Московский государственный университет имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Кафедра автоматизации систем вычислительных комплексов

Малышев Никита Сергеевич

Средство анализа истории изменений текста программы

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:

Ассистент Д.Ю. Волканов

Москва, 2017

Аннотация

В рамках данной работы рассматривается задача разработки и реализации алгоритма анализа истории изменений текста программы в процессе разработки с использованием системы контроля версий. Этот алгоритм нацелен на отслеживание по дереву версий клонов кода некоторых изначально отмеченных при помощи статического анализатора и вручную проблемных фрагментов текста программы.

Был проведён обзор способов представления исходного кода и средств статического анализа, разработан алгоритм, позволяющий отследить, начиная с некоторой версии, изменения, происходившие с помеченными в этой начальной версии фрагментами исходного кода, и разработано средство, реализующее этот алгоритм для работы с репозиториями проектов на языке C в системе контроля версий Git.

Данный алгоритм позволяет отслеживать клоны целевых фрагментов исходного кода заданного размера, которые отличаются от указанных целевых фрагментов в произвольных подвыражениях. Для сравнения фрагментов исходного кода используется представление в виде абстрактного синтаксического дерева с последующим формированием наиболее специализированного общего шаблона и вычислением расстояния между фрагментами по этому шаблону.

Оглавление

1. Введение 4
2. Постановка задачи 9
2.1. Цели работы 9
2.2. Определение клонов кода 10
2.3. Формальная постановка 13
3. Обзор способов представления исходного кода программы 14
3.1. Последовательность символов 14
3.2. Последовательность лексем 15
3.3. Абстрактные синтаксические деревья 16
3.4. Числовые характеристики 18
3.5. Граф зависимостей программы 18
3.6. Итоги обзора 19
4. Обзор средств статического анализа исходного кода программы 21
4.1. Astrée Static Analyzer 23
4.2. BLAST 24
4.3. Coverity 24
4.4. PVS-Studio 25
4.5. Clang Static Analyzer 26
4.6. CppCheck 26
4.7. PC-Lint и FlexeLint 27
4.8. Polyspace Bug Finder 27
4.9. KlocWork 28
4.10. Splint 28
4.11. Итоги обзора 29
5. Алгоритм анализа репозитория 31
5.1. Наиболее специализированный общий шаблон 31
5.2. Расстояние между фрагментами исходного кода 32
5.3. Алгоритм 33
6. Описание реализации 36
6.1. Функционирование реализации 37
6.2. Взаимодействие модулей программы 38
7. Экспериментальное исследование. 42
7.1. Цель исследования 42
7.2. Подготовленный репозиторий. 42

7.3. Реализация алгоритма, предложенного в рамках ВКР. 43
7.4. Cranium. 43
7.5. CCWT. 44
7.6. Tiny JPEG. 44
7.7. LibGOST 15. 44
7.8. C-INI-Parser. 45
7.9. Итоги экспериментального исследования. 45
8. Заключение 53
9. Литература 55
Приложение А. Работа модулей реализации предложенного алгоритма. 60

1. Введение

На сегодняшний день в процессе разработки программ используют системы управления версиями (далее может встречаться сокращение СУВ), такие как **Git ([1], [58])**, **Subversion ([59])** или **Mercurial ([60])** для того, чтобы не потерять результат работы и затем при необходимости иметь доступ к более ранним версиям или разветвить разработку по нескольким разным направлениям.

В ходе разработки программы может возникать необходимость реализовать фрагмент программы, сходный по функционалу с уже существующим фрагментом. В таких случаях вместо создания некоторой общей функции для реализации этого функционала программист может прибегнуть к обычным “копированию и вставке” (copy & paste), то есть попросту копировать существующих фрагмент в другое место в коде программы, быть может с некоторыми изменениями, такими как замена имен переменных, констант или выражений. Такой подход приводит к появлению так называемых “*клонов кода*” - незначительно отличающихся друг от друга синтаксически фрагментов кода программы.

В некоторых случаях подобные действия оправданы тем, что затраты на реорганизацию кода и его последующую верификацию так велики, что разумнее копировать уже проверенный работоспособный код.

Помимо этого, очевидно возможны случаи не намеренного дублирования - например если два программиста, ответственных за разные модули программы, одинаково реализуют сходные по функционалу фрагменты. Причиной также может быть и ситуация, когда один программист сталкивается в процессе разработки с большим количеством типовых задач, либо использование какой-то библиотеки или приложения через прикладной программный интерфейс, что может привести к появлению в коде схожих последовательностей обращений.

