизна и практическая значимость.

В первой главе приводится обзор работ, которые имеют отношение к теме диссертации. Рассматриваются современные методы поиска клонов исходного и исполняемого кода, а также методы сравнения программ и идентификации библиотек.

В разделе 1.1 приведены определения и примеры клонов кода. Клоны кода делятся на четыре типа. Для исходного кода тип-1 — идентичные фрагменты кода, отличающиеся только форматированием (пробелы, табуляции) и комментариями. Тип-2 включает тип-1 и допускает различия в именах переменных, значениях и типах данных. Для исполняемого кода тип-1 — идентичные фрагменты, тип-2 включает тип-1 и допускает различия только в значениях данных и именах регистров. Для исходного и исполняемого кода тип-3 расширяет тип-2 и позволяет добавление или удаление операций/инструкций, а тип-4 — фрагменты, выполняющие одинаковые вычисления, но с разным набором инструкций.

В главе также описаны инструменты и основные подходы нахождения клонов кода, включая методы, основанные на текстах, токенах, метриках, деревьях и графах. Также рассматриваются гибридные методы, которые объединяют несколько подходов.

Каждый из перечисленных подходов имеет свои преимущества и недостатки. Текстовые методы наиболее эффективны для обнаружения клонов типа-1, так как они сравнивают строки кода напрямую, но их невозможно применить для типа-2, типа-3 и типа-4 из-за чувствительности к изменениям форматирования или структуры. Методы, основанные на токенах, лучше подходят для нахождения клонов типа-2, так как они игнорируют форматирование и комментарии, но могут быть ограниченно полезны при анализе типа-3 и типа-4. Методы, основанные на метриках, могут быть полезны для предварительной оценки наличия клонов всех типов, но их точность недостаточна для детального анализа. Подходы, основанные на анализе деревьев, такие как анализ абстрактных синтаксических деревьев (АСД), обеспечивают высокую точность в обнаружении клонов типа-2 и могут выявлять некоторые клоны типа-3, однако обладают низкой точностью. Наконец методы, основанные на графах, например граф зависимостей программы, наиболее подходят для обнаружения клонов типа-3 и типа-4, так как они учитывают функциональную эквивалентность, но их реализация сложна и требует значительных вычислительных ресурсов. Гибридные методы, объединяющие разные подходы, способны дать лучшие результаты, но существующие реализации пока не достигают необходимой точности и эффективности для нахождения клонов типа-3 и типа-4.

Таким образом, выбор подхода к обнаружению клонов кода должен основываться на конкретных требованиях проекта, таких как объем кода, используемый язык программирования, доступные вычислительные ресурсы и цели анализа. Каждый подход имеет свои сильные и слабые стороны, и в некоторых случаях может быть целесообразным комбинировать несколько техник для достижения наилучших результатов.

В разделе 1.2 описаны существующие работы нахождения клонов кода, вычисляющих ЦПИ, и автоматически заменяющих найденный фрагмент на более эффективную реализацию. В ходе исследования была найдена только одна работа, связанная с распознаванием и заменой ЦПИ для архитектуры RISC-V в GCC, представленная Дж. Реннеком и Дж. Бенистоном в 2022 году. Этот метод основывается на последовательном сопоставлении базовых блоков программы с заранее определенным шаблоном вычисления ЦПИ. Если на любом этапе шаблон не совпадает, анализ прерывается. При успешном совпадении всех этапов оригинальный код ЦПИ заменяется табличной реализацией. Однако подход имеет ряд ограничений: он поддерживает только реверсивные ЦПИ 8, 16 и 32 для 8-битных данных, требует строго определенного шаблона вычисления и может быть чувствителен к изменениям, вызванным оптимизациями компилятора.

В разделе 1.3 описаны существующие работы сравнения программ. Разные работы сравнивают программы на разных уровнях: на уровне инструкций, базовых блоков, функций или целых программ. В этом разделе описаны доступные подходы и инструменты, их преимущества и ограничения. Например, методы сравнения на уровне инструкций обеспечивают высокую точность для исполняемого кода, но плохо масштабируются на уровне крупных программ. Сравнение на уровне базовых блоков и функций позволяет работать с более крупными структурами, но при этом теряется детализация анализа.

Автоматическое обнаружение изменений между двумя версиями программы имеет множество практических применений, таких как отслеживание эволюции кода, контроль версий и обеспечение безопасности. Существующие инструменты обычно сопоставляют функции первой версии программы с функциями второй, но их эффективность снижается в случаях, когда в конкретной версии программы присутствуют похожие функции. В таких случаях предлагается использовать сопоставление группы функций первой программы с группой функций второй программы.

В разделе 1.4 детально описаны инструменты идентификации статически связанных библиотек или их функций в исполняемых файлах. Анализируются

подходы, используемые в этих инструментах, а также их применимость для задач идентификации библиотек в различных архитектурах. Ни один из анализируемых инструментов не удовлетворяет требованиям для эффективного поиска статически связанных библиотек из-за ограничений в функциональности, точности и универсальности.

