

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

*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»

Диссертация

на соискание ученой степени  
кандидата технических наук

Научный руководитель:

к. ф.-м. н., доцент

Саргсян Севак Сеникович

**Москва 2025**

## Оглавление

Введение .....	6
Глава 1. Обзор существующих работ.....	13
1.1. Обзор методов и инструментов поиска клонов кода .....	13
1.1.1. Методы поиска клонов кода на основе текста.....	16
1.1.2. Методы поиска клонов кода на основе токенов .....	18
1.1.3. Методы поиска клонов кода на основе деревьев.....	18
1.1.4. Методы поиска клонов кода на основе графов .....	19
1.1.5. Методы поиска клонов кода на основе метрик .....	20
1.1.6. Сравнение методов и итоги исследования.....	20
1.2. Обзор методов нахождения клонов циклической проверки избыточности..	22
1.3. Обзор методов сравнения программ .....	23
1.3.1. Сравнение инструментов и итоги исследования .....	26
1.4. Обзор методов поиска статически связанных библиотек в исполняемых файлах.....	29
1.4.1. Обзор инструмента KARTA .....	29
1.4.2. Обзор инструмента IDA FLIRT .....	31
1.4.3. Обзор инструмента KISS .....	32
1.4.4. Обзор инструмента IdenLib .....	33
1.4.5. Обзор инструмента от Шу Акабане и Такеши Окамото .....	34
1.4.6. Обзор инструмента FunctionSimSearch. ....	34
1.4.7. Обзор инструмента Library identification .....	35
1.4.8. Сравнение инструментов и итоги исследования .....	37
1.5. Выводы исследования .....	38

Глава 2. Унифицированный метод поиска клонов произвольных фрагментов кода в исходном и исполняемом коде .....	40
2.1. Метод нахождения клонов фрагмента кода.....	40
2.1.1. Построение ГЗП .....	41
2.2. Сопоставление графов зависимостей программы .....	42
2.2.1. Построение множества изначально сопоставленных пар вершин .....	44
2.2.2. Выбор нерассмотренных пар вершин из построенного множества начальных вершин и итеративное расширение.....	45
2.3. Реализация .....	51
2.3.1. Построение ГЗП исходного кода.....	51
2.3.2. Построение ГЗП исполняемого кода.....	53
2.4. Оценка точности и полноты .....	54
2.5. Масштабируемость .....	59
2.6. Сравнение с существующими инструментами.....	59
2.7. Оценка FCD на BigCloneBench.....	60
2.8. Выводы .....	62
Глава 3. Метод оптимизации программ при помощи поиска клонов .....	63
3.1. Применения циклической проверки избыточности .....	63
3.2. Циклическая проверка избыточности .....	64
3.2.1. Реализации ЦПИ.....	65
3.2.2. ЦПИ с прямым и обратным порядком бит .....	66
3.3. Описание предлагаемого метода .....	68
3.4. Символическое выполнение на битовом уровне.....	68
3.5. Распознавание ЦПИ .....	71
3.5.1. Первичная идентификация кандидатов ЦПИ .....	72

3.5.2. Извлечение параметров.....	73
3.5.3. Вычисление полинома ЦПИ.....	74
3.5.4. Создание структуры РСЛОС .....	75
3.5.5. Верификация обнаруженного ЦПИ .....	78
3.6. Замена ЦПИ.....	79
3.7. Реализация метода нахождения ЦПИ и его замены.....	80
3.7.1. Внедрение замены ЦПИ в GCC.....	81
3.8. Неподдерживаемые случаи.....	83
3.9. Экспериментальная оценка .....	84
3.9.1. Вычисления ЦПИ в программном обеспечении с открытым исходным кодом.....	84
3.9.2. Влияние оптимизации.....	85
3.10. Выводы .....	92
Глава 4. Двухэтапный метод выявления изменений между версиями программ ..	93
4.1. Схема метода сравнения двух программ .....	93
4.1.1. Первый этап – генерация графа вызовов функций и графов зависимостей программы.....	94
4.1.2. Второй этап - нахождение схожих графов.....	94
4.2. Реализация .....	95
4.3. Результаты .....	96
4.4. Выводы .....	101
Глава 5. Методы идентификации статически связанных библиотек и поиска копий известных уязвимостей с использованием разработанного метода поиска клонов фрагментов кода.....	103
5.1. Идентификация статически связанных библиотек.....	103

5.1.1. Типы библиотек и их связывание.....	103
5.1.2. Схема идентификации статически связанных библиотек.....	104
5.1.3. Результаты идентификации статически связанных библиотек .....	106
5.2. Обнаружение ИУ в программах .....	109
5.2.1. Метод обнаружения ИУ в программах .....	110
5.2.2. Результаты обнаружения клонов ИУ .....	112
5.3. Выводы .....	113
Заключение .....	114
Литература.....	115
Список таблиц .....	128
Список рисунков .....	130
Приложение .....	132

## Введение

### 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 базе 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.

**В первой главе** приводится обзор работ, которые имеют отношение к теме диссертации. Рассматриваются современные методы поиска клонов как в исходном, так и в исполняемом коде, анализируются их преимущества и недостатки. Также исследуются существующие методы сравнения исполняемых

файлов и идентификации статически связанных библиотек.

**Во второй главе** излагается разработанный метод обнаружения клонов фрагментов исходного и исполняемого кода, подробно описываются этапы его реализации. Приводятся результаты экспериментальной оценки метода, включая тестирование его масштабируемости, точности и полноты, а также проводится сравнительный анализ его эффективности с существующими инструментами.

**В третьей главе** излагается разработанный метод обнаружения клонов фрагментов, вычисляющих циклическую проверку избыточности, и замена этого фрагмента на более оптимальную версию. Приводятся результаты тестирования метода, демонстрирующие его практическую применимость, а также улучшения производительности кода после выполненной замены.

**В четвертой главе** описывается разработанный метод сравнения двух программ для выявления пар функций на основе принципа сравнения «многие с многими». Подробно рассматриваются алгоритмы, учитывающие структурные и семантические особенности функций.

**В пятой главе** описывается разработанный метод идентификации статически связанных библиотек. Также описывается процесс анализа кода для идентификации известных уязвимостей в коде. Приводятся результаты экспериментов, включая обнаружение известных уязвимостей в ряде программных продуктов, переданных разработчикам для исправления, некоторые из которых уже устранили.

**В заключении** содержатся выводы разработанных методов.

# Глава 1. Обзор существующих работ

В этой главе приводится обзор работ, которые имеют отношение к теме диссертации. Рассматриваются современные методы поиска клонов как в исходном, так и в исполняемом коде, анализируются их преимущества и недостатки. Также исследуются существующие методы сравнения исполняемых файлов и идентификации статически связанных библиотек.

## 1.1. Обзор методов и инструментов поиска клонов кода

Существует большое количество инструментов и методов обнаружения клонов кода. Однако мало работ, в которых обнаруживаются такие фрагменты кода, которые не являются целыми функциями.

Клоны кода делятся на четыре типа. Клон первого типа - фрагменты кода, которые идентичны, за исключением различий в пробелах и комментариях.

Клон второго типа - в случае исходного кода клоны фрагментов кода могут отличаться именами переменных, значениями, типами, комментариями. В случае исполняемого кода клоны фрагментов кода отличаются значениями данных и именами регистров.

Клон третьего типа - в дополнение к упомянутым выше изменениям, некоторые команды могут быть добавлены или удалены.

Клон четвертого типа – фрагменты кода, которые выполняют одни и те же вычисления, но используют разные наборы инструкций.

На рисунке 1 приведены примеры клонов исполняемого кода, а на рисунке 2 – исполняемого кода, где клон первого типа совпадает с заданным фрагментом. Клон второго типа отличается от заданного фрагмента распределением регистра `ecx` вместо `eax`. Клон третьего типа отличается от заданного фрагмента распределением регистра `ecx` вместо `eax` и заменой операции `imull` в инструкции `imull -4(%rbp), %eax` на `addl`. Клон четвертого типа отличается от заданного фрагмента тем, что вычисляет факториал, но вместо рекурсии написано итеративно.

<i>Фрагмент кода</i>	<i>Клон типа-1</i>
<pre>int factorial_rec (int n) {   if (n &lt;= 1) {     return 1;   } else {     return n * factorial_rec (n - 1);   } }</pre>	<pre>int factorial_rec (int n) {   if (n &lt;= 1) { // Комментария     __return 1;   } else {     __return n * factorial_rec (n - 1);   } }</pre>
<i>Клон типа-2</i>	<i>Клон типа-3</i>
<pre>int factorial_rec (double x) {   if (x &lt; 2) { // Комментария     __return 1;   } else {     __return x * factorial_rec (x - 1);   } }</pre>	<pre>int sum_rec (double x) {   if (x &lt; 2) { // Комментария     __return 1;   } else {     __return x + sum_rec (x - 1);   } }</pre>
<i>Клон типа-4</i>	
<pre>int factorial_iterative(int n) {   int result = 1;   for (int i = 1; i &lt;= n; ++i) {     result *= i;   }   return result; }</pre>	

*Рисунок 1. Пример типов клонов исходного кода.*

<i>Фрагмент кода</i>	<i>Клон типа-1</i>
<pre>factorial_rec:   pushq %rbp   movq %rsp, %rbp   subq \$16, %rsp   movl %edi, -4(%rbp)   cmpl \$1, -4(%rbp)   jg .L2   movl \$1, %eax   jmp .L3 .L2:   movl -4(%rbp), %eax   subl \$1, %eax   movl %eax, %edi   call factorial_rec   imull -4(%rbp), %eax</pre>	<pre>factorial_rec:   pushq %rbp   movq %rsp, %rbp   subq \$16, %rsp   movl %edi, -4(%rbp)   cmpl \$1, -4(%rbp)   jg .L2   movl \$1, %eax   jmp .L3 .L2:   movl -4(%rbp), %eax   subl \$1, %eax   movl %eax, %edi   call factorial_rec   imull -4(%rbp), %eax</pre>

.L3: ret	.L3: Ret
<i>Клон типа-2</i>	
factorial_rec: pushq %rbp movq %rsp, %rbp subq \$16, %rsp movl %edi, -4(%rbp) cmpl \$1, -4(%rbp) jg .L2 movl \$1, %ecx jmp .L3 .L2: movl -4(%rbp), %ecx subl \$1, %ecx movl %ecx, %edi call factorial_rec imull -4(%rbp), %ecx .L3: ret	<i>Клон типа-3</i> sum_rec: pushq %rbp movq %rsp, %rbp subq \$16, %rsp movl %edi, -4(%rbp) cmpl \$1, -4(%rbp) jg .L2 movl \$1, %ecx jmp .L3 .L2: movl -4(%rbp), %ecx subl \$1, %ecx movl %ecx, %edi call sum_rec addl -4(%rbp), %ecx .L3: Ret
<i>Клон типа-4</i>	
factorial_O3: movl \$1, %eax cmpl \$1, %edi jle .L1 .p2align 4,,10 .p2align 3 .L2: movl %edi, %edx subl \$1, %edi imull %edx, %eax cmpl \$1, %edi jne .L2 .L1: ret	

*Рисунок 2. Пример типов клонов исполняемого кода.*

Существует 5 основных методов обнаружения клонов кода: на основе текста, токенов, дерева, графа и метрик. Также существует множество гибридных методов обнаружения клонов кода. Гибридная техника представляет собой совокупность нескольких техник.

### **1.1.1. Методы поиска клонов кода на основе текста**

В текстовых методах обнаружения клонов два фрагмента кода сравниваются друг с другом в виде текста/строк. Фрагменты отмечаются как клоны кода, если они идентичны с точки зрения текстового содержания. Методы обнаружения клонов на основе текста генерируют меньше ложных срабатываний, просты в реализации и не зависят от языка. Но эти методы игнорируют синтаксическую и семантическую информацию кода и могут обнаруживать только относительно простые клоны кода.

Одним из первых инструментов, основанных на текстовом подходе, является Dup [15] [16], разработанный Бейкером. Dup представляет исходный код в виде последовательности строк и обнаруживает клоны построчно. Dup нормализует код, удаляя комментарии и пробелы. Затем он заменяет идентификаторы, переменные и типы специальным параметром, чтобы можно было обнаружить клон, если имя двух переменных отличается. Тем не менее, инструмент Dup не может обнаружить клоны, написанные в разных стилях. Инструмент Dup также можно классифицировать как метод, основанный на токенах, поскольку он токенизирует каждую строку для построчного сопоставления.

Джонсон [17] [18] использует алгоритм отпечатков Карпа-Рабина [19] для обнаружения клонов на основе обычного текста. Сначала генерируется набор подстрок, которые вместе охватывают весь исходный код. Однако ограничение этого метода состоит в том, что для сопоставления рассматриваются последовательности из 50 строк, что приводит к более высокому уровню ложных срабатываний.

Дюкасс и соавт. [20] разработали независимый от языка инструмент Duploc для обнаружения клонов без разбора. Сначала Duploc удаляет комментарии и пробелы в коде и преобразует все символы в строчные. Инструмент хеширует каждую строку в одну из нескольких групп. Затем использует простой строковый метод для обнаружения самой длинной общей подпоследовательности. Duploc способен обнаруживать значительное количество клонов кода, но не способен



идентифицировать переименование, удаление и вставку. В последующей версии в Duploc [21] добавили фильтры для уменьшения ложных срабатываний. На этом шаге из дублированного кода извлекаются интересные шаблоны, такие как размер промежутка, который представляет собой длину не повторяющейся подпоследовательности между парой клонов. Пары клонов фильтруются по минимальной длине последовательности и максимальным размером промежутка.

DuDe [22] - еще один инструмент обнаружения клонов на основе строк, который способен обнаруживать цепочки дубликации, состоящие из нескольких меньших частей кода.

Инструмент NiCad [23] [24] [25], использует два метода обнаружения клонов: на основе текста и абстрактного синтаксического дерева. NiCad состоит из трех фаз.

- 1) Извлекаются функции и разные фрагменты инструкций разбиваются на строки.

- 2) С использованием правил преобразования фрагменты утверждений нормализуются, чтобы игнорировать различия редактирования.

- 3) Проверяются потенциальные клоны на переименования, фильтрации и абстрагирования с использованием динамической кластеризации для простого текстового сравнения потенциальных клонов. Алгоритм самой длинной общей подпоследовательности используется для одновременного сравнения двух потенциальных клонов. Следовательно, каждый потенциальный клон необходимо сравнивать со всеми остальными, что делает сравнение трудоемким и ресурсоемким процессом.

Инструмент SDD (Similar Data Detection) [26], разработанный Сынхаком и Джонгом, полезен для обнаружения клонов кода в системах большого размера. Техника основана на генерации индекса и инвертированного индекса [27] для фрагментов кода и их позиций. Затем для поиска похожих фрагментов используется алгоритм расстояния n-соседей [28].

### ***1.1.2. Методы поиска клонов кода на основе токенов***

Методы обнаружения клонов на основе токенов также называют лексическими подходами, так как они используют лексический анализатор для разделения исходного кода на последовательности токенов. Затем выполняются различные преобразования путем добавления, изменения или удаления некоторых токенов. После чего находят похожие подпоследовательности в последовательности токенов. Методы на основе токенов могут обнаруживать клоны типа 1 и типа 2.

ConQAT [29] [30] и CCFinder [31] [32] используют токены и деревья суффиксов для обнаружения клонов. CP-Miner [33] использует методы добычи данных. SHINOBI [34] и RTF [35] используют массив суффиксов вместо дерева суффиксов. Инструмент Cscope [36] сравнивает токены узлов абстрактного синтаксического дерева (АСД), используя алгоритм на основе дерева суффиксов. Ли и Томпсон [37] используют комбинацию потока токенов и АСД для обнаружения и удаления клонов кода. JPlag [38] и Winnowing [39] являются инструментами обнаружения плагиата.

Существует множество других инструментов, использующих методы на основе токенов, такие как D-CCFinder [40], CCFinderX [41], FCFinder [42], SourcererCC [43], iClones [44] и т.д.

### ***1.1.3. Методы поиска клонов кода на основе деревьев***

Абстрактные синтаксические деревья и деревья синтаксического анализа часто используются в подходах, основанных на деревьях. Методы на основе дерева менее чувствительны к редактированию кода, чем инструменты на основе токенов.

Подход Янга [45] основан на грамматике и строит вариант дерева разбора. Обнаружение основано на методе нахождения наиболее длинной общей последовательности динамического программирования.

Инструмент CloneDR [46] использует хеширование АСД и динамическое программирование. SimScan [47] и ccdiml [48] являются вариантами CloneDR.

Фальке и соавт. [49], Тайрас и Грей [50] используют деревья суффиксов для обнаружения клонов в коде, преобразованном в АСД.

Инструмент Sim [51] преобразует исходные программы в деревья разбора и рассматривает их как строки. Затем применяет алгоритм нахождения самой длинной общей подпоследовательности и динамическое программирование для оценки сходства. DECKARD [52] извлекает векторы признаков из АСД и группирует их с помощью хеширования с учетом местоположения в Евклидовом пространстве. Asta [53] работает над явлением структурной абстракции произвольных поддеревьев АСД. ClemanX [54] [55] конструирует характеристические векторы из поддеревьев АСД и использует хеширование с учетом местоположения. Саебьернсен и соавт. [56] также используют тот же набор методов для обнаружения клонов в ассемблерном коде.

Инструмент CloneDigger [57], Ли и соавт. [58], Браун и соавт. [59] используют антиунификацию для расчета расстояния между АСД. Уолер и соавт. [60], Эванс и соавт. [61] используют АСД, представленные в XML. Чирович и соавт. [62] используют отпечатки синтаксического дерева. CSeR [63] сравнивает АСД с использованием таких метрик, как расстояние Левенштейна. Инструмент JCCD [64] [65] обнаруживает клоны с помощью конвейеров.

#### ***1.1.4. Методы поиска клонов кода на основе графов***

Обычно методы на основе графов используют зависимости управления, зависимости данных или граф зависимостей программы (ГЗП) [66]. Подходы на основе ГЗП более устойчивые, чем предыдущие к вставке и удалению кода, переупорядоченным инструкциям, переплетенному коду и несмежному коду. Однако для обнаружения клонов кода потребуется больше времени.

PDG-DUP [67], Scorpio [68], работы А. Асланяна [69] и С. Саргсяна [70] основаны на ГЗП и используют срезы для поиска изоморфных подграфов. Duplix [71] использует k-ограничивающий итерационный подход для нахождения максимально подобных подграфов.

GPLAG [72] использует ГЗП без информации о зависимостях управления

для обнаружения программного плагиата. Чен и соавт. [73] также предлагают метод сжатия кода на основе ГЗП с учетом синтаксической структуры и зависимостей данных.

### ***1.1.5. Методы поиска клонов кода на основе метрик***

В методах на основе метрик сначала рассчитываются различные типы метрик кода, такие как количество строк и количество функций, и сравниваются эти метрики для поиска клонов. Техника, основанная на метриках, не сравнивает код напрямую. Чтобы найти клоны кода, методы обнаружения клонов используют несколько типов метрик программы. В большинстве случаев для расчета различных типов метрик исходный код или промежуточное представление дизассемблированного исполняемого кода преобразуется в абстрактное синтаксическое дерево или граф зависимостей программы.

Инструменты CLAN [74] и Datrix [75] используют метрики функционального уровня для обнаружения клонов кода. Контояннис и соавт. [76] [77] используют метрики, полученные из АСД-представления кода.

Ланубиле и Маллардо [78] используют метрики для обнаружения клонов функций в веб-приложениях. Дэви и соавт. [79] вычисляют определенные характеристики кодовых блоков, а затем используют нейронные сети для поиска похожих блоков. Перумал и соавт. [80] используют методы метрики и отпечатков для обнаружения клонов в исходном коде. Кодхай и соавт. [81], Дагенайс и соавт. [82] применяют метрики к текстовым представлениям исходного кода. Ли и Сун [83] используют метрическое пространство со значениями координат.

### ***1.1.6. Сравнение методов и итоги исследования***

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 ограничены 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 деревьях, такие как анализ абстрактных синтаксических деревьев (АСД), обеспечивают высокую точность в обнаружении клонов типа-2 и могут выявлять клоны типа-3, однако с 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.

Для оценки эффективности существующих инструментов было проведено сравнительное исследование на наборе данных BigCloneBench [84], содержащем более 8 миллионов размеченных пар клонов. Клоны классифицировались по типам: тип-1, тип-2, и тип-3, с двумя подтипами: слабо измененный тип-3, сильно измененный тип-3.

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

Таблица 1. Сравнительный анализ инструментов поиска клонов кода.

измененный тип-3, где сходство находится в диапазонах [90%,100%) и [70%,90%).

Результаты исследования показывают (Таблица 1), что существующие инструменты эффективно обнаруживают клоны типа-1 и типа-2, однако их эффективность значительно снижается при поиске клонов типа-3. Хотя NiCad демонстрирует относительно лучшие результаты, его полнота при обнаружении сильно измененного типа-3 достигает лишь 73%, что недостаточно для практического применения в различных задачах поиска клонов кода.

## 1.2. Обзор методов нахождения клонов циклической проверки избыточности

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, но не была объединена с основной веткой GCC [85]. Код выполняет сопоставление базовых блоков и их инструкций с предопределенным шаблоном вычисления ЦПИ. Этот процесс проходит в несколько шагов, где каждая функция отвечает за сопоставление определенной части (базового блока) шаблона. Если на каком-либо этапе шаблон не сопоставляется, функция возвращает *false*, и анализ прекращается. Если все функции успешно находят соответствие с шаблоном, исходное вычисление ЦПИ заменяется реализацией на основе таблицы.

Код имеет ряд ограничений, которые могут влиять на его эффективность в обнаружении и оптимизации реализаций ЦПИ. Во-первых, он предназначен для обнаружения и замены только ЦПИ с обратным порядком бит с длиной 8, 16 и 32 бита для 8-битных данных. Это исключает возможность работы с другими вариантами ЦПИ или размерами данных. Во-вторых, код предполагает использование определенного шаблона для реализации ЦПИ, что ограничивает его применимость. Если реализация не соответствует ожидаемому шаблону, обнаружение и замена невозможны. В-третьих, код полагается на структуру и инструкции базовых блоков, которые могут изменяться в результате оптимизаций компилятора или между разными версиями компилятора. Это может приводить к снижению точности и полноты процесса обнаружения и замены, ограничивая

применение метода для конкретных случаев и версий компилятора. Такие ограничения делают метод полезным в определенных сценариях, но менее универсальным для широкого применения.

### 1.3. Обзор методов сравнения программ

Инструменты сравнения программ в ходе могут получать две или более программы для выявления их схожести. Сравнение может быть проведено на уровне базовых блоков, функций или целых программ. Часто исходный код программы не доступен, поэтому в таких случаях сравнение исполняемого кода имеет основополагающее значение.

Для сравнения исходного кода широко используются такие инструменты, как `llvm-diff` [86], `git-diff` [87], `Beyond Compare` [88] и `DiffMerge` [89]. Эти инструменты эффективно выявляют текстовые различия между версиями программ, работая с доступным исходным кодом. Они способны определять изменения в структуре кода, именах переменных, функций, комментариях и определениях структур данных. Однако их возможности ограничены работой только с исходным кодом и не учитывают семантические особенности программы.

Выявление схожести и различий исполняемого кода является значительно более сложной задачей. В процессе компиляции теряется (если программа не скомпилирована с отладочной информацией) большая часть семантики, включая имена переменных, имена функций, комментарии к источникам и определения структур данных. Кроме того, даже если исходный код программы не изменяется, исполняемый файл может измениться. Например, результирующий исполняемый код может значительно измениться при использовании разных компиляторов, разных оптимизаций и выборе различных целевых операционных систем и архитектур.

Первые инструменты сравнения исполняемых файлов – `ExeDiff` [90] и `VMAT` [91]. `ExeDiff` сравнивает два исполняемых файла архитектуры Digital Alpha и генерирует патчи. Для выявления изменений дизассемблирует исполняемый код

в инструкции и ищет идентичные инструкции. VMAT сравнивает функции и базовые блоки. Сначала сопоставляет функции двух исполняемых файлов, потом сопоставляет базовые блоки сопоставленных функций. Для сопоставления функций использует их имена и типы аргументов. А для сопоставления базовых блоков использует технику хеширования. В нем учитываются адреса, коды операций и операнды инструкций, а также информация о зависимостях управления. Хеширование чувствительно к порядку инструкций.

Инструмент Diaphora [92] сопоставляет функции с помощью различных наборов эвристик («лучшие сопоставления», «частичные и ненадежные сопоставления», «ненадежные сопоставления»). Эвристики применяются по очереди, если после применения текущей эвристики остаются несопоставленные функции, применяется следующая из списка.

Работы [91] [93] [94] [95] [96] [97] [98] [99] [100] [101] [102] для определения сходства находят изоморфные подграфы. Работы [91] [93] [94] [95] [96] [97] рассматривают соседние вершины. Сначала идентифицируют начальное множество сопоставленных вершин. Затем сопоставление расширяется путем проверки только соседей уже сопоставленных вершин. В [98] [99] [100] [101] [102] работах используются алгоритмы обратного отслеживания. Они исправляют неправильные сопоставления, повторно обращаясь к решению. Если новое сопоставление не улучшает общее сопоставление, оно отменяется. Алгоритм обратного отслеживания может повысить точность, избегая локального оптимального сопоставления, но это дороже.

Bindiff [94] [95] [103] является популярным инструментом для сравнения исполняемого кода. Для сопоставления функций реализованы разные метрики на основе информации об узлах и ребрах графов зависимостей управления и графов вызовов функций. В работе авторы предлагают метод вычисления хеша для графов (MD-index) [104] и вычисляют разные хеши с использованием MD-индекса. Инструмент также сопоставляет базовые блоки внутри сопоставленных функций.

Работы [105] и [106] нацелены на усовершенствование инструмента BinDiff. В



[105] авторы пытаются сопоставить функции из баз данных вредоносных программ. В работах [107] и [106] (инструмент BinSlayer) авторы используют венгерский алгоритм [108] для более оптимального сопоставления базовых блоков.

Инструмент BinHunt [100] идентифицирует семантические изменения между двумя версиями программы. Он использует символьное выполнение и решатель ограничений, чтобы проверить, имеют ли два базовых блока одинаковую функциональность.

IBinHunt [101] улучшенная версия BinHunt. Для сокращения числа рассматриваемых вершин используется анализ помеченных данных. То есть в нахождение изоморфного подграфа рассматриваются только те базовые блоки, переменные которых зависят от пользовательского ввода. А работа [102] расширяет IBinHunt.

Инструмент Smit [109] находит подобное вредоносное ПО заданного образца в репозитории. Он индексирует графы вызовов вредоносных программ в базе данных и использует расстояние редактирования графа для поиска вредоносных программ с похожими графами вызовов.

Инструменты Idea [110], MBC [111], MutantX-S [112], Expose [113] и iLine [114] используют n-граммы. Для сопоставления инструкций iLine использует векторы булевых признаков (boolean features) и машинное обучение. MutantX-S также использует машинное обучение.

Работа [115] классифицирует инструкции на 14 классов и использует 14-битные значения. Они раскрашивают вершины графа на основе классов инструкций. Так они разбивают граф на k-подграфов, где каждый граф содержит только k связанных вершин, а затем генерируют отпечатки для каждого k-подграфа. Сходство двух графов соответствует максимальному количеству сопоставленных k-подграфов. Этот метод использовали также в [116] [107] работах. А раскрашивание вершин было адаптировано и использовано в [98] [99] работах.

В работах [117] [118] сравнение сходства функций преобразовано в сравнение сходства путей. Сначала они извлекают множество путей выполнения из графа потока управления, затем определяют метрику сходства путей между путями выполнения и, наконец, объединяют сходство путей в сходство функций. Кроме VinHunt и IBinHunt в работе [117] и в VinJuice [119] также используются символьные формулы на уровне базовых блоков. Вместо использования средства доказательства теорем (Theorem prover) VinJuice проверяет, имеют ли две символьные формулы один и тот же хеш после нормализации формул. Семантические хеши более эффективны, чем использование решателя, но они ограничены тем, что могут давать ложноотрицательные результаты.

Инструмент  $\alpha$ Diff использует три семантических признака. Сначала он извлекает внутри функциональный признак каждой двоичной функции, используя глубокую нейронную сеть (ГНС). ГНС работает непосредственно с необработанными байтами каждой функции.  $\alpha$ Diff дополнительно анализирует граф вызовов функций (ГВФ) каждого исполняемого файла и извлекает межфункциональные и межмодульные признаки. Затем на основе этих трех признаков вычисляется расстояние.

### ***1.3.1. Сравнение инструментов и итоги исследования***

Рассмотренные инструменты сопоставляют функции первой версии программы со вторым и имеют высокую точность, когда в конкретной версии программы нет похожих функций. Однако, когда хоть одна из версий программ имеет похожие функции, точность инструментов может падать. Например, если в первой версии исполняемого файла есть функции F1 и F2 похожие на 100%, и во второй версии исполняемого файла есть функции G1 и G2 похожие на 100%, то при сопоставлении функций для инструментов невозможно понять, соответствует ли F1 функции G1, а F2 — функции G2, или наоборот, F1 соответствует G2, а F2 — G1. В таких случаях инструменты выбирают одну перестановку. Желательно, чтобы инструменты выполняли сопоставление функций первого исполняемого

файла с функциями второго по схеме «многие ко многим». В указанном случае инструмент выдаст сопоставления F1 с G1 и G2, и F2 с G1 и G2.

