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

Автореферат
диссертации на соискание ученой степени кандидата физико-математических наук

Москва – 2016

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институте системного программирования Российской академии наук.

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

Официальные оппоненты: **Галатенко Владимир Антонович**
доктор физико-математических наук, старший научный сотрудник, заведующий сектором автоматизации программирования Федерального государственного бюджетного учреждения науки Научно-исследовательского института системных исследований Российской академии наук**,**

Волконский Владимир Юрьевич
кандидат технических наук, старший научный сотрудник, начальник отделения "Системы программирования" публичного акционерного общества "ИНЭУМ им. И.С. Брука".

Ведущая организация: Вычислительный центр им. А.А. Дородницына Российской академии наук Федерального исследовательского центра «Информатика и управление» Российской академии наук

Защита состоится “17” марта 2016 г. в 15 часов на заседании диссертационного совета Д 002.087.01 при Институте системного программирования РАН по адресу: 109004, Москва, ул. А. Солженицына, д.25.

С диссертацией можно ознакомиться в библиотеке и на сайте Федерального государственного бюджетного учреждения науки Институт системного программирования Российской академии наук.
Автореферат разослан “_____”_____2016 г.

Ученый секретарь
диссертационного совета Д 002.087.01,
кандидат физико-математических наук Зеленов С. В.

Общая характеристика работы

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 T3 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 T1 и T2. 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 T3) 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.

Краткое содержание работы.

Во введении неформально определяется понятие клона кода, обсуждаются проблемы, возникающие при копировании исходного кода в процессе разработки, формулируются цели и задачи работы, обосновывается ее актуальность, обсуждаются вопросы практического применения разработанных методов и инструментов, дается краткий обзор работы.

В главе 1 приводится обзор работ, имеющих отношение к теме диссертации.

В разделе 1.1 определяется понятие клона кода. Клонами кода считаются фрагменты исходного кода, которые схожи по нижеприведенным критериям. Существует три основных типа клонов (рисунок 1):

Оригинальный код	Клон типа Т1

`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 Т2	**Клон типа Т3**

`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. Пример трех типов клонов кода.

Т1. Фрагменты кода, которые могут отличаться только пробелами, комментариями и форматированием кода;

Т2. Все клоны типа Т1, а также фрагменты кода, которые могут также различаться: именами переменных; типами переменных; начальными значениями переменных и значениями констант;

Т3. Все клоны типа Т2, а также фрагменты кода, в которых могут быть добавлены или удалены некоторые инструкции и переменные.

Степень схожести фрагмента кода обратно пропорционально количеству выполненных операций редактирования клона (переименование переменных, удаление, добавление и изменение инструкций) в ходе его адаптации.

Пять подходов к поиску клонов кода:

Текстовый подход. Алгоритмы поиска клонов кода считывают хеш-коды одной или нескольких строк исходного кода и сравнивают их. Если хеш-коды совпадают, считается, что соответствующие строки исходного кода являются клонами. Алгоритмы, работающие на основе этого подхода, находят, в основном, клоны типа Т1.

Лексический подход. Алгоритмы, основанные на этом подходе, в первую очередь получают последовательность лексических единиц – токенов – путем

разбора исходного кода, после чего производится поиск совпадающих подпоследовательностей токенов. Алгоритмы, работающие на основе этого подхода, в основном, находят клоны типа Т1 и Т2.

Синтаксический подход. Алгоритмы этого типа используют абстрактное синтаксическое дерево (АСД). Клонами считаются совпадающие поддеревья АСД. Синтаксический подход позволяет находить клоны типа Т1 и Т2. Клоны типа Т3 обнаруживаются с низкой точностью, потому что добавленные и удаленные инструкции могут значительно изменить структуру АСД.

Метрический подход. Алгоритмы, основанные на метрическом подходе, вычисляют ряд метрик для АСД или ГЗП и сравнивают векторы полученных метрик. Эти алгоритм находят все три основных типа клонов. У таких алгоритмов большая производительность, но более низкая точность, чем у алгоритмов, основанных на АСД или ГЗП.

Семантический подход. Используется ГЗП представление программы, который является объединенным графом потока данных и графа потока управления. Алгоритмы, работающие на основе семантического анализа, осуществляют поиск *схожих* подграфов в 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 .