В разделе 1.5 приведены выводы исследования. На основе проведенного обзора можно сделать следующие выводы. Существующие методы нахождения клонов кода демонстрируют высокую точность и полноту (больше 90%) для клонов типа-1 и типа-2, однако для клонов типа-3 оба показателя падают. Также эти методы нацелены на нахождения клонов целых функций, а не произвольных фрагментов кода. Некоторые методы применяются для автоматического рефакторинга кода, однако, они не ориентированы на оптимизацию кода. Существующие методы поиска клонов кода применимы либо для исполняемого кода, либо для исходного кода. Такое ограничение имеют также существующие методы сравнения двух версий программ. Помимо этого, они обеспечивают сопоставление функций один к одному, однако в определенных случаях нужно обеспечить сопоставление многие ко многим. Существующие методы идентификации статически связанных библиотек в большинстве случаев основаны на поиске константных строк и чисел, не учитывая ряд свойств кода.

Таким образом, для достижения цели предлагается использовать усовершенствованный метод на основе графов зависимостей программы, который обеспечивает высокую точность и полноту. Для нахождения клонов фрагментов типа-4, вычисляющих циклическую проверку избыточности, предлагается многоуровневый метод, который позволяет со стопроцентной точностью находить такие фрагменты и заменять их оптимизированными версиями. При сравнении версий программ можно использовать гибридный подход, включающий подходы на основе метрик и графов, который сочетает высокую точность и скорость. Метрики с использованием машинного обучения будут применяться для быстрой фильтрации очевидно несхожих функций, что уменьшит объем данных для дальнейшего анализа. Далее алгоритм на основе графов позволит с высокой точностью выявлять сложные соответствия в оставшейся части. Такое сочетание подходов обеспечит баланс между производительностью и качеством анализа.

Во второй главе описывается унифицированный метод поиска клонов произвольных фрагментов кода в исходном и исполняемом коде, основанный на графах, содержащих зависимости управления и зависимости данных программы. Фрагмент кода и целевая программа могут быть представлены как

в виде исходного кода, так и в виде исполняемого кода. Для исполняемого кода фрагмент задается именем функции, начальным и конечным адресами, а для исходного кода – именем функции, начальным и конечным номерами строк. Важно отметить, что предполагается, что предоставленный фрагмент кода находится в пределах одной функции. Сначала исходный/исполняемый код преобразуется в промежуточное представление. Далее генерируются графы зависимостей программы (ГЗП) и на основе этих графов выполняется поиск клонов фрагмента кода. Также задается минимальный процент сходства, и клоны выявляются, если удовлетворяется следующее условие:

$$\frac{\text{matchedVerticesCount}}{\text{fragmentPDGsVerticesCount}} \times 100\% \geq \text{similarity},$$

где matchedVerticesCount - количество сопоставленных вершин, fragmentPDGsVerticesCount - количество вершин ГЗП фрагмента и similarity - входной минимальный процент сходства.

В разделе 2.1 описывается метод нахождения клонов фрагмента кода, состоящий из двух этапов (Рисунок 1). На первом этапе строятся ГЗП для указанного фрагмента и всех функций в проекте. На втором этапе выполняется анализ и поиск соответствий между ГЗП фрагмента и целевым проектом.

Рисунок 1. Схема нахождения клонов фрагмента кода.

В разделе 2.1.1 приводится определение ГЗП и способы его генерации. ГЗП генерируются для указанного фрагмента и всех функций целевого проекта. Вершины ГЗП представляют собой инструкции промежуточного представления, а ребрам соответствуют зависимости по управлению и по данным между инструкциями. Реализация генерации ГЗП различается для исполняемого и исходного кода поскольку используются разные промежуточные представления. Для генерации ГЗП указанного фрагмента

создается ГЗП для всей функции, содержащий фрагмент и извлекается порожденный подграф, включающий в себя все инструкции указанного фрагмента. Для простоты будем называть его ГЗП фрагмента. Построенные графы затем используются на следующем этапе, в котором инструкции из указанных фрагментов сопоставляются со всеми инструкциями в функциях всего проекта.

В разделе 2.2 описывается процесс сопоставления вершин ГЗП фрагмента с вершинами ГЗП для каждой функции. Алгоритм состоит из двух фаз: формирования множества «начально» сопоставленных пар вершин и их итеративного расширения. Основная особенность метода - возможность обнаружения множественных клонов фрагмента в рамках одной функции. Выбор «начальных» вершин осуществляется по одному из трех критериев: отсутствие родительских узлов, максимальное число входящих зависимостей или максимальное количество соответствующих инструкций промежуточного представления.

Поиск клона начинается с сопоставления пары «начальных» вершин двух ГЗП, а затем ряд процессов пытаются сопоставить оставшиеся вершины ГЗП. Если новое сопоставление уже невозможно, то до этого сопоставленные вершины считаются единым клоном, после чего начинается поиск нового клона путем сопоставления другой пары «начальных» вершин.

Сначала вершины сопоставляются временно, затем, после проверки совместимости подтверждается их совместимость или они считаются несовместимыми.

Алгоритм временного сопоставления включает пять процессов сопоставления, которые применяются последовательно в определенном порядке до тех пор, пока одна пара вершин не будет сопоставлена с использованием одного из этих процессов.

В разделе 2.3 приведены детали реализации метода нахождения клонов фрагмента кода (инструмент FCD). Инструмент поддерживает языки программирования C, C++, Java, Objective-C. Он может анализировать как полностью компилируемые программы, так и синтаксически корректные, но не компилируемые. Кроме того, для анализа исполняемого кода инструмент поддерживает распространенные архитектуры, среди которых x86, x86-64, ARM и PowerPC.

