Федеральное государственное бюджетное учреждение науки
Институт системного программирования Российской академии наук

На правах рукописи

Саргсян Севак Сеникович

Методы поиска клонов кода и семантических ошибок на основе семантического анализа программы

05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Диссертация на соискание ученой степени кандидата физико-математических наук

Научный руководитель:
доктор физико-математических наук,
академик РАН
Иванников Виктор Петрович

Москва – 2016

Содержание

Введение

Актуальность

Широкое распространение свободного программного обеспечения (ПО) привело к частому использованию готовых *фрагментов* исходного кода при разработке нового ПО. Разработчики могут использовать как интернет-ресурсы, так и код, написанный ими самими или их коллегами. Согласно результатам исследований, программы могут содержать до двадцати процентов клонов кода (скопированных фрагментов). Бесконтрольное клонирование может привести к *увеличению размера исходного и бинарного кода программы*, *возникновению семантических ошибок, усложнению поддержки ПО и др.* Известно, что проекты FreeBSD и Linux содержат сотни ошибок, связанных с клонированием кода.

Для разработки высококачественного ПО необходимо находить клоны кода и связанные с ними семантические ошибки в проектах, содержащих десятки миллионов строк исходного кода, при этом обеспечив высокую точность (приемлемый уровень ложных срабатываний) и масштабируемость (приемлемое время работы). Для адаптации скопированного фрагмента кода разработчик может отредактировать его. Иногда отредактированный код может настолько отличаться от оригинала, что определить, откуда его скопировали, практически невозможно. В литературе различаются три основных типа клонов. Клоны типа T1 возникают, когда при вставке клонированного фрагмента кода адаптации к контексту не производится. Клоны типа T2 возникают, когда адаптация сводится к замене идентификаторов. В более сложных случаях, когда адаптация требует не только замены идентификаторов, но и других изменений, возникают клоны типа T3. Из известных *пяти подходов* к поиску клонов кода *(текстового, лексического, метрического, синтаксического и семантического)* только последний позволяет

обнаружить клоны типа Т3 с достаточно высокой точностью. Но большая часть существующих инструментов, базирующихся на семантическом подходе обладают большой вычислительной сложностью, что делает их немасштабируемыми и не позволяет использовать в реальных проектах.

Часть найденных клонированных фрагментов кода может быть не полностью адаптирована к контексту, в который они были вставлены, что приводит к возникновению семантических ошибок (фрагмент вычисляет не то, что ожидает разработчик). Существующие инструменты поиска семантических ошибок могут, в основном, обнаруживать ошибки только в клонах типов Т1 и Т2. Они сначала находят клоны кода путем лексического или синтаксического анализа, после чего производится дополнительный анализ для обнаружения допущенных ошибок. При этом инструменты на основе лексического анализа имеют высокий уровень ложных срабатываний (т.е. низкую точность), поскольку не учитывают контекст программы. Инструменты, использующие синтаксический анализ, не могут найти все семантические ошибки, поскольку после переименования переменных может существенно измениться абстрактное синтаксическое дерево.

Таким образом, на сегодняшний день не существует ни инструментов поиска клонов кода, которые обеспечивали бы требуемый уровень точности и масштабировались до десятков миллионов строк исходного кода, ни достаточно точных инструментов поиска семантических ошибок в клонах. Тема диссертации является актуальной.

Целью диссертационной работы является разработка методов и программных средств для поиска клонов кода (в том числе, типа Т3) и семантических ошибок, возникающих при некорректной адаптации скопированных фрагментов кода. Разрабатываемые методы должны обладать высокой точностью и быть масштабируемы, то есть применимы для анализа десятков миллион строк кода.

Для достижения поставленной цели были сформулированы и решены следующие **задачи:**

1. Выявить недостатки существующих подходов к поиску клонов кода и семантических ошибок, возникающих при неправильном использовании скопированных фрагментов кода.
2. Разработать и реализовать методы поиска клонов кода на основе семантического анализа программы, обладающих высокой точностью и масштабируемых до десятков миллионов строк исходного кода.
3. Разработать и реализовать высокоточные методы нахождения семантических ошибок в клонированных участках кода путем применения комбинированного лексического и семантического анализа.

Основные положения, выносимые на защиту и обладающие научной новизной:

1. Новый четырехфазный метод поиска клонов кода на основе семантического подхода, который масштабируется до десятков миллионов строк кода и обеспечивает высокий уровень (более 90%) истинных срабатываний.

   Набор новых алгоритмов, обеспечивающих выполнение фаз метода:
   - разделение графа зависимостей программы (ГЗП) на подграфы требуемого размера;
   - фильтрация пар подграфов ГЗП с помощью линейных алгоритмов;
   - поиск схожих пар подграфов максимального размера путем расширения пар подграфов за счет идентичных смежных вершин (слайсинг).
   - фильтрация ложных клонов.
2. Два новых метода поиска клонов типов T1 и T2, которые масштабируются до десятков миллионов строк кода и обеспечивают высокий уровень (более 95%) истинных срабатываний:
   - метод, использующий преобразование ГЗП в дерево с последующим поиском изоморфных поддеревьев в преобразованных ГЗП;
   - метод, использующий новую метрику вершин ГЗП.
3. Высокоточный комбинированный метод определения семантических ошибок в клонах типов T1 и T2, использующий лексический и семантический анализ.

4. Архитектура инструмента поиска клонов кода для языков программирования C, C++ и JavaScript; в том числе подсистемы анализа точности реализованных алгоритмов поиска клонов что позволяет улучшать указанные алгоритмы.
5. Реализованы масштабируемый инструмент поиска клонов кода на базе компиляторной инфраструктуры LLVM; генератор ГЗП, на основе JIT-компилятора V8, что позволило применить поиск клонов кода для языка JavaScript; инструмент поиска семантических ошибок. Экспериментальные результаты анализа больших проектов, таких как ядро ОС Linux и ОС Android, подтверждают эффективность реализованных методов.

Теоретическая и практическая значимость

Предложены новые методы поиска клонов кода на основе семантического анализа программы, которые масштабируются до десятков миллионов строк исходного кода и находят клоны с высокой точностью. Предложен комбинированный метод поиска семантических ошибок, возникающих при некорректной адаптации скопированного фрагмента кода. Предложен архитектура инструмент поиска клонов кода на базе компиляторной инфраструктуры LLVM.

Реализованный инструмент обеспечивает поиск клонов кода для всех языков программирования, которые поддерживают трансляцию в промежуточное представление LLVM, в частности C и C++. Архитектура инструмента позволяет добавлять поддержку новых языков, для этого достаточно обеспечить генерацию ГЗП для соответствующего языка.

Реализованные инструменты внедрены в научно-исследовательские и учебные проекты Института системного программирования РАН. Часть разработанных программных средств внедрена в программные продукты коммерческих компаний.

Апробация работы

Основные результаты были представлены в докладах на следующих конференциях:

- Международная научная конференция студентов, аспирантов и молодых учёных «Ломоносов-2014», 7 – 11 апреля 2014 г., Москва;
- 57 научная конференция МФТИ, 24-29 ноября 2014 г., Долгопрудный;
- FOSDEM-2015, 31 января – 1 февраля 2015 г., Брюссель, Бельгия;
- CSIT-2015, 28 сентября – 2 октября 2015 г., Армения, Ереван;
- Открытая конференция по компиляторным технологиям, 2 – 4 декабря 2015 г., Москва.

Публикации

По теме диссертации опубликовано 7 работ, 4 из которых входят в перечень рецензируемых научных изданий ВАК РФ [1, 3, 4, 7]. Список работ приведен в конце работы. В совместной работе [1] личный вклад автора состоит в разработке и реализации преобразований переплетений функций и разбиение целочисленных констант, которые используются для автоматической генерации клонов кода. В совместных работах [2, 3, 5, 6, 7] автору принадлежат описанные в них методы поиска клонов кода, архитектура инструмента поиска клонов кода, обзорные разделы.

Личный вклад автора

Все представленные в диссертации результаты получены лично автором.

Структура и объем работы. Диссертация состоит из введения, 5 глав и заключения. Работа изложена на 103 страницах. Список источников насчитывает 98 наименование. Диссертация содержит 3 таблицы и 49 рисунков.

Глава 1
Обзор работ

1.1. Типы клонов кода и методы их поиска

В статье [8] рассматривается три типа клонов кода и пять основных подходов к их поиску. Каждый подход имеет свои преимущества и ограничения. В зависимости от решаемой задачи и предъявляемых требований, выбирается конкретный подход поиска клонов кода, который позволяет решить задачу оптимальным образом. Существуют комбинированные методы поиска клонов кода, состоящие в использовании разных подходов на разных фазах поиска.

1.1.1. Типы клонов кода

По определению фрагментом кода называется произвольная последовательность строк. Клоны кода типов T1, T2 и T3 определяются следующим образом (см. рисунок 1.1):

T1. Фрагменты кода, которые могут отличаться только пробелами, комментариями и форматированием кода;

T2. Все клоны типа T1, а также фрагменты кода, которые могут также различаться: именами переменных; типами переменных; значениями переменных и констант;

T3. Все клоны типа T2, а также фрагменты кода, в которых могут быть добавлены или удалены инструкции и переменные.

Оригинальный код	**Клон типа T1**

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

Клон типа T2	**Клон типа T3**

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

Рисунок 1.1. Примеры трех типов клонов кода.

Степень схожести фрагментов кода F_1 и F_2 обратно пропорциональна количеству операций редактирования фрагмента F_1 (переименование переменных, удаление, добавление и изменение инструкций), чтобы получился фрагмент F_2.

Для поиска клонов кода используется пять основных подходов [8]: текстовый, лексический, синтаксический, семантический и метрический. Разработаны также инструменты поиска клонов, в которых применяются комбинации нескольких подходов.

1.1.2. Текстовый подход

Методы на основе текстового подхода рассматривают исходный код программы как последовательность строк. Для поиска схожих фрагментов кода используются три основных техники: построчное сравнение, сравнение подстрок, составляющих так называемый отпечаток кода, и сравнение преобразованных с помощью языка TXL фрагментов кода.

Построчное сравнение

Сначала все файлы проекта объединются в один файл. Затем выполняется попарное сравнение всех строк объединенного файла с сохранением результатов

сравнения в матрице *M* (элемент матрицы *M[i][j]=1* если в объединенном файле строка *i* равна строке *j*, в противном случае *M[i][j]=0*).

Инструмент Duploc [9] считает клоном последовательность совпадающих строк, найденных на основе полученной матрицы *M*. Поэтому он может найти только клоны типа T1.

Инструмент DuDe [10], используя матрицу *M*, расширяет полученные клоны. Если для пары клонов (P_1,P_2) существует другая пара клонов (T_1,T_2) такая, что расстояния между парами фрагментов P_1, T_1 и P_2, T_2 достаточно малы, то каждая из пар P_1, T_1 и P_2, T_2 объединяется, причем во фрагмент, полученный объединением P_i и T_i добавляются строки кода, расположенные между P_i и T_i (i = 1, 2). Такой подход позволяет находить больше клонов, чем Duploc. Отметим, что инструмент DuDe позволяте находить даже клоны типа T3, правда, с невысокой точностью.

Сравнение подстрок фрагментов

Johnson [11, 12] вводит понятие отпечатка (fingerprint) фрагмента кода и вместо построчного сравнения всех строк фрагментов кода сравнивает их отпечатки. Такой подход позволяет быстрее производить анализ, так как сравниваются только подмножества строк исходного кода. Следует отметить, что при таком подходе увеличивается уровень ложных срабатываний, посклоьку во фрагменте кода могут находиться строки кода, которые не были выбраны для сравнения.

Тем не менее, инструмент [11, 12] был применен для анализа двух разных релизов компилятора GCC и достаточно точно показал историю изменения файлов [13].

Инструмент sif [14] находит файлы, имеющие общие части. Для этого он сравнивает выбранные из этих файлов подмножества строк (опечатки файлов). На практике инструмент показал, что может найти как точно скопированные файлы, так и файлы, у которых общая часть составляет всего 25%.

Использование языка TXL

Язык TXL [15] по структуре и назначению напоминает РЕФАЛ. Программа на TXL представляет собой описание грамматических правил в нормальной форме Бэкуса—Наура (BNF) и описание правил переписывания термов. В инструментах поиска клонов TXL используется для нормализации фрагментов исходного кода (нормализация состоит в исключении несущественных различий во фрагментах кода).

Инструмент NiCad [16, 17] сначала нормализует исходный код программы с использованием TXL. После этого форматирует преобразованный код таким образом, чтобы потенциальные изменения разработчика после клонирования размещались в отдельных строках. После этого код разбивается на фрагменты, каждый из которых рассматривается как потенциальный клон. Все фрагменты попарно построчно сравниваются для нахождения максимально большого общего множества совпадающих строк, которое и считается клоном.

С использованием NiCad был проведен анализ клонов кода в ряде проектов с открытым исходным кодом, написанных на языках C, Java и Python [18, 19].

Инструмент NiCad имеет ряд расширений, которые позволяют искать клоны кода для других языков, в частности для языка описания веб-сервисов WSDL [20, 21] и даже для графической среды моделирования Simulink [22, 23, 24], в которой программы предсавляются в виде блок-схем. Последняя версия инструмента NiCad [25] масштабируема.

1.1.3. Лексический подход

С помощью лексического анализа из исходного кода можно получить последовательность токенов. Алгоритмы поиска клонов кода ищут совпадающие подпоследовательности токенов. Исходный код, соответствующий совпадающим подпоследовательностям токенов, считается клоном.

Инструмент Dup [26, 27, 28] из последовательности токенов строит суффиксное дерево, на основе которого находит совпадающие максимальные подпоследовательности токенов. Он может находить только клоны типов T1 и T2.

Одним из самых популярных инструментов поиска клонов кода является CCFinder [29, 30], который использует суффиксное дерево для поиска идентичных подпоследовательностей токенов. Если размер анализируемого файла настолько велик, что файл не помещается в память, инструмент анализирует такой файл по частям. Благодаря метрикам оценки схожести найденных клонов, CCFinder дает возможность нахождения фрагментов кода, которые копировались чаще всего. Существует распределенная реализация инструмента D-CCFinder [31], которая предназначена для сверхбольших проектов с десятками миллионов строк исходного кода. CCFinder находит все клоны типов T1 и T2, а также некоторые клоны типа T3.

Инструмент CCFinder был использован для анализа стандартной библиотеки STL [32] и ряда проектов с открытым исходным кодом [33, 34]. CCFinder также был модифицирован для поиска схожих файлов в разных релизах проектов [35].

Инструмент Eunjong [36], используя CCFinder, находит клоны кода. Для найденных клонов он применяет специальные метрики, позволяющие определить кандидатов на рефакторинг.

Инструмент CloneDetective [37] ищет клоны кода используя суффиксное дерево. Он позволяет определять скопированные фрагменты кода в которых содержатся ошибки.

CP-Miner [38] использует для поиска клонов методы добычи данных (data mining). Он определяет повторяющиеся подпоследовательности токенов [39] как клоны кода. Инструмент находит ошибки, связанные с некорректной адаптацией клонированного фрагмента кода. CP-Miner находит только клоны типов T1 и T2. По сравнению с CCFinder он работает медленее.

Инструменты SABSM [40], RTF [41] и SHINOBI [42] используют суффиксный массив [43] для поиска идентичних подпоследовательностей. SHINOBI состоит из двух частей: серверная часть инструмента хранится в репозитории исходного кода проекта; клиентская часть находится в IDE разработчика и работает в режиме реального времени.

Инструменты на основе лексического анализа могут находить все клоны типов T1 и T2 с высокой точностью. Из них CCFinder маштабируется до десятков милионов строк исходного кода. Таким образом, для задач, в которых требуется искать только клоны типов T1 и T2, наилучшим является лексический подход. Клоны типа T3 обнаруживаются инструментами, базирующимися на этом подходе с настолько низкой точностью, что они неприменимы для поиска таких клонов.

1.1.4. Синтаксический подход

Поиск клонов кода осуществляется на основе абстрактного синтаксического дерево (АСД), иногда используется АСД в место дерево разбора (ДР). Эти структуры строятся синтаксическим анализатором, который открыто доступно в многих компиляторах.

Инструмент Yang [44] определяет синтаксическое отличие двух версий кода. Для обеих версий кода он строит АСД, после чего путем применения методов динамического программирования находит пары изоморфных поддеревьев. Вершины, не входящие в такие поддеревья, считаются синтаксическим отличием двух функций. Преимущество этого инструмента по сравнению с linux diff состоит в том, что он показывает не отличающиеся строки фрагментов кода, а конкретные инструкции.

Инструменты Falke et al [45] и Tairas et al [46] производят поиск изоморфных поддеревьев АСД с использованием суффиксного дерева. Инструмент Falke сначала находит клоны типов T1 и T2, после чего находит клоны типа T3, объединяя совпадающие поддеревья АСД. Инструмент Tairas реализован как расширение компиляторной инфраструктуры Microsoft Phoenix [47]. Эти инструменты имеют высокую точность только при поиске клонов кода типов T1 и T2. Они могут пропускать клоны типа T2, если при адаптации скопированного фрагмента не все переменные были переименованы корректно, так как в этом случае структура АСД может измениться.

Широкое применение имеет инструмент DECKARD [48], который работает не над АСД, а над деревом разбора (ДР). Для поддеревьев ДР вычисляются характеристические векторы, после чего производится кластеризация векторов на основе Евклидова расстояния. Клонами считаются поддеревья ДР, для которых Евклидово расстояние соответствующих характеристических векторов достаточно мало. Инструмент DECKARD позволяет находить группы клонов кода. Инструмент поддерживает языки C и Java. По сравнению с Yang et al [44], Falke et al [45] и Tairas et al [46] DECKARD лучше находит клоны типа T3, поскольку вместо поиска изоморфных поддеревьев сравниваются характеристические векторы поддеревьев ДР.

В работе Gray et al [49] DECKARD используется для определения различий между различными версиями нескольких проектов с открытым исходным кодом. В работе Juergens et al [50] DECKARD использован для нахождения семантически схожих фрагментов кода, полученных при независимой реализации одинаковых функциональностей.

Инструмент ClemanX [51, 52] находит клоны кода на основе сравнения характеристических векторов поддеревьев АСД. Инструмент дает возможность следить за клонами в процессе разработки. Каждый раз, когда исходный код обновляется, производится обновление пар найденных клонов. Для каждого добавленного фрагмента кода строится АСД и производится поиск изоморфных поддеревьев. При удалении конкретного фрагмента кода строится АСД и удаляются все пары клонов, которые содержат построенное АСД. Преимущество данного подхода по сравнению с DECKARD заключатся в том, что инструмент работает гораздо быстрее, поскольку поиск клонов кода производится только для обновленного участка кода.

Инструменты Lee et al [53], Brown et al [54] и Clone Digger [55] производят классификацию поддеревьев АСД и заменяют поддеревья одинакового класса вершинами, соответствующими этому классу. Такое преобразование АСД они называют анти-унификацией. Пример анти-унификации: приведение символьных выражений «a[1]» и «a[x+1]» к виду «a[?]». После анти-унификации производится

поиск изоморфных поддеревьев АСД. Для генерации АСД используется синтаксический анализатор CIL. Инструмент Lee реализован для языка C, инструмент Brown – для языка Haskell, инструмент Clone Digger – для языков Python и Java. Эти три инструмента позволяют находить клоны типа T3, в которых различающиеся инструкции становятся одинаковыми после анти-унификации. Инструменты Lee, Brown и Clone Digger находят меньше клонов, чем DECKARD, но имеют более высокую точность.

Инструмент Wahler et al [56] представляет АСД в формате XML. Для поиска совпадающих подпоследовательностей элементов XML используется анализ часто встречающихся элементов, применяемый при добыче данных. Инструмент позволяет найти клоны типов T1 и T2.

Инструмент Chilowicz et al [57] для каждой вершины АСД вычисляет отпечаток таким образом, что отпечаток вершины может быть однозначно получен из отпечатков дочерних узлов. Отпечаток каждой вершины представляет собой поддерево, размер которого больше размера минимального клона, отпечатки сохраняется в базе данных. Точно совпадающие поддеревья находятся с помощью запросов к базе отпечатков.

Инструмент C2D2 [58] позволяет находить клоны кода в проектах, содержащих программные файлы, написанные либо на языке C#, либо на языке Basic. C2D2 использует API CodeDOM из Visual Studio.NET [59], позволяющий получать граф программы, который отображает логические связи между ее структурами. C2D2 преобразует этот граф в дерево, на котором производит поиск изомофных поддеревьев. Инструмент интересен тем, что позволяет производить поиск клонов кода в пределах двух языков.

1.1.5. Семантический подход

Поиск клонов кода осуществляется на основе графа зависимостей программы (ГЗП). ГЗП – направленный граф, объединяющий информацию о потоке данных и потоке управления. Вершины ГЗП помечены кодами операций, а ребра – типами зависимосией (по данным, или по управлению). В ГЗП хранится

вся информация о семантике и структуре функции, что позволяет более точно ее анализировать. Алгоритмы, работающие на основе семантического подхода, осуществляют поиск *схожих*, или изоморфных подграфов в каждой паре ГЗП.

Два ГЗП называются *схожими*, если оба связны и множества типов ребер у обоих ГЗП совпадают. Тип ребра *(u, v)* представляет собой тройку *(U, V, T),* где *U* и *V* – метки вершин *u* и *v*, а *T* – метка ребра *(u, v)*.

Степень схожести пары ГЗП G,H равна $\frac{Nodes(U)}{\min(Nodes(G),Nodes(H))}$, где U — изоморфный подграф графов G,H имеющий максимальный размер, $Nodes(X)$ — количество вершин графа X .

Horwitz [60] использует ГЗП для сравнения двух версий одной программы.

Krinke [61] определяет клоны кода как схожие подграфы ГЗП. Для поиска схожих подграфов выбираются идентичные вершины ГЗП, которые представляют предикаты программы. Эти вершины рассматриваются как начальные схожие подграфы. Начальные подграфы расширяются путем добавления новых идентичных смежных вершин. Метод реализован для программ, написанных на ANSI-C.

Komondoor et al [62] использует для поиска изоморфных подграфов ГЗП технику обратного слайсинга. После того как изоморфные подграфы найдены, они объединяются в группы.

Инструмент Higo et al [63] хеширует вершины ГЗП (на основе исходного кода соответствующего данной вершине), после чего выбирается пара вершин с одинаковыми хешами. Прямой и обратный слайсингы запускаются для выбранной пары вершин. В ходе слайсинга (slicing) новые вершины добавляются в изоморфные подмножества вершин, если они имеют одинаковые хеши.

Horwitz [60], Krinke [61], Komondoor et al [62] и Higo et al [63] находят все три типа клонов кода с высокой точностью, но имеют сложность $O(N^3)$, где N – число вершин ГЗП, что не позволяет использовать эти инструменты для анализа программ, содержащих миллионы строк кода.

Инструмент GPLAG [64] реализован для поиска плагиата в исходном коде программы. Для этого попарно проверяются на изоморфизм ГЗП программ, имеющих достаточно большой размер. Для масштабируемости сначала применяются специальные алгоритмы, сложность которых меньше чем у алгоритма поиска максимальных изоморфных подграфов, определяющие, могут ли пары ГЗП иметь изоморфные подграфы желаемого размера. GPLAG находит все три типа клонов кода. Инструмент GPLAG работает быстрее, чем инструменты [60], [61], [62] и [63], благодаря применению фильтров. Несмотря на это GPLAG не может анализировать проекты с миллионами строк исходного кода.

Gabel et al [65] преобразует ГЗП в дерево, после чего применяет инструмент DECKARD для поиска изоморфных поддеревьев. Масштабируемость достигается за счет потери точности найденных клонов кода (инструмент находит не все клоны Т3 и имеет высокий уровень ложных срабатываний). Gabel может анализировать ядро Linux за приемлемое время.

Инструмент CBCD [66] находит такие клоны кода, которые могут содержать семантические ошибки. В CBCD имеется база фрагментов кода, содержащих семантические ошибки, полученные анализом репозиторий ПО, имеющегося в открытом доступе. Для поиска потенциальных ошибок в проекте CBCD ищет фрагменты кода, которые есть в собранной базе. Для этого используется ГЗП.

Исследование существующих инструментов поиска клонов кода на основе семантического анализа показали, что они могут находить все три типов клонов кода с высокой точностью, но, как правило, плохо масштабируются.

1.1.6. Метрический подход

Алгоритмы вычисляют ряд метрик для фрагментов кода и сравнивают не фрагменты кода, а векторы полученных метрик. Обычно метрики вычисляются для АСД или ГЗП исследуемого фрагмента.

Mayrand et al [67] строит по исходному коду АСД, для которого вычисляет четыре метрики на основе: (1) имен переменных программы; (2) размещения

исходного кода в файле; (3) использованных инструкций; (4) потока управления программы.

Patenaude et al [68] находит клоны и фрагменты кода, которые могут иметь ошибки конструирования. Для сравнения фрагментов кода вычисляются четыре метрики: (1) вычисляется размер класса, количество и типы методов класса; (2) отражает связи и иерархию данного класса в системе; (3) вычисляет сложность методов класса, при этом учитывается размер методов и сложность McCabe [69] (вычисляется на основе графа потока управления); (4) отражает иерархию наследования. Инструмент работает для программ, написанных на языке Java.

Kontogiannis et al [70, 71] для сравнения используются пять метрик: (1) вычисляет количество вызовов функций; (2) вычисляет количество входных и выходных переменных для вызовов функций; (3) учитывает сложность McCabe [69]; (4) метрика Albrecht [72]; (5) метрика Henry-Kafura [73] (вычисляется на основе графа потока данных).

Kodhai et al [74] сначала объединяет все файлы проекта в один, после чего производит стандартизацию текста программы. Стандартизация подразумевает удаление комментариев, пробелов и команд препроцессора. После чего для каждой функции вычисляются семь метрик, на основе которых производится проверка пар функций на клон. Метрики для функции включают количество: (1) эффективных строк исходного кода (количество строк без комментариев и пустых строк); (2) аргументов функции; (3) вызовов функций; (4) локальных переменных; (5) инструкций условии; (6) инструкций циклов; (7) инструкций возвратов из функции. Метод реализован для языка C.

В работе Li et al [75] вектор метрик фрагментов кода определяется таким образом, что множество этих векторов является метрическим пространством. Чем больше схожесть фрагментов кода, тем меньше расстояние между соответствующими векторами метрик. Определяется два основных типа метрик: (1) описывающий структуру программы (иерархию классов); (2) описывающий разные типы данных, использованные во фрагменте.

Все инструменты на основе метрик имеют высокую производительность и масщтабируются до миллионов строк исходного кода, но у них низкая точность. Клоны типов T1 и T2 эти методы находят лучще чем клоны типа T3.

1.1.7. Комбинированные подходы

Для поиска клонов в исходном коде Sutton et al [76] предлагает применять эволюционный алгоритм (ЭА). Исходный код разделяется на фрагменты, каждый из которых считается отдельной группой клонов. Задача ЭА заключается в минимизации количества групп клонов путем объединения схожих групп. В ходе мутации изменяется размер фрагментов кода с целью увеличения количества клонов в каждой группе, одновременно сокращается количество групп.

В работе Maeda [77] изложен метод поиска клонов кода для языков Java, C# и Ruby с использованием XML представления программы под названием PALEX, построение которого происходит на первом этапе анализа. PALEX — представление типов операций, сгенерированное лексическим анализатором. Существует три основных типа операций: сдвиг, редукция и чтение лексемы. На втором этапе строится суффиксное дерево на основе PALEX-представления. Клоны кода определяются как идентичные последовательности операций.

Алгоритм поиска клонов кода, описанный в работе Chilowicz et al [78], направлен на поиск схожих функций. Для каждой функции, после лексического анализа, получают последовательность лексем. Совпадающие участки разных функций определяются с помощью суффиксного дерева. На основе полученных совпадающих последовательностей лексем каждая функция разделяется на подфункции (факторизуется). После факторизации строится граф вызовов программы. С помощью трех специальных метрик определяется схожесть построенных графов.

Инструмент Clone Miner [79], используя RTF[80], получает группы последовательностей идентичных лексем, а затем производит дополнительный анализ для построения множества схожих методов и файлов.