Для качественного анализа ПО требуется уметь находить клоны кода типа Т3 с высоким уровнем истинных срабатываний. Для достижения такого результата необходимо снизить вычислительную сложность семантического анализа, сохранив высокую точность обнаружения клонов кода.

В разделе 1.2 приводится обзор существующих методов поиска семантических ошибок, возникающих при некорректной адаптации

скопированных фрагментов кода к контексту программы. Выделяют пять основных типов ошибок:

Несоответствие потока управления. При адаптации клонированного фрагмента кода не учитывается контекст потока управления. Такие ошибки могут приводить к неожиданным переходам по управлению во время работы программы.

Несоответствие потока данных. При адаптации клонированного фрагмента кода не учитывается контекст потока данных. Некоторые переменные инициализируются неправильно в контексте. Такие ошибки могут приводить к доступу к несуществующей или недоступной памяти.

Неправильно переименованные переменные. При адаптации клонированного фрагмента кода не все переменные были переименованы согласно контексту. Такие ошибки могут приводить к неожиданным переходам по управлению и доступу к несуществующей или недоступной памяти.

Наличие избыточной операции. Скопированный фрагмент кода содержит избыточные вычисления.

Ошибки стиля, форматирования и комментариев. Такие ошибки возникают при адаптации кода. Комментарии могут не совпадать с функциональностью, форматирование и стиль кода могут отличаться от общепринятых.

Классификация ошибок приводится на основе исследования больших проектов FreeBSD и Linux с открытыми исходными кодами. Они содержат соответственно более 113 и 182 ошибок, связанных с клонированием. Исследования показали, что чаще всего ошибки возникают из-за некорректно переименованных переменных.

В главе 2 рассматривается четырехфазный метод поиска клонов кода, на основе семантического анализа программы. На первой фазе ГЗП разделяется на подграфы, каждый из которых рассматривается как потенциальный клон других подграфов. На второй фазе применяются специальные алгоритмы линейной сложности, которые позволяют определить, что данная пара ГЗП не имеет

достаточно большого схожего подграфа. На третьей фазе применяется приближенный алгоритм поиска максимальных схожих подграфов в паре ГЗП. Этот алгоритм применяется только для тех пар, которые не были отфильтрованы на второй фазе. Последняя фаза предполагает фильтрацию найденных результатов. Инструмент проверяет, что строки исходного кода, соответствующие схожим подграфам, расположены в файле последовательно. Также рассматриваются два новых методов поиска клонов типов T1 и T2 на основе изоморфизма деревьев и метрики вершин ГЗП. Приводится сравнение реализованных алгоритмов и результаты анализа некоторых повсеместно известных программных систем.

В разделе 2.1 представлен новый алгоритм разделения ГЗП на подграфы.

Разработанный алгоритм состоит из двух основных шагов.

На первом шаге вершины ГЗП добавляются в список *S*, который сортируется по номерам строк исходного кода соответствующих вершин. Ребра между вершинами *S* рассматриваются как интервалы. Для каждой вершины *S* вычисляется количество пересекающихся интервалов.

На втором шаге *S* разделяется по тем вершинам, у которых количество пересекающихся интервалов минимально. При разделении ставится ограничения на размер каждого подсписка. Подграфы строятся на основе полученных подсписков.

Такой подход позволяет получать подграфы, имеющие приблизительно одинаковое количество вершин. Фрагмент кода, соответствующий одному подграфу, представляет собой последовательные строки исходного кода. Благодаря выбору точек среза вершин с минимальными значениями пересекающихся интервалов, количество ребер между полученными подграфами минимизируется, что обеспечивает семантическую независимость подграфов.

Результаты показали, что разработанный подход позволяет находить, в среднем, в 1.5-2 раза больше клонов по сравнению с существующими методами.

В разделе 2.2 рассматриваются алгоритмы, которые проверяют, что данная пара ГЗП не может иметь достаточно большие схожие подграфы. Сложность

рассматриваемых алгоритмов линейная от количества вершин ГЗП, что позволяет быстро обработать большинство пар ГЗП.

Две вершины ГЗП схожи, если у них одинаковые метки (код операции соответствующей инструкции). Алгоритмы проверяют наличие достаточного количества схожих вершин в рассматриваемой паре ГЗП. В противном случае, рассматриваемая пара графов не может иметь схожие подграфы желаемого размера.