В разделе 2.4 описывается тестовая система для оценки качества инструментов нахождения клонов кода, которая автоматизирует генерацию тестов и вычисление нескольких метрик. Также приведены результаты работы инструмента FCD, полученные с использованием описанной тестовой системы. Инструмент тестировался с использованием реализованной системы

тестирования на проектах OpenSSL, JasPer, c-ares и rsync для всех функций. В таблицах 1 и 2 представлены результаты обнаружения клонов исходного и исполняемого кода. Результаты были усреднены по указанным процентам сходства 100%, 90%, 80% и 70% в тестовой системе. В частности, тестовая система использовалась четыре раза для целевых проектов, каждый раз с одним из указанных процентов сходства. В проведенном эксперименте инструмент показал 100%-ную точность и полноту, когда сгенерированные клоны были на 100% похожи, что указывает об успешном обнаружении всех клонов типа-1 и типа-2 без каких-либо пропусков или ошибок. Кроме того, FCD демонстрировал высокую точность при более низких уровнях сходства, что отражено в усредненных результатах в таблицах.

Тестирование FCD для анализа исходного кода показало, что инструмент достигает высоких значений точности и полноты для большинства проектов. Например, анализ кода c-ares 1.15.0 занял всего 10 секунды, при этом точность составила 97.5%, а полнота — 95.2%. Для более крупных проектов, таких как openssl 1.0.2t (310922 строки кода), время анализа увеличилось до 27 секунд, однако качество поиска клонов осталось высоким (точность 97%, полнота 95.1%). Это подтверждает эффективность FCD при поиске клонов в исходном коде с минимальными временными затратами.

Проект	**Количество строк кода C/C++**	**Скорость FCD**	**Точность (%)**	**Полнота (%)**	**F1-мера (%)**
c-ares 1.15.0	61087	10c	97.5	95.2	96.3
jasper 1.900.1	28279	15c	95.4	93	94.2
openssl 1.0.2t	310922	27c	97	95.1	96
rsync 3.1.3	44832	26c	96	91.9	93.9

Таблица 1. Поиск клонов с низким сходством в исходном коде

Таблица 2 содержит результаты тестирования FCD для анализа исполняемого кода тех же проектов, скомпилированных для различных архитектур (x86-64, x86, ARM). Результаты показывают, что инструмент демонстрирует высокую эффективность на всех архитектурах. Например, для openssl 1.0.2 на архитектуре x86-64 точность составила 99.9%, а F1-мера — 99%. Аналогичные показатели сохраняются и для других проектов, что подтверждает стабильность работы FCD на различных проектах. Отличия в значениях оценок между архитектурами обусловлены особенностями

компиляции и размером исполняемых файлов, но в целом инструмент сохраняет высокую точность и полноту.

Проект	Размер исполняемого кода	Скорость FCD	Архитектура	Точность (%)	Полнота (%)	F1-мера (%)
c-ares 1.15.0	86 КБ	41с	x86-64	98.9	95.6	97.2
c-ares 1.15.0	96 КБ	43с	x86	97.9	93.4	95.6
c-ares 1.15.0	146 КБ	49с	ARM	98.9	95.6	97.2
jasper 1.900.1	1.5 МБ	3м 5с	x86-64	96	92.1	94
jasper 1.900.1	368 КБ	2м 1с	x86	95	90	92.4
jasper 1.900.1	478 КБ	2м 8с	ARM	94.1	89.8	91.9
openssl 1.0.2	536 КБ	1м 10с	x86-64	99.9	98.1	99
openssl 1.0.2	507 КБ	57с	x86	98.8	95.8	97.3
openssl 1.0.2	634 КБ	1м 25с	ARM	97.9	95.6	96.7
rsync 1.3.2	1.7 МБ	3м 34с	x86-64	96	91	93.4
rsync 1.3.2	1.6 МБ	3м 21с	x86	94.9	88.9	91.8
rsync 1.3.2	1.8 МБ	3м 58с	ARM	94.1	88.8	91.4

Таблица 2. Поиск клонов с низким сходством в исполняемом коде.

В разделе 2.5 приведены результаты масштабируемости инструмента. Для этого был проведен поиск клонов фрагментов кода на проектах имеющих миллионы строк кода. Тест проводился на компьютере с четырехъядерным процессором Intel Core i5-3470 3600 МГц и 16 ГБ оперативной памятью. Средняя скорость обработки составляет один миллион строк кода за две минуты для одного фрагмента кода с десятью строками исходного кода.

В разделе 2.6 приведены результаты сравнения инструмента FCD с инструментами, разработанными С. Саргсяном и А. Асланяном в рамках их диссертационных работ, по сравнению которых FCD показал свое превосходство. На тестовой системе инструмент, разработанный С. Саргсяном (находит клоны исходного кода и только для целых функций) имеет в среднем 99% точности и 9% полноты. А инструмент, разработанный А. Асланяном (находит клоны исполняемого кода), имеет в среднем 80% точности и 45% полноты.

В разделе 2.7 приведена оценка инструмента FCD на базе BigCloneBench. Набор данных BigCloneBench представляет собой обширную коллекцию из более чем 8 миллионов помеченных пар клонов. Типы клонов разделяются на следующие категории - тип-1, тип-2, и тип-3, с двумя подтипами: слабо измененный тип-3, сильно измененный тип-3, где сходство находится в диапазонах [90%,100%) и [70%,90%).