Большое количество клонов кода в программе усложняет переработку и реорганизацию кода программы (так называемый “*рефакторинг*” – изменение внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы) и приводит к увеличению числа ошибок и уязвимостей в исходном коде **[2], [3]**.

Кроме клонов кода в ходе разработки в исходном коде программы могут появляться фрагменты, приводящие к уязвимостям, например, выполнению кода или получению чрезмерных прав доступа. Подобные фрагменты могут появляться в коде в силу невнимательности или неопытности программиста, и в случае, если разработчики не знают об их существовании или не удалось найти лучший способ реализовать нужный функционал, так и останутся в коде программы. В случае использования какой-либо системы управления версиями уязвимые фрагменты кода останутся и в тех версиях проекта в репозитории, которые непосредственно или прямо наследуют той версии, в которой впервые появился проблемный фрагмент исходного кода – то есть во всех версиях в дереве версий репозитория, в которые можно попасть из версии, содержащей первое вхождение фрагмента.

Даже если возможность использования злоумышленником уязвимости или возникновения ошибки при работе программы маловероятна, наличие “опасных” фрагментов в программе для разработчика крайне нежелательно. Но поскольку появление проблемных фрагментов в коде программы неизбежно, то в случае, когда удалось найти дефект в какой-то версии программы, например, благодаря обращению сторонних разработчиков или пользователей, разумно будет выяснить, в каких версиях программы присутствуют такие нежелательные фрагменты кода.

Число уязвимых фрагментов кода, как уже было сказано ранее, увеличивается из-за наличия в программе клонов кода. Значит, задачу поиска уязвимостей можно соединить с задачей поиска клонов кода. А за счёт сравнения фрагментов кода более сложным образом, нежели только на точное совпадение, можно находить потенциально уязвимые фрагменты кода, достаточно сильно отличающиеся от исходных.

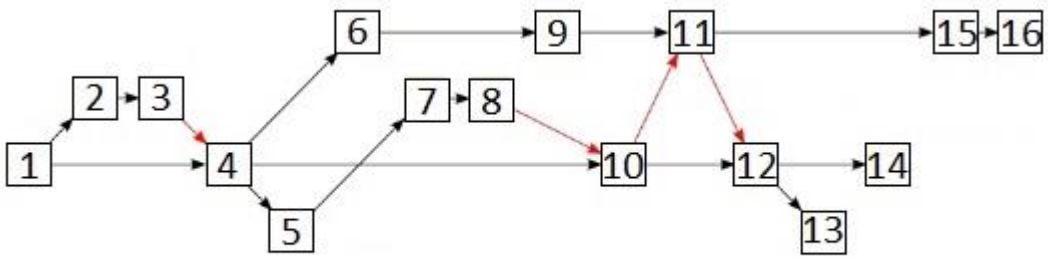

Рисунок 1. Пример дерева версий

На рисунке 1 показан пример дерева версий некоторого проекта. Из изображения очевидно, что если в версиях **1 - 10** есть проблемные фрагменты кода или клоны, то они могут находиться и во всех последующих версиях. Но если интересующие нас фрагменты

найдены в версии **12**, то они окажутся только в версиях **13** и **14**, а в версиях **15** и **16** могут быть найдены, только если разработчик в этих версиях самостоятельно внёс соответствующие изменения в исходный код, то есть проблемные фрагменты из версии **12** в версиях **15** и **16** не смогут появиться из-за наследования.

Итак, в силу описанных выше обстоятельств, в ходе разработки программы в её коде фрагменты, содержащие уязвимости, могут не только появляться, но и затем дублироваться в том же самом программном модуле, либо в других, возможно с некоторыми изменениями. Затем эти фрагменты вполне могут появиться и в последующих версиях программы **[1]**. При этом, имея образец фрагмента, приводящего к уязвимости, можно производить поиск незначительно синтаксически отличающихся от него фрагментов, ограничив спектр возможных изменений клона по отношению к оригиналу.

Существующие детекторы клонов основаны на широком спектре способов представления исходного кода программы (от последовательности символов до графов зависимостей по данным и управлению) и обнаруживают клоны различной степени схожести. Однако большинство из них нацелено на поиск клонов кода наибольшего размера во всех файлах с исходным кодом программы, а не на автоматизированный поиск фрагментов, размер которых не превышает определённого значения и похожих на некоторый исходный, “оригинальный” фрагмент, во всех версиях программы, начиная с некоторой исходной версии **[2], [3], [4], [27]**.