В работе Hummel et al [81] описан алгоритм поиска схожих фрагментов кода. Входной файл преобразуется в последовательность лексем. Затем производится поиск совпадающих подпоследовательностей. Найденные клоны (идентичные подпоследовательности) индексируются таким образом, что поиск клонов кода для добавляемого файла производится быстро. Метод применим только для поиска клонов типа T1 и T2. Его можно эффективно применять для динамически меняющихся проектов, которые находятся в стадии разработки.

1.1.8. Сравнение подходов

Из существующих пяти основных подходов к поиску клонов кода текстовый, лексический, синтаксический и метрический не могут достаточно точно находить клоны типа T3. Текстовый и лексический подходы не могут находить клоны типа T3, поскольку такие клоны различаются как исходным кодом, так и последовательностью лексем. Синтаксический подход не находит клоны типа T3, потому что при удалении или добавленнии инструкций может измениться структура АСД. Метрический подход имеет низкую точность при поиске всех трех типов клонов кода. Только методы на основе семантического анализа умеют находить все клоны с достаточно высокой точностью. Но, как правило, эти методы имеют большую вычислительную сложность и не могут быть применимы для анализа десятков миллион строк исходного кода. Следовательно, для создания высокоточного инструмента поиска клонов кода необходимо снизить вычислительную сложность методов на основе семантического подхода.

1.2. Методы поиска семантических ошибок.

Анализ проектов с открытым исходным кодом показал, что множество семантических ошибок возникает из-за некорректно адаптированных скопированных фрагментов кода. Репозитории проектов FreeBSD и Linux [82], по данным на 2013, содержали более 113 и 182 исправлений подобных ошибок

соответственно. Для поиска ошибок, возникающих из-за неправильной адаптации скопированного кода, как правило, используются методы, основанные либо на лексическом, либо на синтаксическом анализе.

Методы, основанные на лексическом подходе, преобразуют исходный код в последовательность токенов и ищут идентичные последовательности токенов — клонов кода. Затем проверяется корректность переименования переменных. Недостаток такого подхода заключается в большом количестве ложных срабатываний, так как не учитывается контекст участка программы вокруг потенциального клона.

Инструмент CP-miner [38] основан на добыче данных (data mining) и используется для языков C/C++. Он преобразует исходный код в последовательность токенов, в которой находит повторяющиеся подпоследовательности токенов, используя технику анализа часто встречающихся элементов [83, 39]. В найденных подпоследовательностях (клонах) рассматриваются переменные, которые используются больше одного раза. Инструмент проверяет правильность переименования таких переменных.

Методы, основанные на синтаксическом подходе, преобразуют исходный код в АСД и ищут схожие поддеревья. Затем производится анализ найденных поддеревьев для выявления возможных ошибок. Большинство известных инструментов использует репозиторий программы, что является дополнительным ограничением. Каждое изменение в репозитории анализируется, рассматривается его влияние на АСД каждой функции и производится поиск дефектов. Недостаток такого подхода заключается в том, что некорректно переименованные переменные влияют на структуру АСД.

Инструмент CReN [84] встроен в IDE Eclipse и используется для программ, написанных на языке Java. При копировании и вставке фрагментов кода контролирует, чтобы производилось корректное переименование переменных. При копировании фрагмента кода строится его АСД. Когда производится вставка и модификация, инструмент проверяет, чтобы АСД модифицированного фрагмента был изоморфен АСД оригинального фрагмента.

MPAnalyzer [85] анализирует репозиторий проекта для определения шаблонов изменения фрагментов кода. Если изменениям фрагмента кода не соответствует какой-либо шаблон, тогда в данном фрагменте содержится ошибка.

Jiang et al [86], используя DECKARD [48], находит клоны кода на АСД. Для поиска ошибок сравниваются поддеревья АСД, соответствующие клонированным фрагментам кода. Jiang может находить неправильные условия в инструкциях ветвления (условия if), пропущенные проверки (например, проверка объектов на null) и некорректно переименованные переменные.

DejaVu [87] — параллельная и распределенная система поиска семантических ошибок. В ней реализован алгоритм, используемый в DECKARD. Экспериментальный запуск инструмента на 75 миллионах строк коммерческого кода выявил 8000 семантических ошибок, из которых около 2000 являются истинными.

Метод, предложенный в работе Ray et al [82], работает на основе анализа репозитория проекта. Для каждой функции, в которой производились изменение, строится АСД до и после изменения. Для выявления ошибок сравниваются построенные АСД деревья. Для выявления ошибки применяется стандартный анализ потока данных и потока управления. В работе [82] приведена классификация семантических ошибок, возникающих при разработке и портировании кода:

Несоответствие потока управления (Inconsistent control flow - ICF): При некорректной модификации или портировании фрагмента кода не учитывается контекст потока управления. На рисунке 1.2 показан пример ошибки. Цикл, выделенный серым цветом, является лишним, в результате чего оператор «continue» работает для данного цикла. Такие ошибки могут приводить к неожиданным переходам по управлению во время работы программы.

FreeBSD commit: src/sys/kern/sched_4bsd.c
Log: Fix a copy-paste bug in NON-KSE case.
Версия 1.90, Дата: 2006/11/14
Автор: davidxu

```
FOREACH_KSEGRP_IN_PROC(p, kg) {
  awake = 0;
  FOREACH_THREAD_IN_GROUP(kg, td) {
          ...
+       if (ke->ke_cpticks == 0)
+           continue;
        ...
+       if(FSHIFT >= CCPU_SHIFT) {
+           ke->ke_pctcpu += (realstathz == 100)
            ……
+ ... } ... } ... }
```

```
FOREACH_THREAD_IN_PROC(p, td) {
    awake = 0;
    ...
+   if (ke->ke_cpticks ==!= 0)
+       continue;
    {
      ...
+     if(FSHIFT >= CCPU_SHIFT) {
+         ke->ke_pctcpu += (realstathz == 100)
       ….
+ ... } ... }
```

Рисунок 1.2. Несоответствие потока управления.

Некорректное переименование (Inconsistent renaming - IR): в скопированном фрагменте кода не все переменные корректно переименованы. Переменная «bp» не везде была обновлена на «rabp» (рисунок 1.3). Такие ошибки могут приводить к неожиданным переходам по управлению и доступу к несуществующей или недоступной памяти.

FreeBSD commit: src/sys/kern/vfs_bio.c
Log: Fix cut&paste bug which would result in a panic because buffer was being biodone'ed multiple times.
Версия 1.351, Дата: 2003/01/05
Автор: phk

```
+  if (( bp ->b_flags & B_CACHE) == 0) {
       ...
+      bp ->b_iocmd = BIO_READ;
+      bp ->b_flags &= ~B_INVAL;
       ...
+      if (vp->v_type == VCHR)
+          VOP_SPECSTRATEGY(vp, bp );
+      else
+          VOP_STRATEGY(vp, bp );
       ...
}
```

```
+  if (( rabp >b_flags & B_CACHE) == 0) {
       ...
+      rabp ->b_flags |= B_ASYNC;
+      rabp ->b_flags &= ~B_INVAL;
       ...
+      if (vp->v_type == VCHR)
+          VOP_SPECSTRATEGY(vp, bp rabp);
+      else
+          VOP_STRATEGY(vp, bp rabp);
       ...
}
```

Рисунок 1.3. Некорректное переименование.

Несоответствие потока данных (Inconsistent data flow - IDF): во фрагменте кода производится некорректная инициализация переменных. На рисунке 1.4 во втором примере переменная «optarg» не инициализирована, поскольку для её инициализации необходимо вызвать функцию «getopt». Такие ошибки могут приводить к доступу к несуществующей или недоступной памяти.

FreeBSD: src/sbin/gpt/gpt.c
Log: Fix cut-n-paste bug: compare argument s against known aliases, not the global optarg
Версия 1.16, Дата: 2006/07/07
Автор: marcel

```
main(int argc, char *argv[]) {
  ...
  while ((ch = getopt(argc, argv,...)) != -1)
    switch (ch) {
       ...
+      case 'o ':
+      if (strcmp(optarg, "space") == 0) {
+         opt = FS_OPTSPACE;
       ... } ... }
```

```
parse_uuid(const char *s, uuid_t *uuid) {
   ...
   switch (*s) {
+     case 'e':
+     if (strcmp(optarg s, "efi") == 0) {
+        uuid_t efi = GPT_ENT_TYPE_EFI;
      ... } ... } }
```

Рисунок 1.4. Некорректная инициализация optarg.

Избыточные операции (Redundant operations - RDN): во фрагменте портированного кода остаются избыточные инструкции. На рисунке 1.5 показано, как функция «memcpy» была портирована дважды.

Linux commit: src/sys/dev/mxge/if_mxge.c
Log: Looks like a copy-n-paste error, identical lines are a few lines below the ones removed.
Версия 1.16, Дата: 2011/04/29
Автор: John W. Linville

```
   memset(&tsf_tlv, 0x00, sizeof(struct
            mwifiex_ie_types_tsf_timestamp));
    ...
+  emcpy(*buffer,&tsf_tlv,sizeof(tsf_tlv.header));
+ *buffer += sizeof(tsf_tlv.header);
```

```
+  memcpy(*buffer, &tsf_val, sizeof(tsf_val));
+  *buffer += sizeof(tsf_val);
    memcpy(&tsf_val,bss_desc-> time_stamp,
             sizeof(tsf_val));
    ..
+  memcpy(*buffer, &tsf_val, sizeof(tsf_val));
+  *buffer += sizeof(tsf_val);
```

Рисунок 1.5. Некорректная инициализация «optarg».

Несоответствие стиля и комментариев: Фрагмент кода отличается стилем и структурой или комментарии не соответствуют функционалу кода.

Недостаток метода Ray [82] заключается в том, что он не может быть применен без репозитория проекта.

Глава 2.
Методы поиска клонов кода

Предложенный подход основан на семантическом анализе программы. Поиск максимальных схожих подграфов проводится в четыре этапа, что позволяет эффективно и точно решать задачу. Сначала ГЗП разделяются на подграфы (единицы сравнения – ЕС), рассматриваемые как потенциальные клоны один другого. Обработка всех пар ЕС производится в две фазы. На первой фазе за линейное время отсеиваются заведомо несхожие пары ЕС. Если пара ЕС не была отсеяна, к ней на второй фазе применяется приближенный алгоритм поиска максимальных схожих подграфов. После того как максимальные схожие подграфы найдены, производится фильтрация ложных срабатываний путем проверки строк исходного кода, соответствующих схожим подграфам. Если строки фрагмента исходного кода, соответствующие найденному подграфу, находятся на расстоянии, меньшем p (параметр, заданный пользователем) и длина фрагмента больше размера минимального клона (задается пользователем), то они считаются клонами.

2.1. Разделение ГЗП на подграфы

Существует несколько методов разделения ГЗП на ЕС. Один из методов разделяет граф на слабосвязанные компоненты [88], которые рассматриваются как ЕС. Недостаток этого метода в том, что размеры фрагментов могут сильно варьироваться. Фрагмент может быть настолько большим, что он будет содержать фрагменты кода, которые могут оказаться клонами. Другой подход разделяет

граф на подграфы, любые два из которых имеют не более чем N общих ребер [65]. Этот метод был применен в инструменте DECKARD [48] и позволил найти в среднем в два раза больше клонов.

Ниже описан алгоритм разделения ГЗП на ЕС. Для оценки качества предлагаемого алгоритма были реализованы методы разделения [88] (WCC – Weakly Connected Components) и [65] (IST – Interesting Semantic Threads). Результаты тестирования на наборе проектов с открытым исходным кодом (Linux kernel, OpenSSL, LLVM см. раздел 2.8) показали, что число найденных клонов при использовании предложенного алгоритма возрастает в несколько раз (см. раздел 2.8.1).

Ребра ГЗП рассматриваются как интервалы. Разделение графа происходит соответственно тем вершинам, которые имеют минимальное количество пересекающихся ребер. Эффективность данного алгоритма обусловлена следующими особенностями разделения ГЗП на подграфы:

- вершинам полученных подграфов соответствуют последовательные строки исходного кода;
- количество строк кода всех ЕС приблизительно одинаково;
- количество ребер между ЕС сводится к минимуму, что обеспечивает максимальную семантическую независимость полученных подграфов.

2.1.1. Алгоритм разделения ГЗП на ЕС

Алгоритм получает на вход ГЗП, размер (количество строк) исходного кода, соответствующий одному подграфу, и процент, который показывает возможную погрешность размера исходного кода одного подграфа. Алгоритм возвращает множество подграфов, где каждому подграфу соответствует фрагмент кода с последовательными строками кода.

1. Входные параметры: G – ГЗП, N – размер исходного кода, соответствующий подграфу, P – процент возможной вариации N.
2. Каждой вершине ГЗП сопоставить число, которое показывает количество пересекающихся ребер в этой точке. Отсортировать вершины графа по

соответствующим номерам строк исходного кода (после сортировки вершины находятся в векторе SS). Для каждой вершины I рассмотреть все вершины I_j, для которых существует ребро (I, I_j). У всех вершин, которые находятся в интервале $(I, I_j]$ или $[I_j, I)$, если в отсортированном векторе SS вершина I_j меньше чем I, увеличить счетчик пересечений на единицу.

3. Рассмотреть все элементы SS, лежащие в интервале $[N, N + \frac{N * P}{100}]$; элемент, имеющий минимальное значение счетчика пересечений, выбрать точкой пересечения.
4. Элементы вектора SS до точки пересечения добавить в отдельное множество S и удалить из SS. Сохранить S, после чего повторять шаг 3, пока SS не станет пустым, или там окажется меньше, чем N элементов. Если SS пустой, перейти к шагу 5, если в SS оказалось меньше, чем N элементов, добавить во множество S, сохранить S, удалить элементы из SS и перейти к шагу 5.
5. Для всех сохраненных множеств S создать граф GS. Вершины графа GS – это элементы множества S. Для двух элементов E_1 и E_2 из S в графе GS будет ребро, если существует соответствующее ребро в графе G.

Пример: На рисунке 2.1 иллюстрируется работа алгоритма после того, как вершины были отсортированы по соответствующим строкам исходного кода.

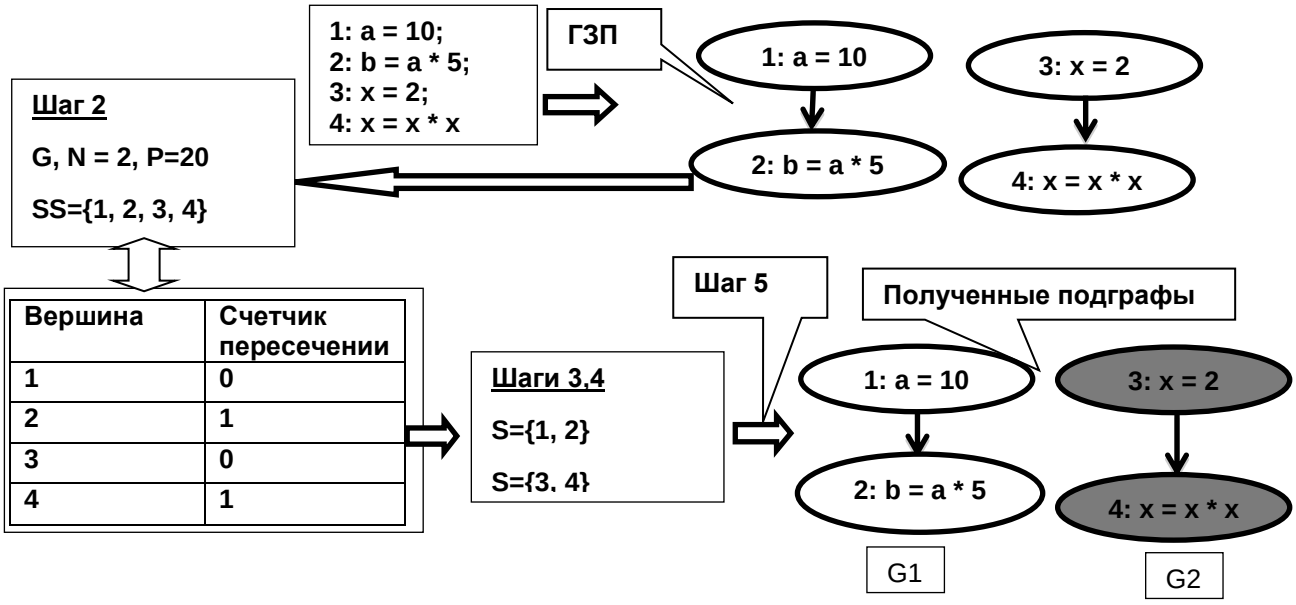

Рисунок 2.1. Пример работы алгоритма разделения ГЗП на ЕС.

Сложность предложенного алгоритма $O(mn^2)$, где n количество вершин графа, m средняя степень вершин.

2.2. Алгоритмы линейной сложности для отсева пар ЕС

Согласно исследованиям [18, 28], клоны кода составляют не более 20 процентов кода программных проектов (хорошо спроектированное ПО имеет меньший процент клонов). Перед проверкой пар ЕС на схожесть целесообразно сначала отсеять пары ЕС, которые заведомо не являются клонами. Такой отсев может быть проведен за линейное время. Каждая вершина графа имеет метку (код соответствующей операции), у схожих вершин метки должны совпадать. Алгоритмы отсева рассматривают наличие у пары ЕС достатчного количества вершин с совпадающими метками. Исходный код, соответствующий таким вершинам, должен быть расположен в тексте программы таким образом, чтобы размер полученного фрагмента был не меньше размера минимального клона.

Первый алгоритм сохраняет метки вершин ГЗП G_1, G_2 в хеш-таблицы T_1, T_2 соответственно. Если $|T_1 \bigcap T_2| \leq k * L$, тогда G_1, G_2 не могут иметь схожие подграфы желаемого размера, где L размер минимального клона, k среднее количество вершин ГЗП соответствующий одной строке исходного кода.

ГЗП имеет ограниченное количество разных типов вершин (метки вершин). Характеристический вектор ГЗП – это специальный вектор длины N, где N – количество различных типов вершин ГЗП. Каждый элемент вектора – это количество вершин в ГЗП, имеющих специальный тип (например, количество вершин, соответствующих операциям сложения).

Второй алгоритм вычисляет характеристический вектор для каждого ГЗП. Если евклидово расстояние между двумя векторами больше, чем заданное число d (пользователь задает степень схожести искомых клонов $sim \in (0,1]$, на основе этого определяется $d = 1 - sim$), тогда соответствующие им ГЗП не содержат желаемые схожие подграфы.

Сложность этих алгоритмов линейная от количества вершин, соответствующей пары ЕС.

2.3. Поиск схожих подграфов на основе слайсинга

Ниже описан приближенный алгоритм поиска схожих подграфов максимального размера в паре ГЗП/ЕС, основанный на слайсинге [62]. Этот алгоритм применяется для пары ЕС, если с помощью алгоритма проверки не доказано, что данная пара не может быть клоном.

Алгоритм поиска схожих подграфов: Предложенный алгоритм является приближенным потому что задача поиска максимальных схожих/изоморфных подграфов NP сложная. Прямой/обратный слайсинг и пошаговая фильтрация несовпадающих вершин работают на основе данного алгоритма. Сначала для каждой вершины первого графа выбирается максимально схожая (см. раздел 2.5) с ней вершина из второго графа. После этого все пары вершин считаются схожими подграфами и расширяются путем добавления подходящих инцидентных вершин. Алгоритм возвращает самую большую пару подграфов, полученный при расширении как максимальные схожие. Надо отметить, что предложенный алгоритм является приближением и не гарантируете максимальность найденных схожих подграфов.

1. Входные параметры: графы G_1 и G_2. Выходные параметры: множества вершин S_1 из G_1 и S_2 из G_2, где S_1 и S_2 – множества вершин схожих подграфов максимального размера.
2. Построить множество пар вершин P. Пара $<v,u>$, где v из G_1, u из G_2, войдет в P только в том случае, если для вершины v вершина u такова, что $<v,u>$ имеет наибольшее количество идентичных соседних вершин. Если пара $<v,u>$ вошла в P, то v и u не будут участвовать в создании новых пар.
3. Для данной пары $<v,u>$ из P создать пустые множества S_1, S_2 и добавить v в S_1, u в S_2.

4. Для всех пар вершин $<v_1, v_2>$, где v_1 из S_1, v_2 из S_2, которые не были рассмотрены в 4 и 5 шагах, проверить, являются ли они идентичными (совпадают ли метки). Если да, перейти к шагу 5. Если нет, пропустить данную пару. Если в ходе данного шага не была найдена пара нерассмотренных идентичных вершин, перейти к шагу 6.
5. Пометить v_1, v_2 как рассмотренные. Найти множества всех нерассмотренных инцидентных вершин v_1, v_2. Построить пересечение множеств этих вершин (критерием равенства элементов считается совпадение меток вершин). Вершины, попавшие в пересечение, добавить в S_1, S_2 соответственно. Перейти к шагу 4.
6. Запомнить пару (S_1, S_2). Повторить шаг 3 для нерассмотренных пар из P. Из всех сохраненных пар (S_1, S_2) вернуть пару максимального размера. Размером пары (S_1, S_2) считается количество элементов наименьшего множества.

Данный алгоритм имеет преимущество по сравнению с существующим слайсингом [64]: одновременно используются прямой и обратный слайсинг, что позволяет найти некоторые идентичные графы, которые невозможно найти, используя только прямой или только обратный слайсинг.

Пример: На рисунке 2.2 показана работа алгоритма для одной пары (<1,1>) вершин (шаги 3, 4, 5, 6).

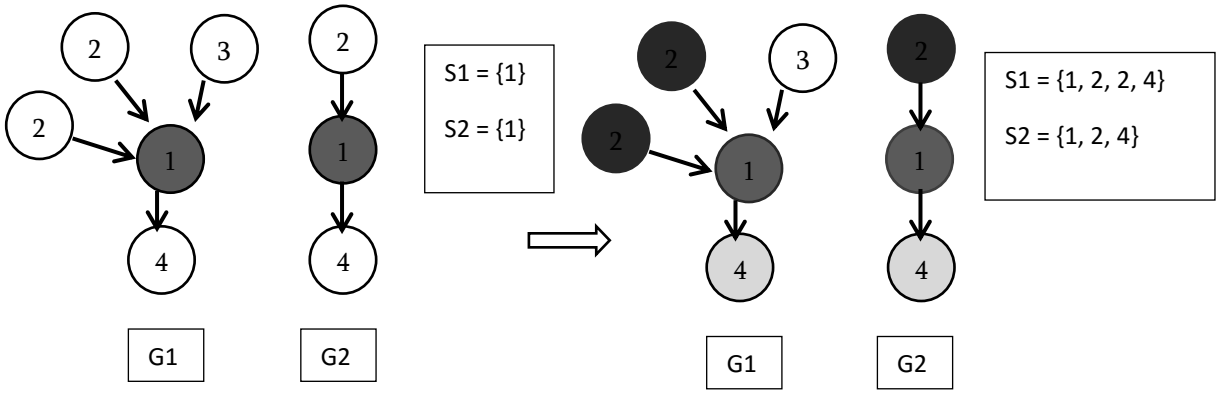

Рисунок 2.2. Пример работы алгоритма: вершины с меткой «2» найдены при обратном слайсинге вершин с меткой «1».

Сложность этого алгоритма $O(nm(n\log(n)+m\log(m))$, где n и m – количество вершин в первом и во втором графе соответственно.

2.4. Поиск схожих подграфов на основе изоморфизма деревьев

Поиска клонов кода на основе изоморфизма деревьев состоит из двух основных этапов. На первом этапе ГЗП преобразуется в дерево (рисунок 2.3), при преобразовании добавляются новые ребра и вершины, что дает возможность сохранить максимальное количество информации об изначальном графе. На втором этапе производится поиск максимальных изоморфных поддеревьев, которые рассматриваются как клоны кода. Такой подход позволяет применить точные алгоритмы поиска максимальных изоморфных поддеревьев, что позволяет быстрее искать клоны типов Т1 и Т2 по сравнению со слайсингом (у этого алгоритма более низкая вычислительная сложность, чем у слайсинга, см. раздел 2.3). Для задач, где надо найти только клоны типов Т1 и Т2 лучше всего подходит этот алгоритм (см. раздел 2.6).

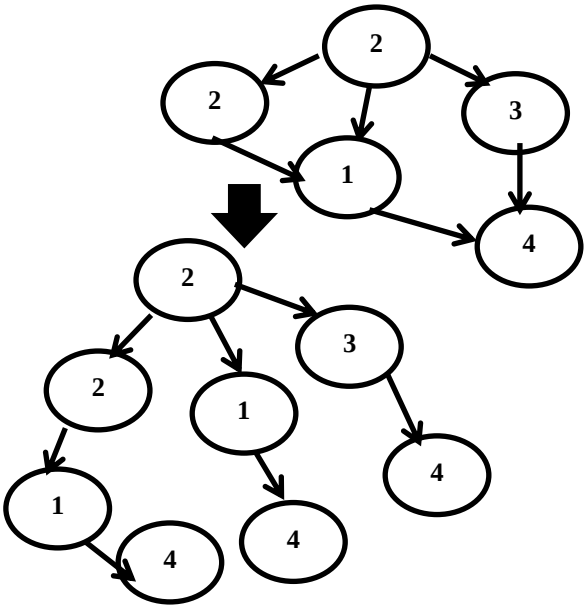

Рисунок 2.3. Преобразование ГЗП в дерево.

2.4.1. Преобразование ГЗП в дерево

Преобразование ГЗП в дерево в свою очередь выполняется в два этапа. Сначала выполняется удаление обратно направленных ребер и топологическая сортировка ГЗП. Сортировка начинается из начальной вершины (вершина, у которой нет входящих ребер, каждый ГЗП имеет такую вершину). Затем в обратном порядке рассматриваются уровни вершин, полученные в результате сортировки. Вершины, у которых больше одного входящего ребра, преобразуются следующим образом.

Предположим, что $V-$ это вершина, у которой есть входящие ребра от вершин $W_1,...,W_n$. Ребра (W_i,V) отсортированы соответственно меткам (тип операции соответствующей инструкции) W_i (W_n – максимальный).

Преобразование А. Предположим, что из вершин $W_1,...,W_n$ только W_n имеет максимальную метку. В этом случае создается $n-1$ новых вершин $V_1,...,V_{n-1}$ с метками как у V. Ребра (W_i,V) заменяются на ребра $(W_i,V_i), i=1,...,n-1.$

Преобразование Б. Предположим, есть несколько вершин из $W_1,...,W_n$ с максимальной меткой: существует такое l, что метки вершин $W_l,...,W_n$ равны друг другу, и $W_{l-1}\neq W_l, l\neq 1$. В этом случае создаются $V_1,...,V_{l-1}$ новые вершины с метками как у V. Ребра (W_i,V) меняются на ребра $(W_i,V_i), i=1,...,l-1.$ Для каждой вершины $W_l,...,W_{n-1}$ поддерево с корнем V копируется, и ребра (W_i,V) меняются на ребра $(W_i,V_i), i=l,...,n-1,$ где V_i- корень скопированного поддерева соответствующей W_i.

Утверждение 3: Для двух изоморфных ГЗП G и H деревья, получаемые с помощью преобразований **А** и **Б**, изоморфны.

Доказательство: Каждая функция имеет ровно одну инструкцию (входная инструкция – это входная инструкция функции, которой передается управление при вызове функции), из которой доступны все остальные инструкции (любая вершина ГЗП доступна из входной вершины ребрами по управлению). Для G и H графов удаление обратно направленных ребер и топологическая сортировка производится однозначным образом (сортировка G и H начинается из входных

инструкций). Из этого следует, что после удаления обратно направленных ребер и топологической сортировки два изоморфных графа останутся изоморфными. На соответствующих уровнях графов будут находиться изоморфные вершины (рисунок 2.4).