FCD был оценен на данном наборе, обеспечив тщательную и детальную оценку широкого спектра типов клонов, в результате чего было установлено, что полнота обнаружения клонов превышает 91% (Таблица 3).

Тип клона	**CCFinderX**	**PMD/CPD**	**SourcererCC**	**Deckard**	**ConQat**	**iClones**	**NiCad7**	**FCD**
Тип-1	100%	100%	100%	60%	62%	100%	100%	100%
Тип-2	93%	94%	98%	58%	60%	57%	100%	100%
Слабо измененный Тип-3	62%	71%	93%	62%	57%	84%	98%	99%
Сильно измененный Тип-3	15%	21%	61%	31%	49%	33%	73%	91%

Таблица 3. Результаты оценки на BigCloneBench.

Кроме того, таблица 3 демонстрирует результаты других инструментов, которые обнаруживают клоны типа-3 и имеют поддержку кода, написанного на языке Java. Большинство инструментов обнаружили клоны типа-1 и типа-2 с высокой полнотой. Однако в случаях клонов слабо измененного типа-3 и сильно измененного типа-3 полнота значительно падает. FCD демонстрирует лучшие показатели полноты 99% и 91%, соответственно. Второй лучший

результат демонстрирует Nicad с 98% и 73% полнотой, соответственно. Таким образом FCD демонстрирует более высокие показатели полноты.

В разделе 2.8 приведены выводы по разработанному унифицированному методу поиска клонов произвольных фрагментов кода.

В третей главе описан метод оптимизации программ, использующих вычисление циклической проверки избыточности (ЦПИ или Cyclic Redundancy Check, CRC), при помощи поиска клонов и подстановки эффективных реализаций ЦПИ с учетом аппаратной платформы. В данном случае речь идет о нахождения клонов фрагментов типа-4. Для оптимизации клон с неоптимальной реализацией заменяется на более эффективную реализацию.

Метод начинается с обнаружения потенциальных реализаций побитового ЦПИ. Затем он вычисляет различные параметры и использует специально разработанную технику символического выполнения для проверки идентифицированных кандидатов. Далее метод заменяет фрагменты кода реализаций побитового ЦПИ на более эффективные альтернативы, такие как реализации на основе таблиц поиска, реализации, использующие инструкцию умножения без переноса (УБП), или прямые реализации с использованием инструкции ЦПИ.

В разделе 3.1 описывается ЦПИ как широко используемый метод обнаружения ошибок, который интегрирован в цифровые сети, протоколы связи и системы хранения данных. Основное назначение ЦПИ – обнаружение случайных изменений цифровых данных, вызванных шумами в каналах передачи. Объясняется процесс вычисления контрольных значений, которые добавляются к блокам данных и используются для проверки целостности при получении данных. Приводятся примеры применения ЦПИ в различных системах.

В разделе 3.2 описывается алгоритм вычисления ЦПИ как метод обнаружения ошибок, основанный на полиномиальном делении. Далее рассматриваются различные типы реализаций алгоритма ЦПИ, включая побитовые и оптимизированные подходы: на основе таблиц и УБП. Также описываются вариации в порядке бит, такие как прямой и обратный порядок.

В разделе 3.3 описывается предлагаемый метод распознавания ЦПИ (Рисунок 2), который начинается с обнаружения потенциальных реализаций побитового ЦПИ путем проверки различных особенностей и использует индивидуальную технику символического выполнения для верификации выявленных кандидатов.

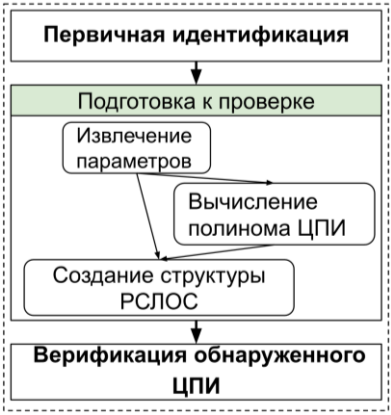

Рисунок 2. Распознавание ЦПИ.

В разделе 3.4 описывается разработанная техника символического выполнения. Ее основная концепция заключается в выполнении процесса, похожего на символическое выполнение, но на уровне битов, где каждый бит переменных программы анализируется индивидуально. Вместо использования существующих универсальных движков символического выполнения был разработан специализированный и легкий движок. Кроме того, традиционные решатели SMT также не используются, поскольку уравнения, генерируемые в ходе этого процесса, просты и могут быть решены без их помощи.

В разделе 3.5 описывается процесс идентификации ЦПИ. Он состоит из пяти шагов: первичная идентификация кандидатов ЦПИ, извлечение параметров, вычисление полинома ЦПИ, создание структуры регистра сдвига с линейной обратной связью (РСЛОС, linear feedback shift register, LFSR) и верификация обнаруженного ЦПИ.

В разделе 3.5.1 описывается алгоритм первичной идентификации кандидатов для вычисления ЦПИ. Он выполняет быстрые проверки, включая анализ циклов с фиксированным числом итераций (8, 16, 32, 64), наличие операций XOR и SHIFT, а также их зависимость от переменных и ветвлений. Алгоритм выделяет потенциальные реализации ЦПИ, исключая фрагменты, не связанные с вычислением ЦПИ.