В рамках исследования был проведен сравнительный анализ двух ведущих инструментов сравнения версий исполняемого кода: Diaphora и Bindiff. Эксперименты проводились на базе проекта GNU Coreutils [120] - набора базовых утилит для работы с файлами, оболочками и текстами операционной системы GNU.

Для всесторонней оценки эффективности инструментов использовались различные версии Coreutils, скомпилированные с применением разных компиляторов (GCC [121] и Clang [122]) и уровней оптимизации. Выбор Coreutils обусловлен тем, что эти утилиты представляют собой стабильный, хорошо документированный код, что позволяет эффективно отслеживать изменения между версиями.

Результаты экспериментов, представленные в таблице 2. Оба инструмента показали высокую точность при сравнении различных версий Coreutils, даже в случаях использования разных компиляторов. Это особенно важно для задач анализа безопасности и отслеживания изменений в программном коде. Однако существенное снижение эффективности наблюдалось при работе с кодом, скомпилированным с агрессивными оптимизациями (флаг -O2).

Сравнительный анализ выявил превосходство Bindiff над Diaphora. Тем не менее, результаты сравнительного анализа подтверждают необходимость разработки более совершенных инструментов для сравнения программ, способных эффективно работать с различными уровнями оптимизации и разными компиляторами. Создание такого инструмента представляет собой важную задачу в области анализа программного обеспечения и информационной безопасности.

Версия 1, компилятор, оптимизация	Версия 2, компилятор, оптимизация, без отладочной информации	BinDiff точность (%)	BinDiff полнота (%)	Diaphora точность (%)	Diaphora полнота (%)
8.30 clang O0	8.30 clang O0	98.8	97.9	98.3	75
8.30 gcc O0	8.30 gcc O0	98.8	97.9	98.3	74.7
8.30 clang O2	8.30 clang O2	98.3	85.9	97.5	59.2
8.30 gcc O2	8.30 gcc O2	98.2	81.7	97.3	55.2
8.30 clang O0	8.30 clang O2	72.2	61.5	93.1	34.3
8.30 gcc O0	8.30 gcc O2	75.5	63.7	91.3	34.5
8.30 clang O2	8.30 clang O3	86.6	76.6	97.2	41.2
8.30 gcc O2	8.30 gcc O3	88.7	75.4	93.5	44.8
8.30 gcc O0	8.30 clang O0	91.2	85.4	95.6	38.1
8.30 gcc O2	8.30 clang O2	81.1	71	93.8	36.7
8.30 gcc O0	8.30 clang O2	71.2	61.5	93	34.5
8.29 gcc O0	8.30 gcc O0	97.8	97.8	97.6	72.5
8.29 gcc O2	8.30 gcc O2	96.6	81.4	96.3	52.8
7.6 gcc O0	8.30 gcc O0	62.9	62.9	92.5	35.5
7.6 gcc O2	8.30 gcc O2	83.4	76.5	95.1	46.5
<b>В среднем</b>		<b>86.8</b>	<b>78.5</b>	<b>95.4</b>	<b>49.0</b>

Таблица 2. Сравнительный анализ инструментов сравнения программ.

## **1.4. Обзор методов поиска статически связанных библиотек в исполняемых файлах**

Использование библиотек – общепринятая практика в разработке программного обеспечения. Они позволяют разработчикам экономить время и усилия, предоставляя готовые решения для распространенных задач. Использование сторонних библиотек при написании программы облегчает работу программиста, но они также могут создавать серьезные угрозы безопасности и препятствовать анализу программ. Также в старой версии библиотеки может быть ошибка, которая исправлена в новой версии, но при этом в программе используется старая версия. Следовательно, в программе необходимо использовать обновленную версию библиотеки.

Часто потребитель имеет только исполняемый код программы, и неизвестно, какие библиотеки там используются. Исполняемые файлы часто статически связаны с библиотеками. Уязвимость в одной из этих библиотек может привести к эксплуатации устройства, особенно если учесть, что эти библиотеки часто обрабатывают данные, контролируемые пользователем. Таким образом, определив библиотеки и их версии в исполняемом коде, можно решить вопрос использования безопасных библиотек.

Данная глава посвящена исследованию методов поиска статически связанных библиотек в исполняемых файлах. Существуют несколько инструментов идентификации связанных библиотек, поэтому рассмотрим принцип их работы более подробно.

### **1.4.1. Обзор инструмента KARTA**

Karta [123] на вход получает исполняемый код и сопоставляет символы исходного кода библиотек к символами исполняемого кода. При разработке метода основное внимание уделено тому, чтобы процесс сопоставления происходил быстро.

Инструмент является архитектурно-независимым: при сопоставлении

исходный код компилируется и из него получается «каноническая форма» для поиска библиотеки в исполняемом коде. Результатом сопоставления являются пары функций (представляющие одну и ту же функцию) - одна из исходного кода библиотек, и вторая из исследуемого исполняемого кода. Также, сопоставляются некоторые символы. Инструмент поддерживает несколько библиотек: libpng, zlib, OpenSSL, OpenSSH, net-snmp, gSOAP, libxml2, libtiff, mDNSResponder, MAC-Telnet, libjpeg-turbo, libjpeg, icu, libvpx, treck.

Сопоставление основано на построении «канонической формы», которая получается из числовых констант, константных строк, количество инструкций ассемблера, размера фрейма стека, порядка вызовов функций. Но сопоставление только на основе «канонической формы» не масштабируется. Если функция в исполняемом коде точно похожа на некоторую функцию исходного кода и не похожа ни на одну другую функцию, то функции сопоставляются. Такие функции называются якорями. Якорные функции сопоставляются прежде, чем получение «канонического представления». Это ограничивает количество функций, для которых нужно построить «каноническое представление». Якоря создаются таким образом, чтобы однозначно идентифицировать библиотеку. Для каждой библиотеки рассматривались все функции, которые имеют уникальные числовые константы и уникальные строки. Если константы достаточно сложны или могут быть сгруппированы так, чтобы быть достаточно уникальными, то функция помечается как якорь. Сопоставление происходит следующим образом:

1. Каждая поддерживаемая библиотека имеет идентификатор, с помощью которого определяется, находится она в исполняемом коде или нет. Идентификаторы основаны на уникальных строках в библиотеке. Как только библиотека найдена, при возможности из идентификатора извлекается информация о точной версии, которая используется в исполняемом коде.
2. Вторым шагом является поиск якорных функций на основе уникальных числовых констант и уникальных строк, которые соответствуют функциям якорей библиотеки.

3. Используя множество сопоставленных функций, которые были определены соответствующими якорями, начинается сопоставление функций между ними.
4. Последним шагом является сопоставление на основе «канонического представления». Сопоставление заканчивается, когда больше нет функций для сопоставления, или когда не удастся найти новые совпадения.

#### **1.4.2. Обзор инструмента IDA FLIRT**

IDA FLIRT [124] (Fast Library Identification and Recognition Technology) - технология быстрой идентификации и распознавания библиотек. Технология позволяет IDA Pro [125] распознавать стандартные библиотечные функции, сгенерированные поддерживаемыми компиляторами [126]. Рассматриваются только программы, написанные на C/C++, и только распознаются и идентифицируются функции, расположенные в сегменте кода, сегмент данных игнорируется.

Для всех функций из всех используемых библиотек создана база для их идентификации. IDA Pro проверяет каждый байт дизассемблируемой программы, чтобы определить, может ли он быть началом функции стандартной библиотеки.

Информация, необходимая для распознавания функций стандартных библиотек, сохраняется в файле сигнатур. Каждая функция представляется шаблоном. Шаблон — это первые 32 байта функции, где отмечены все варианты байтов (статические библиотеки содержат так называемые переменные байты, которые во время статической линковки меняются и, следовательно, не известны на момент создания сигнатуры).

Для хранения функций используется древовидная структура. В узлах дерева хранятся последовательности байтов. И в каждом листе хранятся информация о функциях.

Так как современные библиотеки могут содержать несколько функций, начинающихся с одинаковыми байтами, то первые 32 байта функции не всегда позволяют определить ее. Когда первые 32 байта совпадают для пары функций,

они сохраняются в одном и том же листе дерева. Для решения такой ситуации вычисляются ЦПИ16 для байтов с позиции 33 до первого вариант байта. Но даже этот метод не позволяет распознать все функции.

### **1.4.3. Обзор инструмента KISS**

Инструмент KISS [127] определяет функции и версии библиотек (libc [128]) в исполняемом файле. Предлагаемый метод использует концепции libcdb [129] по созданию большой базы данных с различными типами и версиями библиотек, и подход технологии FLIRT [124] для идентификации библиотечных функций.

В отличие от FLIRT, где для сопоставления функций строятся хеши из первых 32 байтов функции, KISS начинает сопоставление функций, начиная с каждой точки функции.

**Создание базы данных.** Для создания базы данных libc, собираются различные версии библиотеки для разных архитектур. Далее анализируются библиотеки, определяется архитектура, извлекается .text сегмент библиотеки. Извлеченные наборы данных упаковываются в один большой файл с расширением db. В то время как это происходит, алгоритм читает таблицы символов ELF файлов и для каждого извлеченного .text сегмента записывает смещения всех содержащихся функций относительно начала .text сегмента. Размеры и записи всех библиотек объединяются и окончательный результат сохраняется в файле с расширением ofst.

На втором этапе построения загружается ранее сгенерированный файл с расширением db, и к нему применяется метод сжатия - Burrow Wheeler Transformation [130], потом кодирование Хаффмана [131], объединенное с концепцией wavelet-tree [132], генерируя окончательный .cdb сжатый файл базы данных.

**Поиск функции.** Для поиска функций используются сжатая база данных и файл смещения (.cdb и .ofst). Поиск выполняется параллельно в разных процессах. Затем байтовые строки, соответствующие функциям исполняемого файла, сопоставляются с байтовыми строками компрессированной базы данных. Для



быстрой оценки количества подходящих (сопоставляющих) мест в библиотеке используется `sdsl-lib` [133]. Также предоставляется возможность нахождения всех подходящих сопоставлений. Полученные результаты сохраняются в структуре.

**Анализ статически связанной библиотеки.** Модуль `StaticAnalyzer Python` принимает в качестве входных данных базу данных из одной или нескольких библиотек и исполняемый файл, который должен анализироваться. Затем анализатор рассматривает каждый байт исполняемого кода и пытается сопоставить блоки переменного размера из исполняемого кода и статической библиотеки. На последнем этапе все возможные функции-кандидаты извлекаются из базы данных и сравниваются с содержанием статически связанных исполняемых файлов. Сопоставленные результаты представляются пользователю. Для эффективности работа инструмента была распараллелена.

#### **1.4.4. Обзор инструмента `IdenLib`**

`IdenLib` [134] инструмент имеет плагины `x32dbg/x64dbg` [135] и `IDA Pro` [125] для идентификации функций статически линкованных библиотек. Инструмент состоит из двух частей. Первая часть генерирует сигнатуры из `.lib/.obj/.exe` файлов, далее с помощью плагина идентифицирует функции библиотек.

Генерация сигнатур состоит из следующих шагов:

1. Анализ входного файла (`.lib/.obj`) для получения списка адресов функций и имен функций
2. Извлечение последнего кода операции из каждой инструкции
3. Сжатие сигнатур с помощью `zstd`
4. Сохранение сигнатур

Для идентификации функций реализовано два метода сравнения сигнатур: точное совпадение и использование индекса Жаккарда [136]. Индекс Жаккарда находит схожие функции, но количество ложных срабатываний очень большое.

### ***1.4.5. Обзор инструмента от Шу Акабана и Такеши Окамото***

В работе Акабана и Окамото [137] [138] идентификация библиотечных функций осуществляется путем сравнения машинного кода функций, статически связанных с вредоносной программой, с кодом статических библиотечных функций в цепочках инструментов, часто используемых во встраиваемых устройствах. Для сопоставления образцов используется инструмент YARA [139]. Правило YARA для вредоносных программ — это сигнатура байтов, специфичных для этой конкретной вредоносной программы. А правило YARA для идентификации библиотеки соответствует машинному коду в функции библиотеки.

Чтобы сгенерировать правила YARA для идентификации функций, они по очереди извлекали каждую функцию из всех статических библиотек и генерировали правило YARA для каждой функции. Правило YARA состоит из шестнадцатеричной строки машинного кода каждой функции, длина которой не превышает 200 байт. Перемещаемые адреса были заменены подстановочным знаком YARA. Чтобы предотвратить ложную идентификацию маленьких функций, правила YARA не генерировались для функций размером 3 байта и меньше, за исключением подстановочных знаков.

### ***1.4.6. Обзор инструмента FunctionSimSearch.***

В рамках проекта [140] разработан метод для поиска функций статически связанных библиотек в исполняемом файле. Разработанный метод реализован в инструменте FunctionSimSearch [141]. Для каждой функции инструмент извлекает некоторые особенности, которые записываются в 128-битных векторах. В идеале две функции, скомпилированные из одного и того же исходного кода, должны иметь схожие наборы особенностей. В работе рассматриваются следующие особенности: подграфы графа потока управления, n-граммы (последовательность n инструкций) мнемоник дизассемблированных инструкций и константы. Затем из заданного набора особенностей вычисляется хеш на основе алгоритма SimHash [142]. Используя вычисленные хеш-значения, определяется сходство функций на

основе расстояния Хэмминга [143]. Алгоритм SimHash выполняет следующие шаги:

1. Инициализировать нулями вектор с размером 128
2. Для каждой особенности во входном наборе выполнить:
  1. Если бит  $n$  особенности равен 0, вычитать 1 из  $n$ -го элемента.
  2. Если бит  $n$  особенности равен 1, добавить 1 к  $n$ -му элементу.
3. Преобразование вектора путем сопоставления положительным значениям 1, отрицательным значениям  $-0$ .

Одна из проблем с описанным подходом — это то, что каждая особенность во входном наборе рассматривается с одинаковой важностью. В действительности, однако, особенности могут иметь совершенно разные значения. В расчете SimHash легко можно учесть важность отдельных особенностей: вместо добавления  $+1$  или  $-1$  в вектор можно прибавить или вычесть характерные веса. Эти веса выбираются с использованием технологии машинного обучения. Данные генерируются путем компиляции исходного кода с применением нескольких различных компиляторов и оптимизаций компилятора, а затем анализируется информация о символах для создания групп «вариантов функций» (одна и та же функция, скомпилированная различными компиляторами, добавляется в одну группу). Точно так же генерируются известные непохожие пары, путем выбора двух случайных, различных функций.

#### **1.4.7. Обзор инструмента *Library identification***

Инструмент [144] определяет версии заданных разделяемых библиотек и статически связанных библиотек в исполняемых файлах. Для решения задачи предложено шесть алгоритмов.

Сначала создается эталонная база данных библиотек, включающая разные версии, скомпилированные для архитектур x86, MIPS и ARM. Чтобы определить наиболее подходящую версию библиотеки для образца (исполняемый файл или разделяемая библиотека), вычисляется показатель сходства между файлом

образца и каждой версией каждой библиотеки (эталон). В случае статически связанных исполняемых файлов инструмент будет показывать процент сопоставленных функций между версией библиотеки, что может быть показателем того, что данная версия библиотеки действительно содержится в исполняемом файле.

### Реализованные методы сравнения:

1. **Метод bloom.** Сравниваются образец и эталон на основе фильтров Блума [145]. Для получения информации о базовом блоке используется `radare2` [146].
2. **Метод cc1.** Сравниваются значения цикломатической сложности всех функций образца со значениями всех функций в эталоне, используя расстояние Левенштейна [147]. Цикломатическая сложность  $M_f$  функции для функции  $f$ , граф потока управления которой содержит  $N_f$  узлов (т. е. базовых блоков) и  $E_f$  ребра, определяется как:  $M_f = E_f - N_f + 2$ .
3. **Метод cc2.** Сравниваются значения цикломатической сложности всех функций образца со значениями всех функций в эталоне с использованием индекса Жаккарда [136]. Этот метод справляется с переупорядочиванием инструкций.
4. **Метод cc3.** Сравниваются значения цикломатической сложности функций в образце и в эталоне на основе сигнатур [148].  
$$sig(file) = \prod_{f \in file} p_{M_f}$$
, где  $p_n$  обозначает  $n$ -е простое число.
5. **Метод str1.** Метод основан на сравнении строк. Используется расстояние Левенштейна между конкатенациями списков строк как в образце, так и в эталоне.  
$$r = 1 - \frac{Levenshtein(a_{cat}, b_{cat})}{\max(|a_{cat}|, |b_{cat}|)}$$
, где  $a_{cat}$  и  $b_{cat}$  представляют собой конкатенации всех строк из образца и эталона соответственно,  $r$ -коэффициент сходства.
6. **Метод str2.** Сравниваются наборы строк образца и эталона с использованием индекса Жаккарда [136].

#### **1.4.8. Сравнение инструментов и итоги исследования**

В ходе исследований был проведен сравнительный анализ семи наиболее популярных инструментов поиска статически связанных библиотек в исполняемом файле (Таблица 3): IDA FLIRT, Kiss, idenLib, инструмента от Акабана и Окамото, FunctionSimSearch, Library identification и Karta.

IDA FLIRT не предоставляет возможность добавления новых библиотек. idenLib и инструмент от Акабана и Окамото используют базу библиотек, собранную аналитиком. Все три инструмента не определяют версии библиотек, а только определяют функции поддерживаемых библиотек. Также, исходный код IDA FLIRT плагина и инструмента от Акабана и Окамото недоступны.

Инструмент Kiss предназначен только для поиска библиотеки LIBC. Также он предоставляет возможность добавления новых версий LIBC. С помощью инструмента возможно найти ту версию библиотеки, функции которой большей степенью сопоставились функциям входного исполняемого файла.

Инструменты FunctionSimSearch, Library identification предоставляют возможность добавления новых библиотек и используют базу библиотек, собранную аналитиком. С помощью инструментов возможно найти ту версию библиотеки, функции которой большей степенью сопоставились функциям входного исполняемого файла. FunctionSimSearch использует машинное обучение для сопоставления функций, что может занять до нескольких дней на сервере с 50 ядрами. Еще одним недостатком является то, что для обучения сети нужно собрать много вариантов функций, чтобы достигнуть большой точности.

В Library identification реализовано 6 разных методов сравнения функций, где лучший результат на тестах дал метод, сравнивающий наборы строк образца и эталона с использованием индекса Жаккарда. Но даже на простом примере, где библиотека слинкована в программе «hello world», инструмент неправильно определил версию библиотеки.

Karta предоставляет возможность автоматически определять версии связанных библиотек. Однако инструмент имеет ряд недостатков: поддерживаются только несколько библиотек, а для добавления поддержки новых библиотек необходимо вручную вводить уникальные идентификаторы, как описано выше. Определение версии основано только на специальной константной строке, в котором написана версия. В общем случае такой строки может не быть.

<b>Библиотеки</b>	<b>Поддержка разных библиотек</b>	<b>Возможность добавления новых библиотек</b>	<b>Определение функций библиотек</b>
Kiss	Нет	Нет	Да
Ida Flirt	Да	Нет	Да
Library identification	Да	Да	Нет
Karta	Да	Нет	Да

*Таблица 3. Сравнительный анализ инструментов поиска статически связанных библиотек.*

### **1.5. Выводы исследования**

На основе проведенного обзора можно сделать следующие выводы. Существующие методы нахождения клонов кода демонстрируют большую 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 основном основаны на поиске константных строк и чисел, при этом не учитывается 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 уменьшает 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.

## Глава 2. Унифицированный метод поиска клонов произвольных фрагментов кода в исходном и исполняемом коде

В этой главе представляется унифицированный метод поиска клонов произвольных фрагментов кода в исходном и исполняемом коде, основанный на графах, содержащих зависимости управления и данных программы. Этот метод поиска клонов фрагмента кода (FCD) находит клоны выделенного фрагмента функции в целевой программе. Целевая программа и фрагмент кода могут состоять как из исходного кода, так и исполняемого. При исполняемом коде фрагмент задается именем функции, начальным и конечным адресами. При исходном коде – именем функции, начальным и конечным номерами строк. Сначала исходный код или исполняемый код преобразуются в промежуточное представление (ПП). Далее для них генерируются графы зависимостей программы и на основе этих графов выполняется поиск клонов фрагмента.

### 2.1. Метод нахождения клонов фрагмента кода

Предлагаемый метод состоит из двух этапов (Рисунок 3): построение ГЗП и сопоставление этих графов. Этот этап может быть применен как к исходному коду, так и к исполняемому коду. Второй этап фокусируется на анализе и обнаружении соответствий между ГЗП фрагмента и целевого проекта. Следующая формула вычисляет проценты сходства для обнаруженных клонов фрагмента:

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

где `matchedVerticesCount` - количество сопоставленных вершин, `fragmentPDGsVerticesCount` - количество вершин ГЗП фрагмента и `similarity` - вычисленный процент сходства. Если вычисленное значение `similarity` превышает заданный процент сходства, то сопоставленные вершины считаются клонами.





Рисунок 3. Схема нахождения клонов фрагмента кода.

### 2.1.1. Построение ГЗП

ГЗП строятся для указанного фрагмента и всех функций целевого проекта. Граф зависимостей программы — это ориентированный граф, вершины которого являются инструкциями промежуточного представления программы, а ребра строятся на основе зависимостей данных и управления между ними. Примеры ГЗП приведены в разделе реализации.

**Зависимость по данным:** Пусть даны команды  $S_1$  и  $S_2$ . Говорят, что  $S_2$  зависит от  $S_1$ , если  $O(S_1) \cap I(S_2) \neq \emptyset$ , где  $I(S_2)$  — множество памяти, из которых  $S_2$  читает, а  $O(S_1)$  — множество памяти, в которые  $S_1$  записывает.

**Зависимость по управлению:** Зависимость управления определяет порядок выполнения команд. Пусть  $T(S_1)$  - множество команд, которые могут быть выполнены сразу после команды  $S_1$ . Говорят,  $S_2$  зависит от  $S_1$ , если  $S_2 \in T(S_1)$ .

Процесс построения ГЗП различается для исполняемого и исходного кода, а конкретные детали изложены в разделе реализации. Чтобы построить ГЗП для указанного фрагмента, сначала создается ГЗП для всей функции, содержащей фрагмент. Затем извлекается подграф, соответствующий указанному фрагменту, который служит в качестве окончательного ГЗП фрагмента. Это наименьший

индуцированный подграф ГЗП всей функции, который включает все инструкции указанного фрагмента. Для простоты мы будем называть его ГЗП фрагмента. Затем построенные графы используются на следующем этапе, на котором инструкции из указанных фрагментов сопоставляются со всеми инструкциями внутри функций по всему проекту.

## **2.2. Сопоставление графов зависимостей программы**

После построения ГЗП, алгоритм сопоставляет вершины ГЗП фрагмента с вершинами ГЗП каждой функции (Рисунок 4). Важно отметить, что в пределах ГЗП одной функции могут быть обнаружены несколько совпадений, что указывает на существование нескольких клонов указанного фрагмента внутри этой функции.

Алгоритм сопоставления ГЗП фрагмента и функции включает в себя два этапа:

- 1. Построение множества изначально сопоставленных пар вершин,***
- 2. Выбор нерассмотренных пар вершин из построенного множества начальных вершин и итеративное расширение.***

Первая вершина каждой пары принадлежит ГЗП фрагмента, а вторая — ГЗП функции. Соответствующие инструкции для вершин каждой пары выполняют одну и ту же операцию. Алгоритм выбирает одну из нерассмотренных пар из сформированного множества, чтобы начать процесс расширения. Из выбранной пары алгоритм временно сопоставляет ранее не сопоставленные пары вершин, используя определенные процессы. Эти процессы сопоставляют вершины на основе их характеристик и смежных ребер, гарантируя, что вершины с одинаковыми кодами операций образуют пары. Если временно сопоставленные вершины удовлетворяют всем заданным условиям, они окончательно сопоставляются. Этот процесс повторяется для всех вершин, которые еще не были сопоставлены. Фаза расширения завершается, когда новые временно сопоставленные пары больше не обнаруживаются. Результатом этого процесса является список множеств, каждое из которых содержит пары сопоставленных

вершин. Дополнительные детали приводятся далее в тексте.

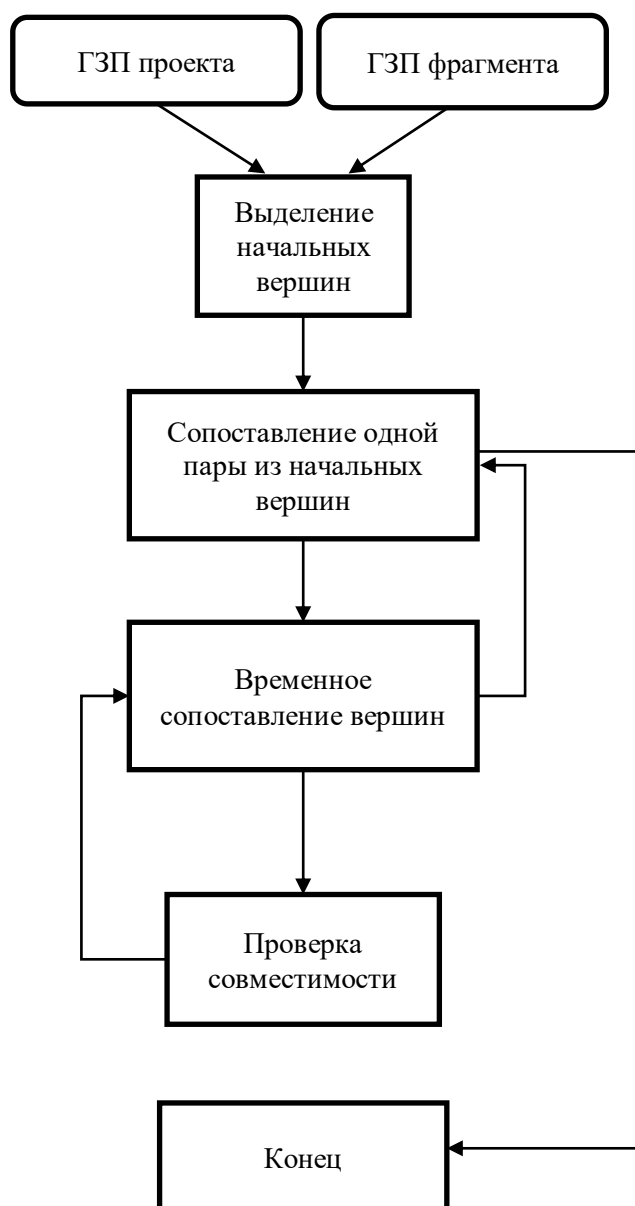


Рисунок 4. Поиск подграфов графов зависимостей программы.

Для простоты будем использовать следующие обозначения:

- *function\_PDG* - ГЗП данной функции,
- *fragment\_PDG* - ГЗП данного фрагмента,
- *intial\_pairs* - множество начальных пар  $(v, v^*)$ , где  $v \in \text{fragment\_PDG}$ ,  $v^* \in \text{function\_PDG}$ ,
- *temporarily\_matched\_pairs* - множество пар  $(v, v^*)$ , где  $v \in \text{fragment\_PDG}$ ,  $v^* \in \text{function\_PDG}$ , которые временно сопоставлены, но должны пройти несколько проверок перед окончательным сопоставлением,

- *matched\_pairs* - множество пар  $(v, v^*)$ , где  $v \in \text{fragment\_PDG}$ ,  $v^* \in \text{function\_PDG}$ , которые окончательно сопоставлены,
- *matched(G)* – множество окончательно сопоставленных пар вершин графа  $G$ ,
- *incompatible\_pairs* - множество  $(v, v^*)$  несовместимых пар вершин, где  $v \in \text{fragment\_PDG}$ ,  $v^* \in \text{function\_PDG}$ ,
- *opcode(v)* - код операции, соответствующий вершине  $v$ ,
- *pred\_ctrl(v)* / *succ\_ctrl(v)* - множество предшествующих/последующих вершин  $v$  по зависимости управления,
- *pred\_data(v)* / *succ\_data(v)*- множество предшествующих/последующих вершин  $v$  по зависимости данных,
- *bb(v)* - список вершин в том же базовом блоке, что и вершина  $v$ ,
- *pred\_bb(v)* / *succ\_bb(v)* - список вершин в базовых блоках предшественника/последователя вершины  $v$ .

### 2.2.1. Построение множества изначально сопоставленных пар вершин

Выбор начальных вершин является первым шагом поиска клонов (Рисунок 4). Цель этого шага — найти те вершины из двух ГЗП, которые наиболее вероятно соответствуют друг другу. Количество таких вершин должно быть как можно меньше для обеспечения эффективности. Именно поэтому начальные вершины выбираются по трем процессам, показавшим лучшие результаты в ходе экспериментальной оценки.

Первый процесс выбирает все вершины без входящих ребер в обоих ГЗП. Эти вершины обычно соответствуют первым инструкциям указанного фрагмента и функции. Затем из полученных множеств вершин процесс строит все возможные комбинации пар, где соответствующие инструкции выполняют одну и ту же операцию, и добавляет их в *intial\_pairs*.