Первый алгоритм сохраняет метки вершин ГЗП G_1, G_2 в хеш-таблицы T_1, T_2 соответственно. Если $|T_1 \bigcap T_2| \leq p$ тогда G_1, G_2 не могут иметь схожие подграфы желаемого размера, где p вычисляется на основе размера минимального искомого клона (прямо пропорционально).

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 имеющий 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 пользователем число, тогда соответствующие им ГЗП не содержат желаемые схожие подграфы.

Раздел 2.3 содержит описание алгоритма поиска максимальных схожих подграфов в паре ГЗП с использованием прямого и обратного слайсинга. Алгоритм включает два шага.

На первом шаге для данной пары ГЗП G_1, G_2 строится множество пар схожих вершин (V_1, V_2), где $V_1 \in G_1, V_2 \in G_2$. Для построения этих пар вычисляются БВ (см. раздел 2.5) вершин G_1, G_2. Выбираются $V_1 \in G_1, V_2 \in G_2$ таким образом чтобы БВ соответствующие (V_1, V_2), имели максимальный схожесть. Процесс повторяется для оставшихся нерассмотренных вершин G_1, G_2.

На втором шаге для каждой полученной пары вершин (V_1,V_2) создаются подграфы T_1,T_2, где T_1 содержит только V_1, а T_2 только V_2. T_1,T_2 расширяются путем прямого и обратного слайсинга. Выбирается нерассмотренная пара вершин $(U_1,U_2),U_1 \in T_1,U_2 \in T_2$ после чего в T_1 и T_2 добавляются идентичные смежные вершины (U_1,U_2). Расширения T_1,T_2 продолжается пока возможно, после чего полученные максимальные схожие подграфы для пары (V_1,V_2) сохраняются. Самая большая полученная пара подграфов возвращается как максимальные схожие подграфы G_1,G_2.

Раздел 2.4 содержит описание метода поиска клонов кода на основе изоморфизма деревьев. Он состоит из двух основных этапов. На первом этапе ГЗП преобразуется в дерево. При преобразовании добавляются новые ребра и вершины, что дает возможность сохранить максимальное количество информации об исходном графе. На втором этапе производится поиск максимальных изоморфных поддеревьев, которые рассматриваются как клоны кода. Такой метод позволяет применить существующие точные алгоритмы поиска максимальных изоморфных поддеревьев.

Преобразование из 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 из вершин $W_1,...,W_n$ только W_n имеет максимальную метку. В этом случае создаются $V_1,...,V_{n-1}$ новые вершины с метками как у V. Ребра (W_i,V) меняются 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 такой 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 1: Для двух изоморфных ГЗП деревья, получаемые с помощью преобразований **А, Б** изоморфны.

В разделе 2.5 приводится алгоритм поиска схожих подграфов на основе метрики. Для каждой вершины ГЗП строится битовый вектор, который содержит информацию обо всех смежных вершинах. 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.

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 $V[i] = V_1[i] \wedge V_2[i], i = 1,..., N.$

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 2: Пусть X_n – множество БВ длины n, тогда функция $Dist: X_n \times X_n \to [0,1)$ где $Dist(V_1,V_2) = \frac{orC(V_1,V_2) - andC(V_1,V_2)}{1 + orC(V_1,V_2)}, V_1,V_2 \in X_n$ является метрикой.

Утверждение 3: Для любых $V_1,V_2 \in X_n, 0 \le Dist(V_1,V_2) < 0$.

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.

Алгоритм поиска схожих подграфов состоит из двух шагов:

1. для данной пары ГЗП строит множества схожих вершин, на основе БВ (определение 5);
2. из каждого множества удаляются вершины до тех пор, пока плотность (определение 6) множества не будет удовлетворять заданным условиям.

Исходя из того, что языки программирования имеют ограниченное количество операций, БВ вершин ГЗП возможно представить как машинное слово. Такой подход позволяет сравнить пару БВ за постоянное время.