В разделах 3.5.2 и 3.5.3 описывается этап извлечения информации о нескольких параметрах ЦПИ и процесс вычисления полинома ЦПИ для использования на последующих этапах.

В разделе 3.5.4 описывается процесс создания структуры РСЛОС для верификации выявленных реализаций ЦПИ. Процесс вычисления ЦПИ с

использованием РСЛОС заключается в сдвиге значений РСЛОС и применении операций XOR к данным и полиномам в зависимости от их значений. Итерации работы РСЛОС эквивалентны циклам побитового вычисления ЦПИ, что позволяет определить, соответствует ли цикл вычислению контрольной суммы. Структура РСЛОС создается с помощью формул для прямого и обратного порядка бит. Формулы учитывают операции XOR с данными, если они применяются на каждой итерации. Это позволяет гибко адаптировать структуру РСЛОС для различных реализаций ЦПИ.

В разделе 3.5.5 описывается метод верификации обнаруженного кандидата ЦПИ с использованием символического выполнения и ранее собранной информации.

Алгоритм состоит из следующих шагов:

1. **Инициализация:** запускается символическое выполнение с первой инструкции цикла, создается объект состояния для отслеживания значений переменных и условий. Переменные инициализируются исходными или символическими значениями.
2. **Символическое выполнение цикла:** каждая инструкция выполняется символически для всех возможных путей. Если обнаруживается инструкция, результат которой используется вне цикла и не относится к ЦПИ, верификация завершается отрицательно, так как цикл выполняет дополнительные вычисления.
3. **Проверка структуры:** после каждой итерации проверяется, что значения ЦПИ соответствуют ожидаемой структуре, определенной через РСЛОС.

В разделе 3.6 представляется алгоритм замены побитового ЦПИ на более эффективную реализацию. Для данной цели реализовано три метода: метод на основе таблиц, метод на основе УБП и прямое использование инструкции CRC32. Последние два метода применимы, если целевая архитектура поддерживает соответствующие инструкции.

В разделах 3.7 и 3.8 приведены детали реализации и перечислены неподдерживаемые случаи.

В разделе 3.9 представлены результаты двух экспериментов. Первое исследование направлено на выявление ЦПИ в открыто доступном ПО. Второй эксперимент оценивает прирост производительности, достигаемый за счет замены побитовой реализации ЦПИ альтернативными реализациями.

В разделе 3.9.1 приведены результаты первого эксперимента, подтверждающие, что все найденные реализации являются ЦПИ. Алгоритм поиска сработал без ошибок, не выдав ни одного ложноположительного

результата. В пакетах, распространяемых Fedora, он выявил 18 случаев вычисления побитовой реализации ЦПИ, в ядре Linux – 2, а на бенчмарке Coremark была найдена одна реализация.

В разделе 3.9.2 представлены результаты оценки производительности после замены фрагментов кода на оптимизированные реализации ЦПИ. Тестирование проводилось на x86-64 (Intel Core i7-3770 @ 3.40GHz), AArch64 (Raspberry Pi 4 Model B Rev 1.5) и RISC-V (StarFive VisionFive 2 v1.38), с использованием побитовой реализации, табличного метода, метода на основе УБП и инструкций CRC32. На архитектуре x86-64 среднее улучшение производительности при использовании метода на основе таблиц составляет 54%, на основе УБП — 43%. На архитектурах RISC-V и AArch64 среднее улучшение при использовании метода на основе таблиц составляет 73% и 50%, соответственно. При использовании инструкции CRC32 на архитектуре x86-64 улучшение достигает 92%, а на AArch64 — 93%. На процессоре с архитектурой RISC-V инструкция CRC32 не поддерживается. Метод на основе УБП не тестировался на RISC-V и AArch64 из-за отсутствия поддержки в вышеупомянутых процессорах.

В разделе 3.10 приведены выводы по разработанному методу оптимизации программ на основе поиска клонов.

В четвертой главе рассматривается двухэтапный метод выявления изменений между версиями программ, который сочетает метрический подход с разработанным методом поиска клонов кода на основе графов, содержащих зависимости управления и зависимости данных, для сопоставления функций в формате «многие ко многим» и отображения измененных инструкций.

В разделе 4.1 рассматривается метод сравнения двух программ состоящий из двух основных этапов: генерация ГЗП и графов вызовов функций (ГВФ), а также поиск схожих функций с использованием полученных графов.

Метод получает в ходе две версии программы и минимальный процент сходства, а затем возвращает набор списков сопоставленных функций. На первом этапе генерируются графы вызовов функций и ГЗП для каждой функции целевой программы. На втором этапе сопоставляются схожие функции. В процессе этого этапа для каждой функции рассчитываются хеш-значения для предварительного сопоставления функций, после чего используется алгоритм нахождения клонов фрагмента кода, описанный в главе 2.

1. На основе ранее сгенерированных ГЗП вычисляются семь хешей. Для вычисления хешей используется модифицированная версия SimHash,

которая является локально-чувствительным хешированием. Ниже перечислены хеши вычисляемые модифицированным SimHash:

1. Хеш на основе инструкций ассемблера/выражений исходного кода.
2. Хеш на основе кодов операций ассемблера/операций исходного кода.
3. Хеш на основе кода операции вершины и соседних ребер ГЗП.
4. Хеш, используя MD-index, основанный на количестве предшествующих и последующих вершин базовых блоков ГЗП.
5. Хеш, используя MD-index, основанный на количестве предшествующих и последующих вершин каждой вершины ГЗП.
6. Хеш на основе сильно связанных компонентов ГЗП.
7. Хеш на основе констант.