Второй процесс собирает вершины с максимальным числом входящих зависимостей данных в *fragment\_PDG*. Затем он собирает вершины из *function\_PDG*, которые имеют равное или большее число входящих зависимостей данных. Подобно первому процессу, этот также создает все возможные комбинации пар из полученных множеств (гарантируя, что соответствующие инструкции выполняют одну и ту же операцию), и добавляет их

в *intial\_pairs*.

Третий процесс идентифицирует все инструкции из фрагмента кода с максимальным количеством соответствующих инструкций промежуточного представления (ПП). Затем он выбирает инструкции из функции с тем же количеством соответствующих инструкций ПП и собирает вершины, соответствующие первым инструкциям ПП упомянутых инструкций. Наконец, подобно другим процессам, он генерирует все возможные комбинации пар из полученных множеств, гарантируя, что соответствующие инструкции выполняют одну и ту же операцию, и добавляет их в *intial\_pairs*.

Если число выбранных начальных вершин велико, то есть превышает соотношение числа вершин ГЗП фрагмента к вершинам ГЗП текущей функции, то выбирается только часть выбранных начальных вершин.

### **2.2.2. Выбор нерассмотренных пар вершин из построенного множества начальных вершин и итеративное расширение**

Эта фаза начинается с выбора пары из множества *intial\_pairs*. Затем она использует эту пару в качестве отправной точки для итеративного сопоставления дополнительных пар. Для каждой итерации она временно сопоставляет нерассмотренные вершины *fragment\_PDG* и *function\_PDG*. Эти временно сопоставленные пары вершин затем проверяются на наличие нескольких условий. Если пара проходит проверки, она добавляется в список *matched\_pairs*, в противном случае она помещается в список *incompatible\_pairs*. Этот итеративный процесс сопоставления продолжается до тех пор, пока не будут обнаружены новые временно сопоставленные вершины.

**Временное сопоставление:** Этот алгоритм сопоставления включает пять различных процессов. Ниже приведены эти процессы:

1. Временное сопоставление соседних вершин через зависимости управления сопоставленных пар вершин,
2. Временное сопоставление вершин соответствующим командам базовых блоков сопоставленных пар,

3. Временное сопоставление вершин команд соседних базовых блоков сопоставленных пар,
4. Временное сопоставление соседних вершин через зависимости данных сопоставленных пар вершин,
5. Временное сопоставление вершин с помощью операционных кодов команд.

Результаты, полученные из этих процессов, затем проверяются на соответствие нескольким условиям (описанным в следующем разделе), и некоторые из временно сопоставленных пар отфильтровываются. Процесс сопоставления завершается, когда никакие новые пары вершин временно не сопоставлены, что означает, что алгоритм исчерпал все возможные соответствия между ГЗП фрагмента и ГЗП функции.

Для каждой заданной пары вершин  $(v, v^*)$ , процедура ***allow\_match(v, v^\*)*** возвращает ***true***, если они могут быть временно сопоставлены, в противном случае возвращает ***false***:

$$\begin{aligned}
 & \mathbf{allow\_match}(v, v^*) \{ \\
 & \quad \mathbf{return} \mathit{opcode}(v) == \mathit{opcode}(v^*) \wedge \\
 & \quad |\mathit{pred\_ctrl}(v)| == |\mathit{pred\_ctrl}(v^*)| \wedge \\
 & \quad |\mathit{succ\_ctrl}(v)| == |\mathit{succ\_ctrl}(v^*)| \wedge \\
 & \quad (v, v^*) \notin \mathit{matched\_pairs} \wedge \\
 & \quad (v, v^*) \notin \mathit{incompatible\_pairs} \}
 \end{aligned}$$

Во всех процессах две вершины  $(v, v^*)$  могут быть временно сопоставлены, если ***allow\_match(v, v^\*)*** возвращает ***true***. Процессы применяются в определенном порядке, и если одна пара уже сопоставлена, другие не будут применены. Изначально ***temporarily\_matched\_pairs*** пусто. Ниже приведены описания пяти процессов временного сопоставления.

**1. Временное сопоставление соседних вершин через зависимости управления сопоставленных пар вершин:** Для каждой пары  $(v, v^*) \in \mathit{matched\_pairs}$  временно сопоставляются вершины из множеств ***pred\_ctrl(v)*** и ***pred\_ctrl(v^\*)*** и добавляются в множество ***temporarily\_matched\_pairs*** (Рисунок 5):

$$temporarily\_matched\_pairs \leftarrow \left\{ (u, u^*) \left| \begin{array}{l} u \in pred\_ctrl(v) \wedge \\ u^* \in pred\_ctrl(v^*) \wedge \\ allow\_match(u, u^*) \end{array} \right. \right\}$$

Делается то же самое для вершин из множеств  $succ\_ctrl(v)$  и  $succ\_ctrl(v^*)$ :

$$temporarily\_matched\_pairs \leftarrow \left\{ (u, u^*) \left| \begin{array}{l} u \in succ\_ctrl(v) \wedge \\ u^* \in succ\_ctrl(v^*) \wedge \\ allow\_match(u, u^*) \end{array} \right. \right\}$$

Если  $temporarily\_matched\_pairs$  не пусто, выполняется переход к *шагу проверки условий*.

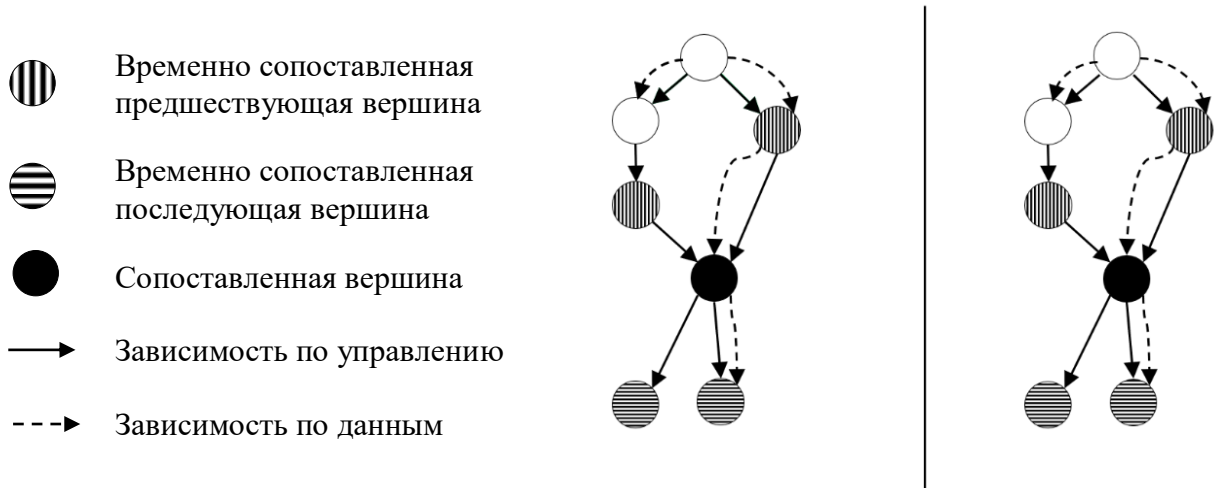


Рисунок 5. Пример сопоставления вершин через зависимости управления.

**2. Временное сопоставление вершин соответствующим командам базовых блоков сопоставленных пар:** Для каждой пары  $(v, v^*) \in matched\_pairs$  временно сопоставляются вершины из списков  $bb(v)$  и  $bb(v^*)$  и добавляются в множество  $temporarily\_matched\_pairs$  (Рисунок 6):

$$temporarily\_matched\_pairs \leftarrow \left\{ (u, u^*) \left| \begin{array}{l} u \in bb(v) \wedge \\ u^* \in bb(v^*) \wedge \\ allow\_match(u, u^*) \end{array} \right. \right\}$$

Если  $temporarily\_matched\_pairs$  не пусто, выполняется переход к *шагу проверки условий*.

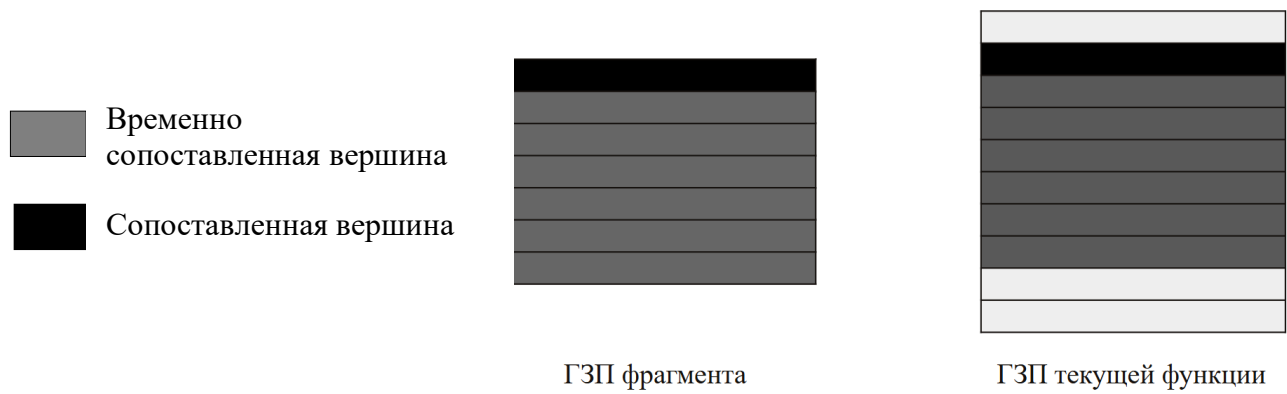


Рисунок 6. Сопоставление инструкций базовых блоков.

3. *Временное сопоставление вершин команд соседних базовых блоков сопоставленных пар:* Для каждой пары  $(v, v^*) \in \text{matched\_pairs}$  временно сопоставляются вершины из списков  $\text{pred\_bb}(v)$  и  $\text{pred\_bb}(v^*)$  и добавляются в множество  $\text{temporarily\_matched\_pairs}$  (Рисунок 7):

$$\text{temporarily\_matched\_pairs} \leftarrow \left\{ (u, u^*) \left| \begin{array}{l} u \in \text{pred\_bb}(v) \wedge \\ u^* \in \text{pred\_bb}(v^*) \wedge \\ \text{allow\_match}(u, u^*) \end{array} \right. \right\}$$

Делается то же самое для вершин из списков  $\text{succ\_bb}(v)$  и  $\text{succ\_bb}(v^*)$ :

$$\text{temporarily\_matched\_pairs} \leftarrow \left\{ (u, u^*) \left| \begin{array}{l} u \in \text{succ\_bb}(v) \wedge \\ u^* \in \text{succ\_bb}(v^*) \wedge \\ \text{allow\_match}(u, u^*) \end{array} \right. \right\}$$

Если  $\text{temporarily\_matched\_pairs}$  не пусто, выполняется переход к *шагу проверки условий*.



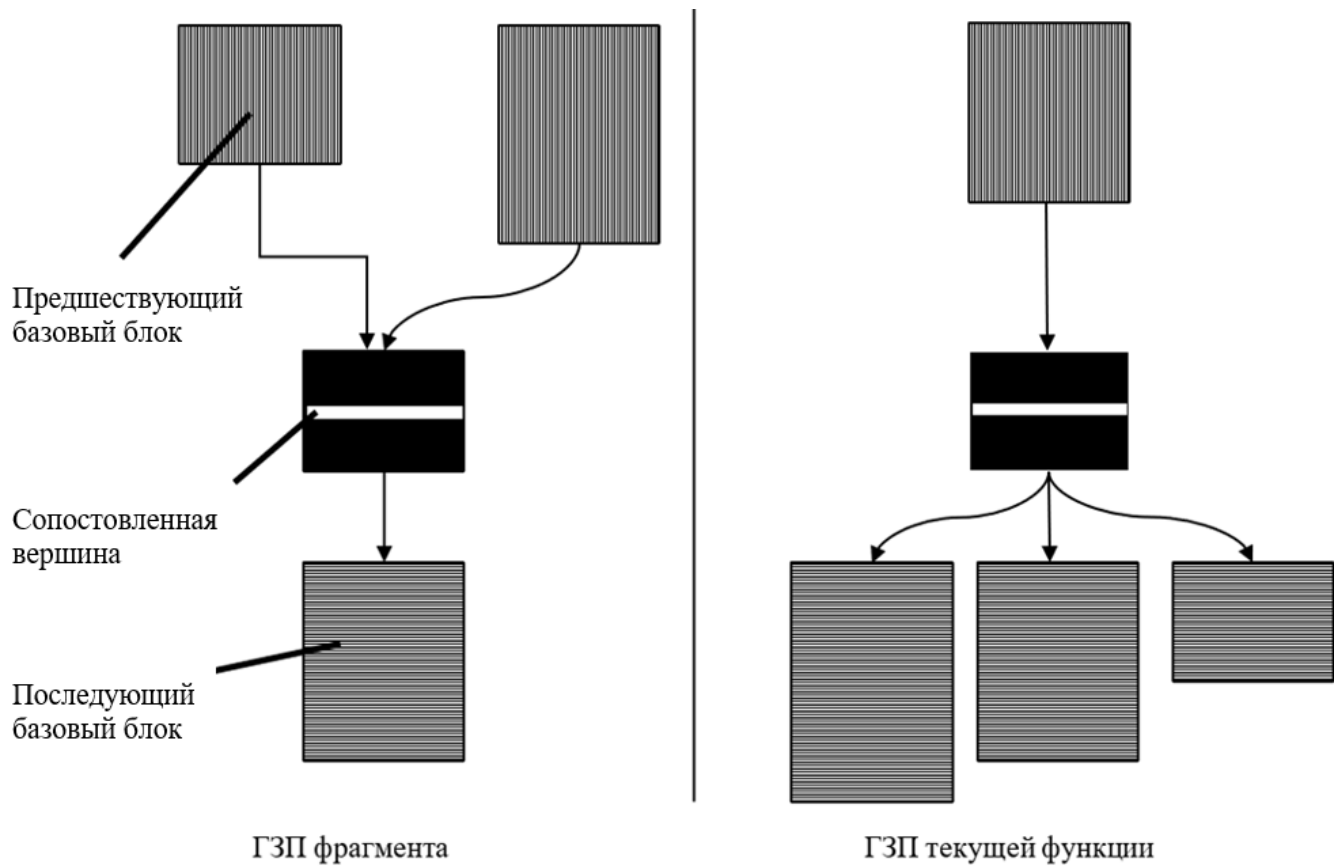


Рисунок 7. Сопоставление вершин соседних базовых блоков сопоставленной пары.






**4. Временное сопоставление соседних вершин через зависимости данных сопоставленных пар вершин:** Для каждой пары  $(v, v^*) \in \text{matched\_pairs}$  временно сопоставляются вершины из множеств  $\text{pred\_data}(v)$  и  $\text{pred\_data}(v^*)$  и добавляются в множество  $\text{temporarily\_matched\_pairs}$  (Рисунок 8):

$$\text{temporarily\_matched\_pairs} \leftarrow \left\{ (u, u^*) \left| \begin{array}{l} u \in \text{pred\_data}(v) \wedge \\ u^* \in \text{pred\_data}(v^*) \wedge \\ \text{allow\_match}(u, u^*) \end{array} \right. \right\}$$

Делается то же самое для вершин из множеств  $\text{succ\_data}(v)$  и  $\text{succ\_data}(v^*)$ :

$$\text{temporarily\_matched\_pairs} \leftarrow \left\{ (u, u^*) \left| \begin{array}{l} u \in \text{succ\_data}(v) \wedge \\ u^* \in \text{succ\_data}(v^*) \wedge \\ \text{allow\_match}(u, u^*) \end{array} \right. \right\}$$

Если  $\text{temporarily\_matched\_pairs}$  не пусто, выполняется переход к *шагу проверки условий*.

-  Предшествующая вершина
-  Последующая вершина
-  Сопоставленная вершина
-  Зависимость по управлению
-  Зависимость по данным

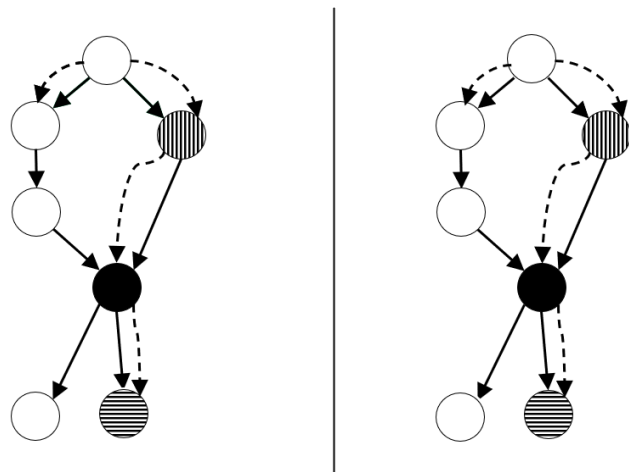


Рисунок 8. Пример сопоставления вершин через зависимости данных.

### 5. Временное сопоставление вершин с помощью операционных кодов команд

Этот метод является простейшим методом сопоставления, который сопоставляет пару «начальных» вершин, которые еще не сопоставлены, не являются несовместимыми и имеют одинаковый код операции.

$$temporarily\_matched\_pairs \leftarrow \left\{ (u, u^*) \left| \begin{array}{l} (u, u^*) \in initial\_pairs \wedge \\ (u, u^*) \notin matched\_pairs \wedge \\ (u, u^*) \notin incompatible\_pairs \wedge \\ pcode(u) == opcode(u^*) \end{array} \right. \right\}$$

**Проверка совместимости:** Следующий шаг — проверка временно сопоставленных пар. После каждой итерации временного сопоставления каждая пара  $(v, v^*) \in temporarily\_matched\_pairs$  проверяется на соответствие нескольким условиям. Если пара удовлетворяет всем условиям, она перемещается в *matched\_pairs*, в противном случае — в *incompatible\_pairs*. Условия описаны ниже:

1. Функция *pred\_condition*( $v, v^*$ ) возвращает *false*, если  $\exists p \in pred\_ctrl(v)$ , такой что  $p \in matched(fragment\_PDG)$ , и  $\nexists p^* \in function\_PDG$ , для которого  $p^* \in pred\_ctrl(v^*)$  и  $(p, p^*) \in matched\_pairs$ . В противном случае возвращает *true*.
2. Функция *succ\_condition*( $v, v^*$ ) возвращает *false*, если  $\exists s \in succ\_ctrl(v)$ , такой что  $s \in matched(fragment\_PDG)$ , и  $\nexists s^* \in function\_PDG$ , для

которого  $s^* \in succ\_ctrl(v^*)$  и  $(s, s^*) \in matched\_pairs$ . В противном случае возвращает *true*.

## 2.3. Реализация

Обнаружение клонов фрагментов кода реализовано в инструменте FCD. Инструмент командной строки принимает следующие входные данные:

1. Путь к проекту и имя функции, содержащей фрагмент кода для анализа.
2. Границы фрагмента кода:
  - a. Для исходного кода — начальные и конечные номера строк.
  - b. Для исполняемого кода — начальные и конечные адреса памяти.
3. Проект, в котором необходимо искать клоны указанного фрагмента.
4. Необязательный параметр минимального процента сходства для фильтрации клонов, которые менее похожи, чем указанное значение. Этот параметр принадлежит интервалу  $(0, 100]$  и имеет значение по умолчанию 90%.

Процесс генерации ГЗП различается для исходного и исполняемого кода, однако совпадающие части графа остаются идентичными.

### 2.3.1. Построение ГЗП исходного кода

Для построения ГЗП исходного кода используются три подхода. Первый подход использует промежуточное представление LLVM [149]. Чтобы получить ГЗП для исходного кода, используется новый проход, добавленный в компилятор LLVM Clang [122], который использует информацию о зависимостях управления, цепочки use-def [150] и анализ псевдонимов (alias). В этом случае построение целевых проектов для извлечения ПП LLVM является явным требованием. Таким образом, при этом подходе ГЗП могут быть сгенерированы только для проекта, который может быть успешно скомпилирован. Однако могут быть случаи, когда фрагмент целевого кода и проект не могут быть скомпилированы. Чтобы решить эту проблему, мы также реализовали построение ГЗП на основе ANTLR [151]. Он использует сгенерированные с помощью ANTLR деревья разборов и требует

только, чтобы проект был синтаксически правильным, без необходимости в полной компиляции проекта. Процесс построения ГЗП для каждой функции включает три основных этапа. Во-первых, деревья разбора анализируются для создания графов зависимостей по управлению. Во-вторых, производится анализ достигающих определений и строятся цепочки use-def. Наконец, генерируются графы зависимостей программы. Метод, основанный на ПП LLVM приемлем для анализа кода написанных на языках C, C++, Objective-C, а метод на основе ANTLR – для C, C++, Java. В дополнение к этим двум подходам также поддерживается построение ГЗП из байт-кода Java. Для реализации этого добавлен компонент генерации ГЗП в проект ASM [152].

ГЗП на основе LLVM имеют более точные зависимости данных благодаря доступному анализу use-def, def-use и points-to для ПП LLVM. Хотя ГЗП на основе ANTLR менее точны, то есть некоторые ребра могут отсутствовать, их качество компенсируется более широкими возможностями их построения, даже для исходного кода, который не может быть скомпилирован. На рисунках 9, 10 и 11 представлены примеры ГЗП на основе LLVM ПП, ANTLR и байт-кода Java.

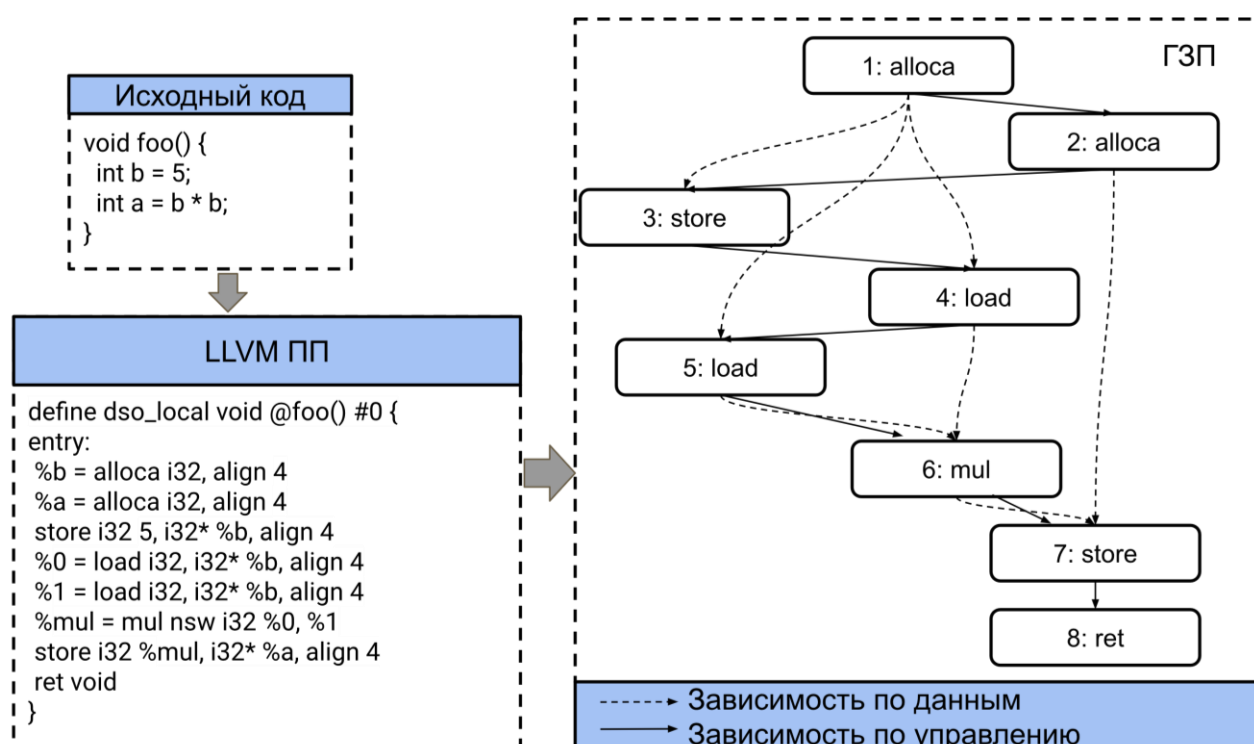


Рисунок 9. Пример ГЗП на основе LLVM ПП.

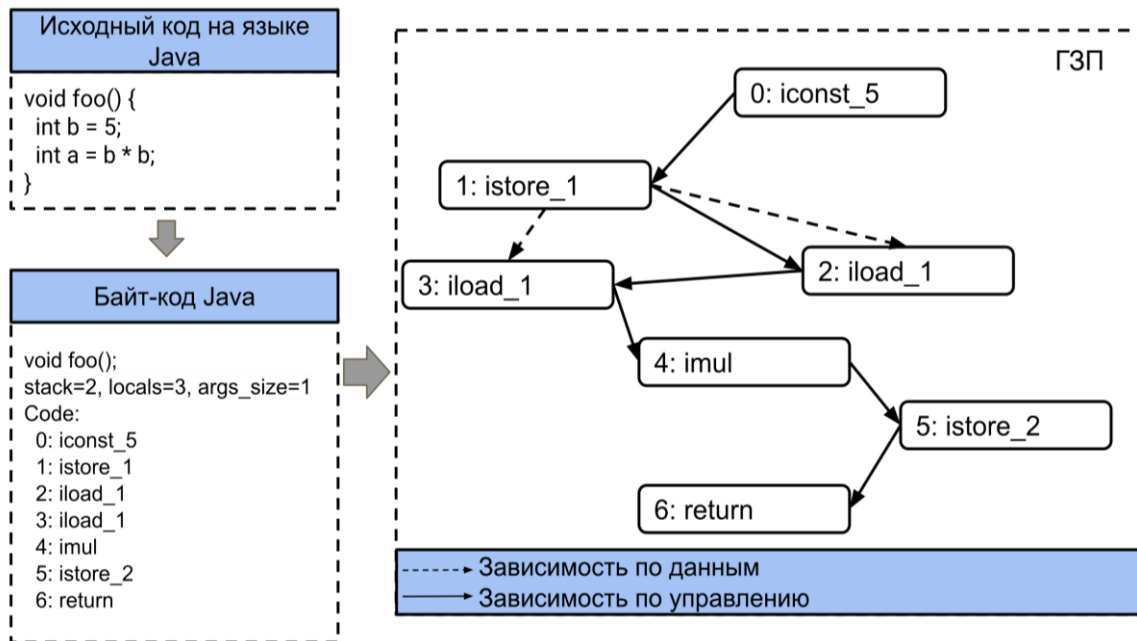


Рисунок 10. Пример ГЗП на основе байт-кода Java.

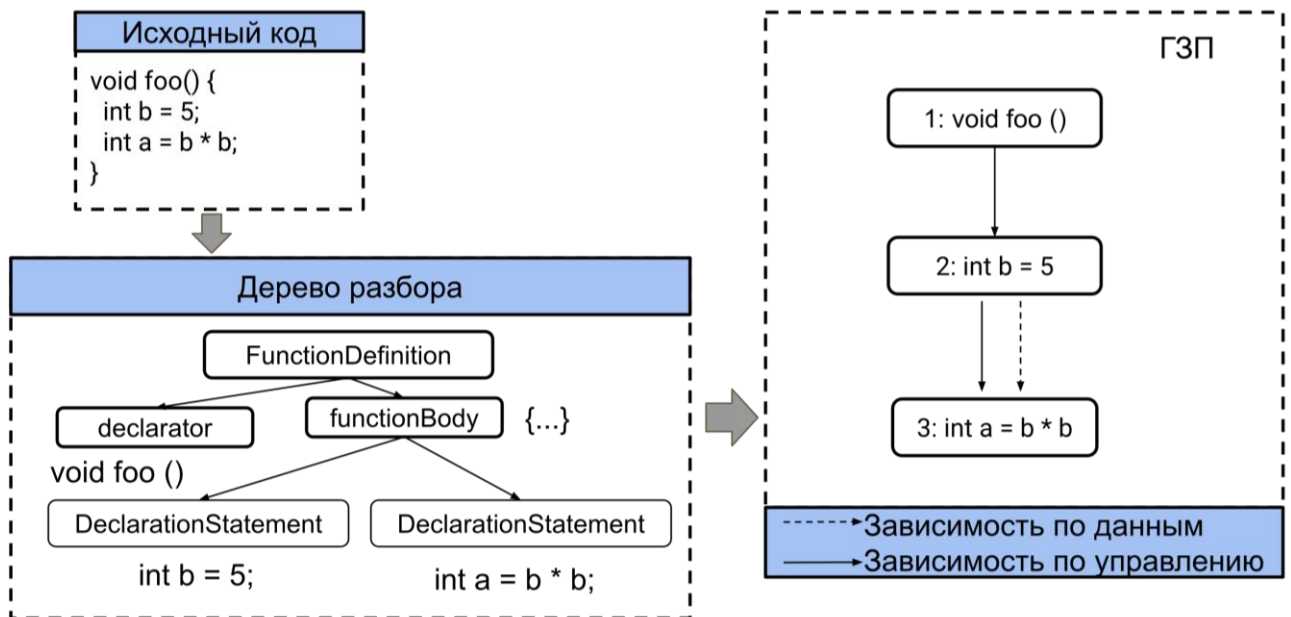


Рисунок 11. Пример ГЗП на основе ANTLR.