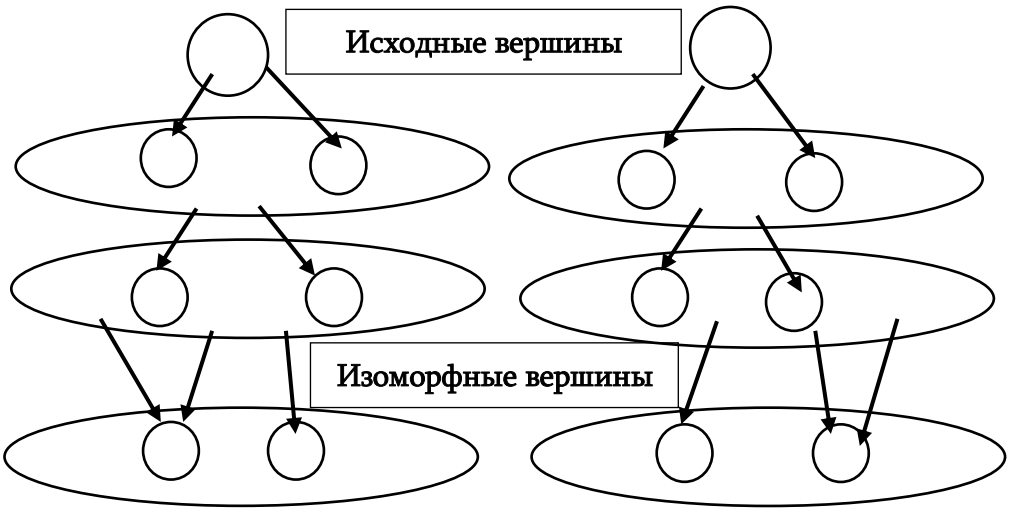

Рисунок 2.4. Изоморфные деревья.

Используя математическую индукцию, докажем, что деревья, получаемые путем применения преобразования **А** и **Б** в обратном порядке над парой топологически сортированных ациклических изоморфных графов, будут изоморфными.

Основание: Рассмотрим последние уровни вершин L^1_n и L^2_n для пары изоморфных графов G и H, полученные в ходе топологической сортировки. Для каждой вершины v из L^1_n существует точно одна изоморфная вершина u из L^2_n.. Из этого следует, что для вершин v и u будут применены одинаковые преобразования, **А** или **Б**. Таким образом, получается, что преобразование всех вершин (вершин L^1_n и L^2_n) последнего уровня обоих графов не изменит отношения изоморфизма.

Шаг индукции: Предположим, что после преобразования всех вершин L^1_n, L^2_n, ….., L^1_i,L^2_i полученные графы остались изоморфными. Докажем, что после преобразования всех вершин L^1_{i-1} и L^2_{i-1} полученные графы будут изоморфными. Рассмотрим произвольную вершину v из L^1_{i-1}, для неё существует точно одна изоморфная вершина u из L^2_{i-1}. Для v и u будут применены одинаковые

преобразования. Если для них было применено преобразование **А**, очевидно, что полученные графы останутся изоморфными.

В случае применения пробразования **Б** полученные графы тоже будут изоморфными. Предположим, что для вершин v (v имеет входящие ребра от вершин $W_1,...,W_n$) из L^1_{i-1} и u (u имеет входящие ребра от вершин $V_1,...,V_n$) из L^2_{i-1} было применено преобразование **Б**. Это значит, что существует такое l, что индексы вершин $W_l,...,W_n, V_l,...,V_n$ равны друг другу, а индекс W_{l-1},V_{l-1} не равен W_l (если $l \neq 1$). В ходе преобразования **Б** для вершин $W_1,...,W_{l-1}$ и $V_1,...,V_{l-1}$ создаются новые вершины $U^1_1,\ldots, U^1_{l-1}$ и $U^2_1,\ldots, U^2_{l-1}$ с индексом как у v (v и u имеют одинаковые индексы). Ребра (W_i, v) и (V_i,u) меняются на ребра (W_i, U^1_i) и (V_i, U^2_i), $i = 1, \ldots, l-1$, соответственно. Как видно, данная часть преобразований не влияет на изоморфизм полученных графов.

Для каждой вершины $W_l,...,W_{n-1}$ и $V_l,...,V_{n-1}$ поддеревья с корнями v и u копируются, и ребра (W_i,v) и (V_i,u) заменяются на ребра (Wi, U^1_i) и (Wi, U^2_i) $i = l, \ldots, n-1$ соответственно, где U^1_i и U^2_i – корни скопированных поддеревьев, соответствующие W_i и V_i (рисунок 2.6). Поддеревья с корнями v и u изоморфны (это следует из индукции), вследствие чего полученные графы будут изоморфными.

Таким образом, преобразования **А** и **Б** любых двух изоморфных вершин текущего уровня не меняет отношения изоморфизма, следовательно, предположения индукции доказаны.

2.4.2. Алгоритм изоморфизма деревьев

Алгоритм нумерует вершины пар полученных деревьев. В ходе нумерации изоморфные вершины получают одинаковые номера.

Алгоритм:

1. На вход получает пару деревьев T_1 и T_2, возвращает множества вершин изоморфных поддеревьев.

2. Вершины первого графа нумеруются значениями 0, а вершины второго – 1, и деревья объединяются. Корни деревьев добавляются в очередь.
3. Берется первый элемент из очереди, и для него генерируется номер (см. ***генератор номеров***). Если выбранный элемент не является листом, его преемники добавляются в очередь. Процесс повторяется, пока очередь не станет пустой.
4. Все вершины дерева T_1, для которых есть вершина с одинаковым номером из T_2 добавляются в множество F. Аналогичным образом все вершины дерева T_2, для которых есть вершина с одинаковым номером из T_1 добавляются в множество S.

В множествах F и S содержатся вершины изоморфных поддеревьев T_1 и T_2.

Генератор номеров:

1. На вход получает вершину U. Алгоритм использует глобальный счетчик C для нумерации.
2. Для U создается вектор *Vec*, содержащий метку U и номер предшественника U (если есть предшественник). Если такой вектор уже существует, в таблице векторов *tabVec* переходим к шагу 3, если нет – к шагу 4.
3. Номеру U присваивается соответствующее значение счетчика *Vec* из *tabVec*. U добавляется в вектор $Isomorphic[U]$ (*Isomorphic* представляет собой вектор векторов вершин дерева).
4. В *tabVec* добавляется вектор *Vec* и текущее значение счетчика C. Номеру U присваивается значение C. U добавляется в вектор $Isomorphic[U]$. Значение C увеличивается на единицу.

Сложность этого алгоритма $O((n+m)^2(\log(n+m))$, где n и m – количество вершин в первом и во втором дереве соответственно.

Пример: На рисунке 2.5 показана работа алгоритма после шага 2. Входные деревья T_1 и T_2 размечены и объединены в одно, после чего корневые вершины добавлены в очередь вершин.

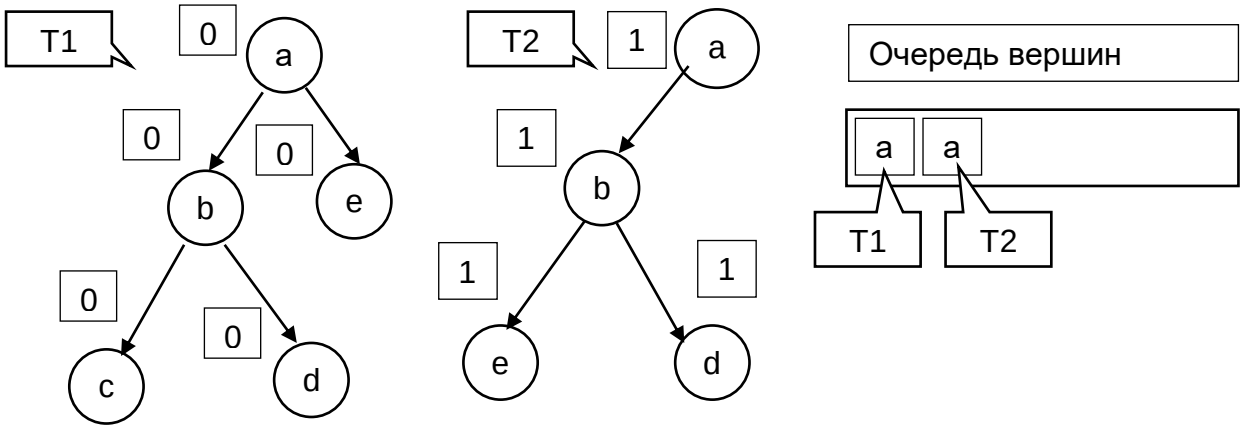

Рисунок 2.5. Объединенное размеченное дерево.

На рисунке 2.6 показана нумерация вершины «a» дерева T1.

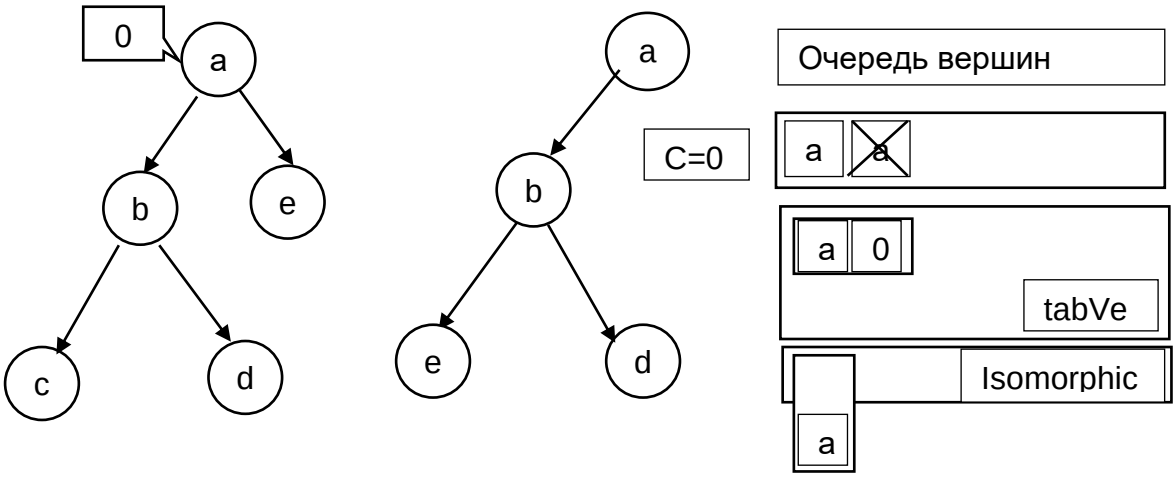

Рисунок 2.6. Нумерация вершин.

Рисунок 2.7 показывает дерево после полной нумерации. Как видно из рисунка, изоморфные вершины получают одинаковые номера.

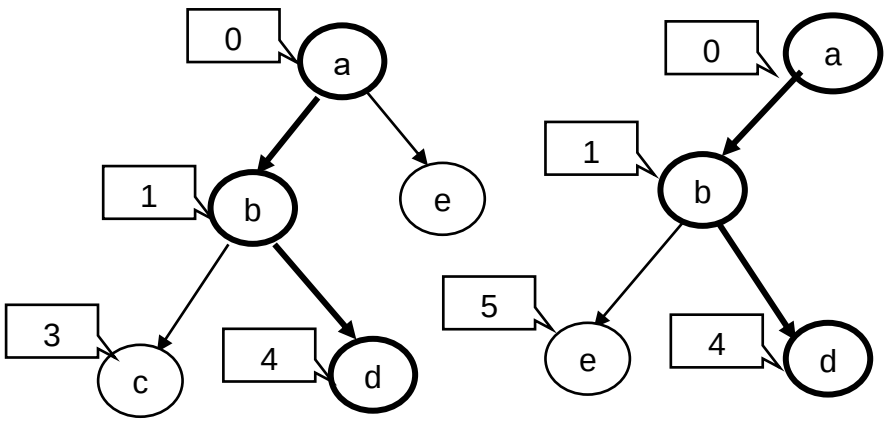

Рисунок 2.7. Изоморфные деревья с одинаковыми номерами.

2.5. Поиск схожих подграфов на основе метрик

В данном разделе приводится алгоритм поиска клонов кода на основе метрик для ГЗП. Для каждой вершины ГЗП строится бинарный вектор, который содержит информацию обо всех соседних вершинах. Бинарные векторы представляются в виде 64-битных целых чисел, поскольку количество различающихся меток вершин ГЗП не больше 60, что позволяет находить сходство двух векторов за константное время.

2.5.1. Битовый вектор:

Битовый вектор (БВ) – это вектор длины $2 * N$, где N – количество всех возможных типов вершин. Тип вершины — это код операции той инструкции, которой он соответствует (метка вершины ГЗП). Типы вершин ГЗП обозначены цифрами от 1 до N. БВ вершины инициализируется следующим образом: позиции $i = 1, \dots, N$ присваивается значение 1, если существует входящее ребро из вершины, типа i. Аналогичным образом позиции $j = N + 1, \dots, 2 * N$ присваивается значение 1, если существует выходящее ребро к вершине, типа $j - N$. Всем остальным позициям присваивается 0. (рисунок 2.8).

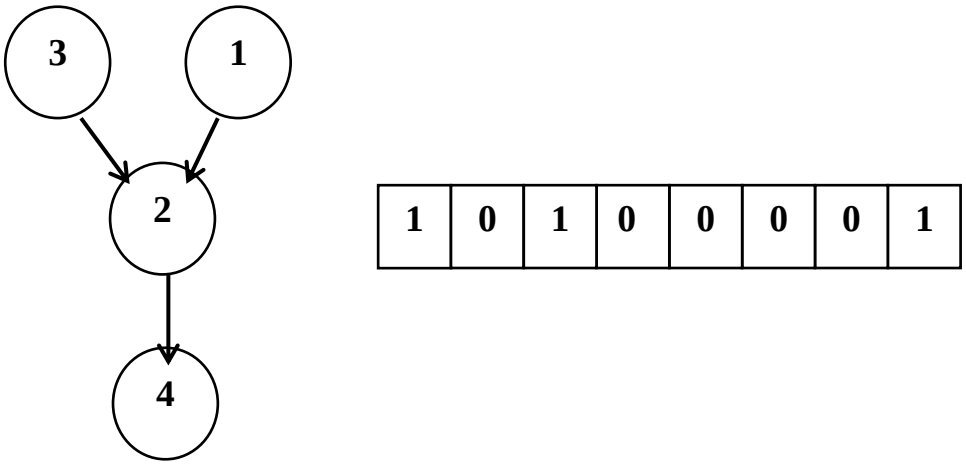

Рисунок 2.8. Пример БВ.

Определение 1: Для двух БВ V_1 и V_2 длины $N, V_1 \wedge V_2$ – это новый БВ V длины N, у которого $V[i] = V_1[i] \wedge V_2[i], i = 1, \dots, N.$

Определение 2: Для двух БВ V_1 и V_2 длины $N, V_1 \vee V_2 -$ это новый БВ V длины N, у которого $V[i] = V_1[i] \vee V_2[i]$, $i = 1,..., N$.

Определение 3: Для двух БВ V_1 и V_2 длины N, $andC(V_1, V_2)$ – это количество элементов со значением 1 в векторе $V_1 \wedge V_2$.

Определение 4: Для двух БВ V_1 и V_2 длины N, $orC(V_1, V_2)$ – это количество элементов со значением 1 в векторе $V_1 \vee V_2$.

Утверждение 1: Пусть $X_n -$ множество БВ длины n, тогда функция $Dist(V_1, V_2) = \dfrac{orC(V_1, V_2) - andC(V_1, V_2)}{1 + orC(V_1, V_2)}$ является метрикой, где V_1, V_2 из X_n.

Доказательство: Необходимо доказать следующие три свойства метрики:

1. $Dist(V_1, V_2) = 0$ только в том случае, если $V_1 = V_2$, — это свойства тождества.
2. $Dist(V_1, V_2) = Dist(V_2, V_1)$ — это свойства симметрии.
3. $Dist(V_1, V_2) \le Dist(V_1, V_3) + Dist(V_2, V_3)$ (1), для любых V_1, V_2, V_3 — это свойства треугольника.

Доказательство-1(<=): Предположим $V_1 = V_2$, из определения $orC(V_1, V_2)$ и $andC(V_1, V_2)$ следует что $andC(V_1, V_2) = orC(V_1, V_2)$. Следовательно, $Dist(V_1, V_2) = 0$.

Доказательство-1(=>): Предположим $Dist(V_1, V_2) = 0$, докажем что $V_1 = V_2$. Допустим существует $V_1 != V_2$, такой, что $Dist(V_1, V_2) = 0$. $Dist(V_1, V_2) = 0$ только в том случае, если $andC(V_1, V_2) = orC(V_1, V_2)$. Так как $V_1 != V_2$, следовательно, в каких-то $M > 0$ позициях значение элементов V_1 и V_2 не совпадают. Из этого следует, что $orC(V_1, V_2) = M + andC(V_1, V_2)$. Следовательно, $andC(V_1, V_2) != orC(V_1, V_2)$, что противоречит нашему предположению.

Доказательство-2: Из определений 3, 4 следует, что $andC(V_1, V_2) = andC(V_2, V_1)$ и $orC(V_1, V_2) = orC(V_2, V_1)$. Из этого следует, что $Dist(V_1, V_2) = \dfrac{orC(V_1, V_2) - andC(V_1, V_2)}{1 + orC(V_1, V_2)} = \dfrac{orC(V_2, V_1) - andC(V_2, V_1)}{1 + orC(V_2, V_1)} = Dist(V_2, V_1)$.

Доказательство-3: V_1, V_2, V_3 можно представить в виде:

$V_1 = (a_1, a_2, ..., a_n)$

$V_1 = (b_1, b_2, ..., b_n)$

$V_1 = (c_1, c_2, ..., c_n)$,

где $a_i, b_i, c_i \in \{0,1\}, 1 \le i \le n$. Из этого следует, что функции $andC, orC$ можно представить в следующем виде:

$orC(V_1, V_2) = (a_1 + b_1 - a_1 * b_1) + (a_2 + b_2 - a_2 * b_2) + (a_n + b_n - a_n * b_n)$.

$andC(V_1, V_2) = a_1 * b_1 + a_2 * b_2 + a_n * b_n$.

Из этого следует, что неравенство (1) можно представить в виде (2):

$$\frac{(a_1 + b_1 - 2 * a_1 * b_1) + ... + (a_n + b_n - 2 * a_n * b_n)}{1 + (a_1 + b_1 - a_1 * b_1) + ... + (a_n + b_n - a_n * b_n)} \le$$
$$\frac{(a_1 + c_1 - 2 * a_1 * c_1) + ... + (a_n + c_n - 2 * a_n * c_n)}{1 + (a_1 + c_1 - a_1 * c_1) + ... + (a_n + c_n - a_n * c_n)} +$$
$$\frac{(b_1 + c_1 - 2 * b_1 * c_1) + ... + (b_n + c_n - 2 * b_n * c_n)}{1 + (b_1 + c_1 - b_1 * c_1) + ... + (b_n + c_n - b_n * c_n)} \quad (2)$$

неравенство (2) будет доказано путем математической индукции.

Основание: При $n = 1$ легко удостовериться, что:

$$\frac{a_1 + b_1 - 2 * a_1 * b_1}{1 + a_1 + b_1 - a_1 * b_1} = \frac{1}{2} * |a_1 - b_1|, a_1, b_1 \in \{0,1\}$$

Неравенство $\frac{1}{2} * |a_1 - b_1| \le \frac{1}{2} * |a_1 - c_1| + \frac{1}{2} * |b_1 - c_1|$ истинно, следовательно, для $n = 1$ неравенство (2) истинно.

Шаг индукции: По индукции, неравенство (2) истинно для всех $a_i, b_i, c_i \in \{0,1\}, 1 \le i \le n-1$. Докажем для случая $i = n$. Рассмотрим два случая:

Случай-1: Существует такое i, что $a_i = b_i = 0$. Без нарушения общности можно рассмотреть случай, когда $a_n = b_n = 0$. В этом случае получится:

$$Dist(V_1,V_2)=\frac{(a_1+b_1-2*a_1*b_1)+...+(a_{n-1}+b_{n-1}-2*a_{n-1}*b_{n-1})}{1+(a_1+b_1-a_1*b_1)+...+(a_{n-1}+b_{n-1}-a_{n-1}*b_{n-1})}\le$$
$$\frac{(a_1+c_1-2*a_1*c_1)+...+(a_{n-1}+c_{n-1}-2*a_{n-1}*c_{n-1})}{1+(a_1+c_1-a_1*c_1)+...+(a_{n-1}+c_{n-1}-a_{n-1}*c_{n-1})}(3)$$
$$+\frac{(b_1+c_1-2*b_1*c_1)+...+(b_{n-1}+c_{n-1}-2*b_{n-1}*c_{n-1})}{1+(b_1+c_1-b_1*c_1)+...+(b_{n-1}+c_{n-1}-b_{n-1}*c_{n-1})}(4)$$

Вышенаписанное следует из предположения индукции.

$$\le\frac{(a_1+c_1-2*a_1*c_1)+...+(a_{n-1}+c_{n-1}-2*a_{n-1}*c_{n-1})+c_n}{1+(a_1+c_1-a_1*c_1)+...+(a_{n-1}+c_{n-1}-a_{n-1}*c_{n-1})+c_n}$$
$$+\frac{(b_1+c_1-2*b_1*c_1)+...+(b_{n-1}+c_{n-1}-2*b_{n-1}*c_{n-1})+c_n}{1+(b_1+c_1-b_1*c_1)+...+(b_{n-1}+c_{n-1}-b_{n-1}*c_{n-1})+c_n}(5)$$

(5) следует из того, что дроби (3) и (4) являются регулярными, следовательно, при добавлении положительного числа они могут стать только больше.

$$=\frac{(a_1+c_1-2*a_1*c_1)+...+(a_n+c_n-2*a_n*c_n)}{1+(a_1+c_1-a_1*c_1)+...+(a_n+c_n-a_n*c_n)}$$
$$+\frac{(b_1+c_1-2*b_1*c_1)+...+(b_n+c_n-2*b_n*c_n)}{1+(b_1+c_1-b_1*c_1)+...+(b_n+c_n-b_n*c_n)}=Dist(V_1,V_3)+Dist(V_2,V_3)(5)$$

(5) следует из того, что $a_i=b_i=0$. Что и требовалось доказать.

Случай-2: Не существует такой i, что $a_i=b_i=0,1\le i\le n$, это значит, если есть $a_i=b_i$ то $a_i=b_i=1$.

Без нарушения общности можно рассмотреть случай, когда $a_1\ne b_1,...,a_k\ne b_k,a_{k+1}=b_{k+1},...,a_n=b_n,0\le k\le n$. Если $a_i\ne b_i$, тогда $b_i=1-a_i$, где $a_i,b_i\in\{0,1\}$. Из вышесказанного следует, что (2) эквивалентен (6).

$$\frac{k}{1+n}\le\frac{(a_1+c_1-2*a_1*c_1)+...+(a_n+c_n-2*a_n*c_n)}{1+(a_1+c_1-a_1*c_1)+...+(a_n+c_n-a_n*c_n)}$$
$$+\frac{(b_1+c_1-2*b_1*c_1)+...+(b_n+c_n-2*b_n*c_n)}{1+(b_1+c_1-b_1*c_1)+...+(b_n+c_n-b_n*c_n)}(6)$$

В неравенстве (6) заменим b_i на $1-a_i, i=1,...,k$, а $b_j, a_j, j=k+1,...,n$ на 1. Получится:

$$\frac{k}{1+n} \le \frac{(a_1+c_1-2*a_1*c_1)+...+(a_k+c_k-2*a_k*c_k)+(1-c_{k+1})+....+(1-c_n)}{1+(a_1+c_1-a_1*c_1)+...+(a_k+c_k-a_k*c_k)+n-k} + \frac{(1-a_1-c_1+2*a_1*c_1)+...+(1-a_n-c_n+2*a_k*c_k)+(1-c_{k+1})+....+(1-c_n)}{1+(1-a_1+a_1*c_1)+(1-a_k+a_k*c_k)+n-k} \quad (6)$$

Докажем неравенство (7), из которого следует (6).

$$\frac{k}{1+n} \le \frac{(a_1+c_1-2*a_1*c_1)+...+(a_k+c_k-2*a_k*c_k)}{1+(a_1+c_1-a_1*c_1)+...+(a_k+c_k-a_k*c_k)+n-k} + \frac{(1-a_1-c_1+2*a_1*c_1)+...+(1-a_n-c_n+2*a_k*c_k)}{1+(1-a_1+a_1*c_1)+(1-a_k+a_k*c_k)+n-k} \quad (7)$$

Без нарушения общности можно рассмотреть случай, когда $a_1 \ne c_1,..., a_l \ne c_l, a_{l+1} = c_{l+1},..., a_k = c_k, 0 \le l \le k$. Из этого следует, что (7) эквивалентно (8).

$$\frac{k}{1+n} \le \frac{l}{1+l+(c_{l+1}+...+c_k)+n-k} + \frac{k-l}{1+(c_1+...+cl)+k-l+n-k} \quad (8)$$

$$\frac{k}{1+n} = \frac{l}{1+n} + \frac{k-l}{1+n} \le \frac{l}{1+n+l-k+(c_{l+1}+...+c_k)} + \frac{k-l}{1+n-l+(c_1+...+c_l)}$$

Последнее неравенство следует из того, что $l-k+(c_{l+1}+...+c_k) \le 0$ и $-l+(c_1+...+c_l) \le 0$. Что и требовалось доказать.

Утверждение 2:

Для любых $V_1, V_2 \in X_n, 0 \le Dist(V_1, V_2) < 0$.

Доказательство:

Очевидно, что для любых $V_1, V_2 \in X_n, Dist(V_1, V_2) \ge 0$, поскольку из определения $orC, andC$ следует $orC(V_1, V_2) \ge andC(V_1, V_2) \ge 0$.

1) Докажем существование таких $V_1, V_2 \in X_n$, что $Dist(V_1, V_2) = 0$.

Рассмотрим $V_1, V_2 \in X_n$, где $V_1 = V_2 = \{0,0,...0\}$, из определения $orC, andC$ следует, что $orC(V_1,V_2) = andC(V_1,V_2) = 0$. Следовательно, $Dist(V_1,V_2) = \frac{orC(V_1,V_2) - andC(V_1,V_2)}{1 + orC(V_1,V_2)} = 0$, что и требовалось доказать.

2) Теперь докажем, что для любой $V_1, V_2 \in X_n, Dist(V_1,V_2) < 1$.

$orC(V_1,V_2) \geq andC(V_1,V_2) \geq 0$ следовательно, $orC(V_1,V_2) - andC(V_1,V_2) \leq orC(V_1,V_2)$, из чего получается, что $Dist(V_1,V_2) = \frac{orC(V_1,V_2) - andC(V_1,V_2)}{1 + orC(V_1,V_2)} \leq \frac{orC(V_1,V_2)}{1 + orC(V_1,V_2)} \leq 1$. Что и требовалось доказать.

Определение 5: Степень с*хожести* двух БВ V_1 и V_2 одинаковой длины определяется как $1 - Dist(V_1,V_2)$.

Определение 6: *Плотностью* множества вершин ГЗП называется $P(S) = \frac{|S|}{\max(S) - \min(S)}$, где S – множество вершин ГЗП, которые отсортированы по номерам соответствующих строк исходного кода, $\max(S)$ – номер строки исходного кода, соответствующая максимальной вершине из S, $\min(S)$ – номер строки исходного кода, соответствующая минимальной вершине из S.

На первом этапе алгоритм для данной пары ГЗП строит множества схожих вершин. На втором этапе из каждого множества удаляются некоторые элементы, пока плотность (определение 6) множества не будет больше *k*sim*, где *sim* степень схожести искомых клонов заданное пользователем, *k* коэффициент полученный на практике (приблизительно равен 0.8).

***Алгоритм MBCCD*:**

1. Входные параметры: G_1 и G_2 — ГЗП, S — уровень сходства клонов, CL — длина минимального клона.
2. Построить таблицы (map) C_1 и C_2 для вершин графов G_1 и G_2. Ключ таблицы — БВ вершины, а значение — сама вершина.
3. Любой элемент n_1 из C_1 удаляется, если нет такого n_2 из C_2, что степень схожести БВ n_1 и n_2 больше S. Для C_2 — аналогично.

4. Построить множества вершин S_1 и S_2 из таблиц C_1 и C_2. Отсортировать множества вершин S_1 и S_2, по строкам исходного кода, соответствующих их вершинам.
5. Рассмотреть первый элемент fe из S_1 и последний элемент le из S_2. Если плотность множества $S_1 \setminus \{fe\}$ больше, чем плотность $S_1 \setminus \{le\}$, удалить le из S_1, в противном случае, удалить fe. Повторять этот шаг, пока плотность S_1 не станет больше S. Для C_2 — аналогично.
6. Если исходный код, соответствующий S_1 и S_2 больше, чем CL, тогда он считается клоном.