Для иного подхода к решению проблемы поиска клонов уязвимых фрагментов кода существуют другие средства - статические анализаторы кода, нацеленные как раз на поиск уязвимостей различными способами. Среди них существуют достаточно мощные средства, однако большинство из них проприетарны, а время их работы при поиске уязвимостей в каждой версии может составлять многие часы **[21], [29]**. Кроме того, не у всех средств статического анализа можно ограничивать множество обнаруживаемых результатов, то есть возможны случаи, когда в ходе анализа будут обнаружены ложноположительные или не интересующие пользователя результаты. Помимо этого, нельзя быть уверенным в том, что будут найдены все интересующие нас, то есть являющиеся клонами, проблемные участки программы.

Предлагается совместить поиск уязвимостей с поиском клонов кода следующим образом. В рассматриваемом репозитории некоторого проекта в нескольких версиях (далее вместо “версии” может встречаться также название “коммит”) в файлах с исходным кодом

известно положение строк, содержащих проблемный код. Эта информация может содержаться в отдельных файлах, в которых для определённых версий указано в каких файлах и в каком месте находятся целевые строки, или представлять собой простой комментарий специального вида перед строкой прямо в тексте программы:

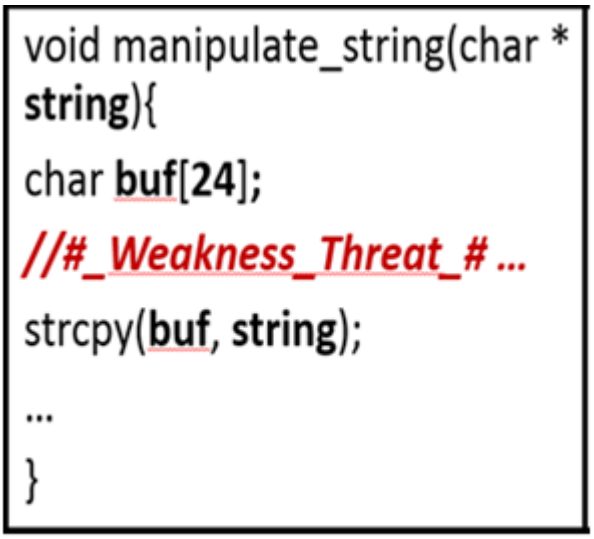

Рисунок 2. Пример комментария - метки

На рисунке 2 приведён пример фрагмента кода на языке C, в котором строка, получаемая от неизвестного источника, копируется в буфер ограниченного размера, что может привести к переполнению буфера и непредсказуемому поведению программы. Комментарий **//#_Weakness_Threat_#** указывает на необходимость отслеживания фрагмента с центром в строке **strcpy(buf, string)**.

При использовании комментария-метки её текст может произвольным - важно только, чтобы он не использовался нигде в программе в качестве обычного комментария.

Определение положения целевого фрагмента, клоны которого мы будем искать, одной строкой оправдано, так как к уязвимости приводит использование какой-то функции, метода или операции. Информация о положении целевых фрагментов может быть произведена как специальным статическим анализатором, так и человеком, проводившим анализ или изначально написавшим потенциально уязвимый фрагмент, не зная, как осуществить тот же функционал без внесения уязвимости.

Теперь рассмотрим структуру выпускной работы по разделам. В разделе 2 будут описаны цели работы, введено обобщённое определение клонов кода, которое впоследствии будет уточнено, и изложена формальная постановка задачи. В следующем разделе 3 проводится обзор способов представления исходного кода, что важно для выбора

алгоритма сравнения фрагментов на этапе обнаружения клонов кода. В разделе 4 изложены результаты проведённого обзора средств, при помощи которых на этапе инициализации предлагаемого в данной работе алгоритма возможно провести поиск целевых фрагментов исходного кода, клоны которых необходимо будет обнаружить. В разделе 5, посвящённом алгоритму анализа репозитория, будут введены строгое определение клонов кода и определение расстояния, которые будут использоваться в работе алгоритма и могли быть введены только после проведения обзора способов представления исходного кода, а также будет изложен предлагаемый алгоритм поиска клонов кода. Раздел 6 содержит описание реализации предложенного в разделе 5 алгоритма. В разделе 7 изложены цели экспериментального исследования и подробности каждого из проведённых экспериментов. В заключении подводятся итоги проведённой работы, а также описываются преимущества и недостатки предложенного метода и альтернативные области применения.

2. Постановка задачи

В рамках выпускной квалификационной работы необходимо разработать и реализовать метод анализа репозитория программы в некоторой системе управления версиями на предмет клонирования фрагментов исходного кода из некоторого начального набора фрагментов исходного кода. Этот набор формируется на основании результата обработки некоторого количества версий, для которых известны их уникальные идентификаторы.