### 2.3.2. Построение ГЗП исполняемого кода

Для генерации ГЗП исполняемого кода используется промежуточное представление REIL [153]. Во-первых, исполняемый код дизассемблируется с помощью IDA Pro [125], так как он поддерживает множество форматов исполняемых файлов для ряда архитектур и автоматически восстанавливает графы зависимостей управления. Затем полученный ассемблер транслируется в

промежуточное представление REIL с помощью BinNavi [103]. Наконец, для каждой функции строится ГЗП с помощью инструмента BinSide [5]. На рисунке 12 представлен ГЗП на основе REIL.

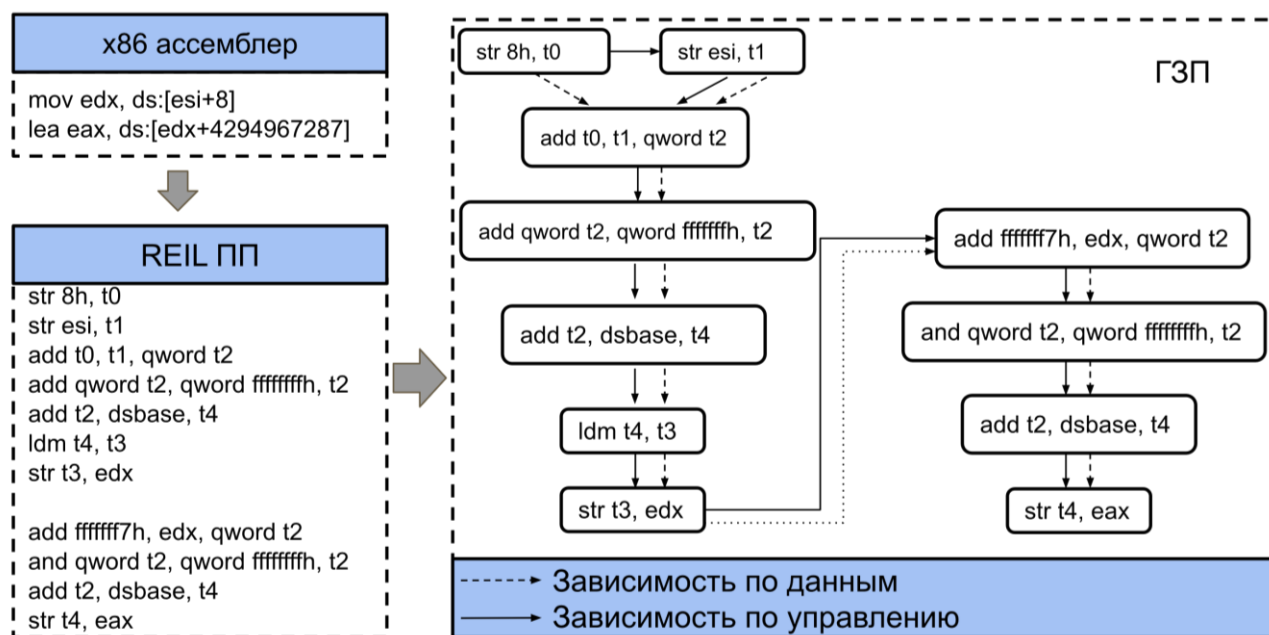


Рисунок 12. Пример ГЗП на основе REIL представления.

### 2.3.3. Реализация алгоритма сопоставления графов

Построенные ГЗП используются для идентификации клонов фрагментов кода как для исходного, так и для исполняемого кода. Алгоритм обнаружения клонов фрагментов кода написан на языке C++. Вывод инструмента состоит из файлов JSON, содержащих информацию об обнаруженных клонах. Эта информация включает имена функций, соответствующих сопоставленным фрагментам, процент сходства, все пары совпавших инструкций и другие детали.

### 2.4. Оценка точности и полноты

Для оценки качества инструмента была разработана и реализована тестовая система, которая оценивает точность и полноту инструмента рядом методов тестирования. Разработанная тестовая система, принимая в качестве входных данных ГЗП некоторого проекта, создает тесты для оценки точности и полноты инструмента. Оценка выполняется с использованием ниже приведенных формул.

$$\text{Точность} = \frac{\text{количество найденных истинно — положительных пар}}{\text{количество всех найденных пар}}$$

$$\text{Полнота} = \frac{\text{количество найденных истинно — положительных пар}}{\text{количество всех истинных пар}}$$

$$F1 \text{ — мера} = 2 \times \frac{\text{Точность} \times \text{Полнота}}{\text{Точность} + \text{Полнота}}$$

Точность отражает, насколько правильно были найдены клоны, а полнота — сколько из всех возможных клонов было обнаружено. F1-мера является комбинированной оценкой, которая учитывает как точность, так и полноту, обеспечивая более полное понимание эффективности метода.

Ниже приведены методы генерации тестов:

- **Поиск базовых блоков** — наибольший базовый блок каждой функции проекта выбирается в качестве фрагмента кода, для которого выделяется ГЗП и выполняется поиск клонов этого фрагмента в ГЗП текущей функции.
- **Поиск циклов** — наибольший цикл каждой функции проекта считается указанным фрагментом кода, для которого выделяется ГЗП и выполняется поиск клонов этого фрагмента в ГЗП текущей функции.
- **Поиск модифицированных базовых блоков** — наибольший по количеству команд базовый блок каждой функции проекта считается указанным фрагментом кода, для которого выделяется ГЗП, изменяются некоторые команды и выполняется поиск клонов этого фрагмента в ГЗП текущей функции. Было внесено 4 типа изменений:
  - LB — Команда, находящаяся в базовом блоке, заменяется командой LOWER\_BOUND. LOWER\_BOUND — искусственно добавленная команда с наименьшим значением идентификатора.

- UB - Команда, находящаяся в базовом блоке, заменяется командой UPPER\_BOUND. UPPER\_BOUND — искусственно добавленная команда с наибольшим значением идентификатора.
- INC - Команда, находящаяся в базовом блоке, заменяется последующей командой.
- DEC - Команда, находящаяся в базовом блоке, заменяется предшествующей командой.
- **Поиск клонов с низким сходством** — из ГЗП каждой функции проекта удаляется X% вершин и делается попытка нахождения ее клона в оригинале (Таблица 4 и Таблица 5) с 100 - X% схожестью.

Инструмент тестировался с использованием реализованной системы тестирования на проектах OpenSSL, Jasper, c-ares и rsync для всех функций. Тест проводился на компьютере с четырехъядерным процессором Intel Core i5-3470 3600 МГц и 16 ГБ оперативной памятью, используя одно ядро. В таблицах 4 и 5 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. Стоит отметить, что сходство между модифицированными и оригинальными ГЗП может не быть точно равным заданному проценту сходства, поскольку в процессе модификации происходят дискретные шаги по снижению сходства за счет поочередного удаления вершин. Например, при задании уровня сходства 70% система может вернуть клоны, которые имеют 69,5% сходства. В таких случаях система обновляет процент сходства и использует немного более низкий порог для последующего обнаружения клонов. 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. В среднем FCD имеет 97%



точности и 94% полноты для исходного кода. Для исполняемого кода в среднем FCD имеет 97% точности и 93% полноты.

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

*Таблица 4. Поиск клонов с низким сходством в исходном коде.*

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 секунд, 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.

Таблица 5 содержит результаты тестирования FCD для анализа исполняемого кода тех же проектов, скомпилированных для различных архитектур (x86-64, x86, ARM).

Проект	Размер исполняемого кода	Скорость FCD	Архитектура	Точность (%)	Полнота (%)	F1-мера (%)
libcares 2.3.0 (c-ares 1.15.0)	86 КБ	41с	x86-64	98.9	95.6	97.2
libcares 2.3.0 (c-ares 1.15.0)	96 КБ	43с	x86	97.9	93.4	95.6
libcares 2.3.0 (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 5. Поиск клонов с низким сходством в исполняемом коде.

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.

## 2.5. Масштабируемость

Чтобы проверить масштабируемость инструмента, был проведен поиск клонов фрагментов кода на некоторых крупных проектах. Тест проводился на компьютере с четырехъядерным процессором Intel Core i5-3470 3600 МГц и 16 ГБ оперативной памятью. Результаты представлены в таблице 6.

Проект	Размер	Время поиска	Количество функций	Количество строк
Linux-5.7-rc4	1.1 ГБ	85 сек.	279330	~27 млн
Postgresql-12.2	141 МБ	47 сек.	17462	~1,3 млн
Apache/httpd-2.4.43	60 МБ	3 сек.	4441	~350 тыс.
FFmpeg-n2.8.16	62 МБ	61 сек.	13307	~700 тыс.

*Таблица 6. Результаты масштабируемости.*

## 2.6. Сравнение с существующими инструментами

Инструмент FCD также был сравнен с инструментами, разработанными С. Саргсяном и А. Асланяном в рамках их диссертационных работ. Сравнение инструментов показало (Таблица 7), что FCD обладает наивысшей универсальностью, поддерживая анализ исходного и исполняемого кода, а также поиск клонов, при этом обеспечивая высокие показатели точности (97%) и полноты (93%), что соответствует значению F1-меры в 95%.

Инструмент	Точность (%)	Полнота (%)	F1-мера (%)	Анализ исходного кода	Анализ исполняемого кода	Поиск клонов фрагментов
FCD	97	93	95	Да	Да	Да
Инструмент, разработанный С.Саргсяном	99	9	17	Да (при наличии опций компиляции)	Нет	Нет
Инструмент, разработанный А.Асланяном	80	45	58	Нет	Да	Нет

Таблица 7. Сравнение инструмента FCD с существующими инструментами.

Инструмент С. Саргсяна достигает максимальной точности (99%), но ограничен анализом исходного кода при наличии опций компиляции и имеет низкую полноту (9%), что приводит к значению F1-меры всего в 17%. Инструмент А. Асланяна ориентирован исключительно на анализ исполняемого кода, показывая умеренные результаты по точности (80%) и полноте (45%), что соответствует значению F1-меры в 58%. Также, в отличие от остальных работ, FCD поддерживает поиск клонов заданного фрагмента, а не целой функции.

## 2.7. Оценка FCD на BigCloneBench

FCD также был оценен на наборе данных BigCloneBench [84], обеспечив тщательную и детальную оценку широкого спектра типов клонов. В результате было установлено, что полнота обнаружения клонов превышает 91%, что позволяет инструменту превосходить другие аналогичные решения. Результаты оценки представлены в таблице 8.

Стоит отметить, что по мере уменьшения процента сходства также начинает уменьшаться полнота. Основной причиной этого является то, что в инструменте FCD и наборе данных BigCloneBench используются разные определения сходства. FCD использует определение сходства на уровне ПП, тогда как BigCloneBench использует определение сходства на уровне исходного кода. Когда сходство на уровне исходного кода высокое, соответствующие ГЗП схожи. Однако по мере увеличения различий в исходном коде, соответствующие ГЗП могут быстро меняться. Это становится очевидным из таблицы для сильно измененного типа-3, где ожидаемое сходство должно находиться в интервале [70%, 90%], а FCD обеспечивает более низкую полноту.

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 8. Результаты оценки на BigCloneBench.

Кроме того, в таблице 8 показаны 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 сильно измененного типа-3 и слабо измененного типа-3 полнота значительно падает. FCD демонстрирует лучшие показатели полноты 99% и 91% соответственно. Второй лучший результат демонстрирует NiCad7 с 98% и 73% полнотой соответственно. Для сравнения, FCD демонстрирует более высокие показатели полноты 99% и 91% соответственно.

## 2.8. Выводы

В данной главе был представлен унифицированный метод поиска клонов фрагментов кода в исходном и исполняемом коде. Разработанный метод основан на ГЗП, что позволяет учитывать не только синтаксические, но и семантические особенности кода. Это делает возможным обнаружение клонов более сложных типов, включая клоны, отличающиеся изменением порядка инструкций или модификациями логики.

Экспериментальная оценка предложенного метода показала его эффективность. В ходе тестирования разработанный инструмент продемонстрировал точность и полноту свыше 90%, что делает метод применимым в различных задачах анализа программного обеспечения. Кроме того, была подтверждена его масштабируемость – метод способен эффективно работать на больших объемах кода (десятки миллионов строк исходного и соответствующего исполняемого кода), обеспечивая анализ в несколько часов.

Таким образом, разработанный метод может применяться для поиска схожих фрагментов кода, что актуально для ряда задач анализа кода, включая рефакторинг, выявление потенциальных уязвимостей, связанных с многократным использованием кода и т. д.

## **Глава 3. Метод оптимизации программ при помощи поиска клонов**

В третьей главе описан метод оптимизации программ, использующих вычисление циклической проверки избыточности (ЦПИ, Cyclic Redundancy Check, CRC) [154]), при помощи поиска клонов и подстановки эффективных реализаций ЦПИ с учетом аппаратной платформы.

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 переносов, или прямые реализации с использованием инструкции ЦПИ.

### **3.1. Применения циклической проверки избыточности**

ЦПИ — широко распространенный метод обнаружения ошибок, интегрированный в цифровые сети, протоколы связи, системы хранения данных и т. д. Его основное назначение — обнаружение случайных изменений цифровых данных, вызванных шумом в каналах передачи. Процесс отправки данных включает добавление коротких проверочных значений к блокам данных. Эти значения рассчитываются с помощью алгоритма ЦПИ, который основан на остатке полиномиального деления содержимого каждого блока данных. При извлечении данных расчет ЦПИ повторяется, и любые различия в полученных значениях указывают на потенциальное повреждение данных, требующее мер по исправлению. Кроме того, ЦПИ можно использовать для исправления обнаруженных ошибок.

Алгоритм ЦПИ глубоко интегрирован в основу многих программных систем, включая протоколы связи, системы хранения данных, алгоритмы сжатия и механизмы обновления прошивки/программного обеспечения. Например, Ethernet

[155], HDLC [156] и USB [157] используют ЦПИ для обнаружения и исправления ошибок в передаваемых данных. Более того, он широко используется в ядре Linux [158], в частности, в файловой системе ext4 [159], при обработке сетевых пакетов [160], в протоколе «точка-точка» (PPP) [161] и т.д.

В результате широкое использование алгоритма ЦПИ в существующем программном обеспечении подчеркивает важность выявления и оптимизации их реализаций. Побитовая или неоптимизированная реализация ЦПИ может привести к снижению производительности, особенно в системах, которые обрабатывают большие объемы данных или имеют строгие требования к задержке. Обнаружение и замена таких реализаций более быстрыми и оптимизированными альтернативами может значительно повысить производительность и эффективность систем.

### **3.2. Циклическая проверка избыточности**

Алгоритм вычисления ЦПИ — это метод обнаружения ошибок на основе деления полиномов. Основная идея алгоритма ЦПИ заключается в обработке входного сообщения как большого двоичного числа и выполнении полиномиального деления с использованием фиксированного делителя полинома [162]. Остаток от этой операции деления становится контрольной суммой ЦПИ, которая добавляется к переданному блоку данных.

Выбор размера и значения полинома также может повлиять на производительность и эффективность алгоритма ЦПИ. Размер полинома обычно измеряется в битах и определяет количество бит в контрольной сумме ЦПИ. Большой размер полинома может обеспечить лучшие возможности обнаружения ошибок, но также он может увеличить вычислительную сложность алгоритма. Значение полинома также важно, так как оно определяет конкретные возможности обнаружения ошибок алгоритма. Размер блока данных — еще один важный параметр, так как он влияет на вычислительную сложность и требования к памяти алгоритма. Например, алгоритм ЦПИ, разработанный для небольших пакетов, может не подходить для больших файлов. Алгоритм ЦПИ имеет



различные реализации, включая побитовые и оптимизированные подходы, а также различные вариации порядка бит.

### **3. 2. 1. Реализации ЦПИ**

#### **3.2.1.1. Побитовая реализация ЦПИ**

Побитовая реализация ЦПИ [163] (Рисунок 13) — это простая реализация алгоритма ЦПИ. Она обрабатывает входные данные побитно, выполняя побитовые операции XOR и SHIFT для вычисления контрольной суммы ЦПИ.

```
1. #define POLYNOM 0x07
2. uint8_t crc8 (uint8_t crcValue, uint8_t data) {
3.     uint8_t i;
4.     crcValue = crcValue ^ data;
5.     for (i = 0; i < 8; i++) {
6.         if (crcValue & 0x80) {
7.             crcValue = (crcValue << 1);
8.             crcValue = crcValue ^ POLYNOM;
9.         } else {
10.            crcValue = (crcValue << 1);
11.        }
12.    }
13.    return crcValue;
14. }
```

*Рисунок 13. Пример побитовой реализации ЦПИ на языке C.*

Несмотря на простоту понимания и реализации, побитовый подход не является самым эффективным для больших объемов входных данных.

#### **3.2.1.2. Табличная реализация ЦПИ**

Табличная реализация ЦПИ (Рисунок 14, пример взят из [164]) оптимизирует вычисление ЦПИ, предварительно вычисляя значения ЦПИ для всех возможных значений байта и сохраняя их в таблице поиска. Она обрабатывает входные данные побайтно, используя предварительно вычисленную таблицу для обновления значения ЦПИ. Табличный метод, как правило, быстрее, но требует больше памяти для хранения предварительно вычисленной таблицы

поиска.

```
1. const unsigned short crc16_tab[256] = { 0x0000,0x1021,0x2042, ...  
    ,0x1ef0};  
2. unsigned short fast_crc16(unsigned short crc, unsigned char *buffer,  
    unsigned long len) {  
3.     while (len--) {  
4.         crc = (crc << 8) ^ crc16_tab[(crc >> 8) ^ *buffer++] ;  
5.     }  
6.     return crc;
```

*Рисунок 14. Пример табличной реализации ЦПИ на языке C.*

### **3.2.1.3. Реализация на основе умножения без переноса**

Умножение без переноса (УБП) [165] — это метод умножения двух чисел или многочленов без необходимости обработки операций переноса в отличие от стандартных алгоритмов умножения. Метод основан на арифметике поля Галуа [166] и предлагает несколько преимуществ по сравнению с традиционными методами умножения многочленов.

УБП также используется для реализации операции умножения многочленов, необходимой для вычислений ЦПИ [167].

### **3.2.2. ЦПИ с прямым и обратным порядком бит**

Реализации ЦПИ также могут различаться по порядку битов, используемому для обработки входных данных и генерации контрольной суммы ЦПИ. Существует два основных способа обработки битов: в прямом порядке (бит-вперед) и в обратном порядке (бит-назад).

#### **3.2.2.1. ЦПИ с прямым порядком бит**

В реализации с прямым порядком бит данные обрабатываются от старшего значащего бита (СЗБ) к младшему значащему биту (МЗБ). В этой модели расчет ЦПИ начинается с СЗБ каждого байта и продвигается к МЗБ. Такой подход называется прямым порядком бит, и он обычно используется в традиционных системах, где биты обрабатываются поочередно с самой старшей позиции.

Пример на рисунке 13 представляет собой реализацию прямого порядка.

### 3.2.2.2. ЦПИ с обратным порядком бит

В реализации с обратным порядком бит данные обрабатываются от МЗБ к СЗБ (Рисунок 15, пример из [168]). В этом случае расчет ЦПИ начинается с МЗБ каждого байта и продвигается к СЗБ. Кроме того, в этом подходе сам многочлен тоже перевернут. Эта техника применяется в некоторых системах, где битовые данные обрабатываются в противоположном порядке.

```
1. ee_u16 crcu8(ee_u8 data, ee_u16 crc) {
2.   ee_u8 i = 0, x16 = 0, carry = 0;
3.   for (i = 0; i < 8; i++) {
4.     x16 = (ee_u8)((data & 1) ^ ((ee_u8)crc & 1));
5.     data >>= 1;
6.     if (x16 == 1) {
7.       crc ^= 0x4002;
8.       carry = 1;
9.     } else
10.      carry = 0;
11.     crc >>= 1;
12.     if (carry)
13.       crc |= 0x8000;
14.     else
15.       crc &= 0x7fff;
16.   }
17.   return crc;
18. }
```

Рисунок 15. Пример реализации ЦПИ с обратным порядком бит.

Кроме того, в пределах каждой вариации порядка битов могут быть дополнительные подтипы, основанные на начальном значении ЦПИ и конечном возвращаемом значении. Например, начальное значение может быть полностью нулями или полностью единицами, а конечное значение может быть как исходным результатом ЦПИ, так и результатом ЦПИ, подвергнутому операции XOR со всеми единицами [162] или с другими значениями.

Таким образом, широкий спектр подходов к реализации ЦПИ, от простого побитового метода до более оптимизированных табличных и основанных на УБП методов, предоставляет разработчикам множество вариантов выбора, позволяя им

выбрать наиболее подходящую реализацию для конкретных требований и ограничений своего приложения.

### **3.3. Описание предлагаемого метода**

Предлагаемый метод начинается с обнаружения потенциальных реализаций побитового ЦПИ путем проверки различных особенностей и использует индивидуальную технику символического выполнения для верификации выявленных кандидатов. Впоследствии метод заменяет фрагменты кода верифицированной реализации побитового ЦПИ более эффективными альтернативами, такими как реализации на основе таблиц поиска с использованием инструкции CLMUL или прямым применением инструкции ЦПИ.

### **3.4. Символическое выполнение на битовом уровне**

Символическое выполнение [169] — это метод, применяемый для систематического анализа поведения программ через исследование всех возможных путей их выполнения. Вместо конкретных входных данных символическое выполнение использует символические значения, которые присваиваются входным переменным. Программа выполняется с этими символическими данными, формируя выражения для каждого возможного пути. Для решения таких выражений механизмы символического выполнения часто используют решатели SMT [170], способные генерировать конкретные значения для полученных выражений. В предложенном методе распознавания ЦПИ символическое выполнение проводится на уровне отдельных битов, что позволяет анализировать каждый бит переменных программы по отдельности. Вместо использования существующих универсальных механизмов символического выполнения был разработан специализированный и легкий механизм, адаптированный под поставленную задачу. Кроме того, традиционные решатели SMT не применяются, поскольку уравнения, возникающие во время этого

процесса, имеют низкую сложность и могут быть решены без их помощи.

Этот специализированный механизм символического выполнения использует абстракцию *Состояния* для отслеживания всех значений переменных и условий на каждом пути программы. Переменные в объекте *Состояния* сохраняются как битовые векторы в двоичном формате. Например, переменная  $int8_t x = 3$ ; представлена битовым вектором (0,0,0,0,0,0,1,1). Размер битового вектора равен размеру типа переменной; например, для  $int8_t$  он равен 8 бит. Для переменных с неопределенными значениями создаются битовые векторы с символическими битами. Например, неинициализированная переменная  $y$  с типом  $int8_t$  представлена битовым вектором ( $y_7, y_6, y_5, y_4, y_3, y_2, y_1, y_0$ ).

Механизм символического выполнения поддерживает операции, которые влияют на значения переменных в объекте *Состояния*, такие как AND, OR и XOR. Полный список поддерживаемых операций и их операндов описан ниже:

- *AND* (Состояние, Результат, Операнд\_1, Операнд\_2)
- *OR* (Состояние, Результат, Операнд\_1, Операнд\_2),
- *XOR* (Состояние, Результат, Операнд\_1, Операнд\_2),
- *SHIFT\_RIGHT* (Состояние, Результат, Операнд\_1, Операнд\_2),
- *SHIFT\_LEFT* (Состояние, Результат, Операнд\_1, Операнд\_2),
- *ADD* (Состояние, Результат, Операнд\_1, Операнд\_2),
- *SUB* (Состояние, Результат, Операнд\_1, Операнд\_2),
- *MUL* (Состояние, Результат, Операнд\_1, Операнд\_2),
- *COMPLEMENT* (Состояние, Результат, Операнд).

Первый параметр каждой операции — объект *Состояния*, который содержит переменные и их значения. Второй параметр — целевая переменная, в которую записывается результат операции. Остальные параметры — представляют собой входные значения для операции. Важно отметить, что приведенный выше множество поддерживаемых операций охватывает все основные инструкции (поддержка условий описана ниже), используемые для побитового вычисления ЦПИ. Таким образом, нет необходимости поддерживать все возможные

инструкции, поскольку символическое выполнение применяется исключительно для тела цикла, отвечающего за расчет ЦПИ. Если в процессе выполнения обнаруживаются инструкции, не относящиеся к вычислению ЦПИ, выполнение прерывается с сообщением о том, что анализируемый код не предназначен для вычисления ЦПИ.

Так как значения переменных хранятся как бит-векторы, операции выполняются на уровне бит. Например, после инструкции  $XOR(St, dest, 7, 10)$   $dest$  будет иметь следующее значение:

$$dest = (0,0,0,0,0,1,1,1)^{(0,0,0,0,1,0,1,0)} = (0^0,0^0,0^0,0^0,0^0,1^1,1^1,1^0) = (0,0,0,0,1,1,0,1).$$

Если переменная, участвующая в операции, имеет символическое значение, то биты в результате будут иметь форму выражения. Например, после команды на строке 15 ( $crc = 0x7fff;$ ) из рисунка 15 представление  $crc$  на уровне бит будет:  $AND(Si, crc, crc, 0x7fff) = (crc_{15}, crc_{14}, \dots, crc_0) \& (0,1,1, \dots, 1) = (crc_{15} \& 0, crc_{14} \& 1, \dots, crc_0 \& 1) = (0, crc_{14}, \dots, crc_0).$

Последний пример показывает, что также выполняются простые оптимизации. Некоторые из них описаны ниже:

- $var \& 0 = 0, 0 \& var = 0,$
- $var \& 1 = var, 1 \& var = var,$
- $var | 0 = var, 0 | var = var$
- $var | 1 = 1, 1 | var = 1$
- $var \wedge var = 0$

Помимо вышеперечисленных операций, механизм символического выполнения обеспечивает операции по сбору условий пути. Список этих условий описан ниже:

- $EQUAL(STATE, OP1, OP2),$
- $NOT\_EQUAL(STATE, OP1, OP2),$
- $GREATER\_THAN(STATE, OP1, OP2),$
- $LESS\_THAN(STATE, OP1, OP2),$

- $GREATER\_OR\_EQUAL(STATE, OP1, OP2)$ ,
- $LESS\_OR\_EQUAL(STATE, OP1, OP2)$ .

Добавленные условия пути также сохраняются в битовом представлении. Например, для условия на строке 6 ( $x_{16} == 1$ ) из рисунка 15 для объекта текущего состояния  $St$  добавляется условие  $EQUAL(St, x_{16}, 1)$ :

$$(x_{16_{15}}, x_{16_{14}}, \dots, x_{16_0}) == (0, 0, \dots, 0, 1) = (x_{16_{15}} == 0, x_{16_{14}} == 0, \dots, x_{16_0} = 1).$$

Для более сложного условия  $GREATER\_THAN(St, x, y)$  это будет:  $(x_7, x_6, x_5, x_4, x_3, x_2, x_1, x_0) > (y_7, y_6, y_5, y_4, y_3, y_2, y_1, y_0) = \{x_7 > y_7 || (x_7 == y_7 \&\& (x_6 > y_6 || (x_6 == y_6) \&\& \dots (x_0 > y_0) \dots))\}$

Механизм символического выполнения начинает свою работу с создания начального состояния  $S_0$ . Во время выполнения, когда встречается условие ветвления, текущий объект состояния дублируется, добавляя условие пути к исходному состоянию и противоположное условие к дублированному состоянию. Например, когда выполнение достигает условия на строке 6 из рисунка 15 ( $if (x_{16} == 1)$ ) с некоторым состоянием  $S_i$ , то состояние  $S_i$  дублируется в новое состояние  $S_{i+1}$ . После этого добавляется условие  $EQUAL(S_i, x_{16}, 1)$  для  $S_i$  и  $NOT\_EQUAL(S_i, x_{16}, 1)$  для  $S_{i+1}$ . Выполнение ветвления пропускается, если добавленное условие всегда ложно, которую можно определить, когда условие не содержит никаких символических битов. Для вычисления ЦПИ соответствующие циклы имеют predetermined счетчики итераций, обычно 8, 16 или 32, что обеспечивает ограниченное количество инструкций для символического выполнения. После завершения выполнения всех итераций механизм символического выполнения предоставляет все состояния, соответствующие каждому возможному пути выполнения.

### 3.5. Распознавание ЦПИ

Фаза распознавания ЦПИ состоит из пяти шагов (Рисунок 16): первичная идентификация кандидатов ЦПИ, извлечение параметров, вычисление полинома

ЦПИ, создание структуры регистра сдвига с линейной обратной связью (РСЛОС) и проверка обнаруженного ЦПИ.