Каждый из вышеперечисленных хешей имеет размер 64 битов. После вычисления всех хешей они попарно сравниваются, используя коэффициент Отиаи и сходство по длине.

Коэффициент Отиаи для двух векторов A и B размерности n вычисляется как:

$$\frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}.$$

Сходство по длине вычисляется следующим образом:

$$\frac{\min(\sum_{i=1}^{n} {A_i}^2, \sum_{i=1}^{n} {B_i}^2)}{\max(\sum_{i=1}^{n} {A_i}^2, \sum_{i=1}^{n} {B_i}^2)}.$$

Таким образом, для каждой пары функций вычисляются 14 значений – m_1, m_2, …, m_{14}, на основе которых они могут предварительно сопоставляться. Для каждого значения определяется вес, так как некоторые хеши имеют более высокий приоритет по сравнению с другими. Пары функций предварительно сопоставляются, если

$$\frac{\sum_{i=1}^{14} m_i w_i}{14} \geq \text{процента сходства}.$$

Веса w_i были установлены с помощью методов машинного обучения. Для обучения создавались пары функций с заданным процентом сходства, после чего вычислялись их хеш-значения.

2. Для всех предварительно сопоставленных функций вычисляются общие части используя алгоритм FCD. Назначим количество общих инструкций промежуточного представления - *commonInstructionsCount,* количество инструкций промежуточного

представления первой функции - *instructionsCount1,* количество инструкций промежуточного представления второй функции - *instructionsCount2*. Функции сопоставляются, если

$$\frac{2\times commonInstructionsCount}{instructionsCount1+\text{instructionsCount2}} \geq \text{процента сходства.}$$

В разделе 4.2 приведены детали реализации метода (инструмент BCC).

В разделе 4.3 приведены результаты разработанного инструмента, который был протестирован на некоторых широко распространенных проектах, скомпилированных для разных архитектур.

В таблице 4 приведены результаты работы инструмента на некоторых версиях проекта Coreutils. В нем указана информация о версии, используемом компиляторе, оптимизациях и архитектуре, а также о размерах программ, точности, полноте и времени анализа. В проведенных экспериментах точность и полнота оставались высокими даже при различных параметрах оптимизаций и компиляторов. Например, комбинации компиляторов Clang и GCC с разными оптимизациями показывали точность в пределах 90-99%.

Версия 1, компилятор, оптимизация, архитектура (бит)	**Размер 1 (МБ)**	**Версия 2, компилятор, оптимизация, архитектура (бит)**	**Размер 2 (МБ)**	**Точность (%)**	**Полнота (%)**	**Время анализа (с.)**
8.30_clang_O0_64	16	8.30_clang_O0_64	7	98.8	98.3	38
8.30_gcc_O0_64	15	8.30_gcc_O0_64	6.3	98.8	98.3	27
8.30_gcc_O0_64	15	8.30_clang_O0_64	7	94.9	89.4	36
8.29_gcc_O0_64	14.8	8.30_gcc_O0_64	6.3	97.8	98.2	28
8.30_gcc_O0_32	13	8.30_gcc_O0_32	5.6	98.4	97.9	33
8.30_gcc_O2_32	19	8.30_gcc_O2_32	5.3	97.9	88	27

Таблица 4. Результат работы инструмента на разных версия Coreutils.

Также представлены результаты сравнения инструментов BCC, BinDiff и Diaphora, которые показали превосходство BCC. В среднем F1-мера для BCC составляет 85,6%, для BinDiff — 82,4%, для Diaphora — 64,7%.

В разделе 4.4 приведены выводы по разработанному двухэтапному методу выявления изменений между версиями программ.

В пятой главе описаны методы идентификации статически связанных библиотек и поиска копий известных уязвимостей с использованием разработанного метода поиска клонов фрагмента кода, способного

анализировать десятки миллионов строк исходного и соответствующего исполняемого кода.

В разделе 5.1 описывается общая структура метода идентификации статически связанных библиотек, его реализация и экспериментальные результаты.

Метод предусматривает анализ исполняемого файла и множества различных библиотек с разными версиями. На первом этапе исполняемый файл сравнивается с каждой библиотекой, и для каждой пары определяются общие функции и степень их сходства. В конце, на основе сравнения всех пар метод выявляет библиотеки и их версии, которые статически связаны с исполняемым файлом (Рисунок 3).

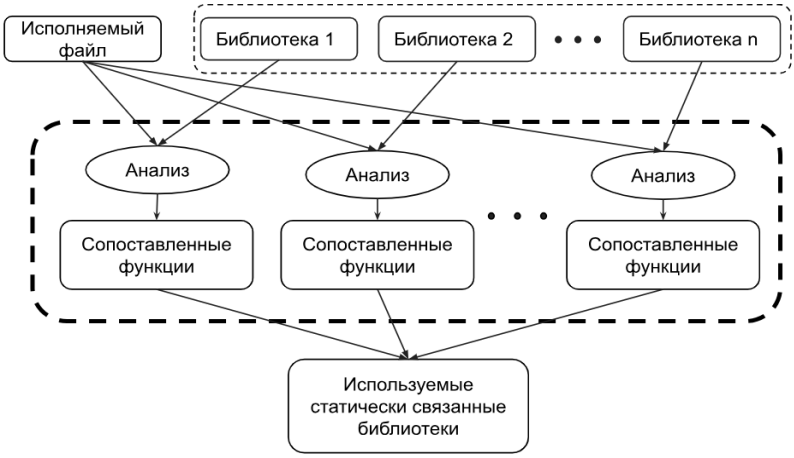

Рисунок 3. Схема идентификации статически связанных библиотек.