В разделе 2.6 объясняется, для каких задач следует применить конкретный алгоритм. Наибольшей точностью обладает подход на основе поиска схожих подграфов (раздел 2.3). Он позволяет находить фрагменты кода, в которых были сделаны существенные изменения. По сравнению с остальными подходами, у этого подхода самая большая вычислительная сложность. Его следует применять в задачах, целью которых является нахождение всех клонов кода. Подход, основанный на изоморфизме деревьев (раздел 2.4), находит клоны типа T1 и T2 с большой точностью. Его следует применять в задачах поиска клонов типов T1 и T2. Подход, основанный на метриках (раздел 2.5) обладает низкой

вычислительной сложностью по сравнению с остальными подходами. Этот подход имеет сравнительно низкую точность, но работает быстрее всех. Его следует применять в таких задачах, где время работы более критично, чем нахождение всех клонов.

В разделе 2.7 приводится сравнение результатов реализованных алгоритмов с существующими инструментами поиска клонов кода. Для получения набора тестов оригинальный файл был изменен несколькими разными способами, чтобы получить все три типа клонов. Все полученные копии – клоны оригинального файла. Инструмент, обладающий наибольшей точностью, должен найти, что все преобразования являются клонами.

Сравнение производилось с тремя известными инструментами. Первый, MOSS, был разработан в Стэнфордском университете для поиска плагиата в курсах по программированию. Второй, CloneDR, был разработан в компании Semantic Designs, которая занимается разработкой различного инструментария для проектирования и анализа ПО. Третий, CCFinder, автором является Toshihiro Kamiya, разработка велась в рамках проекта Exploratory IT Human Resources Project. В Таблице 1 приведены результаты сравнения вышеописанных инструментов с реализованными алгоритмами.

В раздел разделе 2.8 приводится результаты анализа некоторых широко известных библиотек и программных систем. Тестирование производилось на Intel Core i3 CPU 540 с 8 ГБ оперативной памяти. Были проанализированы проекты Linux (13.9 млн. строк кода на C/C++): время анализа составило 34.4 часов, найдено 1965 клонов, из них 73 ложных (3.7 %); Mozilla Firefox (3.8 млн. строк кода на C/C++): время анализа составило 16 часов, найден 91 клон, из них 7 ложных (7.7%).

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

Имя теста	MOSS	CloneDR	CCFinder	слайсинг	изоморфизм	сравнение

				раздел 2.3	дерево раздел 2.4	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	+	+	+	+	+	+

В главе 3 рассматривается предлагаемая архитектура инструмента поиска клонов кода на базе компиляторной инфраструктуре LLVM. Объясняется схема генерации ГЗП графов и их дальнейший анализ. Описывается система автоматической генерации клонов кода, для анализа и улучшения разработанных алгоритмов. Приводится схема параллельного запуска инструмента в многоядерных системах.

В разделе 3.1 рассматривается схема генерации ГЗП проекта и их анализа.

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. 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 разного типа получают одинаковые метки.

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.

Инструмент применим для всех языков программирования, поддерживающих промежуточное представление LLVM.

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

После того как проект скомпилирован и ГЗП получены, производится анализ этих графов в целях поиска клонов кода. Поиск подразделяется на четыре основных фазы, что позволяет эффективным образом найти все клоны.

В разделе 3.2 рассматривается схема автоматической генерации клонов кода. 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. Точность разработанного алгоритма определяется количеством найденных клонов из базы. Предложены два подхода к автоматической генерации клонов кода.

Первый подход использует существующие стандартные и специальные проходы LLVM, которые были разработаны 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.

Второй подход по списку ГЗП проекта строит последовательность пар ГЗП и объединяет каждую пару в один граф. Для объединения используется 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 с помощью семантического анализа находит подходящие подграфы во втором графе и добавляет их в первый. Алгоритм поиска клонов кода сравнивает ГЗП оригинального и объединенного списка.

В разделе 3.3 рассматривается схема запуска инструмента в многоядерных системах.

Файлы, содержащие ГЗП проекта, разделяются на k групп в зависимости от количества ядер и размера оперативной памяти машины.

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

Если файлы, соответствующие паре групп, не помещаются в оперативной памяти, количество групп увеличивается настолько, чтобы любые две группы помещались в оперативной памяти.

В разделе 3.4 рассматриваются скрипты анализа полученных результатов. В инструменте содержатся два специальных скрипта для анализа найденных клонов. Первый скрипт позволяет просмотреть исходный код клонов и соответствующие им графы. Второй скрипт дает возможность проследить историю копирования конкретного фрагмента кода.