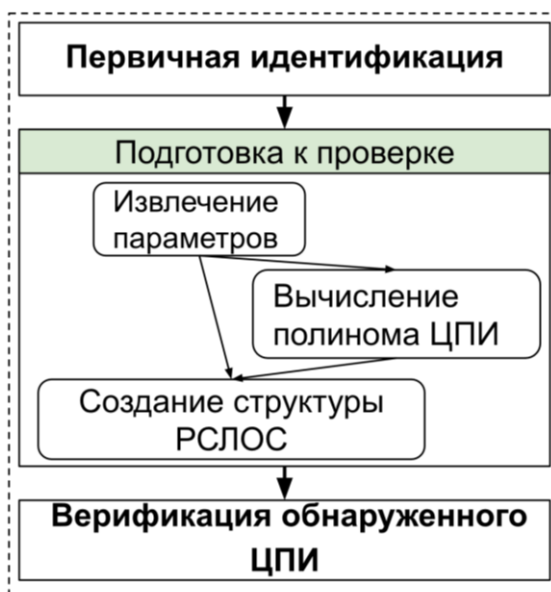


Рисунок 16. Распознавание ЦПИ.

### 3.5.1. Первичная идентификация кандидатов ЦПИ

В начале алгоритм выполняет легкие проверки, чтобы выявить потенциальных кандидатов для вычисления ЦПИ. Эти проверки включают распознавание операций, связанных с ЦПИ, таких как XOR и SHIFT, с некоторым фиксированным числом итераций цикла. Хотя анализ работает на представлении статического одиночного присваивания (SSA) [171], для ясности используется пример исходного кода, показанный на рисунке Рисунок 13. Алгоритм последовательно выполняет несколько проверок, и при успешном прохождении всех из них код идентифицируется как потенциальная реализация ЦПИ. Выполняются следующие проверки:

1. **Обнаружение циклов с фиксированным числом итераций** (8, 16, 32 или 64), которые не содержат вложенных циклов. Например, цикл в строке 5 на рисунке 13 удовлетворяет этому условию, так как он выполняется 8 раз.
2. **Проверка наличия операции XOR** в теле каждого идентифицированного цикла. В рассматриваемом примере инструкция в строке 8 выполняет операцию XOR.
3. **Анализ расположения операции XOR в инструкции ветвления.** Если



XOR выполняется в истинной ветви, условие должно проверять, что старший (для ЦПИ с прямым порядком бит) или младший (для ЦПИ с обратным порядком бит) бит равен 1. Если XOR выполняется в ложной ветви, условие проверяет, что соответствующий бит равен 0.

4. **Определение зависимости операндов XOR и условия ветвления от одной переменной.** В примере такой переменной является `srcValue`.
5. **Идентификация инструкций, связанных с XOR через путь зависимости данных.** Если такие инструкции содержат операцию SHIFT (например, строка 7), второй операнд SHIFT должен быть константой 1. В случае отсутствия SHIFT анализируются инструкции, зависящие от XOR. Если среди них есть SHIFT с константой 1 как вторым операндом, условие считается выполненным.
6. **Проверка симметрии инструкций SHIFT в ветвлениях.** Если идентифицированная инструкция SHIFT расположена в одной ветви (например, строка 10), в другой ветви должна быть аналогичная инструкция SHIFT.

Эти базовые проверки позволяют выявить потенциальные фрагменты кода, выполняющие расчеты ЦПИ, что позволяет нам отфильтровать большую часть кода, не связанного с реализациями ЦПИ.

### **3.5.2. Извлечение параметров**

На данном этапе извлекаются параметры ЦПИ, которые будут использованы на следующих этапах анализа. Список параметров включает:

- **Переменная `Data`.** Переменная, используемая в цикле, соответствующая целевому блоку данных, для которого вычисляется ЦПИ. В примере на рисунке 15 это данные в строке 4. В случае из рисунка 13 переменная данных отсутствует, поскольку, хотя данные передаются в функцию как параметр, они не участвуют в цикле.
- **Переменная `InputCRC`.** Это переменная, содержащая значение ЦПИ перед началом цикла. На рисунке 15 это переменная `src` из строки 4. В некоторых

реализациях до начала цикла выполняется операция XOR между переменной Data и InputCRC. В таком случае InputCRC определяется как результат этой операции. Например, на рисунке 13 это результат инструкции XOR в строке 4.

- **Переменная OutputCRC.** Это переменная, содержащая значение ЦПИ сразу после завершения цикла. На рисунке 13 это переменная crcValue в строке 13, а на рисунке 15 — переменная crc в строке 17. Поскольку анализ проводится на уровне SSA, InputCRC и OutputCRC представлены как разные переменные.
- **Направление вычисления ЦПИ.** Указывает, выполняется ли побитное вычисление ЦПИ в прямом и обратном порядке. Направление определяется на основе операций сдвига. Например, на рисунке 13 операции SHIFT в строках 7 и 10 выполняют сдвиг влево, что соответствует побитному вычислению ЦПИ в прямом порядке бит. На рисунке 15 операции SHIFT выполняют сдвиг вправо, указывая на побитное вычисление ЦПИ в обратном порядке бит.

### **3.5.3. Вычисление полинома ЦПИ**

Полином ЦПИ — это математическое выражение, используемое для расчета и проверки контрольной суммы блока данных ЦПИ. Для различных ЦПИ применяются полиномы определенных степеней. Например, полином 8-й степени используется для ЦПИ-8, а полином 32-й степени — для ЦПИ-32. В двоичной форме полином представлен в виде последовательности битов, где каждый бит соответствует коэффициенту полинома. Например, полином 0x107 (0001.0000.0111 в двоичной форме) математически выражается как  $x^8 + x^2 + x^1 + 1$ .

В примере, показанном на рисунке 13, выполняется вычисление ЦПИ-8, для которого требуется полином степени 8. Полином равен 0x107. В этом примере он совпадает со вторым операндом инструкции XOR, но с опущенным ведущим битом (ведущий бит всегда равен 1). Это позволяет ему вписаться в 8-битную память.

В некоторых реализациях базовый полином может быть не сразу очевиден. Например, на рисунке 15 второй операнд инструкции XOR перевернут и сдвинут. Исходный полином для этой ЦПИ, включая лидирующую 1, равен  $0x18005$ . Далее мы обсудим процесс вычисления полинома ЦПИ.

Чтобы извлечь полином из исходного кода, мы разработали алгоритм на основе пользовательского символического выполнения, обсуждаемого в разделе IV. Алгоритм работает, присваивая значение 1 либо младшему (МЗБ), либо старшему (СЗБ) биту переменной `InputCRC`, в зависимости от того, выполняется ли ЦПИ бит-вперед или бит-назад. Переменной `data` присваивается значение 0, после чего цикл продолжается в символическом режиме. Операция XOR с полиномом происходит, когда MSB или МЗБ переменной `InputCRC` равен 1. Устанавливая этот бит в 1 во время выполнения, можно извлечь полином. После завершения одной итерации вычисленный полином сохраняется в переменной `OutputCRC`. Если в реализации используется полином без ведущей 1, то в `OutputCRC` он также будет представлен без ведущей 1.

В случае ЦПИ, выполняемого в обратном порядке бит, извлеченный полином будет представлен в инверсном виде, а в случае ЦПИ в прямом порядке бит — в исходном виде. Степень полинома определяет размер ЦПИ. Например, на рисунке 13 размер равен 8, а на рисунке 15–16.

#### ***3.5.4. Создание структуры РСЛОС***

РСЛОС [172] — это сдвиговый регистр, входной бит которого вычисляется как линейная функция его предыдущего состояния. Обычно эта линейная функция представляет собой операцию XOR, выполняемую над выбранными битами самого регистра, имитируя полиномиальное деление. РСЛОС используется в различных областях, таких как криптография, обнаружение и исправление ошибок, генерация псевдослучайных чисел и т. д. Он также применяется для вычисления ЦПИ [173].

При вычислении ЦПИ РСЛОС инициализируется predetermined значением, часто состоящим из всех нулей. Полином определяет, какие биты в РСЛОС участвуют в операциях XOR. Пример РСЛОС на рисунке 17 сдвигает свое содержимое влево, а сдвинутый бит подвергается операции XOR с битом данных. Затем результирующее значение подвергается операции XOR со сдвинутым содержимым. Если значение равно 0, РСЛОС просто сдвигает свое содержимое,

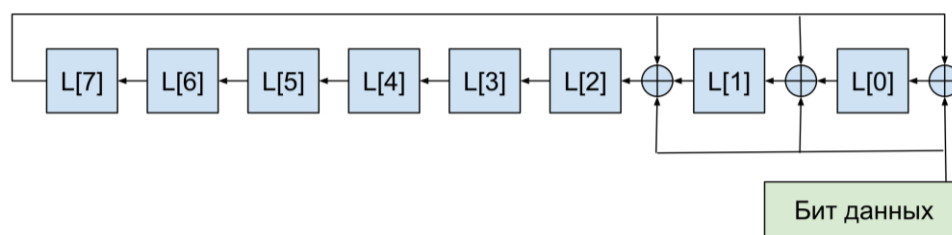


Рисунок 17. РСЛОС с полиномом  $x^8+x^2+x+1$ .

поскольку XOR с 0 не изменяет значения. Этот процесс продолжается до тех пор, пока не будут обработаны все биты данных. Оставшееся значение в РСЛОС представляет собой контрольную сумму ЦПИ.

В некоторых случаях бит данных не подвергается операции XOR со сдвинутым битом, вместо этого он может подвергаться операции XOR с ЦПИ перед работой РСЛОС. В таких случаях структура РСЛОС представлена в виде, показанном на рисунке 18.

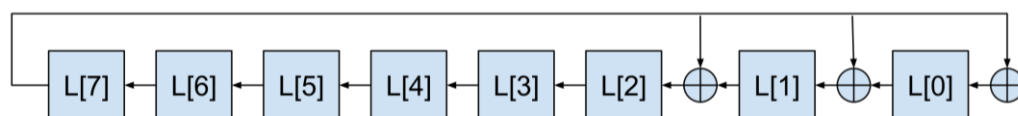


Рисунок 18. РСЛОС с полиномом  $x^8+x^2+x+1$  без операции XOR с битами данных.

Таким образом, каждый цикл РСЛОС эквивалентен одной итерации цикла побитовой реализации ЦПИ. Если проверить, что каждая итерация цикла соответствует циклу РСЛОС, это означает, что цикл вычисляет ЦПИ.

Для проверки идентифицированных кандидатов ЦПИ используется структура РСЛОС, использующая символические значения. Сначала создается массив размером ЦПИ с символическими элементами, затем вычисляем новый массив с

элементами на основе следующих уравнений.  $LFSR[i]$  представляет  $i$ -й бит структуры РСЛОС,  $polynomial[i]$  обозначает  $i$ -й бит полинома ЦПИ.

Для случая ЦПИ в прямом порядке бит, формула создания структуры РСЛОС следующая:

$$LFSR[i] = LFSR[i-1] \oplus (LFSR[size-1] \& polynomial[i]), \overline{i = 1, size - 1}$$

и (1)

$$LFSR[0] = LFSR[size-1] \& polynomial[0].$$

Из формулы следует, что сдвинутые биты РСЛОС подвергаются операции XOR с битами, соответствующими позициям, где коэффициенты полинома равны 1, как это указано в определении РСЛОС.

Если данные используются в операции XOR во время каждой итерации, в формуле (1) выражение  $LFSR[size-1]$ , заменяется на  $LFSR[size-1] \oplus data[size-1]$ .

$$LFSR[i] = LFSR[i-1] \oplus ((LFSR[size-1] \oplus data[size - 1]) \& polynomial[i]),$$

$$\overline{i = 1, size - 1}$$

и (2)

$$LFSR[0] = (LFSR[size-1] \oplus data[size - 1]) \& polynomial[0].$$

Для случая ЦПИ в обратном порядке бит, формула создания структуры РСЛОС следующая:

$$LFSR[i] = LFSR[i+1] \oplus (LFSR[0] \& polynomial[i]), \overline{i = 0, size - 2}$$

и (3)

$$LFSR[size-1] = LFSR[0] \& polynomial[size-1].$$

Если данные подвергаются операции XOR во время каждой итерации, в формуле выражение  $LFSR[0]$ , заменяется на  $LFSR[0] \oplus data[0]$ .

$$LFSR[i] = LFSR[i+1] \oplus ((LFSR[0] \oplus data[0]) \& polynomial[i]), \overline{i = 0, size - 2}$$

и (4)

$$LFSR[size-1] = (LFSR[0] \oplus data[0]) \& polynomial[size-1].$$

На рисунке 19 показана структура РСЛОС, соответствующая реализации ЦПИ из рисунка 13, где ЦПИ-8 вычисляется с использованием полинома  $0x107$  ( $0x07$  без ведущей 1,  $0000.0111$  в двоичной форме).

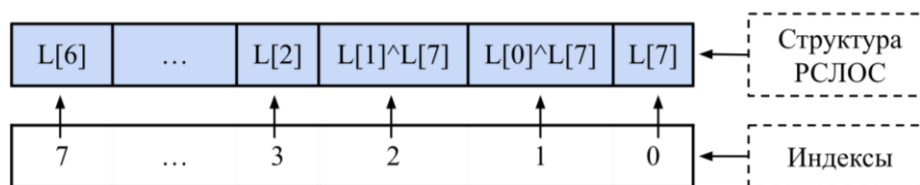


Рисунок 19. Пример структуры РСЛОС.

Сначала создается структура РСЛОС длиной 8 с символьными значениями. Затем первые три бита подвергаются операции XOR с ведущим битом РСЛОС ( $L[7]$ ), поскольку первые три бита полинома равны 1.

Созданная структура РСЛОС будет использоваться на следующем этапе для проверки идентифицированных реализаций ЦПИ.

### 3.5.5. Верификация обнаруженного ЦПИ

На этом этапе выполняется верификация обнаруженного фрагмента кода ЦПИ с использованием символического выполнения и ранее собранной информации. Основные шаги алгоритма проверки следующие:

1. Запустить символическое выполнение с первой инструкции цикла. Построить соответствующий объект состояния, предназначенный для отслеживания всех значений переменных и условий на каждом пути и для каждой точки в программе. Инициализировать все используемые переменные и сохранить их в состоянии. Для переменных, имеющих инициализацию в коде, использовать те же значения, в противном случае присвоить им символические значения.

2. Пройти цикл и символически выполнить каждую инструкцию. Одна и та же инструкция может быть символически выполнена более одного раза, каждый раз для разных путей:

- 2.1. Если встречается инструкция, результат которой используется вне цикла и это не переменная ЦПИ, символическое выполнение завершается с отрицательным результатом проверки. Это означает, что

цикл не вычисляет ЦПИ или помимо ЦПИ вычисляет что-то иное и не может заменяться оптимальной реализацией ЦПИ.

3. После каждой итерации цикла проверяем, что вычисленные значения ЦПИ по всем объектам состояния имеют ту же структуру, что и созданная структура РСЛОС, т. е. удовлетворяет одному из уравнений (1), (2), (3), (4).

Проверка на каждой итерации необходима, поскольку код может содержать вычисления, зависящие от номера итерации. Если проверку выполнять только после первой итерации, это может привести к ложноположительным результатам.

Рассмотрим пример на рисунке 13 с условием  $if(crcValue \& 0x80)$ . Если заменить это условие на  $if((crcValue \& 0x80) \wedge (i \& 1))$ , цикл перестанет вычислять ЦПИ. Однако символический исполнитель, анализирующий код после первой итерации, может ошибочно идентифицировать его как ЦПИ. Это происходит потому, что на первой итерации значение переменной  $i$  равно 0, а выражение  $i \& 1$  также равно 0. В результате условие  $(crcValue \& 0x80) \wedge (i \& 1)$  упрощается до  $crcValue \& 0x80$ , что соответствует структуре LFSR. В то же время на второй итерации значение  $i \& 1$  изменится, и условие перестанет соответствовать структуре LFSR. Поэтому выполнение проверки на каждой итерации позволяет эффективно выявлять такие случаи и отсеивать ложноположительные результаты.

### 3.6. Замена ЦПИ

Для замены побитового ЦПИ на более оптимальную версию реализовано три метода ЦПИ: метод на основе таблиц, метод на основе умножения без переноса и прямое использование инструкции ЦПИ. Последние два метода применимы, если целевая архитектура поддерживает соответствующие инструкции.

Метод на основе таблиц включает в себя создание таблицы поиска и замену цикла вычисления ЦПИ кодом, представленным на рисунке 20. Где  $CRC\_size$  и  $data\_bit\_size$  могут быть 8, 16, 32, 64, а  $CRC\_table$  — это предварительно вычисленный список из 256 значений ЦПИ.

```

for (int i = 0; i < data_bit_size / 8; i++)
    crc = (crc << 8) ^
        crc_table[(crc >> (crc_size - 8)) ^
            (data >> (data_bit_size - (i + 1) * 8) & 0xFF)];

```

Рисунок 20. Реализация ЦПИ на основе таблиц на языке C.

Для метода, основанного на умножении без переноса, генерируется ассемблерный код, соответствующий коду на рисунке 21. Где  $q$  = частное от деления  $x^{2n}$  на  $p$ ,  $p$  — многочлен,  $ps$  — многочлен без ведущей единицы, *clmul* — аппаратная инструкция, выполняющая умножение без переноса, *mask(n)* — это маска, которая обнуляет все биты, кроме младших  $n$  бит.

Когда целевая архитектура предоставляет инструкцию ЦПИ, весь цикл можно заменить этой единственной инструкцией.

```

crc = clmul(clmul(data^crc, q) >> n, ps) & mask(n);

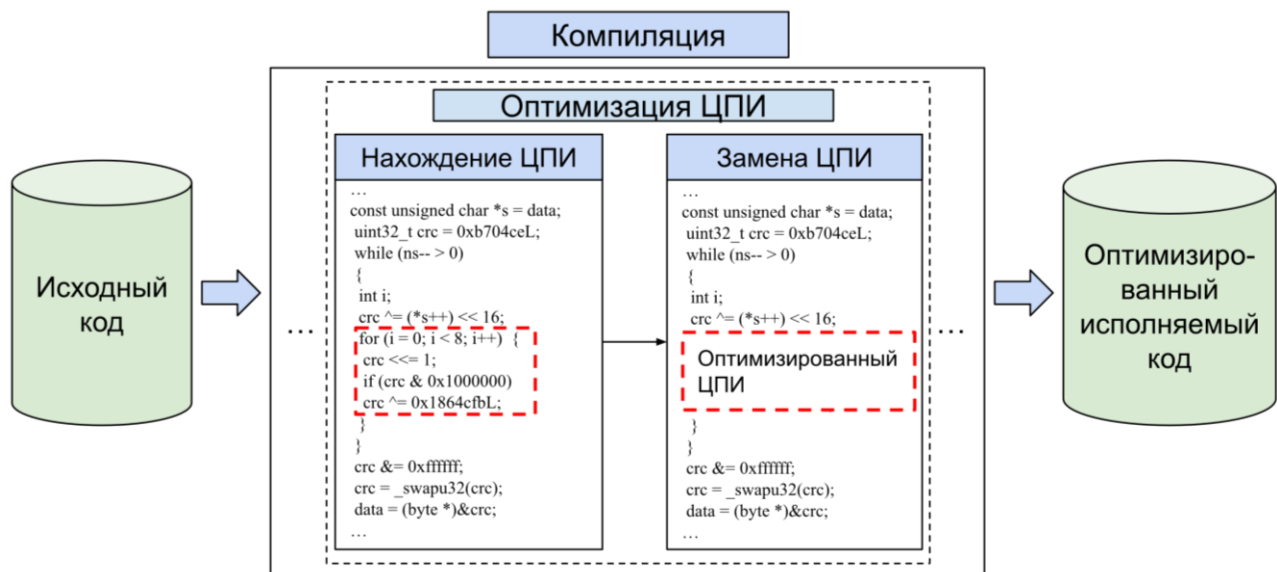
```

Рисунок 21. Реализация ЦПИ на основе умножения без переноса на языке C.

### 3.7. Реализация метода нахождения ЦПИ и его замены

Рисунок 22 иллюстрирует предлагаемый процесс оптимизации ЦПИ. Он автоматически проверяет код всех функций и заменяет побитовые ЦПИ оптимизированными версиями, устраняя необходимость ручного взаимодействия. Предложенный метод нахождения ЦПИ реализован в компиляторе GCC [121] в виде отдельного прохода (pass), основанного на представлении GIMPLE [174]. Проход выполняется после прохода распределения цикла, что позволяет использовать дополнительную информацию о цикле, включая количество его итераций. Код в рамках данного прохода представлен в форме SSA, благодаря чему переменные InputCRC и OutputCRC рассматриваются как разные, хотя в исходном коде они соответствуют одной и той же переменной.





*Рисунок 22. Процесс оптимизации ЦПИ.*

Для включения разработанного прохода добавлена опция компилятора -fortimize-crc. Оптимизация ЦПИ активируется по умолчанию на уровнях оптимизации -O2 и выше. Однако при использовании опции -Os (оптимизация по размеру) метод замены ЦПИ на основе таблицы не применяется, так как он увеличивает размер исполняемого файла из-за необходимости хранения таблицы поиска ЦПИ.

Патчи, реализующие данный проход, интегрированы в компилятор GCC. Подробная информация о реализации доступна по следующей ссылке [175].

### **3.7.1. Внедрение замены ЦПИ в GCC**

После идентификации и верификации побитовой ЦПИ выполняется ее замена на более оптимальную реализацию. Для замены обнаруженных циклов ЦПИ выполняются следующие шаги:

1. Добавляется вызов функции, реализующей оптимальную ЦПИ. Возвращаемое значение этой функции записывается в переменную OutputCRC цикла.
2. Удаляется старая инструкция присваивания вычисленного значения OutputCRC.
3. Условие выхода из цикла модифицируется таким образом, чтобы

выполнение цикла не происходило вовсе.

После выполнения третьего шага запускается проход очистки графа потока управления, который удаляет структуру цикла и заднее ребро. Затем проход устранения мертвого кода удаляет оставшиеся мертвые инструкции. Таким образом, удаление цикла не выполняется в рамках самого прохода оптимизации ЦПИ, а выполняется на более поздних этапах компиляции.

Для реализации более быстрых вычислений ЦПИ были разработаны две внутренние функции GCC: **IFN\_CRC** и **IFN\_CRC\_REV**. Первая из них предназначена для вычислений в прямом порядке бит, а вторая — для вычислений в обратном порядке. Обе функции принимают следующие параметры: начальное значение ЦПИ, данные и полином. На выходе они возвращают вычисленное значение ЦПИ.

Эти функции проверяют, поддерживает ли целевая архитектура специализированный расширитель. Если да, то вызывается соответствующий расширитель; в противном случае используется алгоритм вычисления ЦПИ на основе таблицы. Расширители ЦПИ были реализованы для архитектур **RISC-V**, **i386** и **AArch64**.

#### **Расширитель ЦПИ для RISC-V [176]:**

- Если целевая система поддерживает инструкции `clmul` и `clmulh`, используется вычисление ЦПИ на основе УБП.

#### **Расширитель ЦПИ для AArch64 [177]:**

- Если вычисляется ЦПИ с обратным порядком бит и полиномом `iSCSI` (`0x1EDC6F41`), то при наличии поддержки инструкции `crc32c` она будет использоваться. Конкретный вариант инструкции (`crc32cb`, `crc32ch` или `crc32cw`) выбирается в зависимости от размера обрабатываемых данных.
- Если полином соответствует `HDLC` (`0x04C11DB7`), применяется инструкция `crc32`. Выбор конкретного варианта инструкции (`crc32b`, `crc32h` или `crc32w`) также зависит от размера обрабатываемых данных.
- Если доступна инструкция `rmull`, используется вычисление ЦПИ на основе УБП, как для прямого, так и для обратного порядка бит.

### Расширитель ЦПИ для i386 [178]:

- Если выполняется ЦПИ с обратным порядком бит с полиномом iSCSI (0x1EDC6F41) и поддерживается инструкция crc32, она используется с соответствующими вариантами в зависимости от размера данных.
- Если поддерживается инструкция pclmulqdq, используется вычисление ЦПИ на основе УБП, как для прямого, так и для обратного порядка бит.

Хотя основные принципы вычисления ЦПИ с использованием прямых инструкций CRC32 или УБП остаются неизменными для всех целевых архитектур, конкретные детали реализации различаются из-за различий в наборах инструкций, режимах и других факторах. Следовательно, для каждой целевой платформы требуется отдельный код. Кроме того, 64-битный ЦПИ, основанный на УБП, не поддерживается на 64-битных архитектурах, поскольку  $q(x)$  требует 65 бит памяти. Аналогично, 32-битный ЦПИ, основанный на УБП, не поддерживается на 32-битных архитектурах, поскольку  $q(x)$  требует 33 бит.

### 3.8. Неподдерживаемые случаи

В этом разделе подробно описаны конкретные случаи, когда ЦПИ не может быть обнаружено или заменено из-за ограничений в текущей реализации.

- Переменный полином: значение полинома необходимо при генерации кода для вычисления ЦПИ. В частности, оно используется в методе на основе УБП для вычисления частного и в табличном методе для предварительного вычисления ЦПИ. Поэтому, если полином в коде является переменным, то невозможно генерировать оптимизированный ЦПИ.
- Более крупные типы, используемые для хранения ЦПИ и данных: например, 32-битная переменная ЦПИ может использоваться для вычисления ЦПИ-24. В этом случае точное предсказание фактического размера ЦПИ становится сложным.
- Вычисления, не связанные с ЦПИ в цикле: если цикл вычисления ЦПИ содержит другие вычисления, не связанные с ЦПИ, замена всего цикла

непреднамеренно удалит эти вычисления и изменит семантику.

- Вычисление ЦПИ без ветвей: на этапе первичной идентификации кандидатов ЦПИ 3-й пункт (см. раздел 3.5.1.) проверяет, содержит ли код условный переход.
- Ведущая единица полинома сохраняется: например, если в вычислении ЦПИ-8 полином представлен с использованием 9 бит.
- Большой размер данных, чем ЦПИ: например, для хранения данных используется 64-битная переменная, а для ЦПИ — 32-битная.

### **3.9. Экспериментальная оценка**

В первом эксперименте выполнен поиск вычислений ЦПИ в существующем программном обеспечении, чтобы понять потенциальное влияние реализованной оптимизации. Во втором эксперименте проведен сравнительный анализ оптимизации для оценки улучшения производительности.

#### ***3.9.1. Вычисления ЦПИ в программном обеспечении с открытым исходным кодом***

В рамках подготовки экспериментов были проанализированы различные программные обеспечения, в которых удалось обнаружить вычисления ЦПИ. Все обнаруженные реализации были корректными, так как метод разработан так, чтобы не дать ложноположительных результатов.

В пакетах, распространяемых Fedora, первичная проверка идентифицировала 26 случаев вычислений ЦПИ, и 18 из них были верифицированы. Остальные случаи не были верифицированы, в первую очередь потому, что использовались полиномы с ведущими битами, или ЦПИ с меньшими размерами хранились в больших пространствах памяти.

В целом, в ядре Linux [158] были обнаружены три побитовые реализации ЦПИ [179] [180] [181], из которых не был верифицирован только один случай, поскольку использовался полином с ведущим битом.

Также был протестирован бенчмарк Coremark [168], где была найдена и верифицирована одна реализация ЦПИ. Код представлен на рисунке 15.

### **3.9.2. Влияние оптимизации**

Для оценки повышения производительности реализованных методов автоматической замены ЦПИ было проведено несколько экспериментов на трех архитектурах: 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). Были оценены следующие методы реализации ЦПИ: побитовая реализация, табличная реализация, реализация на основе УБП и прямое использование выделенных инструкций ЦПИ.

Тестирование включало две различные реализации ЦПИ: реализации в прямом (bf) и обратном (bb) для разных размеров ЦПИ и данных. Каждая функция ЦПИ вызывалась 4294967295 (шестнадцатеричное 0xFFFFFFFF) раз, и в конце был выведен рассчитанный ЦПИ. Тесты компилировались отдельно для каждого метода: один раз для табличного ЦПИ, один раз для ЦПИ на основе УБП и один раз для исходного побитового без оптимизации ЦПИ (с оптимизацией -O2).

Затем время работы сгенерированного ЦПИ сравнивалось с временем работы побитовой реализации ЦПИ. Для архитектуры x86-64 измерялась скорость всех методов генерации ЦПИ, включая метод на основе УБП, поскольку используемый процессор поддерживает его. Однако метод на основе УБП не тестировался на RISC-V, и для оценки их производительности количество выполненных инструкций определялось с помощью эмулятора QEMU [182]. В методе на основе УБП не измеряются случаи 64-битного ЦПИ, поскольку ЦПИ-64 на основе УБП не поддерживается.

Таблица 9 содержит экспериментальную оценку на процессоре Intel Core i7-3770. Таблица сравнивает производительность различных реализаций ЦПИ: побитовой, табличной и основанной на УБП. В среднем табличный метод обеспечивает 54%-ное сокращение времени выполнения, а метод на основе УБП обеспечивает 43%-ное сокращение. В некоторых случаях табличный метод не всегда демонстрирует

преимущества: при реализации ЦПИ с обратным порядком бит, где входные и выходные данные каждый раз переворачиваются, наблюдается значительное увеличение времени работы, что в худшем случае приводит к замедлению на 17%.