Множество библиотек может содержать как статические, так и динамические библиотеки. Так как статическая библиотека – это архив объектных файлов, то при сравнении со статической библиотекой сначала из нее извлекаются объектные файлы. Для каждого объектного файла генерируются ГЗП и ГВФ и удаляются повторяющиеся функции. Из ГВФ всех объектных файлов строится один ГВФ. В случае динамической библиотеки обработка выполняется так же, как обработка исполняемого файла.

После построения ГВФ и ГЗП для библиотеки и исполняемого файла, выполняется их сравнение. Для выявления схожести используется метод сопоставления функций, описанный в главе 4. После сопоставления функций

выбираются только те сопоставленные пары функций, которые схожи на 100%.

Для оценки качества инструмента он был протестирован на нескольких версиях Binutils и Coreutils, скомпилированных с использованием разных компиляторов и оптимизаций. Для идентификации были использованы следующие библиотеки: libc.a, libdl.a, libgcc.a, libgcc_eh.a, libgmp.a, libpthread.a, librt.a, libbfd.a, libctf.a, libctf-nobfd.a, libiberty.a, libopcodes.a, libz.a, libcoreutils.a, libver.a.

В таблице 5 приведена средняя точность и полнота нахождения всех библиотек, связанных с исполняемыми файлами, которые входят в состав Binutils и Coreutils. Как видно из результатов, инструмент демонстрирует высокую полноту (98–100%) при различных конфигурациях, в то время как точность варьируется от 89% до 91%, немного снижаясь при повышении уровня оптимизации. В целом, инструмент демонстрирует высокие показатели обнаружения, что подтверждает его эффективность при анализе различных библиотек и исполняемых файлов.

Исполняемый файл	**Архитектура**	**Компилятор**	**Оптимизация**	**Точность (%)**	**Полнота (%)**	**F1-мера (%)**
Binutils	x86-64	Clang	O0	91	98	94
Binutils	x86-64	GCC	O0	91	98	94
Binutils	x86-64	GCC	O2	89	99	94
Coreutils	x86-64	Clang	O0	91	100	95
Coreutils	x86-64	Clang	O2	90	100	95

Таблица 5. Результат работы инструмента.

В разделе 5.2 описывается метод и детали реализации обнаружения известных уязвимостей (ИУ) в программном обеспечении, основанный на поиске клонов фрагментов кода (Рисунок 4). Процесс обнаружения ИУ включает четыре этапа: сбор информации об ИУ и их исправлениях из открытых источников (GitHub, NIST, Mitre), извлечение уязвимых фрагментов кода из фиксирующих патчей с изоляцией исправлений от изменений, не связанных с устранением уязвимостей (например, рефакторинга). Далее, для каждого уязвимого фрагмента формируются ГЗП, которые сохраняются в хранилище. На завершающем этапе создаются ГЗП для целевого проекта, и

проводится их сравнение с ГЗП из хранилища, что позволяет эффективно выявлять клоны известных уязвимостей в программном обеспечении с открытым исходным кодом.

Рисунок 4. Схема нахождения клонов известных уязвимостей.

Для оценки эффективности обнаружения ИУ были проведены два эксперимента. В первом эксперименте была проанализирована версия 12.6 операционной системы Debian, применив обнаружение клонов ИУ к ее пакетам. Целью было выявить пакеты в Debian, которые используют устаревшее стороннее программное обеспечение с известными уязвимостями. Во время второго эксперимента был расширен анализ до примерно 1000 репозиториев GitHub. Этот эксперимент показал, что многие проекты полагаются на устаревшее стороннее программное обеспечение, что потенциально подвергает их рискам информационной безопасности. Чтобы продемонстрировать эффективность инструмента обнаружения клонов ИУ, было выявлено и сообщено об использовании устаревшего кода в следующих проектах: 0ad, PHP, CMake, OpenJPEG, rct, Emscripten, fldigi, POV-Ray, ntopng, pcl, Kamailio, H2O, ITK, Dragonfly, Webdis, Defold и ODrive. В целом было сообщено о семнадцати ошибках, девять из них уже приняты, а две отклонены (поскольку разработчики используют проекты с ошибками в качестве тестов). По шести проблемам пока не было получено ответов.

В разделе 5.3 приведены выводы по разработанным методам идентификации статически связанных библиотек и обнаружения копий известных уязвимостей.

В **заключении** содержатся выводы разработанных методов.

Основные результаты диссертационной работы:

1. Разработан и реализован унифицированный метод нахождения клонов фрагмента кода в исходном и в исполняемом коде, основанный на графах, содержащих зависимости управления и зависимости данных, который позволяет найти клоны произвольных фрагментов кода и обладает высокой точностью, полнотой и производительностью для анализа десятков миллионов строк исходного и соответствующего исполняемого кода.
2. Разработан и реализован метод оптимизации программ, использующих вычисление ЦПИ, при помощи поиска клонов и подстановки эффективных реализаций ЦПИ с учетом аппаратной платформы.
3. Разработан и реализован двухэтапный метод выявления изменений между версиями программ, который сочетает метрический подход с разработанным методом поиска клонов кода на основе графов, содержащих зависимости управления и зависимости данных, для сопоставления функций в формате «многие ко многим» и отображения измененных инструкций.
4. Разработаны и реализованы методы идентификации статически связанных библиотек и поиска копий известных уязвимостей с использованием разработанного метода поиска клонов фрагментов кода.