В главе 4 рассматриваются методы поиска клонов кода для языка JavaScript. Описываются изменения, внесенные в JIT компилятор V8, что позволяет сгенерировать ГЗП для языка JavaScript. Приводится результаты анализа некоторых распространенных тестовых наборов.

В разделе 4.1 рассматривается архитектура Just-In-Time компилятора V8.

Раздел 4.2 описывает трансляция промежуточного представления Hydrogen компилятора V8 в ГЗП. Вершинами ГЗП являются инструкции Hydrogen, ребра получаются путем анализа базовых блоков Hydrogen и анализа зависимостей между инструкциями (use-def).

В разделе 4.3 рассматриваются результаты анализа некоторых повсеместно известных тестовых наборов JavaScript. Тестирование производилось на Intel Core i3 CPU 540 с 8 ГБ оперативной памяти. Были проанализированы следующие тестовые наборы: SunSpider (4.538 строк кода на JavaScript): время анализа составило 19.4с, найден 10 клонов, из них 1 ложный (10%); Octane (357.165 строк кода на JavaScript): время анализа составило 288с, найдено 342 клонов, из них 5 ложный (1.5%).

В разделе 4.4 приводится сравнение разработанного инструмента с инструментом CloneDR.

В главе 5 рассматривается метод поиска семантических ошибок, возникающих при некорректной адаптации скопированного фрагмента кода к контексту, в который он был вставлен. Сначала путем лексического анализа определяются клоны кода типов T1 и T2, после чего применяется семантический анализ для выявления допущенных ошибок. Также приводятся результаты анализа ряда программных систем таких как ОС Linux и Android.

В разделе 5.1 представлена схема четырехфазного метода. На первой фазе строится последовательность токенов для данной функции. На второй фазе обнаруживаются все клоны типов T1 и T2, как максимальные совпадающие подпоследовательности токенов. На третьей фазе строится ГЗП анализируемой функции. На последней фазе анализируются подграфы, соответствующие идентичным последовательностям токенов, для нахождения допущенных ошибок.

В разделе 5.2 рассматривается метод построения токенов функции и метод поиска максимальных совпадающих подпоследовательностей токенов. Последовательность токенов получается из биткода LLVM. Разработан алгоритм, который для последовательности токенов находит все непересекающиеся пары идентичных подпоследовательностей максимального размера. Полученные подпоследовательности фильтруются от неполных конструкций, чтобы получить структурно целые куски кода. Отфильтрованные последовательности достаточно большого размера, проверяются на наличие ошибок.

В разделе 5.3 представлен метод анализа ГЗП для выявления ошибок при копировании. Для поиска ошибок строится ГЗП соответствующей функции. В полученном графе выделяются два подграфа, соответствующие идентичным подпоследовательностям токенов. Полученные подграфы расширяются путем добавления вершин (исходные вершины), соответствующим переменным, которые используются в выделенных подграфах. Если выделенные подграфы не изоморфны, то при копировании, с большой вероятностью, произошла ошибка, поэтому необходимо провести дополнительный анализ. Анализ исходных вершин

дает информацию о возможной ошибке. Если существует исходная вершина, которая входит в выделенные подграфы, и в каждом подграфе имеет разную степень, то переменная, соответствующая этой вершине, содержит ошибочное использование в функции.

В разделе 5.4 описывается метод проверки изоморфизма двух графов. Для ребер графа определяются характеристические числа.

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 $0 \le id(V) < 2^{16}$, $inDeg(V)$ это количество входящих ребер в вершину V, $outDeg(V)$ – количество ребер, выходящих из вершины V.

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

Предложенный подход является приближенным, так как возможно совпадение множеств характеристических чисел ребер двух неизоморфных ГЗП. На практике метод работает с точностью более девяноста процентов.

В разделе 5.5 приводятся результаты анализа некоторых известных библиотек и программных систем. Были проанализированы проекты Android (22.1 млн. строк кода на C/C++): найдено 53 ошибки, из них 17 ложных (32%); Linux (13.9 млн. строк кода на C/C++) найдено 102 ошибки, из них 33 ложных (33%); Mozilla Firefox (3.8 млн. строк кода на C/C++): найдено 6 ошибок, из них 1 ложная (17%).

Заключение содержит выводы и направления дальнейших исследований.

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.

Основные результаты:

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.

Список опубликованных статей по теме диссертации:

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.