Тест	Полином	Побитовой	Табличный		Основанный на УБП	
		Время	Время	Улучшение	Время	Улучшение
bf-crc8-data8	0x121	44с	11с	75%	38с	14%
bf-crc16-data8	0x11021	54с	10с	81%	37с	31%
bf-crc16-data16		1м 18с	20с	74%	34с	56%
bf-crc32-data8	0x182F63B78	48с	12с	75%	37с	23%
bf-crc32-data16		1м 51с	22с	80%	37с	67%
bf-crc32-data32		2м 39с	40с	75%	33с	79%
bf-crc64-data32	0x142F0E1EBA9EA36	7м 10с	37с	91%	-	-
bf-crc64-data64	93	5м 19с	1м 13с	77%	-	-
bb-crc8-data8	0x105	46с	50с	-9%	39с	15%
bb-crc16-data8	0x18005	48с	51с	-6%	40с	17%
bb-crc16-data16		1м 32с	1м 2с	33%	39с	58%
bb-crc32-data8	0x189080041	47с	55с	-17%	40с	15%
bb-crc32-data16		1м 35с	1м 7с	31%	40с	58%
bb-crc32-data32		3м 5с	1м 24с	55%	37с	80%
bb-crc64-data32	0x1f890800	4м 34с	1м 2с	77%	-	-
bb-crc64-data64	5f8908005	6м 11с	1м 57с	68%	-	-
<b>Среднее улучшение</b>			<b>54%</b>		<b>43%</b>	

Таблица 9. Производительность ЦПИ на архитектуре x86-64.

А метод на основе УБП всегда дает улучшение, с минимальным сокращением времени выполнения на 15%. Максимальное ускорение наблюдалось для табличного метода — до 91%, тогда как метод на основе УБП достиг максимального улучшения в 80%. Некоторые ячейки содержат «-», что означает, что УБП не поддерживается для данных размеров.

Таблица 10 содержит результаты тестов производительности побитовой и табличной реализации ЦПИ на платформе StarFive VisionFive 2 v1.38. В среднем табличный метод обеспечил сокращение времени выполнения на 66% по сравнению с побитовым. В худшем случае табличный метод дал 1% замедление, а в лучшем случае — 93% улучшение.

Тест	Полином	Побитовой	Табличный	
		Время	Время	Улучшение
bf-crc8-data8	0x121	4м 31с	20с	93%
bf-crc16-data8	0x11021	2м 37с	26с	83%
bf-crc16-data16		8м 52с	51с	90%
bf-crc32-data8	0x182F63 B78	2м 46с	26с	84%
bf-crc32-data16		5м 31с	51с	85%
bf-crc32-data32		8м 5с	1м 40с	79%
bf-crc64-data32	0x142F0E 1EBA9EA 3693	11м 5с	1м 31с	86%
bf-crc64-data64		26м 9с	3м 3с	88%
bb-crc8-data8	0x105	3м 31с	2м 17с	25%
bb-crc16-data8	0x18005	6м 40с	2м 43с	59%
bb-crc16-data16		6м 18с	3м 3с	52%
bb-crc32-data8	0x1890800 41	2м 44с	2м 46с	-1%
bb-crc32-data16		5м 31с	3м 6с	44%
bb-crc32-data32		9м 31с	3м 46с	60%
bb-crc64-data32	0x1f89080 05f890800 5	10м 20с	3м 51с	63%
bb-crc64-data64		18м 39с	5м 11с	72%
<b>Среднее улучшение</b>				<b>66%</b>

Таблица 10. Производительность ЦПИ на архитектуре RISC-V.

Таблица 11 представляет экспериментальную оценку на Raspberry Pi 4 модель B Rev 1.5. Таблица сравнивает производительность побитовой и табличной реализации и показывает, что в среднем табличный метод достигает 50% улучшения производительности по сравнению с побитовым методом. Однако в

худшем случае табличный метод демонстрирует замедление на 31%, в то время как в лучшем случае достигает улучшения на 77%.

Тест	Полином	Побитовой	Табличный	
		Время	Время	Улучшение
bf-crc8-data8	0x121	1м 21с	19с	77%
bf-crc16-data8	0x11021	1м 16с	21с	72%
bf-crc16-data16		2м 29с	41с	72%
bf-crc32-data8	0x182F63B7 8	1м 16с	24с	68%
bf-crc32-data16		2м 32с	43с	72%
bf-crc32-data32		4м 32с	1м 24с	69%
bf-crc64-data32	0x142F0E1E	5м 5с	1м 25с	72%
bf-crc64-data64	BA9EA3693	9м 41с	2м 54с	70%
bb-crc8-data8	0x105	1м 54с	1м 37с	15%
bb-crc16-data8	0x18005	1м 54с	1м 54с	0%
bb-crc16-data16		3м 49с	2м 14с	41%
bb-crc32-data8	0x18908004 1	1м 35с	2м 4с	-31%
bb-crc32-data16		3м 10с	2м 19с	27%
bb-crc32-data32		6м 21с	2м 58с	53%
bb-crc64-data32	0x1f8908005	6м 21с	2м 51с	55%
bb-crc64-data64	f8908005	12м 46с	4м 16с	67%
<b>Среднее улучшение</b>				<b>50%</b>

Таблица 11. Производительность ЦПИ на архитектуре AArch64.

В таблицах 12 и 13 представлено сравнение времени работы реализаций ЦПИ с полиномом iSCSI (0x1EDC6F41) на архитектурах x86-64 и AArch64 соответственно. В обеих таблицах последний столбец показывает производительность инструкции crc32. Эти тесты состоят из ЦПИ с обратным порядком бит. В таких случаях побитовая реализация ЦИК зачастую превосходит табличный метод, поскольку последний требует каждый раз инвертировать порядок бит как для входных, так и для выходных данных, что приводит к дополнительным временным затратам.



Тест	Побитовой	Табличный	Основанный на УБП	crc32 инструкция
crc32_data32*	2м 35с	1м 18с	38с	4.4с
crc32-data16*	39.6с	54с	40с	4.5с
crc32-data8*	39.5с	53с	40с	4.4с
<b>Среднее улучшение</b>		<b>-7%</b>	<b>24%</b>	<b>92%</b>

*Таблица 12. Сравнение эффективности инструкции crc32 с альтернативными подходами на архитектуре x86-64.*

Тест	Побитовой	Основанный на УБП	crc32 инструкция
crc32_data32*	5м 7с	1м 40с	7.2с
crc32-data16*	1м 19с	1м 57с	7.1с
crc32-data8*	1м 18с	1м 57с	7.2с
<b>Среднее улучшение</b>		<b>-10%</b>	<b>93%</b>

*Таблица 13. Сравнение эффективности инструкции crc32 с альтернативными подходами на архитектуре AArch64.*

Для оценки эффективности оптимизации с использованием инструкции УБП на архитектурах RISC-V и AArch64 было проведено сравнение общего количества инструкций до и после применения оптимизации к файлам, содержащим функции ЦПИ. Количество инструкций было получено с помощью плагина QEMU [183].

Таблицы 14 и 15 представляют результаты тестирования количества выполненных инструкций для побитовой, табличной и основанной на УБП реализаций функций ЦПИ на целевых архитектурах RISC-V и AArch64. Таблица 14 демонстрирует снижение количества инструкций на 58% для табличной реализации и на 89% для реализации на основе УБП на архитектуре RISC-V. Аналогично, таблица 15 показывает сокращение количества инструкций на 62% для табличной реализации и на 86% для реализации на основе УБП на архитектуре AArch64.

Тест	Побитовой	Табличный		Основанный на УБП	
	Количество	Количество	Улучшение	Количество	Улучшение
bf-crc8-data8	64	6	91%	8	88%
bf-crc16-data8	89	14	84%	14	84%
bf-crc16-data16	175	21	88%	8	95%
bf-crc32-data8	51	13	75%	13	75%
bf-crc32-data16	92	19	79%	11	88%
bf-crc32-data32	180	32	82%	7	96%
bb-crc8-data8	68	52	24%	6	91%
bb-crc16-data8	84	61	27%	8	90%
bb-crc16-data16	166	71	57%	8	95%
bb-crc32-data8	152	162	-7%	28	82%
bb-crc32-data16	297	186	37%	31	90%
bb-crc32-data32	226	86	62%	8	96%
<b>Среднее улучшение</b>		<b>58%</b>		<b>89%</b>	

*Таблица 14. Динамическое количество инструкций для функций ЦПИ на архитектуре RISC-V.*

Тест	Побитовой	Табличный		Основанный на УБП	
	Количество	Количество	Улучшение	Количество	Улучшение
bf-crc8-data8	55	7	87%	10	82%
bf-crc16-data8	52	8	85%	11	79%
bf-crc16-data16	100	11	89%	7	93%
bf-crc32-data8	54	10	81%	14	76%
bf-crc32-data16	99	11	89%	11	90%
bf-crc32-data32	195	17	91%	5	98%
bb-crc8-data8	60	37	38%	9	85%
bb-crc16-data8	60	44	27%	11	82%
bb-crc16-data16	116	48	59%	10	91%
bb-crc32-data8	111	121	-9%	25	78%
bb-crc32-data16	213	132	38%	25	88%
bb-crc32-data32	162	58	64%	8	95%
<b>Average improvement</b>		<b>62%</b>		<b>86%</b>	

*Таблица 15. Динамическое количество инструкций для функций ЦПИ на архитектуре AArch64.*

На рисунке 23 подведены итоги общего среднего улучшения, достигнутого за счет применения оптимизации ЦПИ во всех целевых архитектурах. Значения на диаграмме взяты из средних улучшений, указанных в соответствующих таблицах.

Таким образом, табличные реализации, реализации на основе УБП и прямое использование инструкции `src32` показывают значительные улучшения по сравнению с побитовой реализацией ЦПИ, демонстрируя значительные улучшения производительности, которые достигаются посредством представленной оптимизации.

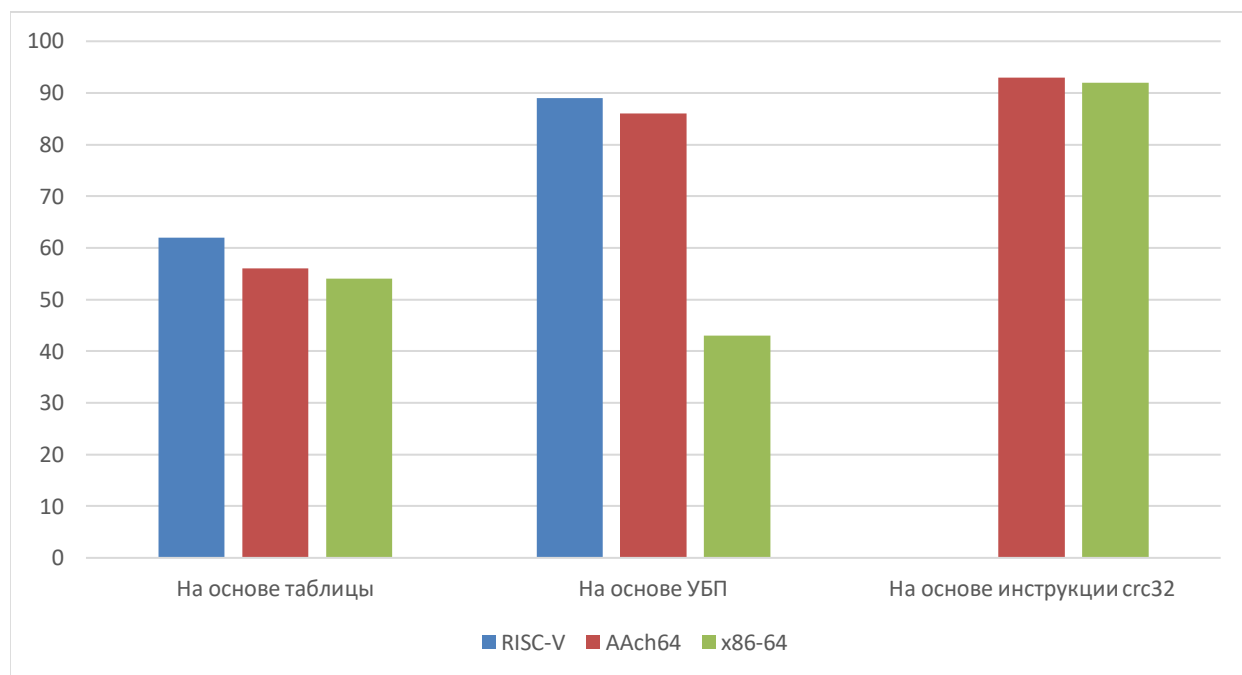


Рисунок 23. Среднее улучшение по сравнению побитовой реализации ЦПИ.

### 3.10. Выводы

В этой главе рассмотрен метод оптимизации программ на основе поиска клонов, ориентированный на автоматическое обнаружение фрагментов кода, вычисляющих ЦПИ, и их замену на более эффективные реализации. Предложенный метод основан на технике символического выполнения, что позволяет выявить ЦПИ без ложноположительных результатов.

Экспериментальная оценка реализованного инструмента показала его эффективность: в тестируемых случаях удалось уменьшить накладные вычислительные расходы за счет использования оптимизированных версий ЦПИ-алгоритмов. Полученное ускорение работы программ подтверждает, что автоматическая оптимизация на основе поиска клонов может быть полезной для повышения эффективности вычислительных задач.

## Глава 4. Двухэтапный метод выявления изменений между версиями программ

Четвертая глава посвящена двухэтапному методу выявления изменений между версиями программ, который сочетает метрический подход с разработанным методом поиска клонов кода на основе графов, содержащих зависимости управления и данных, для сопоставления функций в формате «многие ко многим» и отображения измененных инструкций.

Предлагаемый метод сравнения программ в качестве входных данных получает две программы, и находит все схожие функции первой программы во второй программе, и наоборот.

### 4.1. Схема метода сравнения двух программ

Метод состоит из двух основных этапов (Рисунок 24): генерация ГЗП и графов вызовов функции, и нахождение схожих функций с использованием полученных графов.

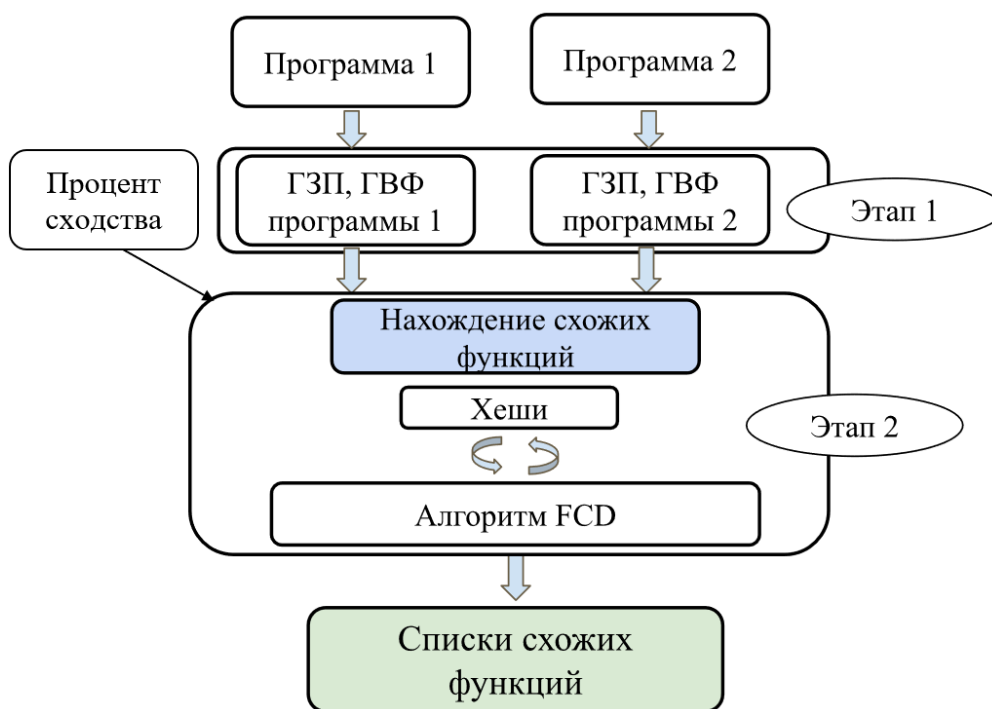


Рисунок 24. Схема сравнения двух программ.

#### **4.1.1. Первый этап – генерация графа вызовов функций и графов зависимостей программы**

На первом этапе генерируется ГВФ и ГЗП для каждой функции целевой программы. Для генерации ГЗП используется тот же метод, что описан в разделе 2, генерация ГВФ описывается в подразделе реализации.

#### **4.1.2. Второй этап - нахождение схожих графов**

Сравнение и сопоставление графов состоит из двух этапов - вычисление хешей и использование алгоритма нахождения клонов фрагмента кода, который рассматривается в главе 2. Хеши используются для предварительного сопоставления функций, после чего используется алгоритм, основанный на поиске клонов фрагмента кода для окончательного сопоставления.

1. На основе ранее сгенерированных ГЗП вычисляются семь хешей. Для вычисления хешей используется модифицированная версия SimHash [142], 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 с использованием MD-index [104], основанный на количестве предшествующих и последующих вершин базовых блоков ГЗП.
5. Хеш с использованием 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 бита. После вычисления всех хешей они попарно сравниваются, используя коэффициент Отиаи (cosine similarity) [184] и сходство по длине.

Коэффициент Отиаи для двух векторов А и В 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 -  $m_1, m_2, \dots, m_{14}$ , на основе которых они могут предварительно сопоставляться. Для каждого значения определяется вес, так как некоторые хеши более приоритетны, чем другие. Парты функций предварительно сопоставляются, если

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

Весы  $w_i$  были установлены с помощью методов машинного обучения. Для обучения создавались пары функций с заданной степенью сходства, после чего вычислялись их хеш-значения.

- Для всех предварительно сопоставленных функций вычисляются общие части, используя алгоритм FCD. Назначим количество общих инструкций промежуточного представления - *commonInstructionsCount*, количество REIL инструкций первой функции - *instructionsCount1*, количество инструкций промежуточного представления второй функции - *instructionsCount2*. Функции сопоставляются, если

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

## 4.2. Реализация

Метод сравнения двух программ реализован в инструменте ВСС. Этот инструмент принимает в качестве входных данных проекты, которые нужно сравнить.

Этап генерации ГЗП И ГВФ для исполняемого кода основан на фреймворке анализа кода BinSide [5]. BinSide использует IdaPro [125] дизассемблер для восстановления структуры программы и представления ее с помощью ГВФ. Каждая функция, в свою очередь, также представлена с помощью ГЗП. Генерация ГЗП для исходного и исполняемого кода описывается в разделе 2.3.

### 4.3. Результаты

Инструмент был протестирован на некоторых широко распространенных проектах, скомпилированных для разных архитектур.

В таблице 16 приведены результаты работы инструмента на некоторых версиях проекта Openssl. В ней указаны сравненные версии, архитектура проектов, размер каждого файла, количество функций в исполняемых файлах, количество сопоставленных функций и средний процент их сходства.

Проект 1	Проект 2	Архитектура	Размер исполняемого файла 1	Размер исполняемого файла 2	Количество функций 1	Количество функций 2	Количество сопоставленных функций	Средний процент схожести
Openssl - 1.0.2s	Openssl - 1.0.2t	Arm	480К	480К	1401	1401	1401	99.98
Openssl - 1.0.2s	Openssl - 1.0.2t	Mips	3.2М	3.2М	5546	5551	5515	97.1
Openssl - 1.0.2s	Openssl - 1.0.2t	X64	3.4М	3.4М	5780	5785	5722	99.5
Openssl - 1.0.2s	Openssl - 1.0.2t	X86	3.2М	3.2М	5751	5757	5700	96.9
Openssl - 1.1.0k	Openssl - 1.1.0l	Arm	500К	500К	1556	1557	1553	99.9
Openssl - 1.1.0k	Openssl - 1.1.0l	Mips	3.2М	3.2М	6179	6180	6125	98.1
Openssl - 1.1.0k	Openssl - 1.1.0l	X64	672К	672К	322	322	322	98.7



Openssl - 1.1.0k	Openssl - 1.1.0l	X86	692K	692K	332	332	332	98.4
Openssl - 1.1.1c	Openssl - 1.1.1d	Arm	560K	560K	1701	1701	1701	99.88
Openssl - 1.1.1c	Openssl - 1.1.1d	Mips	3.8M	3.8M	7231	6938	6872	98.5
Openssl - 1.1.1c	Openssl - 1.1.1d	X64	740K	740K	363	363	361	98.3
Openssl - 1.1.1c	Openssl - 1.1.1d	X86	776K	780K	370	370	359	97.5

*Таблица 16. Результат работы инструмента на разных версиях Openssl.*

А в таблице 17 приведены результаты работы инструмента на проектах libxml, rsync и openssh. Тест проводился на компьютере с четырехъядерным процессором Intel Core i5-3470 3600 МГц и 16 ГБ оперативной памятью, используя одно ядро.

<b>Проект 1</b>	<b>Проект 2</b>	<b>Время работы</b>	<b>Количество функций 1</b>	<b>Количество функций 2</b>	<b>Количество сопоставленных функций</b>	<b>Средний процент схожести</b>
Libxml2 2.9.2	Libxml2 2.9.3	9м 36 с	2591	2610	2567	98.82%
Rsync 3.1.1	Rsync 3.1.2	48 с	663	659	642	98.54%
Openssh 0.7.2	Openssh 0.7.3	18 с	970	970	968	91.90%

*Таблица 17. Результат работы инструмента на разных проектах.*

С помощью тестовой системы инструмент также был протестирован на проекте Coreutils [120]. Исходные файлы были скомпилированы компиляторами g++ (версия 9.2.1) [121] и clang++ (версия 9.0.0) [122] с различными флагами оптимизации для архитектуры x86-64. Тестирование проводилась на компьютере

с четырехъядерным процессором Intel Core i5-3470 3600 МГц и 16 ГБ оперативной памятью.

Результаты получены с использованием тестовой системы, которая компилирует проекты, копирует полученные исполняемые файлы, а затем удаляет из них символы с помощью инструмента strip [120]. Также тестовая система сохраняла оригинальные имена каждой функции в исполняемом файле без отладочной информации, используя соответствующий оригинальный файл (если адреса функций совпадают, они считаются одними и теми же функциями).

В таблице 18 представлены результаты тестирования для различных версий программ, где каждая строка содержит информацию о версии, используемом компиляторе, оптимизации и архитектуре, а также размеры программ, точность, полноту и время анализа. Сопоставление функций считается правильным, если они имеют одинаковые имена (в случае исполняемого файла с удаленными символами, используется оригинальное имя, которое было сохранено в тестовой системе).

В проведенных экспериментах точность оставалась высокой даже при различных настройках оптимизаций и компиляторов. Например, комбинации компиляторов Clang и GCC с разными оптимизациями (O0, O2, O3) показывали точность в пределах 90–99%. Полнота результатов также варьировалась в зависимости от параметров компиляции. Более высокие оптимизации (O2, O3) зачастую обеспечивали несколько более низкую полноту по сравнению с O0, что связано с изменениями в структуре и оптимизациях кода. Время анализа зависело от размера программ и выбранных параметров компиляции. Например, для программ, скомпилированных с использованием компилятора GCC для 64-битных версий, анализ занимал меньше времени по сравнению с 32-битными версиями. Это можно объяснить меньшими размерами обрабатываемого кода.

<b>Версия 1, компилятор, оптимизация, архитектура (бит)</b>	<b>Размер 1 (МБ)</b>	<b>Версия 2, компилятор, оптимизация, архитектура (бит)</b>	<b>Размер 2 (МБ)</b>	<b>Точность (%)</b>	<b>Полнота (%)</b>	<b>Время анализа (с.)</b>
8.30_clang_O0_64	16	8.30_clang_O0_64	7	98.8	98.3	38
8.30_clang_O2_64	17	8.30_clang_O2_64	4.9	97.2	85.6	22
8.30_clang_O0_64	16	8.30_clang_O2_64	4.9	79.5	67.4	49
8.30_clang_O2_64	17	8.30_clang_O3_64	5.9	90.7	81.3	36
8.30_gcc_O0_64	15	8.30_gcc_O0_64	6.3	98.8	98.3	27
8.30_gcc_O2_64	24	8.30_gcc_O2_64	5.2	98.2	82.4	24
8.30_gcc_O0_64	15	8.30_gcc_O2_64	5.2	84.8	69.2	54
8.30_gcc_O0_64	15	8.30_clang_O0_64	7	94.9	89.4	36
8.30_gcc_O2_64	24	8.30_clang_O2_64	4.9	88.1	76	41
8.30_gcc_O0_64	15	8.30_clang_O2_64	4.9	78.9	67.7	47
8.30_gcc_O2_64	24	8.30_gcc_O3_64	6.3	89.5	76.5	42
7.6_gcc_O0_64	19.3	8.30_gcc_O0_64	6.3	72.1	69.5	84
7.6_gcc_O2_64	19.3	8.30_gcc_O2_64	5.2	85.4	77.9	43
8.29_gcc_O0_64	14.8	8.30_gcc_O0_64	6.3	97.8	98.2	28
8.29_gcc_O2_64	24.9	8.30_gcc_O2_64	5.2	97	82.2	28
8.30_clang_O0_32	19	8.30_clang_O0_32	6.3	98	80.2	51
8.30_clang_O2_32	18	8.30_clang_O2_32	4.5	97.5	76.3	21
8.30_clang_O0_32	19	8.30_clang_O2_32	4.5	87.8	67.3	46
8.30_clang_O2_32	18	8.30_clang_O3_32	4.7	90.1	71.7	91
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
8.30_gcc_O0_32	13	8.30_gcc_O2_32	5.3	84.1	72.7	72
8.30_gcc_O0_32	13	8.30_clang_O0_32	6.3	88.5	71.4	42
8.30_gcc_O2_32	19	8.30_clang_O2_32	4.5	90	70.8	48
8.30_gcc_O0_32	13	8.30_clang_O2_32	4.5	84.2	65.1	62
8.30_gcc_O2_32	19	8.30_gcc_O3_32	7.4	88.1	79.1	78
7.6_gcc_O0_32	13.3	8.30_gcc_O0_32	5.6	77.1	75.1	77
7.6_gcc_O2_32	13.3	8.30_gcc_O2_32	5.3	87.9	85.4	30
8.29_gcc_O0_32	13.3	8.30_gcc_O0_32	5.6	97.4	97.8	34
8.29_gcc_O2_32	19.2	8.30_gcc_O2_32	5.3	96.7	87.9	27
8.30_gcc_O0_32	19	8.30_gcc_O0_32	6.3	99.5	80.8	27
8.30_gcc_O2_32	23	8.30_gcc_O2_32	5.3	98.9	78.1	15

Таблица 18. Результат работы инструмента на Coreutils.

Например, для версии 8.30, скомпилированной компилятором Clang с оптимизационным флагом O0 для архитектуры x86-64, размер которой составляет 16 МБ в оригинальном виде и 7 МБ после удаления символов отладки, точность составила 98,8%, а время анализа — 38 секунд. Для той же версии, скомпилированной тем же компилятором для той же архитектуры, но с оптимизационным флагом O2 (17 МБ в оригинальном виде и 4,9 МБ после удаления символов отладки), точность снизилась до 97,2%, но время анализа уменьшилось до 22 секунд. Таким образом, изменения в компиляторе, оптимизации и архитектуре влияли на точность, полноту и время анализа, но в большинстве случаев инструменты демонстрировали высокую эффективность при различных конфигурациях.

Также инструменты VCC, BinDiff (версия 6) и Diaphora (версия 2.0.3) были сравнены на вышеупомянутой тестовой системе, результаты приведены в таблице 19.

Как видно из таблицы, точность и полнота инструментов снижаются, если программы скомпилированы с различными флагами оптимизации. Это показывает, что компиляторы генерируют крайне различные бинарные коды при использовании разных оптимизаций. Даже результаты с различными компиляторами, но с одинаковыми оптимизациями, оказываются выше.

Из таблицы видно, что результаты инструмента VCC превосходят результаты конкурентов. Наилучшую точность показывает Diaphora, но полнота этого инструмента значительно ниже, чем у других. В среднем, F1-мера для VCC составляет 85,6%, для BinDiff — 82,4%, для Diaphora — 64,7%. Более того, разница в F1-мере становится больше, когда различия между версиями анализируемых программ больше (сделано много изменений) или когда программы компилируются с разными флагами компиляции.

Версия 1, компилятор, оптимизация	Версия 2, компилятор, оптимизация, без отладочной информации	ВСС точность (%)	ВСС полнота (%)	BinDiff точность (%)	BinDiff полнота (%)	Diaphora точность (%)	Diaphora полнота (%)
8.30_clang_O0	8.30_clang_O0	98.8	98.3	98.8	97.9	98.3	75
8.30_gcc_O0	8.30_gcc_O0	98.8	98.3	98.8	97.9	98.3	74.7
8.30_clang_O2	8.30_clang_O2	97.2	85.6	98.3	85.9	97.5	59.2
8.30_gcc_O2	8.30_gcc_O2	98.2	82.4	98.2	81.7	97.3	55.2
8.30_clang_O0	8.30_clang_O2	79.3	67.4	72.2	61.5	93.1	34.3
8.30_gcc_O0	8.30_gcc_O2	84.8	69.2	75.5	63.7	91.3	34.5
8.30_clang_O2	8.30_clang_O3	90.7	81.2	86.6	76.6	97.2	41.2
8.30_gcc_O2	8.30_gcc_O3	90	77	88.7	75.4	93.5	44.8
8.30_gcc_O0	8.30_clang_O0	94.9	89.4	91.2	85.4	95.6	38.1
8.30_gcc_O2	8.30_clang_O2	89.1	76.1	81.1	71	93.8	36.7
8.30_gcc_O0	8.30_clang_O2	78.9	67.7	71.2	61.5	93	34.5
8.29_gcc_O0	8.30_gcc_O0	97.8	98.2	97.8	97.8	97.6	72.5
8.29_gcc_O2	8.30_gcc_O2	97	82.2	96.6	81.4	96.3	52.8
7.6_gcc_O0	8.30_gcc_O0	72.1	69.5	62.9	62.9	92.5	35.5
7.6_gcc_O2	8.30_gcc_O2	85.4	77.9	83.4	76.5	95.1	46.5
<b>В среднем</b>		<b>90.2</b>	<b>81.4</b>	<b>86.8</b>	<b>78.5</b>	<b>95.4</b>	<b>49.0</b>

Таблица 19. Сравнение инструмента ВСС с существующими инструментами.