Результатом обработки указанной части версий будет являться информация о том, клоны кода каких фрагментов (назовём их “помеченными”) текста программы необходимо искать в оставшихся версиях программы.

Анализ репозитория состоит в обнаружении в оставшихся версиях клонов кода помеченных фрагментов и формировании непересекающихся множеств клонов для последующего вывода.

Далее опишем цель работы, определение клонов кода и формальную постановку.

2.1. Цели работы

Целью выпускной квалификационной работы является разработка и реализация алгоритма анализа репозитория некоторого проекта на предмет клонирования некоторых целевых фрагментов исходного кода, а также оценка его способности обнаруживать ошибки в исходном коде. Для достижения поставленной цели необходимо решить следующие задачи:

- исследовать проблематику обнаружения клонов исходного кода программы,
- выбрать подходящее определение клонов кода,
- выбрать подходящий способ получения информации о целевых фрагментах, клоны кода которых необходимо обнаружить,
- разработать алгоритм, позволяющий находить в репозитории проекта клоны некоторых целевых фрагментов кода, обнаруженных статическим анализатором в инициализирующих версиях,
- реализовать предложенный алгоритм для выбранной системы управления версиями, языка программирования и способа получения информации о целевых фрагментах,

- провести экспериментальное исследование реализации алгоритма на нескольких репозиториях,

2.2. Определение клонов кода

Для поиска клонов кода необходимо решить, какие два фрагмента исходного кода считать клонами. Несмотря на то, что само понятие клонов широко используется, в настоящее время не существует общепринятого определения. Однако все исследователи сходятся во мнении, что клонами следует считать те фрагменты кода, которые имеют незначительные синтаксические отличия. Поэтому, введём своё определение.

Разумно рассматривать клоны с точки зрения возможности их появления в процессе написания программы. Процесс внесения клонов по своей сути обратен процессу удаления клонов из текста программы при реорганизации (рефакторинге) кода. Для того, чтобы избавить исходный код от клонов, применяется две операции **[22, 23]**:

- выделение общего кода в новую функцию или метод класса,
- перемещение общего метода классов в их базовый класс;

Минимальным фрагментом кода, который может быть выделен в функцию или метод, очевидно является последовательность операторов. Однако из-за существования составных и вложенных операторов можно неоднозначно истолковать понятие последовательности операторов, поэтому рассматриваться будут только фрагменты кода, представляющие собой непрерывные последовательности операторов.

```
l1: int a = 1, b = 2;

l2: if (a > b)

l3:       a = b;

l4: if (a < b) {

l5: …
```

В представленном примере последовательности **(l1, l2)** или **(l1, l2, l3, l4)** может входить в клон, а **(l1, l2, l4)** – нет.

Перемещение общего метода классов в их базовый класс (или выделения родительского класса, если у классов с одинаковым методом нет общего базового класса)

требует семантической эквивалентности фрагментов, однако проверка этого свойства является неразрешимой задачей **[2], [22], [23]**, поэтому на практике часто требуют синтаксической эквивалентности, то есть отличия фрагментов могут состоять только в именах локальных переменных, добавлении/удалении комментариев и незначащих пробелов.

Выделение общего кода в новую функцию допускает более широкие отличия между фрагментами кода, а именно – замены одних синтаксических единиц на другие. Синтаксической единицей называется такой фрагмент исходного кода, к которому могут быть применены синтаксические правила языка программирования. Например, константы и подвыражения являются частными случаями синтаксических единиц, "**x**" или "**x / 3**" являются синтаксическими единицами, а "**x /**" таковой не является. Соответственно, можно ввести следующее определение клонов кода.

Определение 1. Два фрагмента исходного кода F_1 и F_2 программы являются клонами кода, если они представляют собой непрерывные последовательности операторов равного размера (то есть из одинакового числа операторов) и один может быть получен из другого путём замены некоторых синтаксических единиц на любые другие, кроме перестановки операторов в последовательности местами.

Требование равного размера фрагментов вытекает из того факта, что не будь этого требования, из-за того, что последовательность операторов также является синтаксической единицей, то один оператор можно было бы считать клоном последовательности из **m** операторов, а ограничение на перестановку операторов вытекает из того факта, что неизвестна семантика последовательности и на деле фрагменты кода, получаемые при помощи перестановки операторов могут иметь разный функционал.

К сожалению, данное определение хоть и даёт неплохое представление о том, что такое клоны кода, но не является достаточно точным для использования в рамках алгоритма и тем более для написания программы, так как не предоставляет возможности ограничить спектр находимых клонов – любые две последовательности операторов могут считать клонами. Введём ещё одно, более формальное определение клонов кода, так называемое параметризованное определение, которое позволит ограничить множество клонов при помощи функции расстояния между ними **[2], [22], [23]**.