Сложность: Благодаря примененной оптимизации, алгоритм имеет высокую эффективность и масштабируем, что позволяет анализировать миллионы строк исходного кода. Языки программирования имеют ограниченное количество операций, обычно их число не превышает 60. Это позволяет БВ вершины ГЗП представить как 64-битное целое число. Благодаря этому, сравнение двух БВ производится за константное время. Для БВ каждой вершины первого ГЗП алгоритм рассматривает БВ всех вершин второго графа, что требует $n \cdot m$ операций, где n — количество вершин первого графа, а m — второго. Таким образом сложность получается $O(nm)$.

2.6. Сравнение реализованных алгоритмов

Наибольшей точностью обладает алгоритм поиска максимально схожих подграфов на основе слайсинга (раздел 2.3). Он позволяет находить фрагменты кода, в которых были сделаны существенные изменения. Высокая точность достигается благодаря тому, что алгоритм полностью учитывает семантику клонированного и модифицированного фрагментов кода. Схожие подграфы, которые были найдены в ходе работы алгоритма, часто получаются изоморфными или имеют большие изоморфные подграфы. По сравнению с остальными подходами, у этого подхода самая большая вычислительная сложность - $O(nm(n\log(n) + m\log(m))$, где n и m – количество вершин в первом и во втором

графе соответственно. Его следует применять в задачах, целью которых является нахождение всех клонов кода. Этот алгоритм не следует применять для задач, для которых время работы является ключевым фактором.

Подход, основанный на изоморфизме деревьев (раздел 2.4), находит клоны типа T1 и T2 с большой точностью. Клоны типа T3 обнаруживаются с низкой точностью, так как в ходе преобразования схожих, но не изоморфных ГЗП в дерево, полученные деревья могут иметь большие различия. Сложность этого алгоритма $O((n+m)^2(\log(n+m))$, где n и m – количество вершин в первом и во втором графе соответственно. С точки зрения вычислительной сложности, этот алгоритм занимает промежуточное положение между алгоритмом на основе метрик и на основе слайсинга. Его следует применять в задачах поиска клонов типов T1 и T2. Этот алгоритм позволяет довольно быстро находить такие клоны с высокой точностью.

Подход, основанный на метриках (раздел 2.5), обладает низкой вычислительной сложностью ($O(nm)$, где n и m – количество вершин в первом и во втором графе соответственно) по сравнению с остальными подходами. Этот подход имеет сравнительно низкую точность, но работает быстрее всех. В ходе работы алгоритм может пропустить некоторые клоны. Сравнительно больше пропускает клонов типа T3. Он также имеет относительно большой уровень ложных срабатываний. Уровень ложных срабатываний обусловлен тем, что относительно несхожих фрагментов исходного кода могут получаться схожие метрики. Этот алгоритм следует применять в таких задачах, где время работы более критично, чем нахождение всех клонов.

2.7. Сравнение с существующими методами

Было проведено сравнение с тремя широко известными инструментами. MOSS [89] разработан в Стэнфордском университете для поиска плагиата в работах студентов. CloneDR [90] разработан компанией Semantic Designs, которая занимается разработкой инструментария для проектирования и анализа ПО.

Разработка CCFinder(X) [29] (автор Toshihiro Kamiya) велась в рамках проекта Exploratory IT Human Resources Project [91]. В таблице 2.1 описан набор тестов, на основе которого проводилось сравнение инструментов.

Таблица 2.1. Описание тестового набора.

Имя теста	**Описание**

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

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

Оригинальный файл изменялся несколькими способами для получения всех трех типов клонов (см. раздел 1.1.1). В работе [92] детально описаны все преобразования. В таблице 2.2 показан результат сравнения инструментов на данном наборе тестов. Как видно из таблицы, CCD обладает наибольшей точностью.

Таблица 1. Результаты сравнения инструментов.
«+» - клон найден, «-» - клон не найден.

Имя теста	MOSS	CloneDR	CCFinder	слайсинг, раздел 2.3	изоморфизм деревьев, раздел 2.4	сравнение метрик, раздел 2.5
copy01.cpp	+	+	+	+	+	+
copy02.cpp	+	+	+	+	+	+
copy03.cpp	+	+	+	+	+	+
copy04.cpp	+	+	+	+	+	+
copy05.cpp	+	+	+	+	+	+
copy06.cpp	-	+	-	+	+	+
copy07.cpp	+	+	-	+	+	+
copy08.cpp	-	-	-	+	+	+
copy09.cpp	-	+	-	+	+	+
copy10.cpp	-	+	-	+	+	+
copy11.cpp	-	-	-	+	-	+
copy12.cpp	+	+	-	+	+	-
copy13.cpp	+	+	-	+	+	+
copy14.cpp	+	+	+	+	+	+
copy15.cpp	+	+	+	+	+	+

2.8. Результаты тестирования

Компьютер, на котором проводилось тестирование – Intel Core i3 CPU 540 с 8ГБ оперативной памяти. Для тестирования были выбраны следующие проекты с открытыми исходными кодами: Linux 2.6, Firefox Mozilla, LLVM/Clang и OpenSSL. Linux является операционной системой семейства Unix. Firefox Mozilla является браузерным движком, разработанным компанией Mozilla. LLVM/Clang – компиляторная инфраструктура, работа над которой была начата в университете Illinois как исследовательский проект. В данный момент она широко используется

как для оптимизации, так и для разных анализов программ. OpenSSL является криптографической библиотекой, поддерживающей безопасность на уровне транспортировки и сокетов (Transport Layer Security – TLS и Secure Sockets Layer – SSL). Она разработан на основе библиотеки SSLeay, разработчиками которой являются Эрик Янг (Eric Young) и Тим Хадсон (Tim Hudson).

2.8.1. Генерация ГЗП

Рисунок 2.9 показывает количество строк исходного кода проанализированных проектов. Ядро Linux 2.6 содержит приблизительно 13.9 миллиона строк исходного кода. Firefox Mozilla, LLVM/Clang и OpenSSL содержат соответственно 5.1, 1.4 и 0.27 миллиона строк кода, написанных на C/C++.

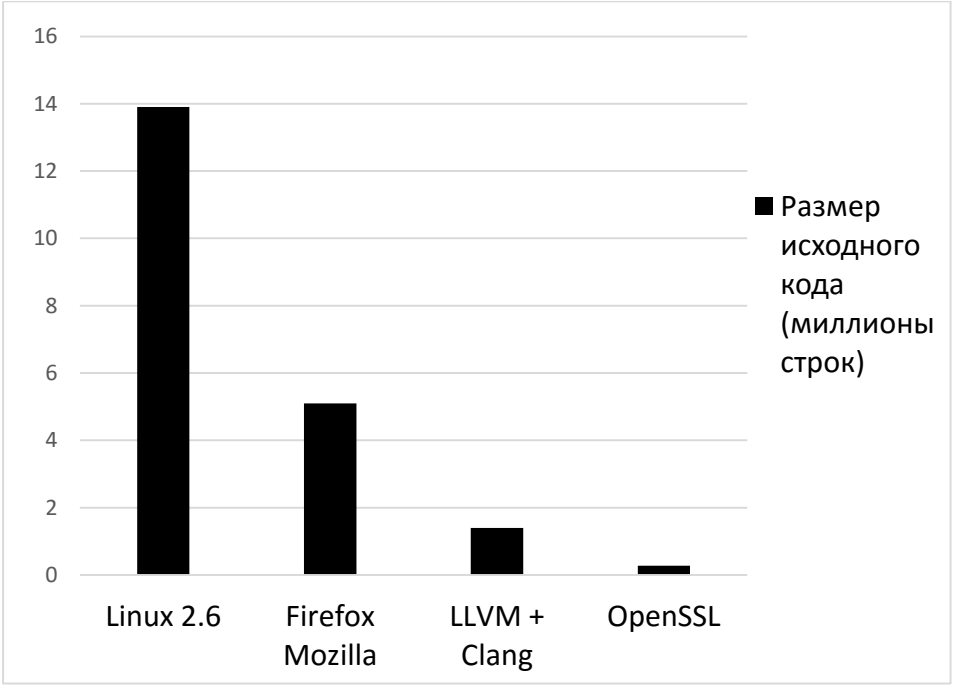

Рисунок 2.9. Количество строк исходного кода.

На рисунке 2.10 приводится сравнение времени компиляции с учетом генерации ГЗП. Время компиляции ядра Linux 2.6 увеличивается с 1.3 до 2 часа, что составляет порядка 50%. Для остальных проектов ухудшения составляют менее 50%.

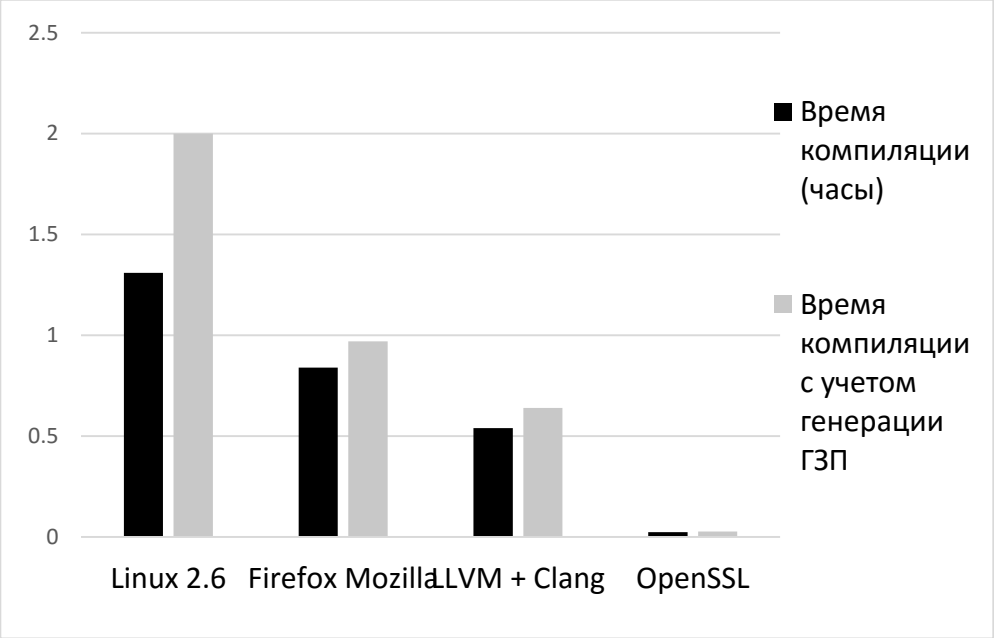

Рисунок 2.10. Сравнения времени компиляции.

Рисунок 2.11 показывает размер полученных ГЗП. Размер ГЗП для ядра Linux 2.6 составляет 439 мегабайт. Для Firefox Mozilla, LLVM/Clang и OpenSSL размер составляет 329, 200 и 12 мегабайт соответственно.

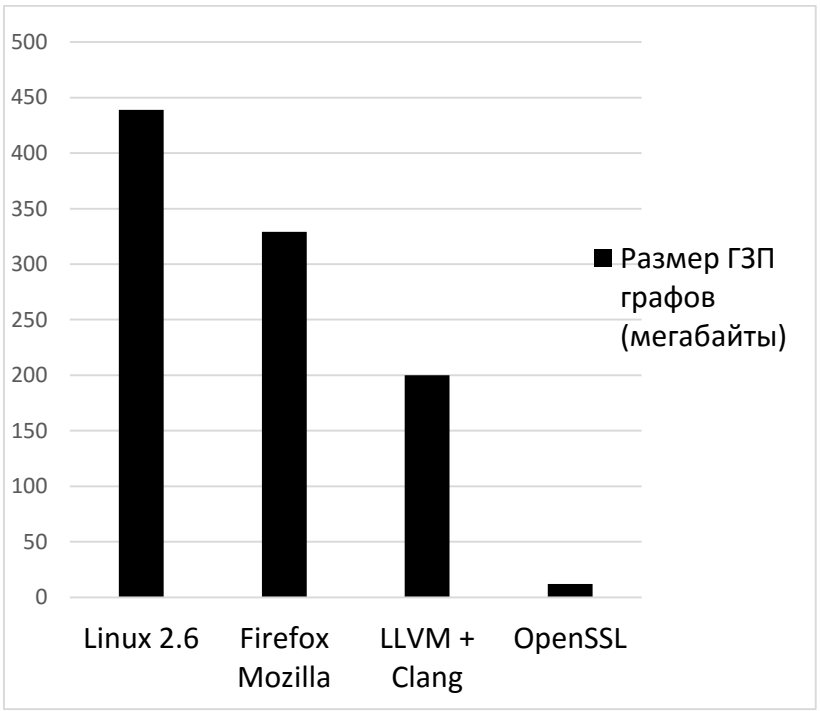

Рисунок 2.11. Размер полученных ГЗП.

2.8.2. Разделение ГЗП

Размер ЕС настраивается двумя параметрами: минимальным размером ЕС (25 строк исходного кода) и верхней границей вариации ЕС (100%), т.е. размер ЕС варьируется в интервале [25, 50] (рисунок 2.12). Указанные параметры выбраны для тестирования и поиска наиболее интересных клонов. Рисунок 4 показывает количество полученных ЕС после разделения ГЗП. Для Linux 2.6, Firefox Mozilla, LLVM/Clang и OpenSSL количество ГЗП вырастает с 33369, 39325, 7884 и 1054 соответственно до 40257, 48921, 11776 и 1588. В среднем, количество ГЗП увеличивается на 25%.

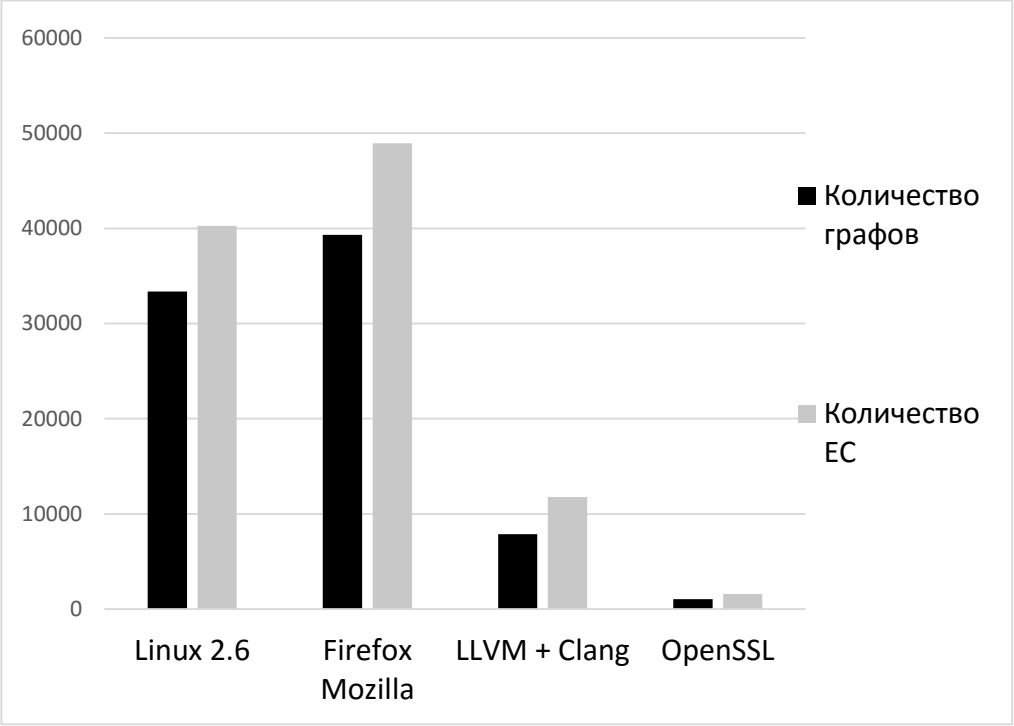

Рисунок 2.12. Количество полученных ЕС.

На рисунке 2.13 показано время работы алгоритма, разделяющего ГЗП на ЕС. Разделение ГЗП на ЕС для Linux 2.6 занимает только 72 секунды. Для Firefox Mozilla, LLVM/Clang и OpenSSL 46, 26, 2.3 секунды соответственно.

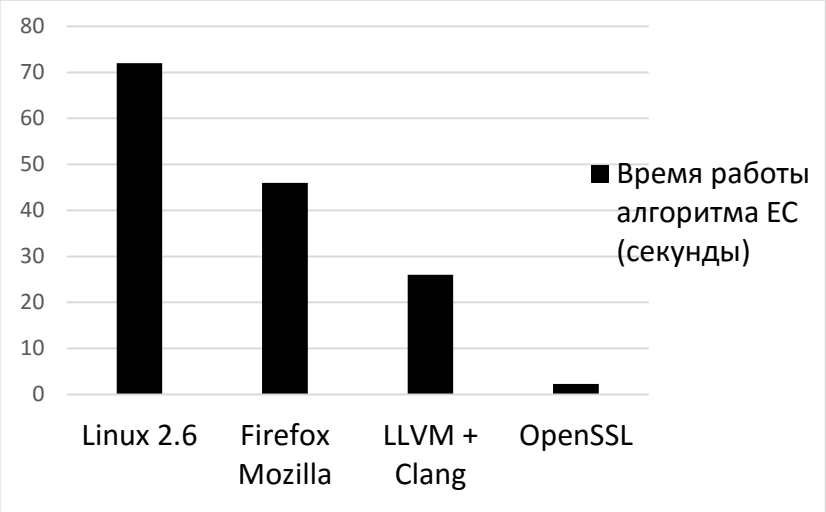

Рисунок 2.13. Время работы алгоритма, разделяющего ГЗП на ЕС.

2.8.3. Количество найденных клонов

На рисунке 2.14 приводится сравнение найденных клонов кода в зависимости от разделяющего алгоритма. Благодаря применению разработанного алгоритма, количество найденных клонов кода для ядра Linux 2.6 возросло с 913 (алгоритм IST) до 1965. Для проектов Firefox Mozilla, LLVM/Clang и OpenSSL количество найденных клонов возросло соответственно с 485, 62, 19 до 708, 134, 50. При применении разделяющего алгоритма WCC количество найденных клонов кода для Linux 2.6, Firefox Mozilla, LLVM/Clang и OpenSSL составляет 628, 351, 31 и 17 соответственно.

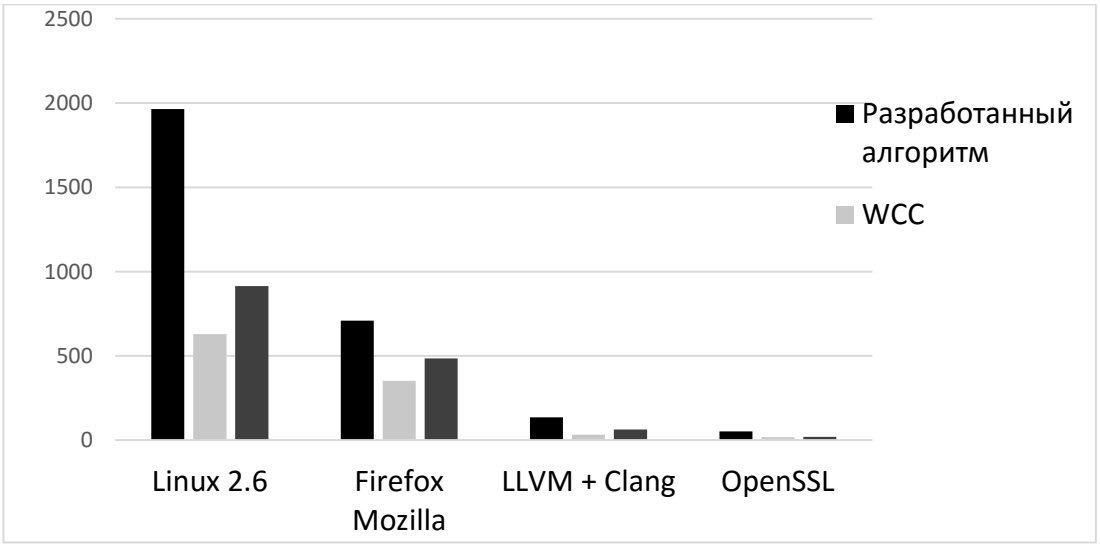

Рисунок 2.14. Сравнение разделяющих алгоритмов.

2.8.4. Результаты работы алгоритма на основе слайсинга

На рисунке 2.15 показано время работы алгоритма. Размер минимального клона – 25 строк, схожесть клонов – более 90 процентов. Ядро Linux 2.6 анализируется за 34.43 часа. Время анализа Firefox Mozilla, LLVM/Clang и OpenSSL составляет 15.81, 3.52 и 0.023 часа соответственно.

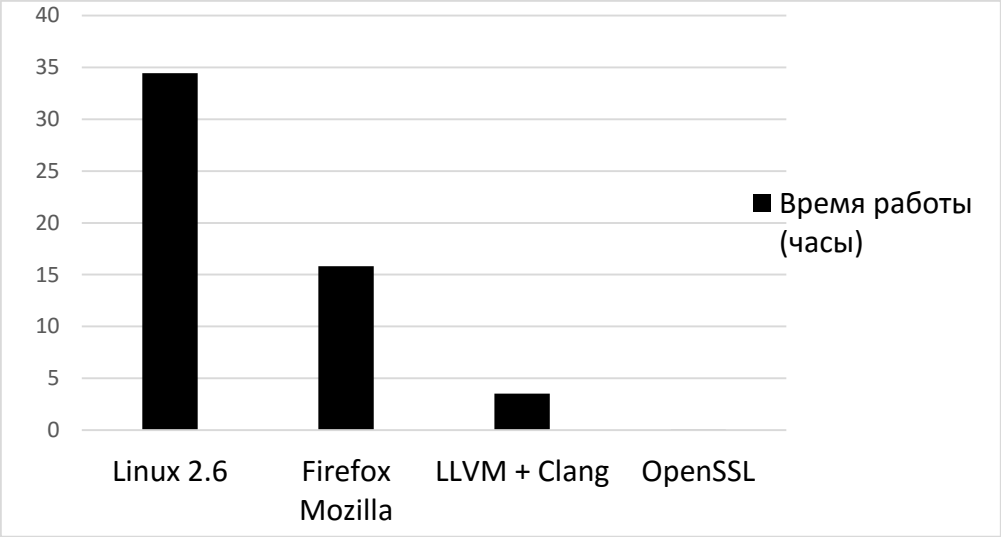

Рисунок 2.15. Время работы алгоритма поиска клонов кода.

Для ядра Linux 2.6 из 1965 найденных клонов только 73 оказались ложными. Для проектов Firefox Mozilla, LLVM/Clang и OpenSSL из найденных 708, 134 и 50 клонов ложными оказались соответственно 41, 6 и 3 (рисунок 2.16).

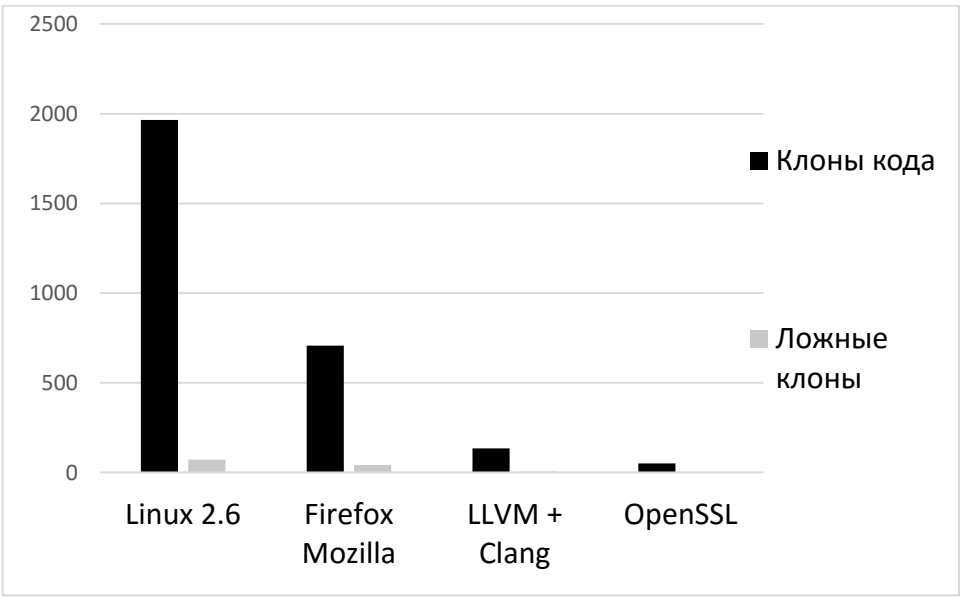

Рисунок 2.16. Количество найденных клонов и уровень ложных срабатываний.

Примеры клонов: Ниже (таблица 2.3) приведен пример найденного клона для проекта Firefox Mozilla.

Таблица 2.3. Пример найденной клон.

	клон 1	**клон 2**
firefox-29.0.1	**mozilla-release/intl/icu/source/common/ucnv_u16.c** **59: length=(int32_t)(pArgs->sourceLimit-source);** **60: if(length<=0) {** **61: /* no input, nothing to do */** **62: return;** **63: }** **64:** **65: cnv=pArgs->converter;** **66:** **67: /* write the BOM if necessary */** **68: if(cnv->fromUnicodeStatus ==** **UCNV_NEED_TO_WRITE_BOM) {** **69: static const char bom[]={ (char)0xfe, (char)0xff };** **70: ucnv_fromUWriteBytes(cnv,** **71: bom, 2,** **72: &pArgs->target, pArgs->targetLimit,** **73: &pArgs->offsets, -1,** **74: pErrorCode);** **75: cnv->fromUnicodeStatus=0;** **76: }** **77:** **78: target=pArgs->target;** **79: if(target >= pArgs->targetLimit) {** **80:** ***pErrorCode=U_BUFFER_OVERFLOW_ERROR;** **81: return;**	**mozilla-release/intl/icu/source/common/ucnv_u16.c** **658: length=(int32_t)(pArgs->sourceLimit-source);** **659: if(length<=0) {** **660: /* no input, nothing to do */** **661: return;** **662: }** **663:** **664: cnv=pArgs->converter;** **665:** **666: /* write the BOM if necessary */** **667: if(cnv->fromUnicodeStatus ==** **UCNV_NEED_TO_WRITE_BOM) {** **668: static const char bom[]={ (char)0xff, (char)0xfe };** **669: ucnv_fromUWriteBytes(cnv,** **670: bom, 2,** **671: &pArgs->target, pArgs->targetLimit,** **672: &pArgs->offsets, -1,** **673: pErrorCode);** **674: cnv->fromUnicodeStatus=0;** **675: }** **676:** **677: target=pArgs->target;** **678: if(target >= pArgs->targetLimit) {** **679:** ***pErrorCode=U_BUFFER_OVERFLOW_ERROR;** **680: return;**

2.8.5. Результаты работы алгоритма на основе изоморфизма деревьев

В данной секции представлены результаты работы алгоритма на основе изоморфизма деревьев. Размер минимального клона – 25, схожесть клонов – более 90 процентов. Ядро Linux 2.6 анализируется за 11.7 часа. Время анализа Firefox Mozilla, LLVM/Clang и OpenSSL составляет 7.9, 1.8 и 0.01 часа соответственно (рисунок 2.17).

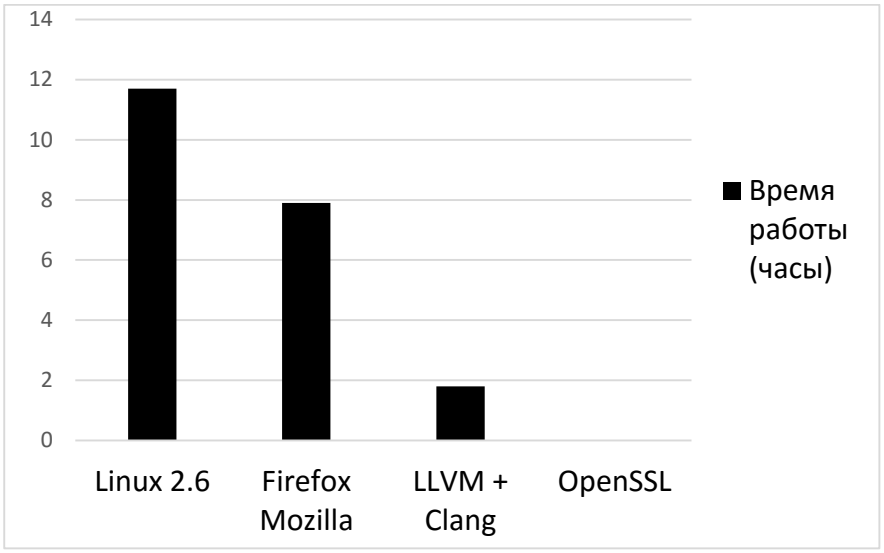

Рисунок 2.17. Время работы алгоритма поиска клонов кода.