#### 4.4. Выводы

В данной главе представлен метод выявления изменений между версиями программ, объединяющий анализ на основе метрик и разработанный метод поиска клонов кода. Основной особенностью метода является возможность сопоставления функций по принципу «многие ко многим».

Метод реализуется в два этапа: на первом этапе используются хеши для быстрой фильтрации непохожих случаев, на втором этапе выполняется детальный анализ их ГЗП. Такой подход позволяет эффективно находить изменения, и также минимизировать количество ложных срабатываний.

Экспериментальная оценка метода на реальных программных проектах показала его высокую точность при анализе различных версий программ. Таким образом, предложенный метод может быть применен в задачах анализа изменений программ, выявления модификаций в исполняемых файлах, а также в задачах реверс-инжиниринга для определения различий между версиями программного обеспечения.

## **Глава 5. Методы идентификации статически связанных библиотек и поиска копий известных уязвимостей с использованием разработанного метода поиска клонов фрагментов кода**

В этой главе описываются методы идентификации статически связанных библиотек и поиска копий известных уязвимостей с использованием разработанного метода поиска клонов фрагментов кода.

### **5.1. Идентификация статически связанных библиотек**

В этом разделе описывается метод идентификации версий статически связанных библиотек, который использует метод сопоставления функций, описанный в главе 4.

#### ***5.1.1. Типы библиотек и их связывание***

Библиотека – это пакет кода, который предназначен для повторного использования многими программами. С точки зрения организации и использования библиотек, они бывают статическими и динамическими.

Статическая библиотека — это коллекция объектных файлов, которые присоединяются к программе во время линковки программы. В результате связывания со статической библиотекой программа включает все используемые ей функции, что увеличивает ее размер, но делает более автономной. Так как, чтобы пользователи могли запускать программу, нужно распространять только исполняемый файл. Поскольку библиотека становится частью программы, это гарантирует, что с программой всегда будет использоваться правильная версия библиотеки. С другой стороны, поскольку копия библиотеки становится частью каждого исполняемого файла, который ее использует, это может привести к потере большого количества места

Динамическая библиотека загружается операционной системой по «требованию» запущенной программы уже в ходе ее выполнения. Загрузчик

проверяет все библиотеки, прилинкованные к программе на наличие требуемых объектных файлов, затем загружает их в память и присоединяет их в копии запущенной программы, находящейся в памяти. Одним из преимуществ динамических библиотек является то, что одна копия библиотеки может быть совместно использована многими программами, что экономит память.

При статическом связывании все подключенные к проекту библиотеки вшиваются в исполняемый файл, для большей автономности программы. При динамическом связывании библиотеки, подключенные к проекту, распространяются отдельно от исполняемого файла, но вместе с ним, что соответствует концепции модульности.

### ***5.1.2. Схема идентификации статически связанных библиотек***

Метод для выявления статически связанных библиотек в исполняемом файле заключается в следующем: на вход подается исполняемый файл для анализа, а также набор библиотек с разными версиями, которые могут быть статически связаны с этим файлом. Для каждой пары «исполняемый файл – библиотека» определяются общие функции и рассчитывается степень схожести функций. На основе результатов сравнения всех пар определяется, какие библиотеки и их версии с наибольшей вероятностью являются статически связанными с исполняемым файлом. В завершение формируется список библиотек и их версий, которые предположительно связаны с данным исполняемым файлом (Рисунок 25).





Рисунок 25. Схема идентификации статически связанных библиотек.

### 5.1.2.1. Сравнение исполняемого файла с одной библиотекой

Процесс идентификации статически связанной библиотеки начинается с анализа исполняемого файла и заданной библиотеки (статической или динамической), которые необходимо сравнить (Рисунок 26). Для этого исследуются функции исполняемого файла и их возможное совпадение с функциями библиотеки. Дальнейшие шаги зависят от типа библиотеки. Так как статическая библиотека – это архив объектных файлов, то сначала из нее извлекаются объектные файлы. Для каждого объектного файла генерируются ГЗП и ГВФ. Так как одна и та же функция может присутствовать в нескольких объектных файлах, из их ГЗП сохраняется только один. Из ГВФ всех объектных файлов строится один ГВФ.

В случае динамической библиотеки обработка выполняется так же, как обработка исполняемого файла: анализируется весь файл, на основании которого формируются его ГВФ и ГЗП.

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%.

Эти функции позволяют определить, связана ли данная библиотека или ее версия с исполняемым файлом.

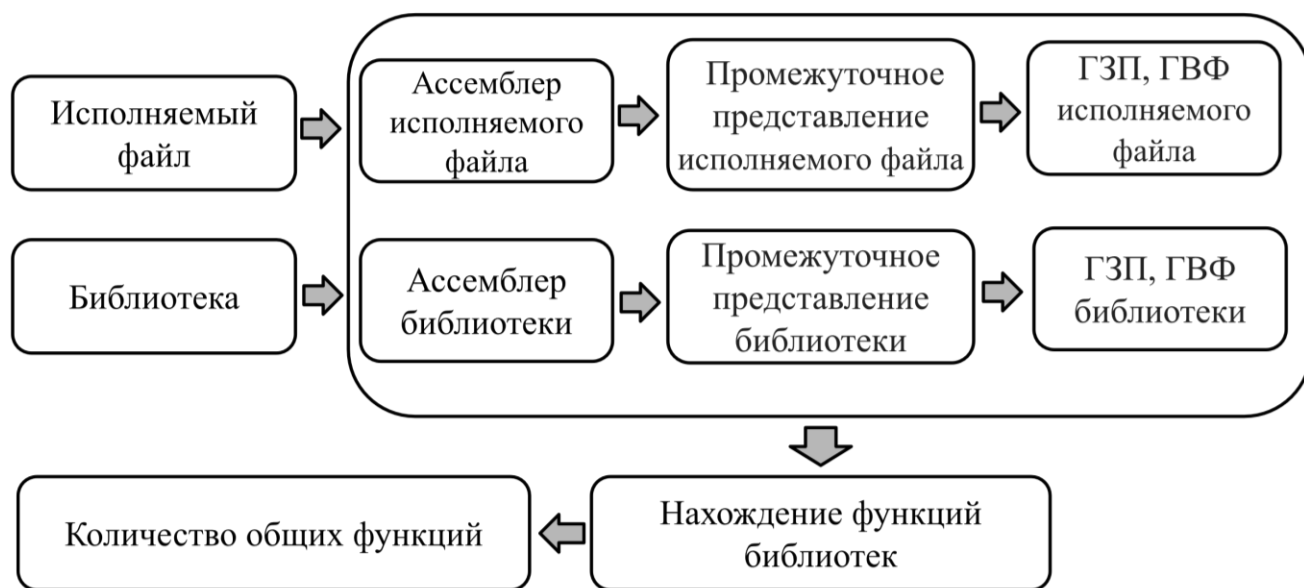


Рисунок 26. Схема сравнения исполняемого файла с одной библиотекой.

### 5.1.3. Результаты идентификации статически связанных библиотек

Для выявления того, насколько хорошо работает инструмент, он был протестирован на нескольких версиях Binutils [185] и Coreutils [120], 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.

Ниже представлены таблицы 20, 21 и 22, показывающие результаты оценки точности и полноты обнаружения функций определенной библиотеки в исполняемых файлах, взятых из пакетов Binutils или Coreutils. В таблицах 20 и 21 приведены данные по обнаружению функций библиотек libgcc.a, libgcc\_eh.a, libiberty.a и libbfd.a в различных исполняемых файлах пакета Binutils (addr2line, objcopy, strip), скомпилированных с оптимизационными флагами -O2 и -O0, соответственно. В целом, точность и полнота обнаружения одинаковы в обоих случаях. Для addr2line точность обнаружения функций из libgcc.a составила 85%, а полнота — 100%. Для библиотеки libiberty.a эти показатели выше — 90% и

100% соответственно. В то же время функции libbfd.a не использовались в этом файле (в таблице поставлено -). Аналогичные оценки наблюдаются и для других исполняемых файлов: в большинстве случаев точность остается высокой, а полнота достигает 100%, за исключением отдельных случаев, где полнота немного снижается при использовании libiberty.a. Для библиотеки libiberty.a показатели немного отличаются в зависимости от примененных оптимизаций. Например, для файла strip точность обнаружения функций из libbfd.a снизилась до 90%, а полнота — до 91.4%. Это свидетельствует о том, что уровень оптимизации компилятора оказывает влияние на точность и полноту обнаружения.

Таблица 22 демонстрирует точность и полноту работы инструмента на исполняемых файлах Coreutils, скомпилированных с компилятором GCC с флагом оптимизации O0. Например, для исполняемого файла cp точность обнаружения функций из libcutils.a составила 92.5%, а полнота — 93.75%. Для libgcc.a эти показатели составили 86.67% и 100% соответственно, а для libgcc\_eh.a — 100% и 100%. Аналогичные результаты наблюдаются и для исполняемых файлов dirname и mkdir, где точность колебалась в диапазоне 87.8%–100%, а полнота оставалась на уровне 100%.

<b>Библиотека</b> <b>Исполняемый файл</b>	libgcc.a	libgcc_eh.a	libiberty.a	libbfd.a
addr2line	85%, 100%	100%, 100%	90%, 100%	-
Objcopy	85%, 100%	100%, 100%	90%, 94.1%	92.7%, 91.4%
Strip	80%, 100%	100%, 100%	95%, 93.75%	92%, 91.2%

*Таблица 20. Точность и полнота работы инструмента на исполняемых файлах binutils\_2.35 (GCC O0).*

<b>Библиотека</b> <b>Исполняемый</b> <b>Файл</b>	<b>Библиотека</b>			
	libgcc.a	libgcc_eh.a	libiberty.a	libbfd.a
addr2line	85%, 100%	100%, 100%	90%, 94.1%	-
Objcopy	85%, 100%	100%, 100%	90%, 94.1%	90.7%, 91.67%
Strip	85%, 100%	100%, 100%	95%, 93.75%	90%, 91.4%

*Таблица 21. Точность и полнота работы инструмента на исполняемых файлах binutils\_2.35 (GCC O2).*

<b>Библиотека</b> <b>Исполняемый</b> <b>Файл</b>	<b>Библиотека</b>		
	libcoreutils.a	libgcc.a	libgcc_eh.a
Cp	92.5%, 93.75%	86.67%, 100%	100%, 100%
Dirname	88.6%, 100%	80%, 100%	100%, 100%
Mkdir	87.8%, 93.3%	86.67%, 100.0%	100%, 100%

*Таблица 22. Точность и полнота работы инструмента на исполняемых файлах coreutils\_8.30 (GCC O0).*

А в таблице 23 приведены усредненные показатели точности и полноты для всех библиотек, связанных в исполняемых файлах пакетов 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 23. Результат работы инструмента.

## 5.2. Обнаружение ИУ в программах

Известные уязвимости (ИУ) представляют собой ранее обнаруженные и задокументированные слабые места в программном обеспечении, которые могут использоваться злоумышленниками. Метод обнаружения известных уязвимостей с использованием метода поиска клонов фрагментов кода позволяет автоматически выявлять фрагменты программ, содержащие уязвимый код, основываясь на его сходстве с уже известными уязвимостями. Этот подход эффективен для обнаружения ИУ в исходных кодах и скомпилированных программах, что делает его полезным инструментом для анализа безопасности и

поиска потенциально уязвимых компонентов.

В этом разделе описывается метод поиска копий известных уязвимостей с использованием разработанного метода поиска клонов фрагментов кода.

### **5.2.1. Метод обнаружения ИУ в программах**

Предложенный метод обнаружения клонов фрагментов использовался для обнаружения клонов известных уязвимостей в программном обеспечении с открытым исходным кодом посредством комплексного четырехэтапного процесса (Рисунок 27). Во-первых, доступная информация о ИУ и соответствующие исправления были собраны из ресурсов с открытым исходным кодом, таких как GitHub, Nist и Mitre. Затем на основе фиксирующих патчей были извлечены фрагменты кода, соответствующие известному ИУ, в частности функции исходного кода в уязвимой версии, к которым были применены фиксирующие патчи.

Чтобы изолировать фактические исправления ошибок от рефакторинга кода в патче, алгоритм сравнивает предыдущие и обновленные версии измененных функций. Если окажется, что эти версии являются клонами типа 2 (исключением являются изменения значений и типов переменных), алгоритм исключает эту часть патча из дальнейшего анализа. Такой подход помогает исключить некоторые случаи изменений, например переименование переменных, которые не связаны с исправлениями ошибок.

ГЗП создаются для фрагментов ИУ и сохраняются в хранилище. На заключительном этапе генерируются ГЗП для целевого проекта, и предложенный метод обнаружения клонов фрагментов применяется для сравнения ГЗП фрагментов ИУ с ГЗП целевых проектов, обеспечивая комплексное обнаружение клонов ИУ в программном обеспечении с открытым исходным кодом.



Рисунок 27. Процесс нахождения клонов известных уязвимостей.

### 5.2.1.1. Реализация нахождения клонов ИУ

Для сбора исходного кода проектов используются API GitHub и утилита Debian «apt-get source». Информация о ИУ извлекается из открытых ресурсов, таких как GitHub, NIST и MITRE. Для извлечения фрагментов исходного кода, соответствующих собранным ИУ, эти ИУ связываются с их соответствующими исходными репозиториями. После определения исходного репозитория коммит, исправляющий ИУ, выявляется из истории разработки.

Обнаружение клонов ИУ осуществляется на основе ГЗП, сгенерированных тремя методами. Использование генерации на основе ANTLR упрощает процесс, так как не требует сборки исходного кода, что позволяет легко генерировать ГЗП для собранных проектов и целевых фрагментов кода, связанных с известными

ИУ. Другие методы также подходят для обнаружения клонов ИУ, если доступны команды сборки.

Далее из полного исправления исключаются изменения, связанные с рефакторингом кода, такие как переименование переменных. Это достигается с использованием FCD для старых и исправленных версий функций с процентом сходства 100%.

После подготовки всех необходимых данных FCD применяется для обнаружения клонов фрагментов кода, содержащих ИУ, в различных проектах. Каждый найденный клон рассматривается как потенциально новая ошибка, которая может содержать ту же уязвимость, что и исходный код, связанный с ИУ.

### ***5.2.2. Результаты обнаружения клонов ИУ***

Для оценки эффективности обнаружения клонов было проведено два эксперимента. В первом эксперименте была проанализирована 12.6 версия операционной системы Debian (самая последняя на момент тестирования), к ее пакетам применено обнаружение клонов ИУ. Целью было выявить пакеты в Debian, которые используют устаревшее стороннее программное обеспечение с известными уязвимостями. Во время второго эксперимента был расширен анализ примерно до 1000 репозиториях GitHub. Эти эксперименты показали, что многие проекты полагаются на устаревшее стороннее программное обеспечение, что потенциально подвергает их эксплуатации или уязвимостям безопасности.

Чтобы продемонстрировать эффективность инструмента обнаружения клонов ИУ, было выявлено и сообщено об использовании устаревшего кода в следующих проектах: 0ad [186], PHP [187], CMake [188], OpenJPEG [189], rct [190], Emscripten [191], fldigi [192], POV-Ray [193], ntopng [194], pcl [195], Kamailio [196], H2O [197], ИТК [198], Dragonfly [199], Webdis [200], Defold [201], and ODrive [202]. В целом, было сообщено о семнадцати ошибках, девять из них уже приняты, а две отклонены (поскольку эти проекты используются в качестве тестов). По шести проблемам пока не было получено ответов. Таблица 24 отображает сведения об общедоступных находках, которые получили обратную



связь от сопровождающих соответствующих проектов. Из-за проблем безопасности два из находок остаются конфиденциальными.

Имя проекта	Ссылка на проблему
Cmake	<a href="https://gitlab.kitware.com/cmake/cmake/-/issues/26112">https://gitlab.kitware.com/cmake/cmake/-/issues/26112</a>
OpenJPEG	<a href="https://github.com/uclouvain/openjpeg/issues/1539">https://github.com/uclouvain/openjpeg/issues/1539</a>
PointCloudLibrary	<a href="https://github.com/PointCloudLibrary/pcl/issues/6080">https://github.com/PointCloudLibrary/pcl/issues/6080</a>
Oad	<a href="https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=1036970">https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=1036970</a>
ITK	<a href="https://github.com/InsightSoftwareConsortium/ITK/issues/4777">https://github.com/InsightSoftwareConsortium/ITK/issues/4777</a>
Dragonfly	<a href="https://github.com/dragonflydb/dragonfly/issues/3830">https://github.com/dragonflydb/dragonfly/issues/3830</a>
Webdis	<a href="https://github.com/nicolasff/webdis/issues/251">https://github.com/nicolasff/webdis/issues/251</a>
Defold	<a href="https://github.com/defold/defold/issues/9527">https://github.com/defold/defold/issues/9527</a>
ODrive	<a href="https://github.com/odriverobotics/ODrive/issues/751">https://github.com/odriverobotics/ODrive/issues/751</a>

Таблица 24. Сообщенные и устраненные проблемы.

### 5.3. Выводы

В данной главе рассмотрены методы автоматического выявления статически связанных библиотек и обнаружения известных уязвимостей в программах.

Метод идентификации библиотек позволяет автоматически определять, какие сторонние библиотеки были статически связаны в исполняемый файл, а также выявлять их версии. Это особенно важно для задач обеспечения безопасности, поскольку устаревшие и уязвимые версии библиотек могут представлять угрозу.

Дополнительно предложен метод поиска копий известных уязвимостей, основанный на анализе клонов кода. Таким образом, разработанные инструменты обеспечивают автоматизированный анализ исходного и исполняемого кода с целью выявления потенциальных угроз.

## Заключение

Основные результаты работы заключаются в следующем:

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 циклической проверки избыточности (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, способного анализировать десятки миллионов строк исходного и соответствующего исполняемого кода.

## Литература

- [1] М. Арутюнян, «ПОИСК КЛОНОВ КОДА НА ОСНОВЕ СЕМАНТИЧЕСКОГО АНАЛИЗА ПРОГРАММ,» в *СБОРНИК НАУЧНЫХ СТАТЕЙ СНО ЕГУ МАТЕРИАЛЫ ЕЖЕГОДНОЙ НАУЧНОЙ СЕССИИ 2016 ГОДА*, Ереван, 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 (Труды ИСП РАН)*, т. 31, № 6, стр. 89-98, 2019.
- [4] М. 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, IVMEM 2019, 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, 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), Moscow, Russia, 2020*.
- [7] М. С. Арутюнян, Р. А. Оганнисян и Х. С. Смбатьян, «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] М. 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 (IVMEM), Nizhny Novgorod, Russia, September 2021*.
- [10] М. 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*, Venice, Italy, 2024.
- [11] М. Arutunian, S. Sargsyan, M. Mehrabyan, L. Bareghamyanyan и H. Aslanyan,

- «Automatic Recognition and Replacement of Cyclic Redundancy Checks for Program Optimization,» *IEEE Access*, т. 12, стр. 192146 - 192158, 2024.
- [12] М. Arutunian, S. Sargsyan, Н. Hovhannisyanyan, G. Khroyan, А. Mkrtchyan, Н. Movsisyan, А. Avetisyan и Н. Aslanyan, «Accurate Code Fragment Clone Detection and Its Application in Identifying Known CVE Clones,» *International Journal of Information Security*, т. 24, № 55, January 2025.
- [13] 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.
- [14] Ш. Курмангалеев, А. Асланян, М. Арутюнян, Р. Оганесян, В. Варданян и С. С.С, «LibraryIdentifier». РФ Патент ЭВМ № 2021665076, 17 09 2021.
- [15] В. S. Baker, «A Program for Identifying Duplicated Code,» в *Proceedings of Computing Science and Statistics: 24th Symposium on the Interface*, 1992.
- [16] В. S. Baker, «Parameterized Diff,» *Symposium on Discrete Algorithms*, стр. 854-855, 1999.
- [17] J. H. Johnson, «Identifying redundancy in source code using fingerprints.,» в *Conference of the Centre for Advanced Studies on Collaborative Research*, Toronto, Ontario, Canada, vol. 2, стр. 171-183, 1993.
- [18] J. H. Johnson, «Substring Matching for Clone Detection and Change Tracking.,» в *Proc. International Conference on Software Maintenance*, 1994.
- [19] R. М. Карп и М. О. Rabin, «Efficient randomized pattern-matching algorithms,» *IBM Journal of Research and Development*, т. 31, № 2, стр. 249-260, 1987.
- [20] S. Ducasse, М. Rieger и S. Demeyer, «A Language Independent Approach for Detecting Duplicated Code,» в *International Conference on Software Maintenance - 1999*, стр. 109-118, 1999.
- [21] S. Ducasse, О. Nierstrasz и М. Rieger, «On the effectiveness of clone detection by string matching.,» *Journal of Software Maintenance and Evolution: Research and Practice*, т. 18.1, стр. 37-58, 2006.
- [22] R. Wettel и R. Marinescu, «Archeology of code duplication: Recovering duplication chains from small duplication fragments,» *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, стр. 8, 2005.
- [23] С. К. Roy и J. R. Cordy, «NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalizationProgram Comprehension,» в *ICPC 2008. The 16th IEEE International Conference*, 2008.
- [24] С. К. Roy, «Detection and analysis of near miss software clones,» в *25th IEEE International Conference on Software Maintenance (ICSM)*, 447-450, 2009.
- [25] С. К. Roy и J. R. Cordy, «A mutation/injection-based automatic framework for evaluating code clone detection tools,» в *Proceedings of the IEEE International Conference on Software Testing Verification and Validation Workshops*, Denver, Colorado, USA, стр. 157-166, 2009.

- [26] S. Lee и J. Iryoung, «SDD: high performance code clone detection system for large scale source code,» в *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005.
- [27] D. Cutting и J. Pedersen, «Optimization for dynamic inverted index maintenance,» в *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, 1989.
- [28] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman и A. Y. Wu, «An optimal algorithm for approximate nearest neighbor searching,» в *Proceedings of the fifth annual ACMSIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics*, 1994.
- [29] E. Juergens, F. Deissenboeck и B. Hummel, «CloneDetective – a workbench for clone detection research,» в *Proceedings of 31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, стр. 603-606, 2009.
- [30] E. Juergens, F. Deissenboeck, B. Hummel и S. Wagner, «Do code clones matter?,» в *Proceedings of 31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, стр. 485-495, 2009.
- [31] «CCFinder,» [В Интернете]. Available: <http://www.ccfinder.net>.
- [32] T. Kamiya, S. Kusumoto и K. Inoue, «CCFinder: A multilinguistic token-based code clone detection system for large scale source code,» в *IEEE Transactions on Software Engineering.*, 2002.
- [33] Li, Lu, Myagmar и Zhou, «CP-Miner: Finding copy-paste and related bugs in large-scale software code,» в *IEEE Transactions on Software Engineering.*, 2006.
- [34] T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, S. Kawaguchi и H. Iida, «SHINOBI: a tool for automatic code clone detection in the IDE,» в *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE'09)*, Lille, France, стр. 313-314, 2009.
- [35] H. Basit, S. Pugliesi, W. Smyth, A. Turpin и S. Jarzabek, «Efficient Token Based Clone Detection with Flexible Tokenization,» в *Proceedings of the Joint Meeting of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Dubrovnik, Croatia, стр. 513-515, 2007.
- [36] R. Koschke, R. Falke и P. Frenzel, «Clone Detection Using Abstract Syntax Suffix Trees,» в *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, Benevento, Italy, стр. 253-262, October 2006.
- [37] H. Li и S. Thompson, «Clone detection and removal for Erlang/OPT within a refactoring environment,» в *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09)*, Savannah, GA, USA, стр. 169–178, 2009.
- [38] L. Prechelt, G. Malpohl и M. Philippsen, «Finding plagiarisms among a set of programs with JPlag,» *Journal of Universal Computer Science*, т. 8(11), стр. 1016-1038, November 2002.

- [39] S. Schleimer, D. S. Wilkerson и A. Aiken, «Winnowing: local algorithms for document fingerprinting,» *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, № San Diego, California, стр. 7685, June 2003.
- [40] S. Livieri, Y. Higo, M. Matsushita и K. Inoue, «Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder,» в *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, USA, стр. 106-115, 2007.
- [41] Т. Kamiya, «CCFinderX,» [В Интернете]. <https://github.com/gpoo/ccfinderx>.
- [42] Y. Sasaki, T. Yamamoto, Y. Hayase и K. Inoue, «Finding file clones in FreeBSD ports collection,» в *Proceedings of the 7th IEEE Working Conference on Mining Software Repositories*, Cape Town, South Africa, стр. 102-105, 2010.
- [43] H. Sajjani, V. Saini, J. Svajlenko, C. K. Roy и C. V. Lopes, «SourcererCC: scaling code clone detection to big-code,» в *ICSE '16: Proceedings of the 38th International Conference on Software Engineering*, 2016.
- [44] N. Göde и R. Koschke, «Incremental clone detection,» в *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, Kaiserslautern, Germany, стр. 219-228, 2009.
- [45] W. Yang, «Identifying syntactic differences between two programs,» *Software Practice and Experience*, т. 21, № 7, стр. 739-755, 1991.
- [46] I. D. Baxter, A. Yahin, L. Moura и M. Sant'Anna, «Clone detection using abstract syntax trees,» в *Proceedings of the 14th International Conference on Software Maintenance (ICSM '98)*, Bethesda, Maryland, USA, стр. 368-378, 1998.
- [47] «SimScan,» [В Интернете]. <http://www.blue-edge.bg/download.html>.
- [48] «Project Bauhaus,» [В Интернете]. <http://www.bauhaus-stuttgart.de>.
- [49] R. Falke, P. Frenzel и R. Koschke, «Empirical evaluation of clone detection using syntax suffix trees,» *Empirical Software Engineering*, т. 13, № 6, стр. 601-643, 2008.
- [50] R. Tairas и J. Gray, «Phoenix-based clone detection using suffix trees,» в *Proceedings of the 44th Annual Southeast Regional Conference (ACM-SE'06)*, Melbourne, Florida, USA, стр. 679-684, 2006.
- [51] D. Gitchell и N. Tran, «Sim: a utility for detecting similarity in computer programs,» *ACM SIGCSE Bulletin*, т. 31, № 1, стр. 266-270, 1999.
- [52] L. Jiang, G. Mishnerghi, Z. Su и S. Glondu, «DECKARD: Scalable and accurate treebased detection of code clones,» в *Proceedings of 29th International Conference on Software Engineering (ICSE'07)*, Minneapolis, MN, USA, стр. 96-105, 2007.
- [53] W. S. Evans, C. W. Fraser и F. Ma, «Clone detection via structural abstraction,» *Software Quality Journal*, т. 17, № 4, стр. 309-330, 2009.
- [54] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofah и T. N. Nguyen,

- «ClemanX: Incremental clone detection tool for evolving software,» в *Proceedings of 31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, 437-438, 2009.
- [55] T. T. Nguyen, H. A. Nguyen, J. M. Al-Kofahi, N. H. Pham и T. N. Nguyen, «Scalable and incremental clone detection for evolving software,» в *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM '09)*, Edmonton, AB, стр. 491-494, 2009.
- [56] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan и Z. Su, «Detecting code clones in binary executable,» в *Proceedings of International Symposium on Software Testing and Analysis*, Chicago, Illinois, USA, стр. 117–127, 2009.
- [57] P. Bulychev и M. Minea, «Duplicate code detection using anti-unification,» в *Proceedings of Spring/Summer Young Researchers' Colloquium on Software Engineering*, St. Petersburg, Russia, стр. 51–54, 2008.
- [58] H. Lee и K. Doh, «Tree-pattern-based duplicate code detection,» в *Proceedings of International Workshop on Data-intensive Software Management and Mining*, Philadelphia, PA, USA, стр. 7–12, 2009.
- [59] C. Brown и S. Thompson, «Clone detection and elimination for Haskell,» в *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'10)*, Madrid, Spain, стр. 111–120, 2010.
- [60] V. Wahler, D. Seipel, J. W. Gudenberg и G. Fischer, «Clone detection in source code by frequent itemset techniques,» в *Proceedings of the 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM'04)*, Chicago, IL, USA, стр. 128–135, 2004.
- [61] W. Evans и C. Fraser, «Clone Detection via Structural Abstraction,» в *Proceedings of the 14th Conference on Reverse Engineering (WCRE'07)*, Vancouver, BC, Canada, October 2007.
- [62] M. Chilowicz, E. Duris и G. Roussel, «Syntax tree fingerprinting for source code similarity detection,» в *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC'09)*, Vancouver, British Columbia, Canada, стр. 243–247, 2009.
- [63] F. Jacob, D. Hou и P. Jablonski, «Actively comparing clones inside the code editor,» в *Proceedings of 4th International Workshop on Software Clones*, Cape Town, SA, стр. 1–8, 2010.
- [64] B. Biegel и S. Diehl, «JCCD: a flexible and extensible API for implementing custom code clone detectors,» в *Proceedings of 25th International Conference on Automated Software Engineering, (ASE'10)*, Antwerp, Belgium, стр. 167–168, 2010.
- [65] B. Biegel и S. Diehl, «Highly configurable and extensible code clone detection,» в *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE'10)*, Beverly, MA, USA, стр. 237–241, 2010.
- [66] J. Ferrante, K. Ottenstein и J. Warren, «The program dependence graph and its use in,» *Trans. on Prog. Lang. and Syst. (TOPLAS)*, стр. 319-349, 1987.