Публикации по теме научного доклада:

1. М. Арутюнян, «Поиск клонов кода для языка программирования JAVA на основе семантического анализа программы,» *СБОРНИК НАУЧНЫХ СТАТЕЙ СНО ЕГУ, МАТЕРИАЛЫ ЕЖЕГОДНОЙ НАУЧНОЙ СЕССИИ 2016 ГОДА*, т. 1. 5, № 22, стр. 125-133, 2017.
2. М. С. Арутюнян, Г. С. Иванов, В. Г. Варданян, А. К. Асланян, А. И. Аветисян и Ш. Ф. Курмангалеев, «Анализ характера изменений программ и поиск неисправленных фрагментов кода,» *Труды Института системного программирования РАН (Труды ИСП РАН)*, т. 31, № 1, стр. 49-58, 2019.
3. Г. С. Иванов, П. М. Пальчиков, А. Ю. Тарасов, Г. С. Акимов, А. К. Асланян, В. Г. Варданян, А. С. Арутюнян и Г. С. Керопян, «Исследование и разработка межпроцедурных алгоритмов поиска дефектов в исполняемом коде программ,» *Труды Института системного программирования РАН*, т. 31, № 6, стр. 89-98, 2019.
4. M. Arutunian, H. Aslanyan, V. Vardanyan, V. Sirunyan, S. Kurmangaleev и S. Gaissaryan, «Analysis of Program Patches Nature and Searching for

Unpatched Code Fragments,» в *2019 Ivannikov Memorial Workshop*, стр. 53-56, Velikiy Novgorod, Russia, September 2019.

5. H. Aslanyan, M. Arutunian, G. Keropyan, S. Kurmangaleev и V. Vardanyan, «BinSide: Static Analysis Framework for Defects Detection in Binary Code,» в *2020 Ivannikov Memorial Workshop, IVMEM 2020*, стр. 9-14, Orel, Russia, 2020.
6. S. Sargsyan, V. Vardanyan, H. Aslanyan, M. Harutyunyan, M. Mehrabyan, K. Sargsyan, H. Hovhannisyan, H. Movsisyan, J. Hakobyan и S. Kurmangaleev, «GENES ISP: code analysis platform,» в *2020 Ivannikov Ispras Open Conference (ISPRAS)*, стр. 35-39, Moscow, Russia, 2020.
7. М. С. Арутюнян, Р. А. Оганнисян и Х. С. Смбатян, «БЕНЧМАРКИНГ ИНСТРУМЕНТОВ СРАВНЕНИЯ,» *ВЕСТНИК РОССИЙСКО-АРМЯНСКОГО УНИВЕРСИТЕТА*, № 1, стр. 150-156, 2021.
8. М. Арутюнян, «Идентификация статически слинкованных библиотек в исполняемых файлах,» в *Тезисы конференции «Ломоносов-2021»*, Москва, 2021.
9. M. Arutunian, H. Hovhannisyan, V. Vardanyan, S. Sargsyan, S. Kurmangaleev и H. Aslanyan, «A Method to Evaluate Binary Code Comparison Tools,» в *2021 Ivannikov Memorial Workshop*, стр. 3-5, Nizhny Novgorod, Russia, September 2021.
10. M. Arutunian, M. Mehrabyan, S. Sargsyan. и H. Aslanyan, «Precise Code Fragment Clone Detection,» в *VALID 2024: The Sixteenth International Conference on Advances in System Testing and Validation Lifecycle*, стр. 7-14, Venice, Italy, September 2024.
11. M. Arutunian, S. Sargsyan, M. Mehrabyan, L. Bareghamyan и H. Aslanyan, «Automatic Recognition and Replacement of Cyclic Redundancy Checks for Program Optimization,» *IEEE Access*, т. 12, стр. 192146 - 192158, 2024.
12. M. Arutunian, S. Sargsyan, H. Hovhannisyan, G. Khroyan, A. Mkrtchyan, H. Movsisyan, A. Avetisyan и H. Aslanyan, «Accurate Code Fragment Clone Detection and Its Application in Identifying Known CVE Clones,» *International Journal of Information Security*, т. 24, № 55, 2025.

Свидетельства о государственной регистрации программы для ЭВМ автора по теме диссертации:

13. Курмангалеев Ш.Ф., Саргсян С.С., Варданян В. Г., Асланян А. К., Акопян Д.А., Арутюнян М.С., Меграбян М.С., Мовсисян О.М., Саргсян К.Г., Оганесян Р.А., «ISP Genes». РФ, свидетельства о государственной регистрации ЭВМ № 2020663670, 30.10.2020.
14. Курмангалеев Ш.Ф., Асланян А.К., Арутюнян М.С., Оганесян Р.А., Варданян В.Г., Саргсян С.С, «LibraryIdentifier». РФ, свидетельства о государственной регистрации ЭВМ № 2021665076, 17.09.2021.