Количество найденных клонов и уровень ложных срабатываний показан на рисунке 2.18. Для ядра Linux 2.6 из 1341 найденных клонов 31 оказались ложными. Для проектов Firefox Mozilla, LLVM/Clang и OpenSSL из найденных 408, 93 и 41 клонов соответственно ложными оказались 26, 13 и 5 соответственно.

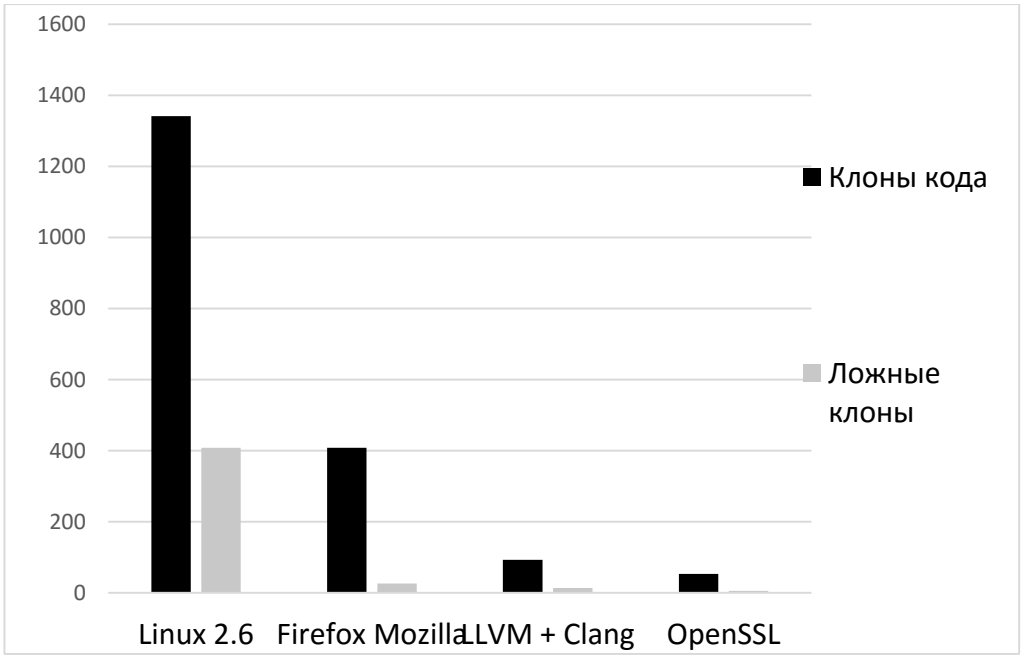

Рисунок 2.18. Количество найденных клонов и уровень ложных срабатываний.

2.8.6. Результаты работы алгоритма на основе метрик

Ниже (см. рисунок 2.19) представлены результаты работы алгоритмов поиска клонов на основе метрик. Размер минимального клона – 25 (минимум 25 совпадающих строк), схожесть клонов – более 90 процентов. На рисунке 2.15 показано время работы алгоритма поиска клонов кода. Ядро Linux 2.6 анализируется за 5340 секунд. Время анализа Firefox Mozilla, LLVM/Clang и OpenSSL составляет 2235, 1720 и 13 секунд соответственно.

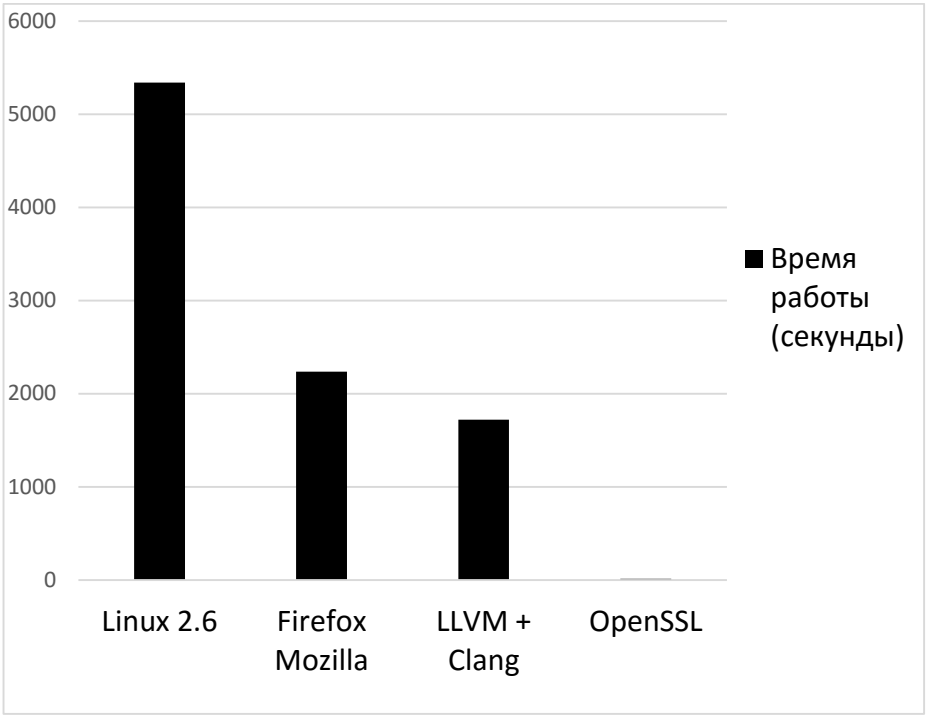

Рисунок 2.19. Время работы алгоритма поиска клонов кода.

Рисунок 2.20 показывает количество найденных клонов в проанализированных проектах и уровень ложных срабатываний. Все найденные клоны были проанализированы вручную. Для ядра Linux 2.6 из 127 найденных клонов только 6 оказались ложными. Для LLVM/Clang из 83 найденных клонов только 3 оказались ложными. Для Firefox Mozilla и OpenSSL было найдено 15 и 13 клонов соответственно, ложных клонов не было обнаружено.

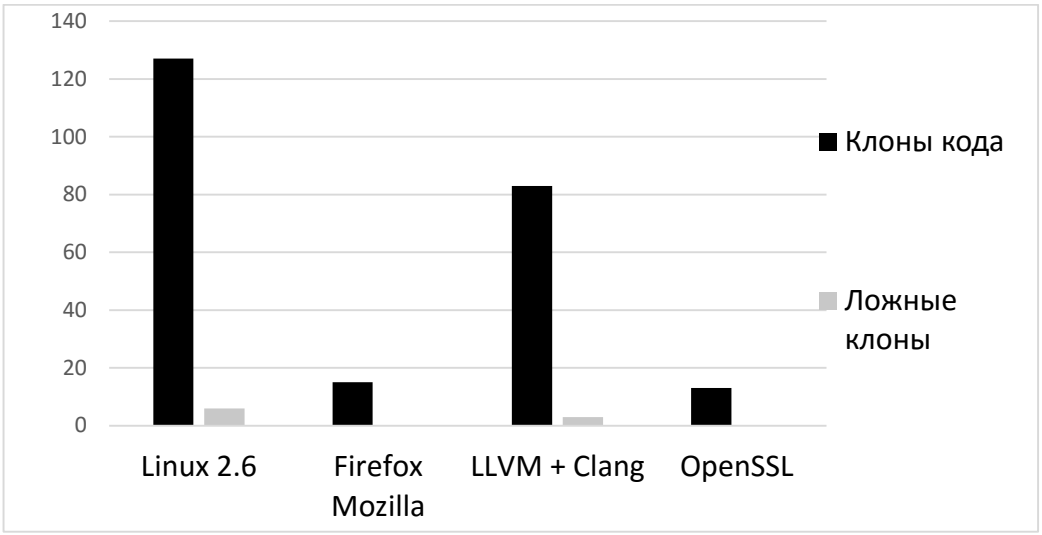

Рисунок 2.20. Количество найденных клонов и уровень ложных срабатываний.

2.8.7. Сравнение результатов работы реализованных алгоритмов

На рисунке 2.21 приведены результаты сравнения времени выполнения реализованных алгоритмов на основе метрик, слайсинга и изоморфизма деревьев. Из результатов видно, что дольше всех на всех тестах работает алгоритм на основе слайсинга. По сложности второе место занимает алгоритм на основе изоморфизма деревьев. Время работы этого алгоритма также включает время преобразования ГЗП в дерево. Быстрее всех работает алгоритм на основе метрик.

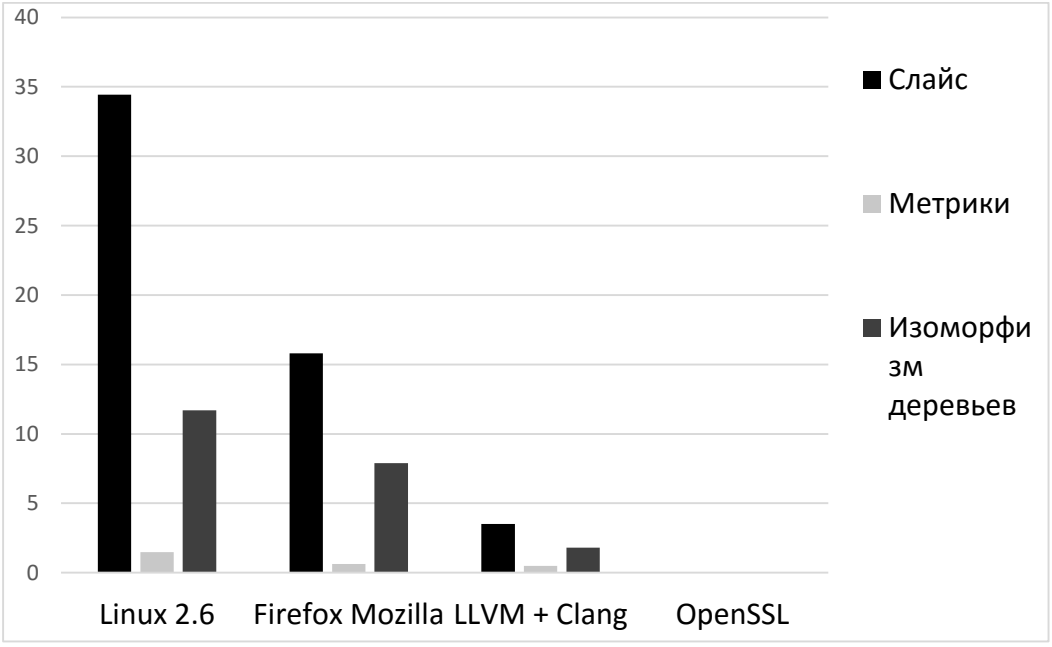

Рисунок 2.21. Сравнение времени работы.

Рисунок 2.22 показывает количество найденных каждым алгоритмом клонов. Более 90 процентов клонов кода, найденных алгоритмами на основе метрик и на основе деревьев, содержатся в клонах, найденных алгоритмом на основе слайсинга. Это еще раз показывает, что наибольшей точности можно достичь, используя метод поиска максимально изоморфных/схожих графов.

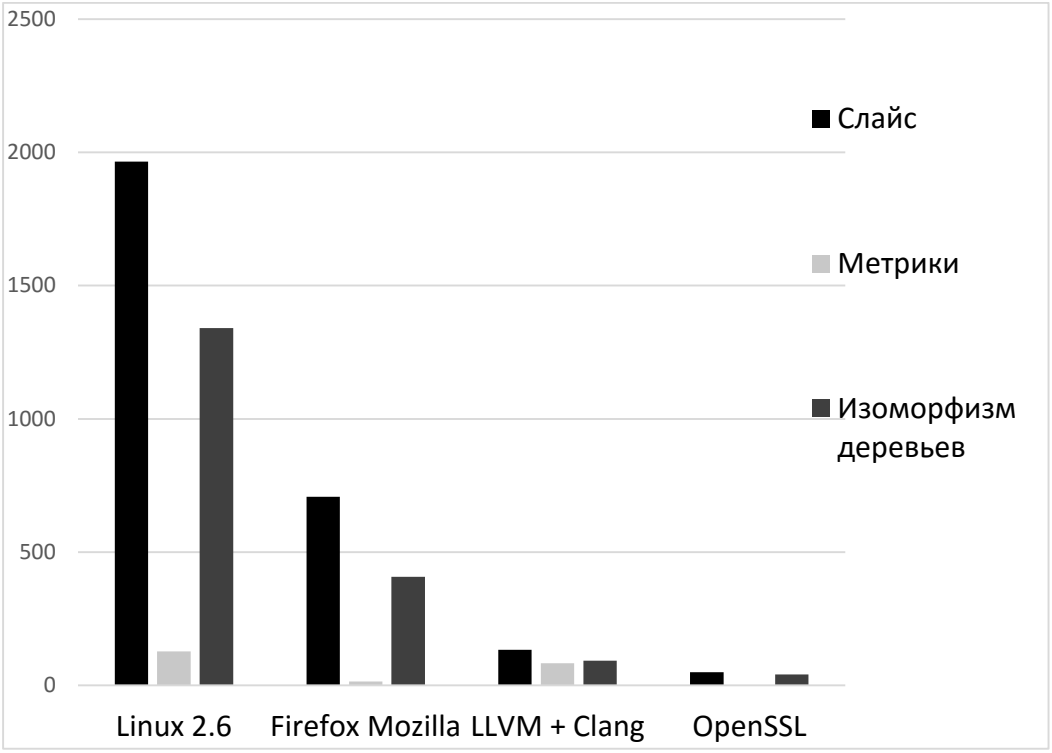

Рисунок 2.22. Сравнение найденных клонов.

Глава 3. Архитектура инструмента поиска клонов кода

Для построения инструмента поиска клонов кода программ в качестве базы выбран LLVM [93]. LLVM является компиляторной инфраструктурой с открытым исходным кодом на языке C++. В рамках этого проекта представлен инструментарий для разработки ПО. LLVM содержит статический компилятор, компоновщик, виртуальную машину и JIT-компилятор, которые работают над единым внутренним представлением программы (биткод). Биткод может быть представлен тремя разными формами: в текстовом виде; в виде структур данных в оперативной памяти; в двоичном виде. Инфраструктура LLVM дает возможность сохранить биткод в промежуточных объектных файлах для дальнейшей оптимизации, в том числе динамической. Все предоставляемые LLVM возможности по обработке внутреннего представления (в том числе различные анализы, преобразование и т.п.) могут быть применимы к биткоду. Поэтому компиляторная инфраструктура LLVM является удобной платформой для семантического анализа программы. Используя анализы LLVM из биткода, можно получить информацию о потоке данных и о потоке управления. Из биткода также можно получить информацию о номерах строк исходного кода, из которого он был построен, если доступно отладочная информация (при компиляции дана опция «-ggdb»).

Основными особенностями LLVM являются: реализация на Си++; модульная и расширяемая архитектура; статический компилятор с возможностью динамической компиляции биткода.

Для LLVM есть несколько компиляторов переднего плана: C, C++, Objective-C (Clang, GCC/dragonegg).

Генерация ГЗП проекта производится на основе биткода во время компиляции проекта. Поиск клонов кода производится отдельно. Для этого реализован отдельный инструмент, который на вход получает директорию с ГЗП графами и производит поиск клонов. Он также содержит систему автоматической генерации клонов кода, для анализа и улучшения разработанных алгоритмов. Реализованный инструмент поддерживает параллельную обработку для многоядерных систем. Одновременно могут быть обработаны несколько пар графов для определения клонов кода.

3.1. Схема поиска клонов кода

Наша модель инструмента поиска клонов кода основана на семантическом подходе, так как цель инструмента – найти все клоны с наибольшей точностью. Инструмент также должен быть масштабируемым для анализа проектов с исходным кодом в несколько миллионов строк. Он состоит из двух основных частей. Первая часть создает ГЗП-граф из внутреннего представления LLVM в ходе компиляции проекта (рисунок 3.1). Этот этап разработан как проход компиляторной инфраструктуры LLVM. Вторая часть инструмента отвечает за анализ ГЗП-графов в целях нахождения клонов. Поиск клонов кода производится в четыре этапа: разделение ГЗП на ЕС; фильтрование несхожих пар ЕС; поиск максимальных схожих подграфов; фильтрация ложных срабатываний. Вторая часть разработана как инструмент пакета LLVM (т.н. LLVM tool) [93].

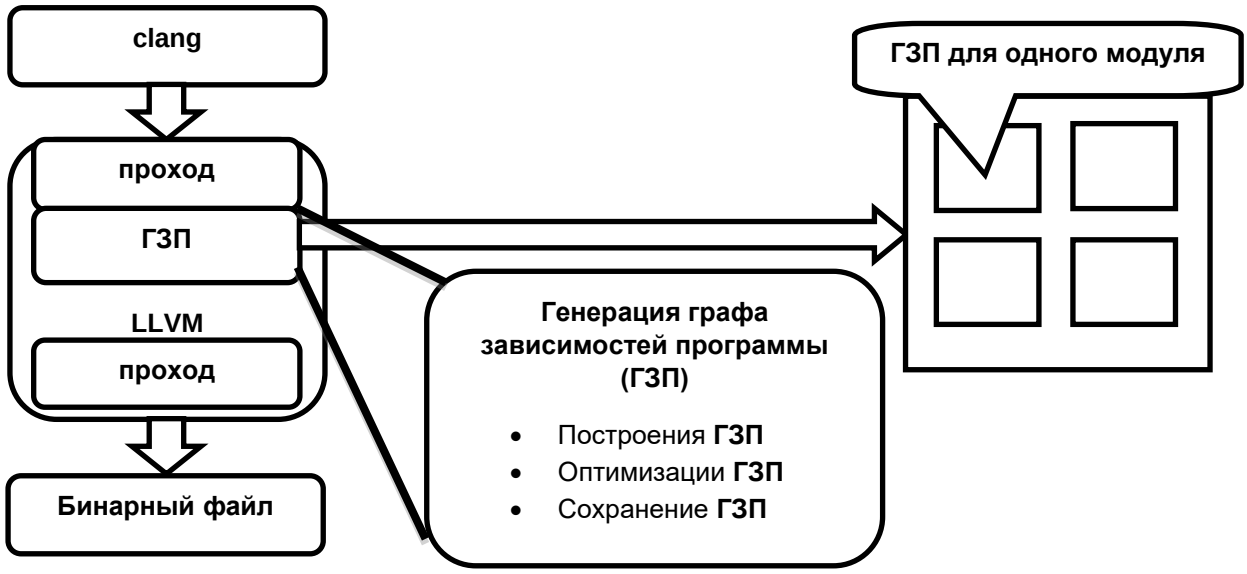

Рисунок 3.1. Архитектура инструмента: генерация ГЗП.

3.1.1. Схема генерации ГЗП

Генерация ГЗП обеспечивается компиляторным проходом LLVM на основе промежуточного представления LLVM (биткод). Вершинами графа являются инструкции биткода, ребра получаются путем анализа потока данных и потока управления (см. рисунок 3.2).

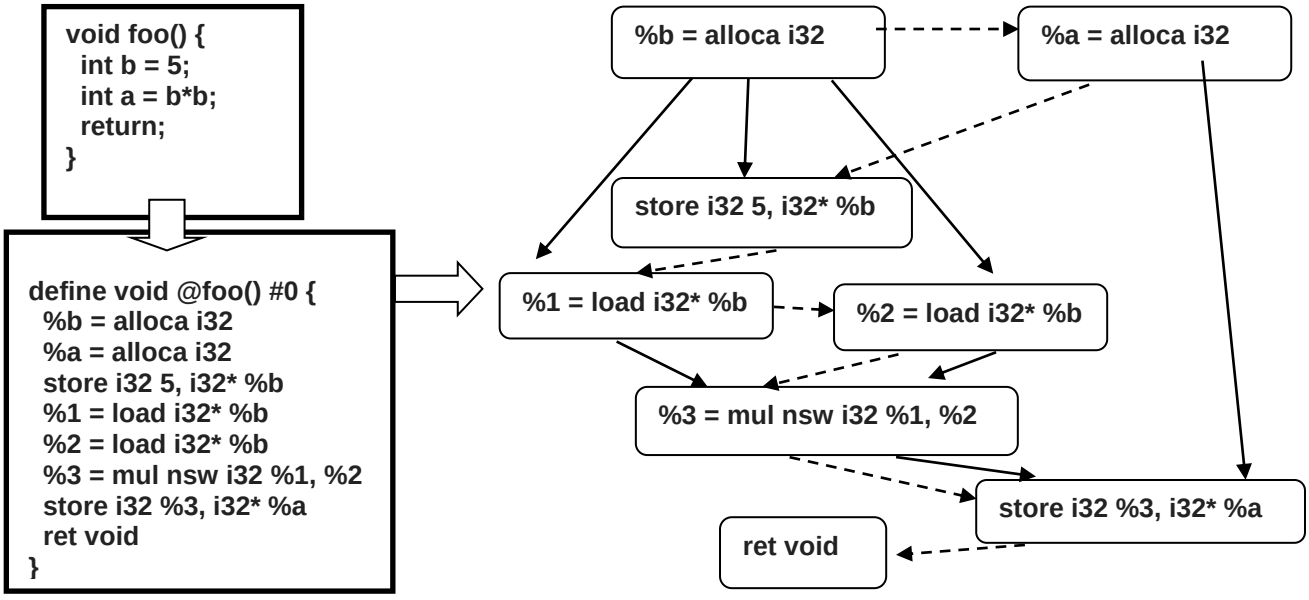

Рисунок 3.2. Пример ГЗП.

Инструмент обеспечивает генерацию ГЗП с тремя разными уровнями детализации, что позволяет эффективно искать клоны конкретного типа. В графе

первого уровня находится только ребра, полученные LLVM use-def анализом. Граф второго уровня содержит также ребра, полученные с использованием анализа алиасов. Максимальное количество информации имеется в графе третьего уровня детализации: в нем есть все ребра первого и второго уровней, а также ребра, отображающие зависимости по управлению. В задачах, где требуется быстро найти только клоны типов T1 и T2, используется граф первого или второго уровня. Для эффективного поиска клонов типа T3 необходимо использовать граф третьего уровня, что снижает скорость работы.

По умолчанию генерируется ГЗП первого уровня (только на основе LLVM use-def анализа). Для генерации графов второго и третьего уровней детализации предусмотрены отдельные опции пользователя.

Инструмент позволяет группировать вершины ГЗП по операциям и типам переменных. Существует три типа группировки вершин ГЗП:

1. вершины ГЗП, которым соответствуют логические операции, получают одинаковые метки (код операции соответствующей инструкции).
2. вершины ГЗП, которым соответствуют арифметические операции, получают одинаковые метки.
3. вершины ГЗП, которым соответствуют переменные программы, получают одинаковые метки, независимо от типов этих переменных.

При логической группировке вершины ГЗП, соответствующие разным логическим операциям, рассматриваются как идентичные. Если в клонированном участке кода одна логическая операция была изменена на другую, инструмент будет считать, что эти фрагменты полностью совпадают. В случае группировки по типу, изменение типов переменных не будет влиять на степень схожести фрагментов кода. Для каждого типа группировки вершин ГЗП предусмотрены отдельные опции пользователя.

После того как ГЗП будут построены, они оптимизируются и сохраняются в файлах. Под оптимизациями ГЗП подразумеваются следующие операции:

1. Удаление вершин, которые не имеют никаких ребер.

2. Удаление вершин, которым не соответствует исходный код. Такие вершины могут возникнуть из-за того, что биткод LLVM представляется в виде SSA формы.

Возможен поиск клонов кода в рамках нескольких проектов, для этого необходимо только поместить ГЗП этих проектов в одну директорию и запустить инструмент для них.

3.1.2. Разделение ГЗП на подграфы

После загрузки ГЗП в память, они разделяются на ЕС (рисунок 3.3). Все пары ЕС рассматриваются как потенциальные клоны друг друга. Разделение графа на ЕС должно производиться аккуратно, чтобы реальные клоны оказались в отдельных ЕС. В противном случае в ЕС попадет только часть реального клона, в результате чего клон найдется частично или вообще не будет найден. Задача состоит в том, чтобы в итоге каждый ЕС полностью содержал потенциальный клон.

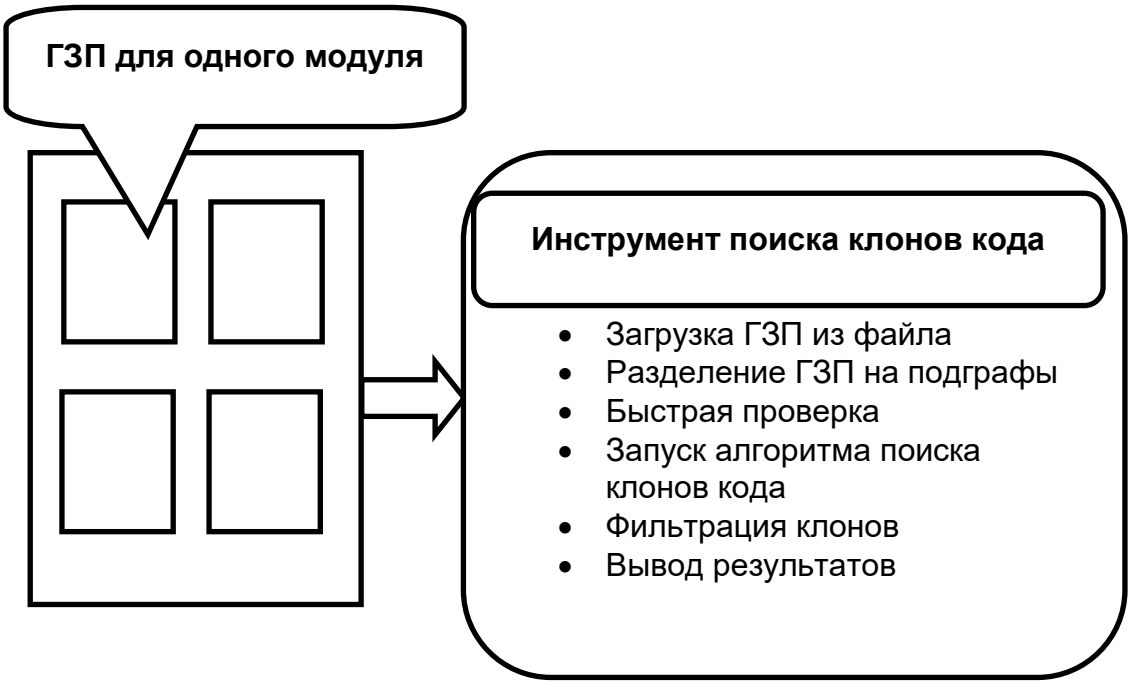

Рисунок 3.3. Архитектура инструмента: анализ ГЗП.

3.1.3. Схема поиска клонов кода

После разделения ГЗП на ЕС начинается процесс их сравнения в целях нахождения клонов. Инструмент содержит два типа алгоритмов сравнения. Первый тип алгоритмов проверяет пару ЕС на то, что они не являются клонами. Сложность таких алгоритмов – линейная, зависит от количества вершин в паре ЕС. Второй тип – это приближенные алгоритмы поиска схожих подграфов. Вычислительная сложность этих алгоритмов довольно велика, она может достигать кубической степени от вершин графа. Так как большинство пар ЕС не являются клонами, для них нецелесообразно будет применять алгоритмы второго типа. Таким образом, сначала будут запущены алгоритмы первого типа, которые за линейное время докажут, что большинство пар ЕС не являются клонами. Алгоритмы второго типа будут запущены для тех пар ЕС, которые не были обработаны алгоритмами первого типа. Больше 80 процентов пар ЕС (согласно нашим экспериментам) анализируются алгоритмами первого типа. Только для малой части запускаются тяжеловесные алгоритмы второго типа.

3.1.4. Фильтрация ложных срабатываний

Последний шаг в процессе поиска клонов кода – фильтрация ложных срабатываний. Найденные пары изоморфных подграфов дополнительно проверяются алгоритмами фильтрации. Необходимость применения фильтра возникает из-за того, что мы определяем понятие клона для исходного кода программы, а ищем клоны как изоморфные подграфы. Получается, что клон должен быть некой последовательностью строк в файле (не обязательно следующих одна за другой, но обязательно расположенных на незначительном расстоянии). Цель фильтрации состоит в том, чтобы проверить, что строки фрагмента кода, соответствующий схожему подграфу, находятся достаточно близко друг другу. Эту проверку нужно сделать после того, как схожие подграфы найдены; в противном случае, алгоритм поиска клонов может пропустить некоторые клоны кода.