- [67] R. Komondoor и S. Horwitz, «Using slicing to identify duplication in source code,» в *Proceedings of the 8th International Symposium on Static Analysis (SAS'01)*, Paris, France, стр. 40-56, 2001.
- [68] Y. Higo и S. Kusumoto, «Code clone detection on specialized PDG's with heuristics,» в *Proceedings of the 15th European Conference on Software Maintenance and Reengineering (CSMR'11)*, Oldenburg, Germany, стр. 75–84, 2011.
- [69] А. Асланян, *Методы статического анализа для поиска дефектов в исполняемом коде программ*, Москва: диссертация на соискание ученой степени кандидата физико-математических наук, 2019.
- [70] С. С. Саргсян, *Методы оптимизации алгоритмов статического и динамического анализа программ*, Москва: диссертация на соискание ученой степени доктора технических наук, 2024.
- [71] J. Krinke, «Identifying similar code with program dependence graphs,» в *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01)*, Stuttgart, Germany, стр. 301–309, 2001.
- [72] C. Liu, C. Chen, J. Han и P. S. Yu, «GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis,» в *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'06)*, Philadelphia, USA, стр. 872-881, August 2006.
- [73] W.-K. Chen, B. Li и R. Gupta, «Code Compaction of Matching Single-Entry MultipleExit Regions,» в *Proceedings of the 10th Annual International Static Analysis Symposium (SAS'03)*, San Diego, CA, USA, стр. 401-417, June 2003.
- [74] J. Mayrand, C. Leblanc и E. M. Merlo, «Experiment on the automatic detection of function clones in a software system using metrics,» в *Proceedings of the 12th International Conference on Software Maintenance (ICSM'96)*, Monterey, CA, USA, стр. 244–253, 1996.
- [75] J.-F. Patenaude, E. Merlo, M. Dagenais и B. Lague, «Extending software quality assessment techniques to java systems,» в *Proceedings of the 7th International Workshop on Program Comprehension (IWPC'99)*, Pittsburgh, PA, USA, pp 49-56, May 1999.
- [76] K. Kontogiannis, R. Demori, E. Merlo, M. Galler и M. Bernstein, «Pattern matching for clone and concept detection,» *Automated Software Engineering*, т. 3, № 1-2, стр. 77-108, 1996.
- [77] K. Kontogiannis, «Evaluation experiments on the detection of programming patterns using software metrics,» в *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE'97)*, Amsterdam, The Netherlands, стр. 44–54, 1997.
- [78] F. Lanubile и T. Mallardo, «Finding function clones in web applications,» в *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR'03)*, Benevento, Italy, стр. 379–386, 2003.
- [79] N. Davey, P. Barson, S. Field и R. J. Frank, «The Development of a Software



- Clone Detector,» *International Journal of Applied Software Technology*, т. 1, № 3/4, стр. 219-236, 1995.
- [80] A. Perumal, S. Kanmani и E. Kodhai, «Extracting the similarity in detected software clones using metrics,» в *Proceedings of International Conference on Computer and Communication Technology*, Allahabad, Uttar Pradesh, India, стр. 575–579, 2010.
- [81] E. Kodhai, S. Kanmani, A. Kamatchi, R. Radhika и B. V. Saranya, «Detection of Type-1 and Type-2 code clones using textual analysis and metrics,» в *Proceedings of 2010 International Conference on Recent Trends in Information, Telecommunication and Computing*, Kochi, Kerala, India, стр. 241–243, 2010.
- [82] M. Dagenais, E. Merlo, B. Laguë и D. Proulx, «Clones occurrence in large object oriented software packages,» в *Proceedings of the 8th IBM Centre for Advanced Studies Conference (CASCON'98)*, Toronto, Ontario, Canada, стр. 192–200, 1998.
- [83] Z. O. Li и J. Sun, «A metric space based software clone detection approach,» в *Proceedings of 2nd International Conference on Software Engineering and Data Mining*, Chengdu, China, стр. 111–116, 2010.
- [84] «BigCloneBench,» [В Интернетe]. <https://github.com/clonebench/BigCloneBench/>.
- [85] J. Rennecke и J. Beniston, «GCC CRC patches,» [В Интернетe]. <https://www.mail-archive.com/gcc-patches@gcc.gnu.org/msg281497.html>.
- [86] «llvm-diff - LLVM structural ‘diff’,» [В Интернетe]. <https://llvm.org/docs/CommandGuide/llvm-diff.html>. [Дата обращения: 02 2025].
- [87] «git-diff,» [В Интернетe]. <https://git-scm.com/docs/git-diff>. [Дата обращения: 02 2025].
- [88] «Beyond Compare,» [В Интернетe]. <https://www.scootersoftware.com/>. [Дата обращения: 02 2025].
- [89] «SourceGear DiffMerge,» [В Интернетe]. <https://sourcegear.com/diffmerge/>. [Дата обращения: 02 2025].
- [90] B. S. Baker, U. Manber и R. Muth, «Compressing Differences of Executable Code,» в *ACMSIGPLAN Workshop on Compiler Support for System Software*, 1999.
- [91] Z. Wang, K. Pierce и S. McFarling, «BMAT - A Binary Matching Tool,» в *In Second ACM Workshop on Feedback-Directed and Dynamic Optimization*, 1999.
- [92] «Diaphora,» [В Интернетe]. <https://github.com/joxeankoret/diaphora>.
- [93] Z. Wang, K. Pierce и S. McFarling, «BMAT – A binary matching tool for stale profile propagation,» *Journal of Instruction-Level Parallelism 2*, стр. 1-20, 2000.
- [94] H. Flake, «Structural comparison of executable objects,» в *Detection of Intrusions and Malware & Vulnerability Assessment*, 2004.
- [95] T. Dullien и R. Rolles, «Graph-based comparison of executable objects,»

- Symposium sur la Securite des Technologies de l'Information et des Communications*, 2005.
- [96] Y. R. Lee, B. Kang и E. G. Im, «Function matching-based binary-level software similarity,» *Research in Adaptive and Convergent Systems*, 2013.
- [97] H. Aslanyan, A. Avetisyan, M. Arutunian, G. Keropyan, S. Kurmangaleev и V. Vardanyan, «Scalable Framework for Accurate Binary Code Comparison,» в *2017 Ivannikov ISPRAS Open Conference (ISPRAS)*, Moscow, 2017.
- [98] D. Bruschi, L. Martignoni и M. Monga, «Detecting Self-mutating Malware Using Control-flow Graph Matching,» в *Detection of Intrusions and Malware and Vulnerability Assessment. Springer-Verlag*, 2006.
- [99] D. Bruschi, L. Martignoni и M. Monga, «Code normalization for self-mutating malware,» *IEEE Security and Privacy* 5, 2, стр. 46-54, March 2007.
- [100] D. Gao, M. K. Reiter и D. Song, «Binhunt: Automatically finding semantic differences in binary programs,» в *ICICS*, 2008.
- [101] J. Ming, M. Pan и a. D. Gao, «iBinHunt: Binary Hunting with Inter-Procedural Control Flow,» в *ICISC*, Seoul, Korea, 2012.
- [102] J. Ming, D. Xu и D. Wu, «Memoized semantics-based binary diffing,» в *IFIP International Information Security Conference. Springer*, 2015.
- [103] «BinDiff,» [В Интернете]. <https://www.zynamics.com/bindiff.html>.
- [104] T. Dullien, E. Carrera, S.-M. Eppler и S. Porst, «Automated Attacker Correlation for Malicious Code,» 2010.
- [105] I. Briones и A. Gomez, «Graphs, entropy and grid computing: Automatic comparison of malware,» в *Proceedings of Virus Bulletin International Conference*, 2008.
- [106] M. Bourquin, A. King и Ed. Robbins, «BinSlayer: Accurate Comparison of Binary Executables,» в *ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.
- [107] S. Cesare, Y. Xiang и W. Zhou, «Control flow-based malware variant detection,» *IEEE Transactions on Dependable and Secure Computing* 11, 4, стр. 307-317, 2014.
- [108] H. W. Kuhn, «The Hungarian Method for the assignment problem,» в *Naval Research Logistics Quarterly*, 1955.
- [109] X. Hu, T. Chiueh и K. G. Shin, «Large-scale malware indexing using function-call graphs,» в *ACM Conference on Computer and Communications Security*, 2009.
- [110] I. Santos, F. Brezo, J. Nieves, Y. K. Peña, B. Sanz, C. Laorden и P. G. Bringas, «Idea: Opcode-Sequence-Based Malware Detection,» в *Symposium on Engineering Secure Software and Systems. Springer Berlin Heidelberg*, 2010.
- [111] B. Kang, T. Kim, H. Kwon, Y. Choi и E. G. Im, «Malware Classification Method via Binary Content Comparison,» в *Research in Applied Computation Symposium. ACM*, 2012.

- [112] X. Hu, K. G. Shin, S. Bhatkar и K. Griffin, «MutantX-S: Scalable Malware Clustering Based on Static Features,» в *USENIX Annual Technical Conference*, 2013.
- [113] B. H. Ng и A. Prakash, «Expose: Discovering potential binary code re-use,» в *Annual Computer Software and Applications Conference. IEEE*, 2013.
- [114] J. Jang, M. Woo и D. Brumley, «Towards Automatic Software Lineage Inference,» в *22nd USENIX Security Symposium*, 2013.
- [115] C. Kruegel, E. Kirda, D. Mutz, W. Robertson и G. Vigna, «Polymorphic Worm Detection Using Structural Information of Executables,» в *Symposium on Recent Advance in Intrusion Detection. Springer Berlin Heidelberg*, 2005.
- [116] M. Lindorfer, A. D. Federico, F. Maggi, P. M. Comparetti и S. Zanero, «Lines of Malicious Code: Insights into the Malicious Software Industry,» в *Annual Computer Security Applications Conference, ACM*, 2012.
- [117] L. Luo, J. Ming, D. Wu, P. Liu и S. Zhu, «Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection,» *International Symposium on Foundations of Software Engineering*, 2014.
- [118] L. Luo, J. Ming, D. Wu, P. Liu и S. Zhu, «Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection,» *n. IEEE Transactions on Software Engineering 12*, стр. 1157-1177, 2017.
- [119] A. Lakhota, M. D. Preda и R. Giacobazzi, «Fast location of similar code fragments using semantic “juice”,» *ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.
- [120] «Coreutils - GNU core utilities,» [В Интернете]. <https://www.gnu.org/software/coreutils/>.
- [121] R. M. Stallman и the GCC Developer Community, *Using the GNU Compiler Collection*, Boston, MA 02110-1301 USA: GNU Press, 2003.
- [122] «Clang: a C language family frontend for LLVM,» LLVM Project, [В Интернете]. <https://clang.llvm.org/>.
- [123] E. Itkin, «Karta – Matching Open Sources in Binaries,» [В Интернете]. <https://research.checkpoint.com/karta-matching-open-sources-in-binaries/>.
- [124] «IDA F.L.I.R.T. Technology,» Hex-Rays, 27 05 2015. [В Интернете]. <https://www.hex-rays.com/products/ida/tech/flirt/index.shtml>.
- [125] «IDA Pro disassembler,» Hex-Rays, [В Интернете]. <https://www.hex-rays.com/products/ida>.
- [126] «Compilers,» Hex Reys, [В Интернете]. <https://www.hex-rays.com/products/ida/tech/flirt/compilers.shtml>.
- [127] M. v. Tschirschnitz, «Library and Function Identification by Optimized Pattern Matching on Compressed Databases: A close to perfect identification of known code snippets,» *Conference: the 2nd Reversing and Offensive-oriented Trends*

- Symposium*, 2018.
- [128] «The GNU C Library,» [В Интернете]. <https://www.gnu.org/software/libc/>. [Дата обращения: 02 2025].
- [129] «libcdb,» Karlsruhe Institute for Technology CTF Team, 2018. [В Интернете]. <https://kitctf.de/tools/>.
- [130] Michael Burrows и David Wheeler, «A block-sorting lossless data compression algorithm,» *Technical Report 124, Digital Equipment Corporation*, 1994.
- [131] D. A. Huffman, «A method for the construction of minimum-redundancy codes.,» *Proceedings of the IRE*, 1952.
- [132] G. Navarro, «Wavelet Trees for All,» *Proceedings of 23rd Annual Symposium on Combinatorial Pattern Matching (CPM)*, 2012.
- [133] J. Bader, T. Beller, S. Gog и M. Petri, «Sdsl - succinct data structure library,» 2018. [В Интернете]. <https://github.com/simongog/sdsl-lite>.
- [134] idenLib, [В Интернете]. <https://github.com/secrary/idenLib>.
- [135] «x64dbg,» [В Интернете]. <https://x64dbg.com/>.
- [136] J. P., «Distribution de la flore alpine dans le Bassin des Dranses et dans quelques regions voisines,» *Bull. Soc. Vaudoise sci. Natur.* , т. 37, стр. 241-272, 1901.
- [137] Akabanea, Shu; Okamoto, Takeshi, «Identification of library functions statically linked to Linux malware,» *24th International Conference on Knowledge-Based and Intelligent Information & Engineering, Procedia Computer Science*, т. 176, стр. 3436-3445, 2020.
- [138] S. Akabane, «stelftools,» [В Интернете]. <https://github.com/shuakabane/stelftools>.
- [139] V. Alvarez, «YARA,» 2013. [В Интернете]. <https://yara.readthedocs.io/en/v4.1.0/index.html>.
- [140] «project0,» [В Интернете]. <https://googleprojectzero.blogspot.com/2018/12/searching-statically-linked-vulnerable.html>.
- [141] «FunctionSimSearch,» [В Интернете]. <https://github.com/googleprojectzero/functionsimsearch>.
- [142] M. S. Charikar, «Similarity estimation techniques from rounding algorithms,» в *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, 2002.
- [143] R. W. Hamming, «Error detecting and error correcting codes,» *The Bell System Technical Journal*, т. 29, № 2, стр. 147-160, 1950.
- [144] T. Rinsma, «Automatic Library Version Identification, an Exploration of Techniques,» 2017.
- [145] B. H. Bloom, «Space/time trade-offs in hash coding with allowable errors.,» *Communications of the ACM*, 1970.
- [146] «radare2,» [В Интернете]. <https://www.radare.org/r/>.
- [147] В. И. Левенштейн, «Двоичные коды с исправлением выпадений, вставок и замещений символов,» в *Доклады Академий Наук СССР*, 1965.

- [148] J. Koret, «Cosa Nostra - Graph Based Malware Clustering Toolkit.,» 2016.
- [149] «LLVM Language Reference,» LLVM Project, 2003. [В Интернете]. <https://llvm.org/docs/LangRef.html>.
- [150] K. Kennedy, «Use-definition chains with applications,» *Computer Languages*, т. 3, № 3, стр. 163-179, 1978.
- [151] «ANTLR,» [В Интернете]. <https://www.antlr.org/>.
- [152] «ASM,» [В Интернете]. <https://asm.ow2.io/>.
- [153] T. Dullien и S. Porst, «REIL: A platform-independent intermediate representation of disassembled code for static code analysis,» в *CanSecWest Conference*, 2007.
- [154] W. W. Peterson и D. T. Brown, «Cyclic codes for error detection.,» в *Proceedings of the IRE*, 49(1), 1961.
- [155] 3.1.1 Packet format, 802.3-2018 – IEEE Standard for Ethernet. IEEE, 2018.
- [156] G. Chandil и P. Mishra, «Design and Implementation of HDLC Controller by Using Crc-16,» *Computer Science, Engineering*, 2012.
- [157] S. R. Ruckmani и P. Anbalagan, «High Speed cyclic Redundancy Check for USB,» *DSP Journal*, т. 6, № 1, стр. 45–49, 2006.
- [158] A. Israeli и D. G. Feitelson, «The Linux kernel as a case study in software evolution,» *Journal of Systems and Software*, т. 83, № 3, стр. 485-501, March 2010.
- [159] A. Mathur, M. Cao и S. Bhattacharya, «The new ext4 filesystem: current status and future plans,» в *Proceedings of the Linux Symposium*, Ottawa, Ontario Canada, 2007.
- [160] «Linux network subsystem,» [В Интернете]. <https://www.kernel.org/doc/html/latest/networking/index.html>.
- [161] «PPP: CRC-CCITT,» [В Интернете]. [https://github.com/torvalds/linux/blob/master/drivers/net/ppp/ppp\\_async.c](https://github.com/torvalds/linux/blob/master/drivers/net/ppp/ppp_async.c).
- [162] R. N. Williams, «A PAINLESS GUIDE TO CRC ERROR DETECTION ALGORITHMS,» 1993.
- [163] Sunshine, «Understanding and implementing CRC (Cyclic Redundancy Check) calculation,» 2023. [В Интернете]. [http://www.sunshine2k.de/articles/coding/crc/understanding\\_crc.html#updates](http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html#updates).
- [164] «CRC16,» [В Интернете]. <https://www.mikrocontroller.net/attachment/91385/crc16.c>.
- [165] «Carry-less multiplication,» [В Интернете]. [https://en.wikipedia.org/wiki/Carry-less\\_product](https://en.wikipedia.org/wiki/Carry-less_product).
- [166] M. Thamer, «Finite Field Arithmetic (Galois field),» в *Information Theory 4th Class in Communications*.
- [167] V. Gopal, E. Ozturk, J. Guilford, G. Wolrich, W. Feghali, M. Dixon и D. Karakoyunlu, «Fast CRC Computation for Generic Polynomials Using PCLMULQDQ Instruction,» 2009.

- [168] «CoreMark, CRC,» [В Интернете]. <https://github.com/eembc/coremark/tree/main>. [Дата обращения: 09 2024].
- [169] J. C. King, «Symbolic Execution and Program Testing,» *Communications of the ACM*, т. 19, № 7, стр. 385-394, 1976.
- [170] C. Barrett, R. Sebastiani, S. A. Seshi и C. Tinelli, «Chapter 12: Satisfiability Modulo Theories,» в *Handbook of Satisfiability*, IOS Press, 2008, стр. 737-797.
- [171] K. D. Cooper и L. Torczon, «Intermediate Representations,» в *Engineering a Compiler (Third Edition)*, 2023, стр. 221-268.
- [172] A. Canteaut, «Linear Feedback Shift Register,» в *Encyclopedia of Cryptography and Security*, Boston, MA, Springer, 2011, стр. 726–729.
- [173] P. Geremia, «Cyclic Redundancy Check Computation: An Implementation Using the TMS320C54x,» Computer Science, Engineering, 1999.
- [174] J. Merrill, «Generic and gimple: A new tree representation for entire functions,» в *GCC Developers Summit*, 2003.
- [175] «GCC CRC optimization,» Commit hashes: bb46d05a, c5126f0a, 74eb3570, 062ad209, 148e2046, dcc6101c, 4d2b9202, 113e902e, [В Интернете]. <https://github.com/gcc-mirror/gcc>. [Дата обращения: 12 2024].
- [176] «CRC optimization: RISC-V CRC expander,» [В Интернете]. <https://github.com/gcc-mirror/gcc/commit/74eb3570e6fba73b0e2bfce2a14d7696e30b48a8>. [Дата обращения: 12 2024].
- [177] «CRC optimization: AArch64 CRC expander,» [В Интернете]. <https://gcc.gnu.org/pipermail/gcc-patches/2024-November/668250.html>. [Дата обращения: 12 2024].
- [178] «CRC optimization: i386 CRC expander,» [В Интернете]. <https://gcc.gnu.org/pipermail/gcc-patches/2024-May/652615.html>. [Дата обращения: 09 2024].
- [179] «crc8,» [В Интернете]. <https://github.com/spotify/linux/blob/master/drivers/i2c/i2c-core.c>.
- [180] «start\_bunzip,» [В Интернете]. [https://github.com/spotify/kernel-common/blob/9a2e9190b35c15cff5c05d041716ff2adf46f3eb/lib/decompress\\_bunzip2.c#L629](https://github.com/spotify/kernel-common/blob/9a2e9190b35c15cff5c05d041716ff2adf46f3eb/lib/decompress_bunzip2.c#L629).
- [181] «drm\_dp\_msg\_data\_crc4,» [В Интернете]. [https://github.com/spotify/kernel-common/blob/9a2e9190b35c15cff5c05d041716ff2adf46f3eb/drivers/gpu/drm/drm\\_dp\\_mst\\_topology.c#L102](https://github.com/spotify/kernel-common/blob/9a2e9190b35c15cff5c05d041716ff2adf46f3eb/drivers/gpu/drm/drm_dp_mst_topology.c#L102).
- [182] F. Bellard, «QEMU, a Fast and Portable Dynamic Translator,» в *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, USA, 2005.
- [183] «QEMU TCG Plugins,» tests/plugins/insn, [В Интернете]. <https://www.qemu.org/docs/master/about/emulation.html>.
- [184] «Cosine Similarity,» [В Интернете].

- [https://en.wikipedia.org/wiki/Cosine\\_similarity](https://en.wikipedia.org/wiki/Cosine_similarity).
- [185] «GNU Binutils,» [В Интернетe]. <https://www.gnu.org/software/binutils/>.
  - [186] «0ad,» [В Интернетe]. <https://packages.debian.org/search?keywords=0ad>.
  - [187] «PHP,» [В Интернетe]. <https://github.com/php/php-src>.
  - [188] «CMake,» [В Интернетe]. <https://gitlab.kitware.com/cmake/cmake>.
  - [189] «OpenJPEG,» [В Интернетe]. <https://github.com/uclouvain/openjpeg>.
  - [190] «rct,» [В Интернетe]. <https://github.com/Andersbakken/rct>.
  - [191] «Emscripten,» [В Интернетe]. <https://github.com/emscripten-core/emscripten>.
  - [192] «Fldigi,» [В Интернетe]. <https://github.com/w1hkj/fldigi>.
  - [193] «POV-Ray,» [В Интернетe]. <https://github.com/POV-Ray/povray>.
  - [194] «ntopng,» [В Интернетe]. <https://github.com/ntop/ntopng>.
  - [195] «PointCloudLibrary,» [В Интернетe]. <https://github.com/PointCloudLibrary/pcl>.
  - [196] «Kamailio,» [В Интернетe]. <https://github.com/kamailio/kamailio>.
  - [197] «H2O,» [В Интернетe]. <https://github.com/h2o/h2o>.
  - [198] «ITK: The Insight Toolkit,» [В Интернетe]. <https://github.com/InsightSoftwareConsortium/ITK>.
  - [199] «Dragonfly,» [В Интернетe]. <https://github.com/dragonflydb/dragonfly>.
  - [200] «Webdis,» [В Интернетe]. <https://github.com/nicolasff/webdis>.
  - [201] «Defold,» [В Интернетe]. <https://github.com/defold/defold>.
  - [202] «ODrive,» [В Интернетe]. <https://github.com/odriverobotics/ODrive>.

## Список таблиц

Таблица 1. Сравнительный анализ инструментов поиска клонов кода. ....	21
Таблица 2. Сравнительный анализ инструментов сравнения программ. ....	28
Таблица 3. Сравнительный анализ инструментов поиска статически связанных библиотек.....	38
Таблица 4. Поиск клонов с низким сходством в исходном коде.....	57
Таблица 5. Поиск клонов с низким сходством в исполняемом коде. ....	58
Таблица 6. Результаты масштабируемости. ....	59
Таблица 7. Сравнение инструмента FCD с существующими инструментами. ....	60
Таблица 8. Результаты оценки на BigCloneBench. ....	61
Таблица 9. Производительность ЦПИ на архитектуре x86-64. ....	86
Таблица 10. Производительность ЦПИ на архитектуре RISC-V. ....	87
Таблица 11. Производительность ЦПИ на архитектуре AArch64. ....	88
Таблица 12. Сравнение эффективности инструкции csc32 с альтернативными подходами на архитектуре x86-64.....	89
Таблица 13. Сравнение эффективности инструкции csc32 с альтернативными подходами на архитектуре AArch64. ....	89
Таблица 14. Динамическое количество инструкций для функций ЦПИ на архитектуре RISC-V.....	90
Таблица 15. Динамическое количество инструкций для функций ЦПИ на архитектуре AArch64. ....	91
Таблица 16. Результат работы инструмента на разных версиях Openssl. ....	97
Таблица 17. Результат работы инструмента на разных проектах. ....	97
Таблица 18. Результат работы инструмента на Coreutils. ....	99
Таблица 19. Сравнение инструмента BCC с существующими инструментами. ..	101
Таблица 20. Точность и полнота работы инструмента на исполняемых файлах binutils_2.35 (GCC O0). ....	107
Таблица 21. Точность и полнота работы инструмента на исполняемых файлах binutils_2.35 (GCC O2). ....	108



Таблица 22. Точность и полнота работы инструмента на исполняемых файлах coreutils_8.30 (GCC O0). .....	108
Таблица 23. Результат работы инструмента. ....	109
Таблица 24. Сообщенные и устраненные проблемы.....	113

## Список рисунков

Рисунок 1. Пример типов клонов исходного кода. ....	14
Рисунок 2. Пример типов клонов исполняемого кода. ....	15
Рисунок 3. Схема нахождения клонов фрагмента кода. ....	41
Рисунок 4. Поиск подграфов графов зависимостей программы. ....	43
Рисунок 5. Пример сопоставления вершин через зависимости управления. ....	47
Рисунок 6. Сопоставление инструкций базовых блоков. ....	48
Рисунок 7. Сопоставление вершин соседних базовых блоков сопоставленной пары.....	49
Рисунок 8. Пример сопоставления вершин через зависимости данных ....	50
Рисунок 9. Пример ГЗП на основе LLVM ПП.....	52
Рисунок 10. Пример ГЗП на основе байт-кода Java. ....	53
Рисунок 11. Пример ГЗП на основе ANTLR. ....	53
Рисунок 12. Пример ГЗП на основе REIL представления. ....	54
Рисунок 13. Пример побитовой реализации ЦПИ на языке C.....	65
Рисунок 14. Пример табличной реализации ЦПИ на языке C.....	66
Рисунок 15. Пример реализации ЦПИ с обратным порядком бит. ....	67
Рисунок 16. Распознавание ЦПИ. ....	72
Рисунок 17. РСЛОС с полиномом $x^8+x^2+x+1$ . ....	76
Рисунок 18. РСЛОС с полиномом $x^8+x^2+x+1$ без операции XOR с битами данных. ....	76
Рисунок 19. Пример структуры РСЛОС. ....	78
Рисунок 20. Реализация ЦПИ на основе таблиц на языке C.....	80
Рисунок 21. Реализация ЦПИ на основе умножения без переноса на языке C. ....	80
Рисунок 22. Процесс оптимизации ЦПИ. ....	81
Рисунок 23. Среднее улучшение по сравнению побитовой реализации ЦПИ. ....	92
Рисунок 24. Схема сравнения двух программ. ....	93
Рисунок 25. Схема идентификации статически связанных библиотек. ....	105
Рисунок 26. Схема сравнения исполняемого файла с одной библиотекой.....	106

Рисунок 27. Процесс нахождения клонов известных уязвимостей..... 111

# Приложение

СВИДЕТЕЛЬСТВА О ГОСУДАРСТВЕННОЙ РЕГИСТРАЦИИ ПРОГРАММ ДЛЯ ЭВМ

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**

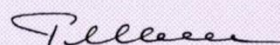
Дата поступления **23 октября 2020 г.**

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): <a href="#">2021665076</a>	Авторы: <b>Курмангалеев Шамиль Фаимович (RU), Асланян Айк Каренович (AM), Арутюнян Мариам Сероповна (AM), Оганесян Рипсима Ашотовна (AM), Варданын Ваагн Геворгович (AM), Саргсян Севак Сеникович (AM)</b>
Дата регистрации: <b>17.09.2021</b>	
Номер и дата поступления заявки: <b>2021663858 07.09.2021</b>	
Дата публикации: <a href="#">17.09.2021</a>	
Контактные реквизиты: Тел.: +7(903)700-79-86; e-mail: <a href="mailto:m.kalugin@ispras.ru">m.kalugin@ispras.ru</a>	Правообладатель: <b>Федеральное государственное бюджетное учреждение науки Институт системного программирования им. В.П. Иванникова Российской академии наук (RU)</b>

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