3.2. Интегрированная система тестирования

Имеется отдельная опция, которая позволяет запустить инструмент в отладочном режиме. В этом случае автоматически генерируется база клонов для данного проекта, и для генерированных клонов запускаются реализованные алгоритмы поиска клонов кода. Точность разработанного алгоритма определяется количеством найденных клонов из базы. Предложены два подхода к автоматической генерации клонов кода.

Первый подход (рисунок 3.4) использует существующие стандартные проходы LLVM, а также специальные проходы, которые были разработаны в Институте ситемного программировании РАН [1] для обфускации биткода. Инструмент в режиме тестирования для каждой функции получает два графа: первый, оригинальный, получается на основе исходного промежуточного представления LLVM; второй граф – на основе преобразованного промежуточного представления. Алгоритм поиска клонов кода запускается для оригинального и преобразованного графа. Так как преобразования LLVM не меняют семантику программы, полученные графы должны быть найдены как клоны.

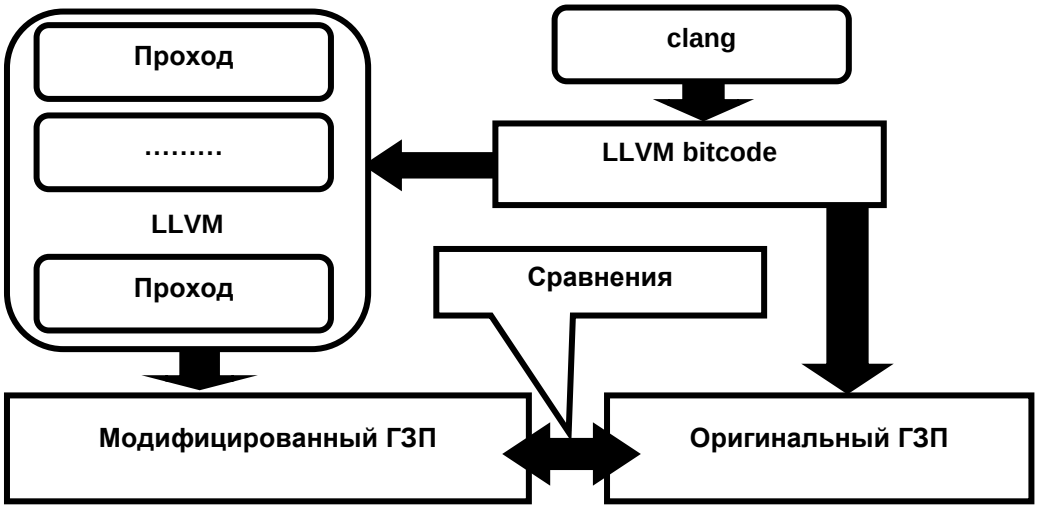

Рисунок 3.4. Схема генерации клонов кода.

Ниже приводится влияние обфусцирующих пробразование на вид ГЗП:

3.2.1. Влияние преобразования вызовов функций на ГЗП

Цель данного преобразования создать функцию-переходник для вызова функции. Вызов нужной функции в переходнике определяется на основе трудно вычисляемого предиката. Вызов изначальной функции в коде заменяется на вызов переходника. Такое преобразование почти не затрагивает поток управления и данных изначальной функции. Поэтому соответствующие ГЗП оригинальной и обфусцированной функции имеют незначительные различия, что позволяет предложенным алгоритмам (2.2, 2.3, 2.4) с легкостью найти сгенерированные клоны кода.

3.2.2. Влияние преобразования «диспетчер» на ГЗП

Базовым блокам функции присваиваются номера. В начале функции вставляется блок «диспетчер». Последовательности выполнения базовых блоков определяется диспетчером на основе информации переданной ей предыдущим выполненным блоком. Это преобразование затрагивает в основном поток управления и добавляет новые инструкции. ГЗП обфусцированной функции содержит дополнительные вершины и ребра по управлению. Новые ребра по данным существуют только между новыми инструкциями. Таким образом, ГЗП оригинальной функции содержится в графе обфусцированной функции как изомерный подграф (возможно, за исключением некоторых ребер по управлению). Такие клоны кода находятся при использовании алгоритмов на основе слайсинга (2.3) и изоморфизма деревьев (2.4).

3.2.3. Влияние преобразования строковых констант на ГЗП

Во время компиляции все константные строки, кроме тех, что содержатся в агрегатных типах (массивы, контейнеры из стандартной библиотеки), шифруются. Создаются специальные функции: шифровщик и дешифратор. Перед каждым использованием строки вызывается дешифратор, а после – шифрующая функция. Это преобразование в основном затрагивает поток данных. ГЗП обфусцированной функции содержит дополнительные вершины,

соответствующие вызовам шифровщика и дешифратора. Также содержит новые ребра по данным между парами вершин, соответствующие инструкциям константной строки и вызова шифровщика/дешифратора. В этом случае ГЗП оригинальной функции содержится в ГЗП обфусцированной функции как изомерный подграф. Алгоритмы на основе слайсинга (2.3) и изоморфизма деревьев (2.4) определяют, что обфусцированная функция является клоном.

3.2.4. Влияние преобразования графа потока управления на ГЗП

В графе потока управления образуются несводимые участки путем добавления «фиктивных» ребер из заголовков циклов в их тела. Для добавления таких ребер генерируются непрозрачные предикаты, на основе которых добавляются фиктивные ребра. Такое преобразование в большей части затрагивает поток управления. ГЗП обфусцированной функции содержит новые вершины, соответствующие добавленным непрозрачным предикатам и ребра по управлению. Новые ребра по данным могут быть добавлены только между вершинами, соответствующим непрозрачным предикатам. Все три разработанных алгоритма (2.2, 2.3, 2.4) находят обфусцированную функцию как клон оригинала.

3.2.5. Влияние переплетения циклов на ГЗП

Для всех циклов функции добавляются «фиктивные» ребра из одного в другой. Фиктивные ребра получаются на основе непрозрачных предикатов. Такое преобразование в основном влияет на поток по управлению. Но также может затрагивать поток данных в зависимости от использованных непрозрачных предикатов. ГЗП обфусцированной функции отличается от оригинального как ребрами потока данных и управления, так и новыми вершинами. Только алгоритм на основе слайсинга (2.3) определяет, что обфусцированная функция является клоном оригинала.

3.2.6. Влияние переплетения функций на ГЗП

Переплетение функций состоит из четырех основных этапов. На первом шаге объединяются сигнатуры двух функций и генерируется дополнительный параметр, на основе которого будет работать код нужной функции. На втором шаге объединяются базовые блоки обоих функций в переплетенную. Создается предикат на основе сгенерированного дополнительного параметра, который решает, код какой функции должен работать. На третьем этапе генерируются специальные базовые блоки, которые будут работать при выполнении кода обоих функций. На последнем этапе вызов переплетенных функций заменяется на вызов созданной функции. Такое преобразование значительно меняет ребра потока данных и управления. Добавляет множество новых вершин. ГЗП оригинальной и обфусцированной функции имеют сильные отличия. Только алгоритм на основе слайсинга (2.3) способен обнаружить такие клоны.

3.2.7. Влияние усложнения анализа потока данных на ГЗП

Локальные переменные переносятся в глобальную область видимости. Производится специальный анализ графа вызовов функций, который позволяет определить, изменение какой глобальной переменной в какой функции не будет влиять на корректную работу программы. После этого в каждой функции производится некоторые вычисления с теми глобальными переменными, которые в данном точке можно «испортить». Такое преобразование влияет на потоки данных и управления. Основная структура ГЗП обфусцированной функции не меняется. Все три алгоритма (2.2, 2.3, 2.4) находят такие клоны. Алгоритмы на основе метрик (2.2) и изоморфизма деревьев (2.4) имеют сравнительно низкую точность и могут пропускать некоторые клоны данного вида.

3.2.8. Влияние разбиения целочисленных констант на ГЗП

Положительные константы в коде программы разбиваются на слагаемые произвольным образом. Вместо использования оригинальной константы используется сумма разбитых чисел. Данное преобразование в основном касается

потока данных, также добавляются новые инструкции. Обычно ГЗП обфусцированной функций не сильно отличается от оригинальной ГЗП. Все три (2.2, 2.3, 2.4) алгоритма находит такие клоны с достаточно высокой точностью.

3.2.9. Влияние размножения тел функции на ГЗП

Для каждого вызова функции генерируется копия тела функции, после чего каждый вызов оригинальной функции заменяется на вызов соответствующей копии. Это преобразование не касается потока данных и управления, поэтому ГЗП копированных функции изоморфны друг другу. Все три (2.2, 2.3, 2.4) алгоритма находят такие клоны с большой точностью.

3.2.10. Влияние вставки ложных циклов на ГЗП

В ходе пробразования выбираются последовательность базовых блоков, где первый блок имеет одно входящее ребро, а последний – одно выходящее. Для создания ложного цикла добавляется «фиктивное» ребро из последнего блока в первый базовый блок. Это преобразование в основном влияет на поток управления и добавляет новые инструкции. Оно может влиять на поток данных, если непрозрачный предикат будет использовать переменные программы. Обычно ГЗП оригинальной и обфусцированной функций не сильно отличаются, поэтому алгоритмы на основе слайсинга (2.3) и изоморфизма деревьев (2.4) находят такие клоны.

3.2.11. Влияние формирования непрозрачных предикатов на ГЗП

Для формирования непрозрачных предикатов используются выражения, значение которых всегда ложно или истинно. Например, выражение $(x \text{ ^ } 3 - x)\ mod\ 3 = 0$ всегда истинно. Данное преобразование отдельно не используется, его используют другие преобразования для добавления фиктивных ребер. Оно в основном касается потока управления и добавляет новые инструкции. Если в ходе преобразования многократно используются такие предикаты, тогда ГЗП обфусцированной функции может сильно отличаться от оригинального графа.

Второй подход по списку ГЗП проекта строит последовательность пар ГЗП и объединяет каждую пару в один граф (рисунок 3.5). Алгоритм поиска клонов кода сравнивает ГЗП оригинального и объединенного списков.

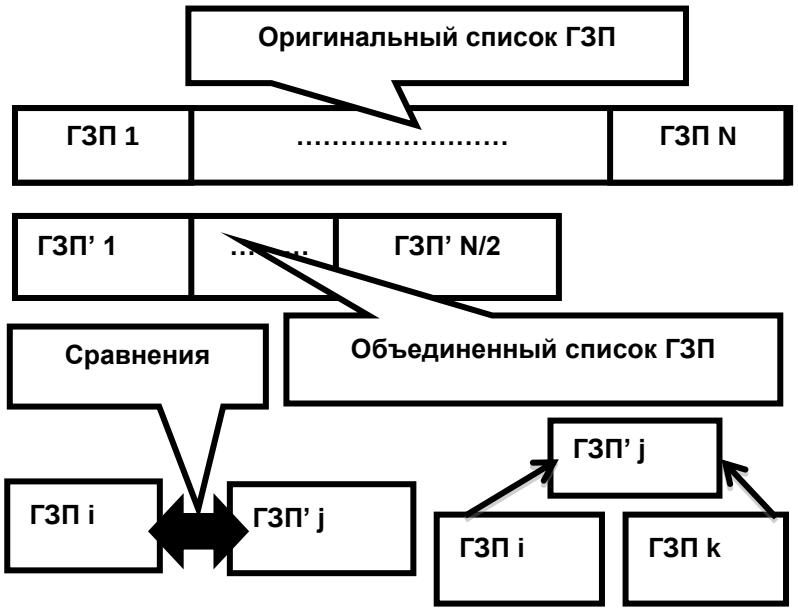

Рисунок 3.5. Схема генерации клонов кода.

3.2.12. Объединение ГЗП

Для объединения используются три разных метода. Первый метод объединяет два графа без добавления новых вершин и ребер. Второй метод объединяет два графа и произвольным образом добавляет ребра между вершинами этих графов. Третий метод (см. **Алгоритм UN** ниже) с помощью семантического анализа находит подходящие подграфы во втором графе и добавляет их в первый.

Алгоритм UN: На вход получает пару ГЗП G_1, G_2, числа P и H. P – максимальное количество подграфов, которые из G_2 будут добавлены в G_1. H – максимальное количество вершин каждого подграфа, выбранного из G_2.

1. Поочередно берутся вершины из G_1 (обозначим V), если в G_2 есть вершина U с индексом как у V, переходить на шаг 2. В противном случае, пропускать V.
2. Запускается обход в ширину для графа G_2 с вершины V. Если в ходе обхода было выбрано H вершин, обход прекращается, и выделанный подграф

добавляется в G_1 (вершина V заменяется на вершину U в выделенном подграфе). В противном случае, в G_1 добавляется подграф, полученный после обхода в ширину.

3. Если в ходе выполнения шага 2 было добавлено P подграфов из G_2 в G_1, алгоритм завершает работу.

3.2.13. Оценка точности реализованных алгоритмов

В данном разделе приводятся результаты тестирования точности реализованных алгоритмов поиска клонов кода на основе слайсинга (раздел 2.3), изоморфизма деревьев (раздел 2.4) и метрик (раздел 2.2). Для проектов Linux 2.6, Firefox Mozilla, LLVM/Clang и OpenSSL автоматически генерируется множество клонов кода. После этого алгоритмы поиска клонов кода запускаются для оригинальной и генерированной пары ГЗП. На рисунке 3.6 приведен усредненный процент точности реализованных алгоритмов по всем проектам.

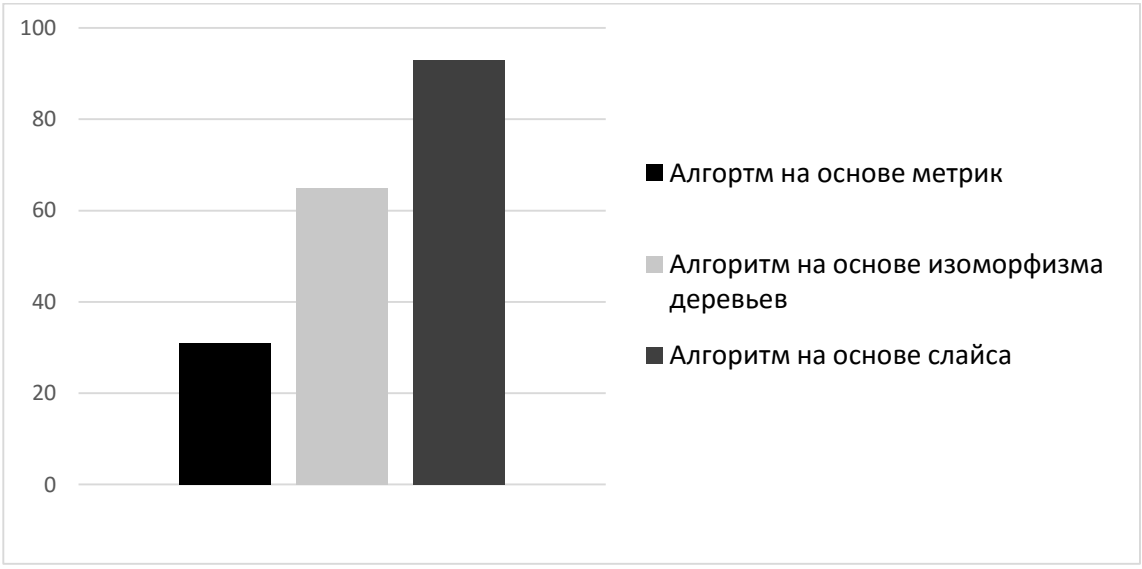

Рисунок 3.6. Сравнение точности реализованных алгоритмов.

3.3. Запуск на многоядерных системах

Описывается схема запуска инструмента на многоядерных системах и анализ полученных результатов. Файлы, содержащие ГЗП проекта, разделяются

на x групп в зависимости от количества ядер и размера оперативной памяти машины.

Утверждение 4: Чтобы система была полностью загружена $x \geq \frac{-1+\sqrt{1+8p}}{2}$, где p – количество ядер.

Доказательство: Предположим, что целевая машина имеет p процессоров, и файлы, содержащие ГЗП проекта, необходимо разделить на x групп, чтобы все p процессоры имели задачи для выполнения. В случае разделения файлов на x групп в целом будет $x+\frac{x(x-1)}{2}$ задач, x – задачи для проверки внутри каждой группы, $\frac{x(x-1)}{2}$ – задачи для проверки всех неповторяющихся пар групп. Количество задач должно быть равным количеству процессоров.

$x+\frac{x(x-1)}{2}=p$, из этого следует $2x+x(x-1)=p$, получается, что для нахождения x надо решить следующее квадратичное уравнение:

$x^2+x-2p=0$.

$D=\sqrt{1+8p}$, отсюда получается $x=\frac{-1+\sqrt{1+8p}}{2}$, отрицательное решение $x=\frac{-1-\sqrt{1+8p}}{2}$ не рассматривается, так как количество групп должно быть положительным числом, что и доказывает сформулированное выше утверждение.

Если файлы, соответствующие паре групп, не помещаются в оперативной памяти, количество групп увеличивается настолько, чтобы любые две группы помещались в оперативной памяти. Сначала производится поиск в рамках ГЗП одной группы файлов, после чего рассматриваются все пары групп файлов, и производится сравнение ГЗП соответствующих групп. Такой подход позволяет параллельно анализировать большие проекты.

В инструменте содержатся два специальных скрипта для анализа найденных клонов. Первый скрипт позволяет просмотреть исходный код клонов и

соответствующие им графы (рисунок 3.7). Скрипт поддерживает сохранение найденных клонов в HTML и EXCEL форматах.

Рисунок 3.7. Инструмент просмотра результатов.

Второй скрипт дает возможность проследить историю копирования конкретного фрагмента кода. Также позволяет интерактивно продвигаться по дереву файлов, содержащему клоны данного фрагмента (рисунок 3.8).

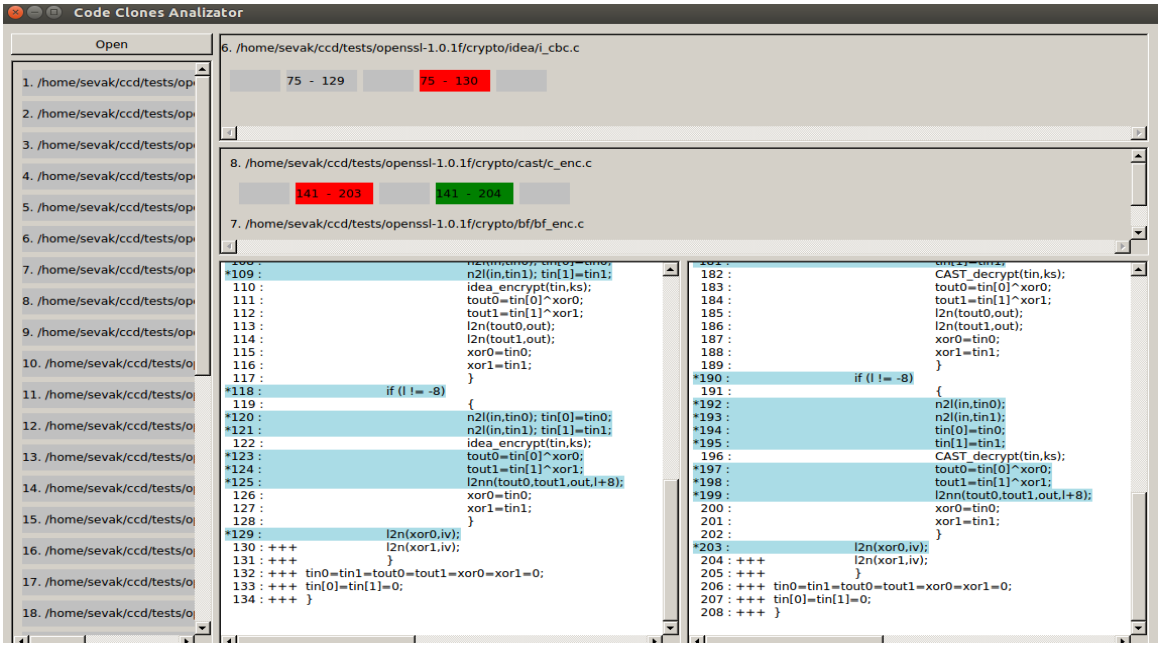

Рисунок 3.8. Инструмент просмотра результатов.

Глава 4.
Поиск клонов кода для языка JavaScript

В последние годы широкое распространение получили программы на скриптовых языках. На сегодняшний день одним из популярных скриптовых языков является JavaScript. Благодаря росту производительности персональных компьютеров и встраиваемых систем JavaScript стало возможно использовать для программирования веб-приложений. Уже существуют операционные системы для сотовых телефонов и планшетов, в которых язык JavaScript является одним из основных языков для разработки приложений (Tizen[94] и FirefoxOS[95]). В таких системах пользовательское приложение представляет собой набор хранящихся на устройстве веб-страниц, написанных на языке JavaScript. В связи с этим все больше возрастают требования к производительности программ-скриптов. Быстрый рост количества JavaScript приложений ставит вопрос об эффективном выполнении таких приложений. Вопрос решается применением динамической (Just-In-Time) компиляции. Некоторые большие компании, такие как Google, Mozilla и Apple, предлагают собственные динамические компиляторы для языка JavaScript. Динамический компилятор работает во время выполнения программы, и время, затраченное им на компиляцию, добавляется к общему времени выполнения программы. Благодаря собранной информации о профилях работающей функции динамические компиляторы могут оптимизировать «горячие» участки кода во время выполнения скриптов.

Быстрый рост количества исходного кода приложений, написанных на JavaScript, также приводит к возникновению все большего количества клонов кода. До последнего времени JavaScript считался интерпретируемым языком, и для него ранее не производились сложные анализы. Оптимизирующие динамические компиляторы, использующие промежуточное представление языка JavaScript появились относительно недавно. Поэтому на сегодняшний день не существует инструмента поиска клонов кода для языка JavaScript, который был бы основан на семантическом анализе программы. Существующие инструменты основаны на текстовом, лексическом или синтаксическом анализе, что не позволяет достичь наибольшей точности. В данной главе представлен метод поиска клонов кода для языка JavaScript на основе семантического анализа программы. В качестве базы инструмент использует V8 JIT [96] компилятор компании Google. На основе промежуточного представление (Hydrogen) V8 получается набор ГЗП для скриптов. Поиск клонов кода производится путем применения разработанного инструмента поиска клонов кода (см. главу 3).

4.1. Архитектура динамического компилятора V8

Динамический компилятор языка JavaScript V8 [96] состоит из двух отдельных компиляторов (рисунок 4.1). Первый компилятор, Full-Codegen, предназначенный для быстрой генерации машинного кода, генерирует неоптимизированный бинарный код по АСД исходного кода. Во время интерпретации неоптимизированного кодасобирается профильная информация (например, информация о типе переменных). Функции, которые вызываются часто («горячие» функции), передаются второму компилятору (Crankshaft) вместе с собранной информацией о профиле программы. Crankshaft генерирует промежуточное представление каждой «горячей» функции в SSA форме (оно называется Hydrogen). После того как все оптимизации над представлением Hydrogen выполнены, оно транслируется в машинно-зависимое представление

Lithium. Это представление используется для распределения регистров, после чего генерируется бинарный код.

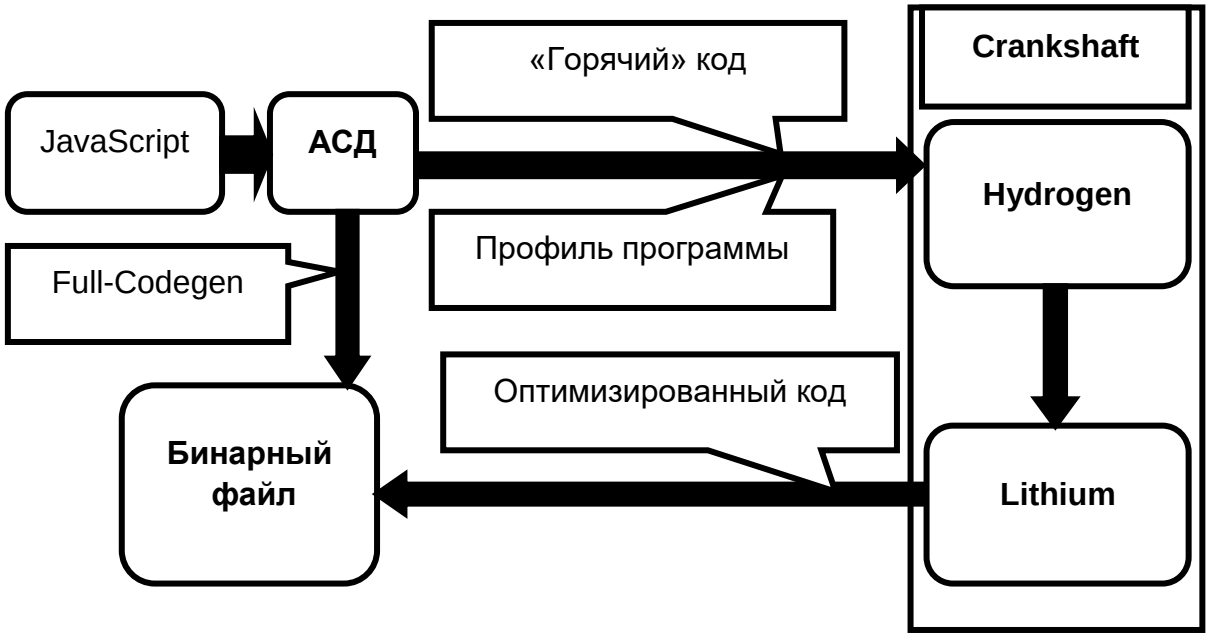

Рисунок 4.1. Архитектура динамического компилятора V8.

4.2. Построение графа зависимостей программы по представлению Hydrogen

4.2.1. Промежуточное представление Hydrogen

Представление Hydrogen по своей структуре похоже на промежуточное представление LLVM. Оно представляет собой граф потока управления, состоящий из базовых блоков. Базовые блоки содержат последовательности инструкций в SSA форме. Каждая инструкция помимо операндов содержит список инструкций, использующих ее результат. Таким образом, представление Hydrogen содержит все зависимости между инструкциями как по данным, так и по управлению.

4.2.2. Граф зависимотей программы

Граф зависимостей программы **(**ГЗП) содержит зависимости по данным и по управлению. ГЗП – направленный граф, в котором вершинами являются

инструкции программы, а ребра соответствуют зависимостям между инструкциями. Различаются два типа ребер: ребра первого типа представляют зависимости по управлению, ребра второго тиа – зависимости по данным. Как известно, есть три типа зависимостей по данным:

1. Истинная зависимость. Значение, записанное в память первой инструкцией, читается во второй.
2. Антизависимость. Значение, записанное в память второй инструкцией, читается в первой.
3. Зависимость по выходу. Обе инструкции записывают значение по одному и тому же адресу.

4.2.3. Построение графа зависимостей программы по представлению Hydrogen

Для каждой инструкции представления Hydrogen строится новая вершина ГЗП. Тип (метка) новой вершины ГЗП определяется кодом операции соответствующей инструкции представления Hydrogen. В вершине ГЗП сохраняется ссылка на исходный код инструкции. Между двумя вершинами ГЗП добавляется зависимость по данным, если между соответствующиими инструкциями представления Hydrogen имеется зависимость по данным. Тип зависимости отмечается на добавленном ребре. Между двумя вершинами ГЗП добавляется зависимость по управлению, если инструкция, соответствующая первой вершине, выполняется перед инструкцией, соответствующей второй вершине (рисунок 4.2).

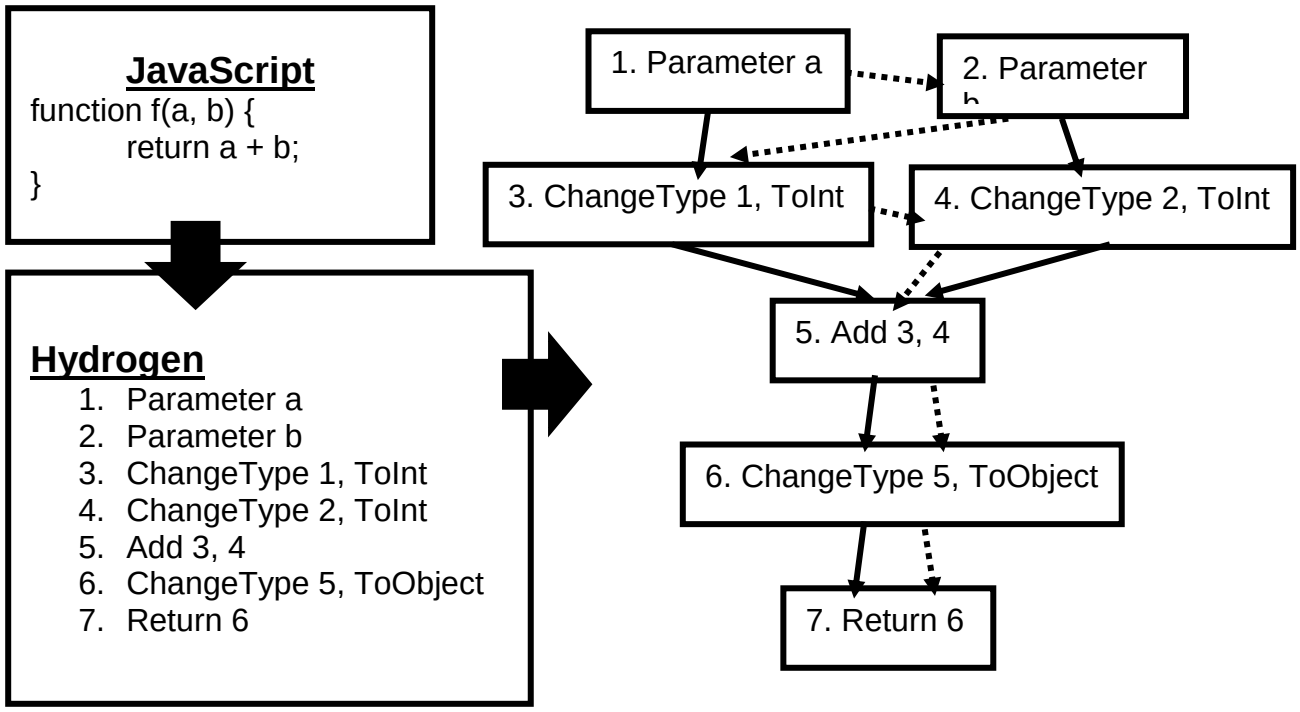

Рисунок 4.2. Пример трансляции представления Hydrogen в ГЗП.

На рисунке 4.3 показаны изменения, которые были сделаны в динамическом компиляторе V8, чтобы сгенерировать ГЗП для приложений, написанных на языке JavaScript.

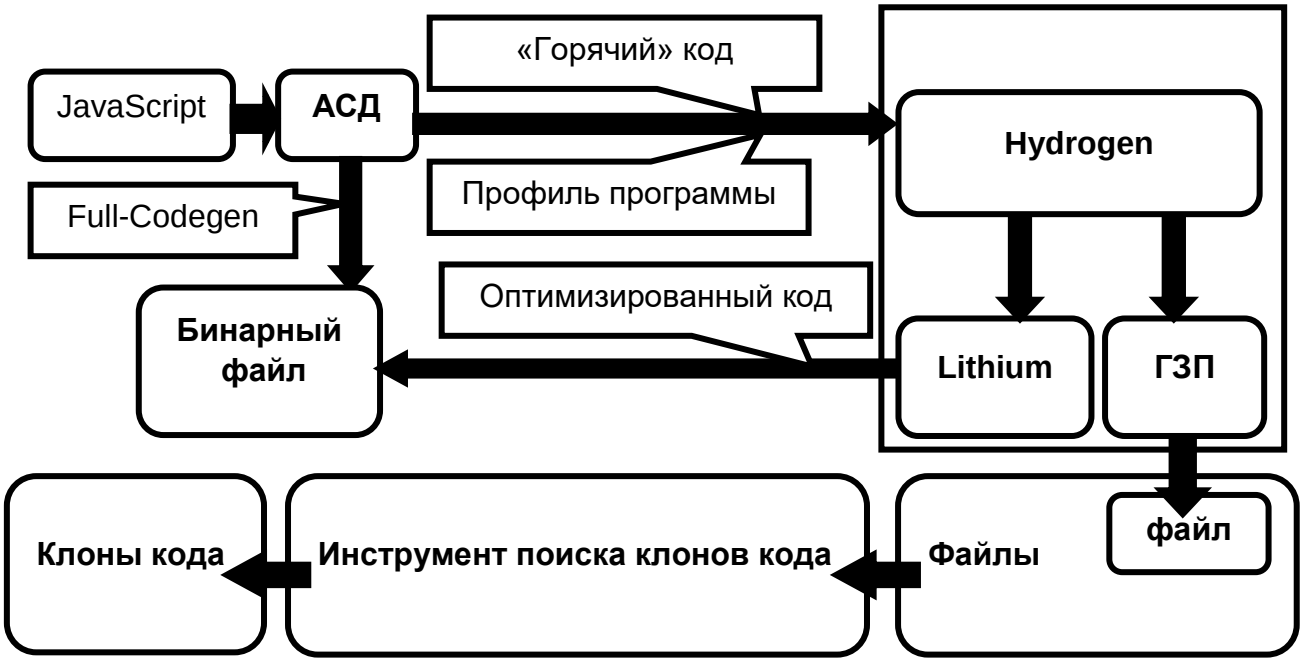

Рисунок 4.3. Инструмент поиска клонов кода для языка JavaScript на основе динамического компилятора V8.

Полученный ГЗП сохраняется в файле. Инструмент поиска клонов кода (см. главы 2, 3) запускается для сгенерированных ГЗП файлов, как только компилятор V8 заканчивает свою работу.

Следует отметить, что формат ГЗПточно такой же, что и формат ГЗП для языков C/C++, генерируемый компилятором LLVM. Это дает возможность использовать разработанные инструменты поиска клонов кода и для языка JavaScript.

4.3. Результаты тестирования

Тестирование инструмента для JavaScript проводилось на Intel Core i3 CPU 540 с 8 ГБ оперативной памяти. Размер минимального клона составлял 10 строк исходного кода. Были проанализированы тестовые наборы SunSpider, Kraken и Octane.

Тестовый набор SunSpider создали разработчики браузера WebKit (компания Apple). В нем содержатся различные вычисления из реальных проектов:

1. графические вычисления;
2. доступ и операции с разными полями объектов;
3. бытовые операции;
4. рекурсивные вызовы;
5. криптографические вычисления;
6. операции с объектами, представляющими время (DateTime objects);
7. вычисления, связанные с регулярными выражениями;
8. строковые операции.

Тестовый набор Kraken, разработанный компанией Mozilla, содержит следующие тесты:

1. алгоритм поиска A*;
2. обработка аудио (звука);
3. обработка изображений;

4. разбор данных в формате JSON;
5. обработка криптографических данных.

Тестовый набор Octane разработан компанией Google. В нем содержатся тесты для проверки пропускной способности интерактивных веб приложений и сборки мусора. Ниже приведен список тестов:

1. Симуляция ядра операционной системы. Содержит функционалы, отвечающие за загрузку объектов, вызов функции, оптимизацию кода и удаление повторного кода.
2. Тесты ориентированы на полиморфизм и ООП.
3. Операции с регулярными выражениями, полученные на основе анализа более 50 популярных веб-страниц.
4. Решение уравнений 2D. Содержит чтение и запись массивов, а также операции с числами с плавающей точкой.
5. Криптографические вычисления, содержащие битовые операции.
6. Автоматическое управление памятью, которое позволяет быстро создавать и удалять объекты.
7. Тесты, измеряющие задержку сборщика мусора.
8. 3D симуляция.
9. Загрузка кода: измеряет время разбора и компиляции JavaScript.

На рисунке 4.4 показано количество строк проанализированных проектов. Octane содержит более 350.000 исходного кода, написанного на JavaScript.

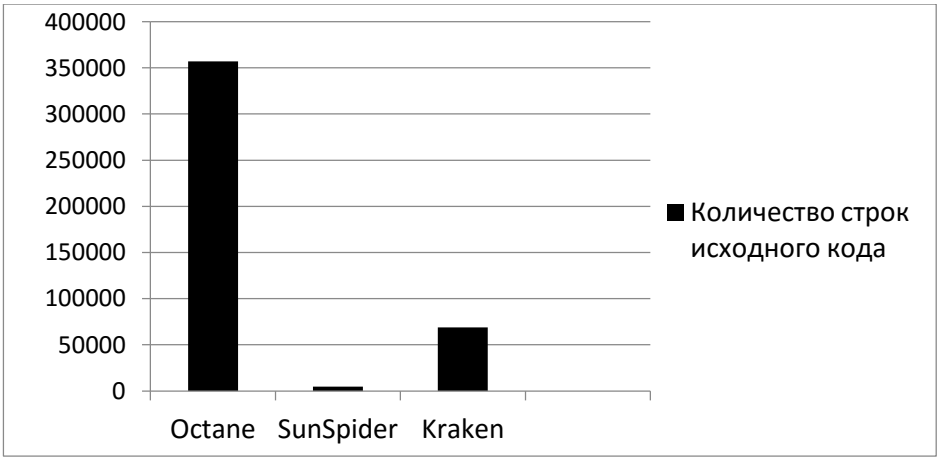

Рисунок 4.4. Количество строк исходного кода проанализированных проектов.

На рисунке 4.5 показан размер ГЗП проанализированных проектов. Для Octane он составляет 26.2 мегабайта, для SunSpider и Kraken 0.8 и 2.5 соответственно.

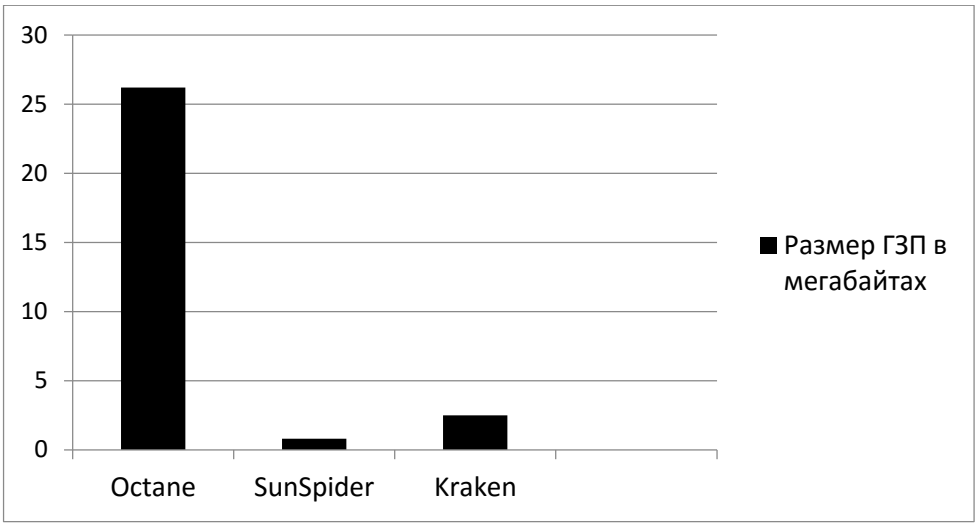

Рисунок 4.5. Размер ГЗП анализированных проектов

На рисунке 4.6 показано время поиска клонов кода. Для Octane оно составляет 428 секунд, для SunSpider и Kraken 19.4 и 14.2 секунды соответственно.

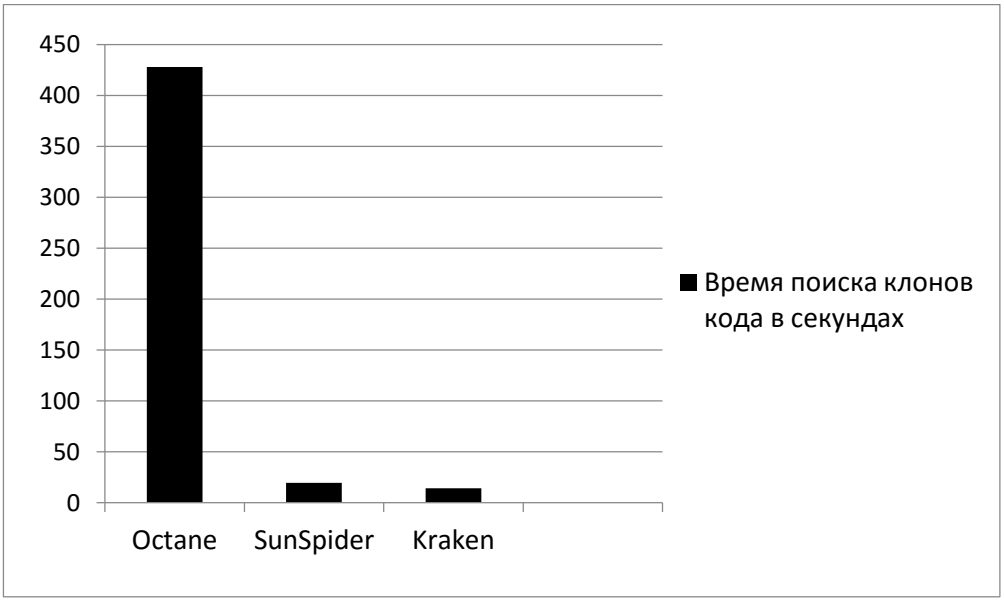

Рисунок 4.6. Время поиска клонов кода проанализированных проектов.

На рисунке 4.7 показано количество найденных клонов и количество ложных срабатываний. Инструмент нашел 342 и 10 клонов для Octane и SunSpider соответственно. SunSpider имеет одно ложное срабатывание. Kraken не содержит ни одного клона.

Рисунок 4.7. Количество найденных клонов и ложных срабатываний.

4.4. Сравнение разработанного инструмента с инструментом CloneDR

Разработанный инструмент поиска клонов кода для языка JavaScript был сравнен с инструментом CloneDR [92]. Тестирование проводилось на наборе тестов Octane. Для обоих инструментов размер минимального клона составлял 10 строк исходного кода, а схожесть – более 90 процентов. CloneDR нашел 35 клонов, а разработанный инструмент 342. Оба инструмента имели 21 совпадающий клон.

Глава 5.
Методы поиска семантических ошибок

Семантические ошибки, связанные с копированием кода, возникают из-за неполной адаптации скопированного фрагмента кода к контексту. Чаще всего ошибки возникают из-за не обновленных имен переменных в скопированном участке. Предложенный подход сначала с помощью лексического анализа определяетклоны кода типов T1 и T2, после чего применяет семантический анализ для выявления допущенных ошибок.

5.1. Схема работы инструмента

Представленный метод поиска семантических ошибок состоит из двух основных этапов. На первом этапе обнаруживаются все клоны типов T1 и T2 с помощью лексического анализа функции. Второй этап строит ГЗП для этой функции, чтобы найти ошибки, допущенные при копировании. Генерация лексем и построение ГЗП проекта производится во время компиляции проекта (рисунок 5.1). Новый проход LLVM [8] строит последовательность лексем на основе промежуточного представления. Для каждой функции производится поиск клонов. Клонами считаются совпадающие последовательности лексем.

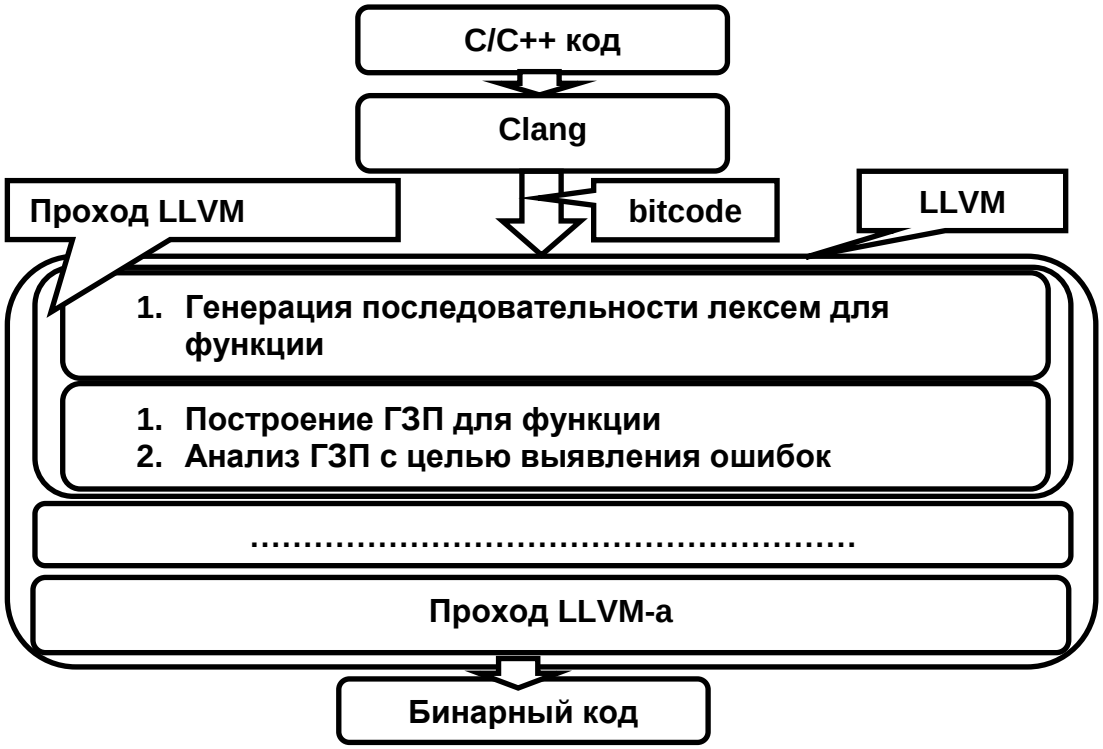

Рисунок 5.1. Поиск клонов кода на базе компиляторной инфраструктуры LLVM.

Основная причина возникновения семантических ошибок – некорректное переименование переменных при адаптации скопированного участка кода. Лексический анализ не чувствителен к именам переменных, что помогает найти скопированные участки кода, содержащие не переименованные переменные (семантические ошибки). При использовании АСД и ГЗП, имена переменных могут повлиять на вид дерева и графа (рисунок 5.2).

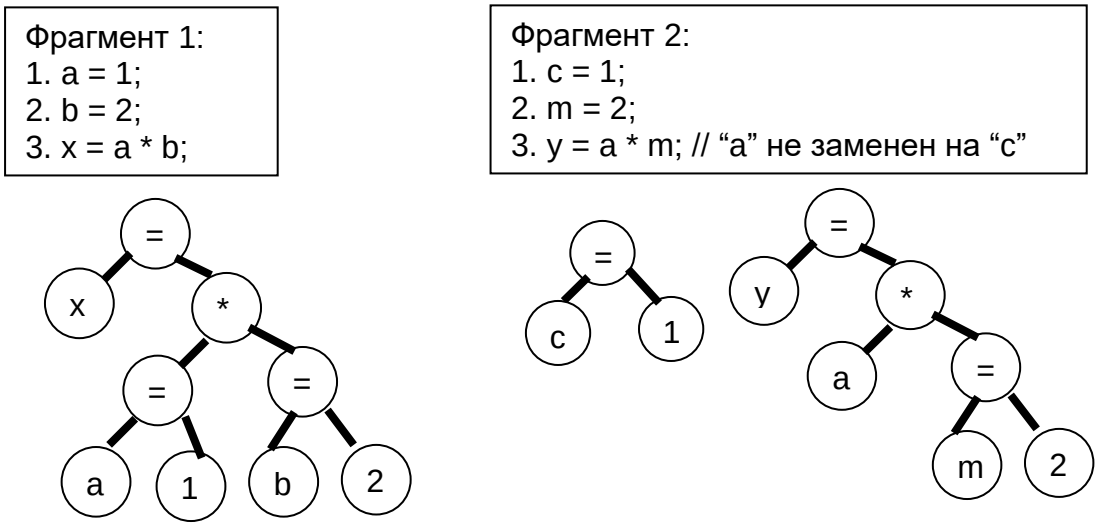

Рисунок 5.2. Влияние имен переменных на вид АСД после некорректного переименования.

Для функции, в которой найден клон, строится ГЗП для проверки корректности копирования.

Такой подход имеет ряд преимуществ. Он позволяет без повторного анализа исходного кода получить лексемы и графы для больших проектов, содержащих миллионы строк исходного кода. Не требуется проводить дополнительный анализ зависимостей между единицами компиляции.

По сравнению с существующими методами [40, 87], предложенный подход обладает большей точностью, благодаря тому, что использует представление ГЗП, содержащее всю необходимую информацию о программе, для поиска ошибок.

5.2. Поиск клонов кода на основе лексического анализа

На первом этапе на основе промежуточного представления LLVM получается последовательность лексем функции (рисунок 5.3). После этого производится поиск всех непересекающихся пар идентичных подпоследовательностей максимального размера.

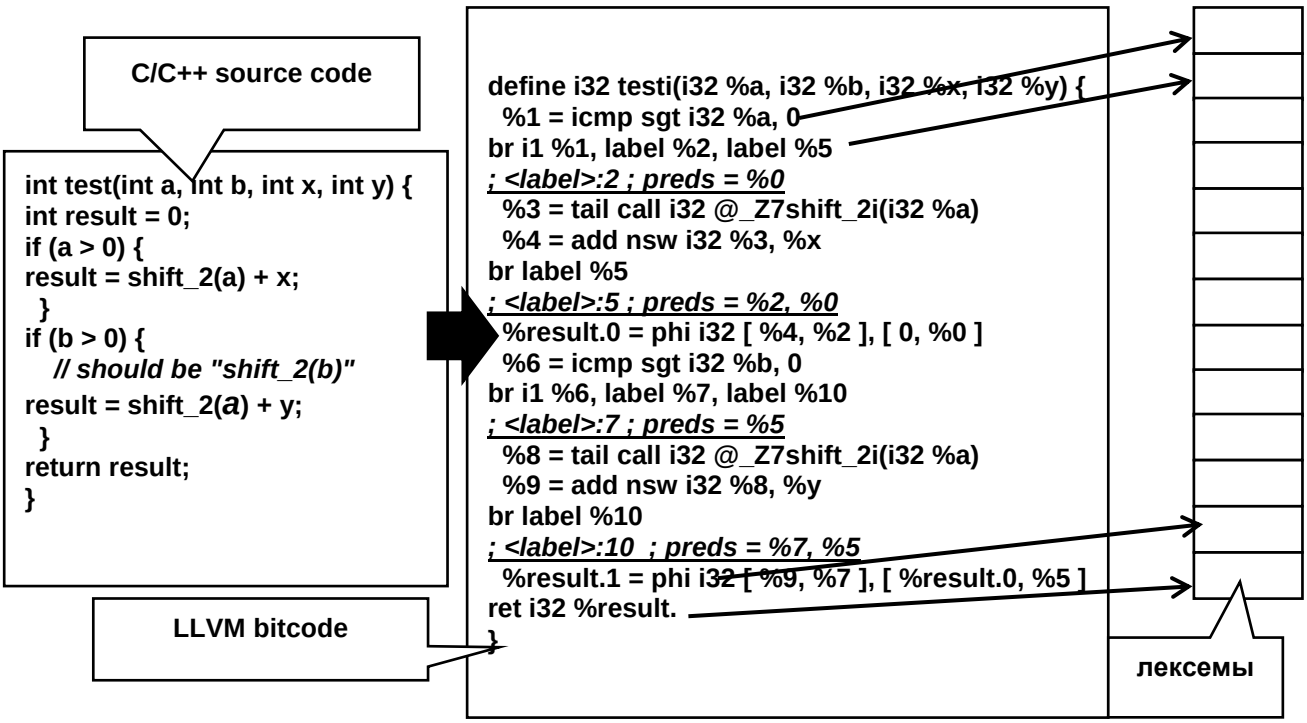

Рисунок 5.3. Построение последовательности лексем.

Для найденных идентичных последовательностей выполняется частичный разбор, чтобы определить синтаксически полные конструкции языка. Идентичные последовательности фильтруются от неполных конструкций: если в последовательность попала только часть оператора, она удаляется. Если отфильтрованные идентичные последовательности имеют достаточно большой размер, они передаются на второй этап для проверки.

5.3. Поиск семантических ошибок

Для поиска ошибок в скопированном фрагменте кода строится ГЗП соответствующей функции. В полученном графе выделяются два подграфа, соответствующие идентичным последовательностям лексем. Полученные подграфы расширяются путем добавления вершин (назовем эти вершины исходными вершинами), соответствующих переменным, которые используются в выделенных подграфах (рисунок 5.4).

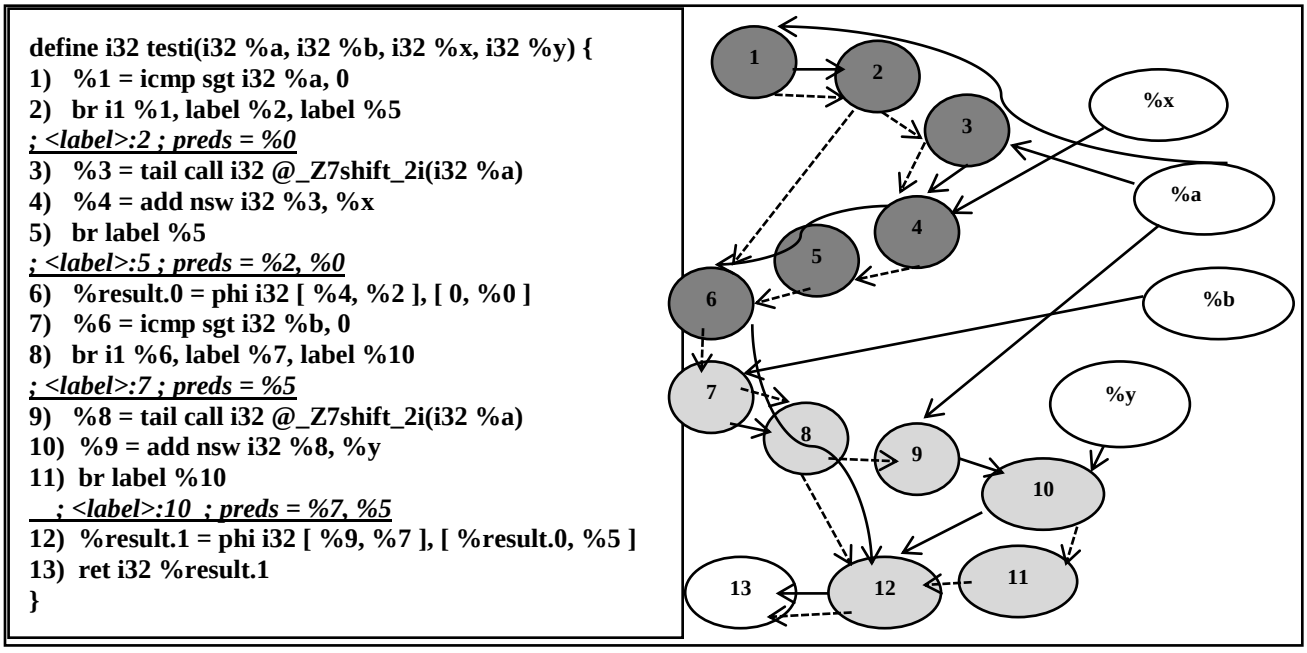

Рисунок 5.4. ГЗП для функции. {1, 2, 3, 4, 5, 6, %x, %a} и {7, 8, 9, 10, 11, 12, %a, %b, %y} выделенные подграфы.

Если выделенные подграфы не изоморфны, значит, при копировании с большой вероятностью произошла ошибка, поэтому необходим дополнительный анализ. Анализ исходных вершин дает информацию о возможной ошибке.

Если есть исходные вершины, которые входят в выделенные подграфы и в каждом подграфе имеют разную степень (рисунок 5.5), тогда переменные, соответствующие этим вершинам, содержат ошибочное использование.

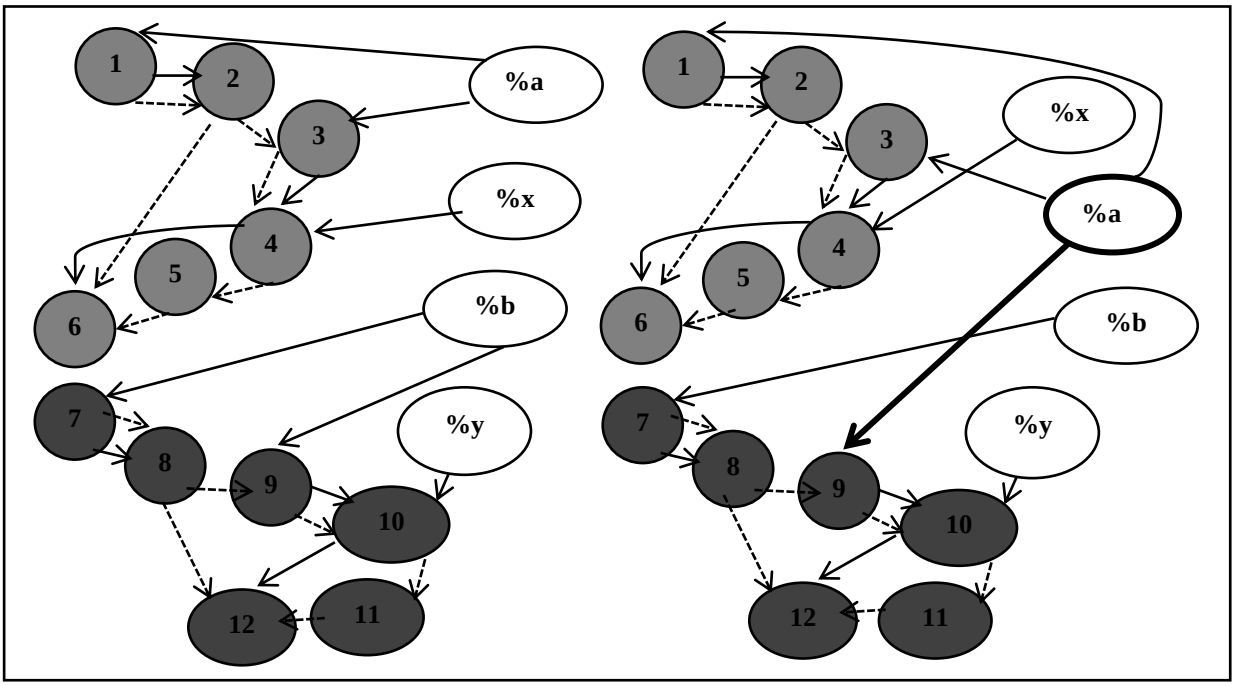

Рисунок 5.5. Изменение структуры ГЗП при неправильном копировании кода.

5.4. Изоморфизм пары графов

Для проверки изоморфизма пары ГЗП сравниваются характеристические числа, вычисленные для ребер соответствующих графов. Каждая вершина ГЗП имеет метку (численную), которая представляет собой код операции соответствующей инструкции промежуточного представления LLVM. Характеристическое число ребра получается на основе меток инцидентных вершин и степени одной вершины (рисунок 5.6).

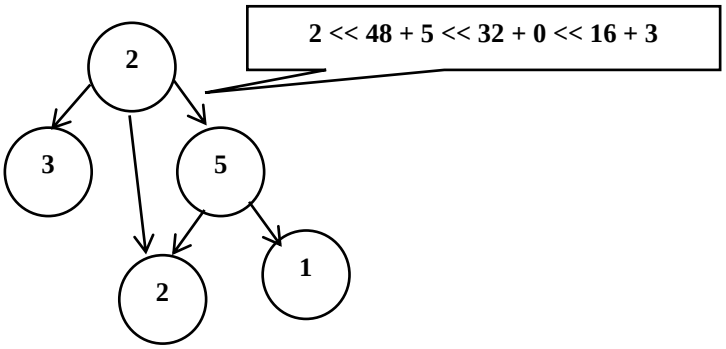

Рисунок 5.6. Характеристическое число ребер ГЗП.

Определение 7: *Характеристическое число* ребра (V,U) графа G – это $M(V,U) = id(V) * 2^{48} + id(U) * 2^{32} + inDeg(V) * 2^{16} + outDeg(V)$, где $id(V)$ – метка вершины V графа G и $0 \le id(V) < 2^{16}$, $inDeg(V)$ – количество входящих ребер в вершину V, $outDeg(V)$ – количество ребер, выходящих из вершины V.

Утверждение 5: Множества характеристических чисел ребер двух изоморфных графов равны.

Доказательство: Предположим, что графы G и H изоморфны: $f : G \to H$, где f – взаимно-однозначное отображение (каждому элементу одного множества соответствует ровно один элемент другого множества, при этом существует обратное отображение, которое обладает тем же свойством). Для любого ребра (v,u) графа G существует ребро $(f(v), f(u))$ из H, такое, что будут удовлетворены следующие условия:

1) $id(v) = id(f(v))$.
2) $id(u) = id(f(u))$.
3) $inDeg(v) = inDeg(f(v))$
4) $outDeg(v) = outDeg(f(v))$

Из приведенных выше четырех условий следует, что $M(v,u) = M(f(v), f(u))$. Из этого получается: $\{M(v,u) \mid (v,u) \in G\} \subseteq \{M(v,u) \mid (v,u) \in H\}$ (1).

$f : G \to H$ является взаимно-однозначным отображением, следовательно, существует взаимно-однозначное отображение $g : H \to G$, которое является

обратным отображением $f: G \to H$ и удовлетворяет условиям 1-4. Исходя из этого, аналогичным образом получается:

$\{M(v,u) \mid (v,u) \in H\} \subseteq \{M(v,u) \mid (v,u) \in G\}$ (2).

Учитывая (1) и (2), получаем $\{M(v,u) \mid (v,u) \in G\} = \{M(v,u) \mid (v,u) \in H\}$, что и требовалось доказать.

Из рисунка 5.6 видно, что характеристическое число ребра представляется в виде 64-битного целого числа. Первые 16 бит – метка первой вершины, следующие 16 бит – метка второй вершины, после этого следует количество входящих и выходящих ребер начальной вершины, которым также выделено по 16 бит. Алгоритм проверки изоморфизма вычисляет и сохраняет характеристические числа каждого графа в отдельном множестве, после чего проверяет равенство полученных множеств. Если они не равны, тогда пара ГЗП не может быть изоморфна.

5.5. Результаты тестирования

На рисунке 5.7 представлены результаты анализа нескольких проектов с открытым исходным кодом. Проекты Linux 2.6, Firefox Mozilla, LLVM/Clang и OpenSSL описаны в разделе 2.8. Android [97] является операционной системой для смартфонов, планшетных компьютеров, электронных книг, цифровых проигрывателей, наручных часов, игровых приставок, нетбуков, телевизоров и других устройств. На нем работает более миллиарда смартфонов и планшетов. Изначально разрабатывалась компанией Android Inc., которая затем была преобретена компанией Google. Qemu [98] обеспечивает эмуляцию аппаратного обеспечения различных платформ. Например, бинарный код, работающий на архитектуре ARM, можно запустить на X86, используя Qemu. Благодаря использованию техники бинарной трансляции, Qemu обеспечивает высокую производительность..

Для Android 4.3, Linux 2.6, Firefox Mozilla, LLVM/Clang ,OpenSSL и Qemu было найдено 53, 102, 6, 20, 3 и 3 ошибок соответственно. Из них ложными оказались соответственно 17, 33, 1, 7, 1 и 0.

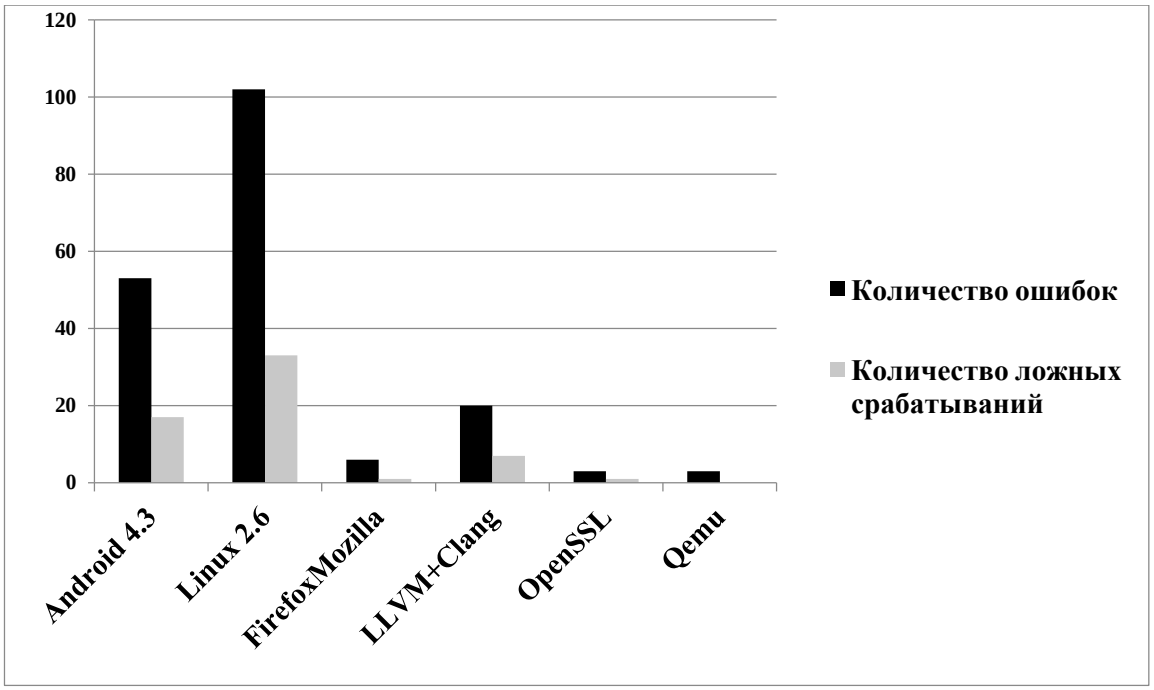

Рисунок 5.7. График найденных семантических ошибок и ложных срабатываний.

На рисунке 5.8 приведен пример семантической ошибки. Разработчик не переименовал переменную 'userName' в 'password' (строка 277) после копирования.

Android 4.3 - AccountSetupBasics.java

```
-------------------------------------------------------------
271|       String userName = SetupData.getUsername();
272|       if (userName != null) {
273|           mEmailView.setText(userName);
274|           SetupData.setUsername(null);
275|       }
-------------------------------------------------------------
276|       String password = SetupData.getPassword();
277|       if (userName != null) {
278|           mPasswordView.setText(password);
279|           SetupData.setPassword(null);
280|       }
```

Рисунок 5.8. Пример найденной семантической ошибки.

Заключение

В диссертации:

1. Проведен анализ существующих методов поиска клонов кода и поиска семантических ошибок, возникающих при неправильном копировании исходного кода.
2. Предложен метод построения набора ГЗП проекта на основе промежуточного представления компиляторной инфраструктуры LLVM, что позволяет получать набор ГЗП во время компиляции проекта без дополнительных накладных расходов.
3. Предложен метод построения ГЗП проекта на основе промежуточного представления Hydrogen JIT компилятора V8.
4. Предложен метод разделения ГЗП на подграфы, позволяющий увеличить количества ненайденных истинных клонов.
5. Предложен четырехфазный метод поиска клонов кода: (1) разделение ГЗП на подграфы, (2) отсеивание пар ГЗП, не содержащих клонов, алгоритмами линейной сложности, (3) обнаружение максимальных схожих подграфов с помощью слайсинга, (4) фильтрация ложных срабатываний. Метод масштабируется до десятков миллионов строк кода.
6. Предложено два алгоритма поиска максимальных изоморфных подграфов. (1) ГЗП преобразуется в дерево и находятся максимальные изоморфные поддеревья. (2) Используется специальная метрика (см. раздел 2.5).
7. Предложена схема автоматической генерации клонов кода, которая позволяет производить оценку точности реализованных алгоритмов.

8. Предложен метод обнаружения семантических ошибок в неверно клонированных участках кода. Применение комбинированного подхода, обеспечивает высокую точность метода.

На основе предложенных методов разработан и реализован инструмент поиска клонов кода; инструмент поиска семантических ошибок в клонированных фрагментах кода. Разработанные инструменты включены в компиляторную инфраструктуру LLVM.

Среди направлений дальнейшей работы по данной тематике можно выделить наиболее важные:

1. Поиск уязвимостей на основе существующих шаблонов. Идея заключается в построении базы ГЗП для программ, содержащих уязвимость. Для поиска уязвимостей в проекте производится поиск схожих подграфов из базы.
2. Применение инструмента в задачах оптимизации размера кода. Идея заключается в том, чтобы инструмент автоматически создавал функцию для группы клонов типа T1 или T2 и заменял клоны вызовами этой функции.

Литература

1. Курмангалеев Ш., Корчагин В., Савченко В., Саргсян С. Построение обфусцирующего компилятора на основе инфраструктуры LLVM // Труды Института системного программирования РАН. 2012 Т. 23. С. 77-92.
2. Sargsyan S., Kurmangaleev S., Baloian A., Aslanyan H. Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph // Mathematical Problems of Computer Science. 2014. Т. 42. С. 54-62.
3. Саргсян С., Курмангалеев Ш., Белеванцев А., Асланян А., Балоян А. Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ // Труды Института системного программирования РАН. 2015. Т. 27. № 1. С. 39-50.
4. Саргсян С. Поиск семантических ошибок, возникающих при некорректной адаптации скопированных участков кода // Труды Института системного программирования РАН. 2015. Т. 27. № 2. С. 93-104.
5. Sargsyan S., Kurmangaleev S., Vardanyan V., Zakaryan V. Code Clones Detection Based on Semantic Analysis for JavaScript Language // 10th International Conference on Computer Science and Information Technologies. 2015. С. 182-185.
6. Avetisyan A., Kurmangaleev S., Sargsyan S., Arutunian M., Belevantsev A. LLVM-Based Code Clone Detection Framework // 10th International Conference on Computer Science and Information Technologies. 2015. С. 178-182.
7. Саргсян С., Курмангалеев Ш., Белеванцев А., Аветисян А. Масштабируемый и точный поиск клонов кода // журнал «Программирование». 2015. № 6. С. 9-17.

8. Roya Ch.K., Cordya J.R., Koschkeb R. Comparison and evaluation of code clone detection techniques and tools, A qualitative approach // Science of Computer Programming. 2009. T.74. № 7. C. 470-495.
9. Ducasse S., Rieger M., Demeyer S. A language independent approach for detecting duplicated code // 15th International Conference on Software Maintenance (ICSM). 1999. C. 109-119.
10. Wettel R., Marinescu R. Archeology of code duplication: Recovering duplication chains from small duplication fragments // 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. 2005.
11. Johnson J.H. Substring matching for clone detection and change tracking // 10th International Conference on Software Maintenance. 1994. C. 120-126.
12. Johnson J. Identifying redundancy in source code using fingerprints // Centre for Advanced Studies on Collaborative Research (CASCON). 1993. C. 171-183.
13. Johnson J. Visualizing textual redundancy in legacy source // Centre for Advanced Studies on Collaborative research (CASCON). 1994. C. 171-183.
14. Manber U. Finding similar files in a large file system // Winter 1994 Usenix Technical Conference. 1994.
15. Cordy J.R. The TXL source transformation language // Science of Computer Programming. 2006. T. 61. № 3. C. 190-210.
16. Roy C.K., Cordy J.R. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization // 16th IEEE International Conference on Program Comprehension (ICPC). 2008. C. 172-181.
17. Roy C.K. Detection and analysis of near miss software clones // 25th IEEE International Conference on Software Maintenance (ICSM). 2009. C. 447-450.
18. Roy C.K., Cordy J.R. An empirical study of function clones in open source software systems // 15th Working Conference on Reverse Engineering (WCRE). 2008. C. 81-90.
19. Roy C.K., Cordy J.R. Are scripting languages really different // 4th International Workshop on Software Clones. 2010. C. 17-24.
20. Web Services Description Language, URL: http://www.w3.org/TR/wsdl

21. Martin D., Cordy J.R. Analyzing web service similarities using contextual clones // 5th International Workshop on Software Clones. 2011. С. 41-46.
22. Manar H. Alalfi, James R. Cordy, Thomas R. Dean, Matthew Stephan, Andrew Stevenson Models are code too: Near-miss clone detection for Simulink models // 28th IEEE International Conference on Software Maintenance (ICSM). 2012. С. 295-304.
23. Matthew Stephan, Manar H. Alafi, Andrew Stevenson, James R. Cordy Towards Qualitative Comparison of Simulink Model Clone Detection Approaches // 6th International Workshop on Software Clones (IWSC). 2012. С. 84-85.
24. James R. Cordy SIMONE: Architecture-Sensitive Near-miss Clone Detection for Simulink Models // First International Workshop on Automotive Software Architecture (WASA). 2015. С. 1-2.
25. James R. Cordy, Chanchal K. Roy Tuning Research Tools for Scalability and Performance: The NICAD Experience // Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010). 2014. Т. 79. С. 158-171.
26. Baker B. A program for identifying duplicated code // Computing Science and Statistics: 24th Symposium on the Interface. 1992. Т. 24. С. 49-57.
27. Baker B. Parameterized pattern matching: Algorithms and applications // Journal Computer System Science. 1996. Т. 52. № 1. С. 28-42.
28. Baker B. On finding duplication and near-duplication in large software systems // 2nd Working Conference on Reverse Engineering (WCRE). 1995. С. 86-95.
29. CCFinder, URL: http://www.ccfinder.net
30. Kamiya T., Kusumoto S., Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code // IEEE Transactions on Software Engineering. 2002. Т. 28. № 7. С. 654-670.
31. Livieri S., Higo Y., Matsushita M., Inoue K. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder // 29th International Conference on Software Engineering (ICSE). 2007. С. 106-115.

32. Basit H., Rajapakse D., Jarzabek S. Beyond templates: a study of clones in the STL and some general implications // 27th International Conference on Software Engineering (ICSE). 2005. С. 15-21.
33. Kozlov D., Koskinen J., Sakkinen M., Markkula J. Exploratory analysis of the relations between code cloning and open source software quality // 7th International Conference on the Quality of Information and Communications Technology. 2010. С. 358-363.
34. Monden A., Okahara S., Manabe Y., Matsumoto K. Guilty or not guilty: using clone metrics to determine open source licensing violations // IEEE Software. 2011. Т. 28. № 2. С. 42-47.
35. Saha R.K., Asaduzzaman M., Zibran M.F., Roy C.K., Schneider K.A. Evaluating code clone genealogies at release level: An empirical study // 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM). 2010. С. 87–96.
36. Eunjong Choi, Norihiro Yoshida, Takashi Ishio, Katsuro Inoue, Tateki Sano Extracting code clones for refactoring using combinations of clone metrics // 5th International Workshop on Software Clones (IWSC). 2011. С. 7-13.
37. Juergens E., Deissenboeck F., Hummel B., Wagner S. Do code clones matter? // 31st International Conference on Software Engineering (ICSE). 2009. С. 485-495.
38. Li Z., Lu S., Myagmar S., Zhou Y. CP-Miner: Finding copy-paste and related bugs in large-scale software code // IEEE Transactions on Software Engineering. 2006. Т. 32. № 3. С. 176-192.
39. Yan X., Han J., Afshar R. CloSpan: Mining Closed Sequential Patterns in Large Datasets // Third SIAM International Conference on Data Mining. 2003. С. 166-177.
40. Jian-lin Huang, Fei-peng Li Quick similarity measurement of source code based on suffix array // International Conference on Computational Intelligence and Security. 2009. С. 308-311.

41. Basit H., Pugliesi S., Smyth W., Turpin A., Jarzabek S. Efficient token based clone detection with flexible tokenization // 6th European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE). 2007. C. 513-515.
42. Yamashina T., Uwano H., Fushida K., Kamei Y., Nagura M., Kawaguchi S., Iida H. SHINOBI: A real-time code clone detection tool for software maintenance // Graduate School of Information Science, Nara Institute of Science and Technology. 2008.
43. Udi Manber, Gene Myers Suffix Arrays: A New Method for On-Line String Searches // SIAM Journal on Computing. 1993. T. 22. № 5. C. 935-948.
44. Yang W. Identifying syntactic differences between two programs // Software Practice & Experience. 1991. T. 21. № 7. C. 739-755.
45. Falke R., Frenzel P., Koschke R. Empirical evaluation of clone detection using syntax suffix trees // Empirical Software Engineering. 2008. T.13. № 6. C. 601-643.
46. Tairas R., Gray J. Phoenix-based clone detection using suffix trees // 44th Annual Southeast Regional Conference (ACM-SE). 2006. C. 679-684.
47. Phoenix Compiler, URL: http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx
48. Jiang L., Misherghi G., Su Z., Glondu S. DECKARD: Scalable and accurate treebased detection of code clones // 29th International Conference on Software Engineering (ICSE). 2007. C. 96-105.
49. Tairas R., Gray J. Sub-clone refactoring in open source software artifacts // ACM Symposium on Applied Computing (SAC). 2010. C. 2373-2374.
50. Juergens E., Deissenboeck F., Hummel B. Code similarities beyond copy & paste // 14th European Conference on Software Maintenance and Reengineering (CSMR). 2010. C. 78-87.
51. Nguyen T.T., Nguyen H.A., Pham N.H., Al-Kofahi J.M., Nguyen T.N Scalable and incremental clone detection for evolving software // 25th IEEE International Conference on Software Maintenance (ICSM). 2009. C. 491–494.

52. Nguyen T.T., Nguyen H.A., Pham N.H., Al-Kofahi J.M., Nguyen T.N. ClemanX: Incremental clone detection tool for evolving software // 31st International Conference on Software Engineering (ICSE). 2009. C. 437-438.
53. Lee H., Doh K. Tree-pattern-based duplicate code detection // International Workshop on Data-intensive Software Management and Mining. 2009. C. 7-12.
54. Brown C., Thompson S. Clone detection and elimination for Haskell // ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM). 2010. C. 111-120.
55. Clone Digger, URL: http://sourceforge.net/projects/clonedigger/
56. Wahler V., Seipel D., Gudenberg J.W., Fischer G. Clone detection in source code by frequent itemset techniques // 4th IEEE International Workshop Source Code Analysis and Manipulation (SCAM). 2004. C. 128-135.
57. Chilowicz M., Duris E., Roussel G. Syntax tree fingerprinting for source code similarity detection // 17th IEEE International Conference on Program Comprehension (ICPC). 2009. C. 243-247.
68. Kraft N., Bonds B., Smith R. Cross-language clone detection // 20th International Conference on Software Engineering and Knowledge Engineering (SEKE). 2008. C. 6.
59. Introducing Visual Studio .NET, URL: https://msdn.microsoft.com/en-us/library/fx6bk1f4(v=vs.71).aspx
60. Horwitz S. Identifying the semantic and textual differences between two versions of a program // Conference on Programming Language Design and Implementation (PLDI). 1990. C. 234-245.
61. Krinke J. Identifying similar code with program dependence graphs // 8th Working Conference on Reverse Engineering (WCRE). 2001. C. 301-309.
62. Komondoor R., Horwitz S. Using slicing to identify duplication in source code // 8th International Symposium on Static Analysis (SAS). 2001. T. 2126. C. 40-56.
63. Higo Y., Kusumoto S. Code clone detection on specialized PDG's with heuristics // 15th European Conference on Software Maintenance and Reengineering (CSMR). 2011. C. 75-84.

64. Liu C., Chen C., Han J., Yu P. GPLAG: Detection of software plagiarism by program dependence graph analysis // 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD). 2006. С. 872-881.
65. Gabel M., Jiang L., Su Z. Scalable detection of semantic clones // 30th International Conference on Software Engineering (ICSE). 2008. С. 321-330.
66. Jingyue Li, Michael D. Ernst CBCD: Cloned Buggy Code Detector // 34th International Conference on Software Engineering (ICSE). 2012. С. 310-320.
67. Mayrand J., Leblanc C., Merlo E.M. Experiment on the automatic detection of function clones in a software system using metrics // 12th International Conference on Software Maintenance (ICSM). 1996. С. 244-253.
68. Patenaude J.F., Merlo E., Dagenais M., Laguë B. Extending software quality assessment techniques to Java systems // 7th International Workshop on Program Comprehension (IWPC). 1999. С. 49-56.
69. McCabe T.J. A complexity measure // IEEE Transactions on Software Engineering. 1976. Т. 12. № 3. С. 308-320.
70. Kontogiannis K., Demori R., Merlo E., Galler M., Bernstein M. Pattern matching for clone and concept detection // Automated Software Engineering. 1996. С. 77-108.
71. Kontogiannis K. Evaluation experiments on the detection of programming patterns using software metrics // 3rd Working Conference on Reverse Engineering (WCRE). 1997. С. 44-54.
72. Albrecht A.J. Measuring application development productivity // IBM Applications Develop. 1979. С. 83-92.
73. Henry Sallie, Dennis Kafura Software Structure Metrics Based on Information Flow // IEEE Transactions on Software Engineering. 1981. Т. 7. № 5. С. 510-518.
74. Kodhai E., Kanmani S., Kamatchi A., Radhika R., Saranya B.V. Detection of Type-1 and Type-2 code clones using textual analysis and metrics // International

Conference on Recent Trends in Information, Telecommunication and Computing. 2010. С. 241-243.

75. Li Z.O., Sun J. A metric space based software clone detection approach // 2nd International Conference on Software Engineering and Data Mining. 2010. С. 111-116.
76. Sutton A., Kagdi H., Maletic J.I., Volkert G. Hybridizing evolutionary algorithms and clustering algorithms to find source-code clones // Genetic and Evolutionary Computation Conference (GECCO). 2005. С. 1079-1080.
77. Maeda K. Syntax sensitive and language independent detection of code clones // World Academy of Science, Engineering and Technology 60. 2009. С. 350-354.
78. Chilowicz M., Duris É., Roussel G. Finding similarities in source code through factorization // Electronic Notes in Theoretical Computer Science, 8th Workshop on Language Descriptions, Tools and Applications (LDTA). 2009. Т. 238. № 5. С. 47-62.
79. Basit H., Jarzabek S. A data mining approach for detecting higher-level clones in software // IEEE Transactions on Software Engineering. 2009. Т. 35. № 4. С. 497-514.
80. Basit H., Puglisi S., Smyth W., Turpin A., Jarzabek S. Efficient token based clone detection with flexible tokenization // 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. 2007. С. 513-516.
81. Hummel B., Juergens E., Heinemann L., Conradt M. Index-based code clone detection: Incremental, distributed, scalable // 26th IEEE International Conference on Software Maintenance (ICSM). 2010. С. 1-9.
82. Ray B., Miryung Kim, Person S., Rungta, N. Detecting and characterizing semantic inconsistencies in ported code // 28th International Conference on Automated Software Engineering (ASE). 2013. С. 367-377.
83. Agrawal R., Srikant R. Mining Sequential Patterns // the Eleventh International Conference on Data Engineering. 1995. С. 3-14.

84. Jablonski P., Hou D. CReN: a tool for tracking copy-and-paste code clones and renaming identifiers // 2007 OOPSLA workshop on eclipse technology eXchange. 2007. C. 16-20.
85. Yoshiki Higo, Shinji Kusumoto MPAnalyzer: a tool for finding unintended inconsistencies in program source code // 29th ACM/IEEE international conference on Automated software engineering. 2014. C. 843-846
86. Lingxiao Jiang, Zhendong Su, Edwin Chiu Context-based detection of clone-related bugs // 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. 2007. C. 55-64.
87. Mark Gabel, Junfeng Yang, Yuan Yu, Moises Goldszmidt, Zhendong Su Scalable and Systematic Detection of Buggy Inconsistencies in Source Code // ACM international conference on Object oriented programming systems languages and applications. 2010. C. 175-190.
88. Bauer V., Hauptmann B. Assessing cross-project clones for reuse optimization // 7th International Workshop on Software Clones. 2013. C. 60-61.
89. A System for Detecting Software Plagiarism, URL: http://theory.stanford.edu/~aiken/moss/
90. Clone Doctor: Software Clone Detection and Reporting, URL: http://www.semdesigns.com/products/clone/
91. IPA: Exploratory IT Human Resources Project, URL: http://www.ipa.go.jp/english/humandev/third.html
92. Chanchal K. Roya, James R. Cordya, Rainer Koschkeb Comparison and evaluation of code clone detection techniques and tools: A qualitative approach // Science of Computer Programming. 2009. T. 74. № 7. C. 470-495.
93. The LLVM Compiler Infrastructure, URL: www.llvm.org
94. Tizen, URL: https://www.tizen.org/
95. Firefox OS, URL: https://www.mozilla.org/en-US/firefox/os/2.0/
96. Google's high performance open source JavaScript engine, URL: https://developers.google.com/v8/

97. Android, URL: https://www.android.com/

98. QEMU, URL: http://wiki.qemu.org/Main_Page