Для этого введём понятие абстрактного синтаксического дерева (АСД, abstract syntax tree) – конечного, размеченного, ориентированного дерева, в котором внутренние вершины

помечены операторами языка программирования, а листья – соответствующими операндами **[24]**. Непрерывные участки кода всегда допускают построение для них АСД (поскольку, даже если часть последовательности входит в составной оператор, эту часть можно представить в виде АСД), а значит и рассматриваемые согласно определению 1 фрагменты тоже.

```
x = a + b;
y = f(x,j);
cout << y;
```

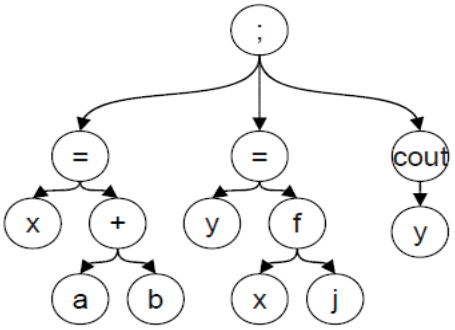

Рисунок 3. Пример абстрактного синтаксического дерева.

На рисунке 3 приведён пример абстрактного синтаксического дерева небольшого фрагмента исходного кода на C++.

Легко видеть, что поддеревья АСД программы соответствуют синтаксическим единицам исходного кода этой программы, а значит, заменив в определении 1 синтаксические единицы фрагментов на поддеревья соответствующих фрагментам АСД, мы получим равносильное определение.

Для формализации определения введём функции ρ, сопоставляющей двум фрагментам F_1 и F_2 число $\rho(F_1, F_2)$, называемое расстоянием между F_1 и F_2.

Теперь описаны возможные отличия между фрагментами исходного кода и введены функции, которые позволят ограничивать спектр находимых клонов, и можно ввести следующее параметризованное определение клонов кода.

Определение 2. Два фрагмента исходного кода программы, являющиеся непрерывными последовательностями операторов с равным числом операторов, F_1 и F_2 формируют клон удалённости p, если абстрактное синтаксическое дерево одной последовательности может быть получено из абстрактного синтаксического дерева другой последовательности путём замены некоторых поддеревьев на другие, кроме перестановки местами поддеревьев с корнем в вершинах сравниваемых деревьев глубины 1, и справедливо равенство $\rho(F_1, F_2) = p$.

Существует большое количество альтернативных определений клонов исходного кода, с некоторыми из них данное определение имеет общие черты, однако в наибольшее степени предложенное определение схоже с предложенным в работе **[2]**. Определение 2 отличается использованием понятия непрерывных последовательностей операторов и только функции расстояния вместо введения метрики.

2.3. Формальная постановка

Рассматривается репозиторий проекта в некоторой системе контроля версий $\mathbf{R = \{V_1, \ldots, V_n\}}$, представляющий собой множество из **n** версий, где каждой из версий $\mathbf{V_i}$ соответствует уникальный идентификатор этой версии и множество файлов с исходным кодом $\mathbf{files_i}$.

Входными данными является множество $\mathbf{P = \{V_p^1, \ldots, V_p^m\}}$, $\mathbf{m \leq n}$, - подмножество множества версий R. После обработки версий из множества P мы получаем множество **marked(P)**, содержащее для каждой версии $\mathbf{V_p^i}$ информацию о положении в файлах этих версий из $\mathbf{files_p^i}$ фрагментов, клоны которых необходимо искать.

Множество $\mathbf{Clusters = \{Cluster_1, \ldots, Cluster_k\}}$ представляет собой неупорядоченное множество непересекающихся упорядоченных по порядку возрастания номера соответствующей версии множеств клонов кода, где каждое множество $\mathbf{Cluster_i}$ содержит такие фрагменты исходного кода, что они являются клонами первого элемента этого множества (назовём этот первый элемент “оригиналом”), но не являются клонами оригиналов из $Cluster_j$, $j \neq i$, $j = 1, \ldots, k$. Множество Clusters инициализируется элемента множества marked(P) и мощность множества Clusters может быть меньше мощности marked(P), так как в версиях из P уже могут быть обнаружены клоны кода, которые попадут в один кластер.

Необходимо разработать алгоритм, для всех версий $\mathbf{V_i}$ из R, таких что $\mathbf{i > p_1}$, $\mathbf{i \neq p_2, \ldots, p_m}$ (так как версии из P считаются уже проверенными) находящий содержащиеся в $files_i$ клоны кода помеченных фрагментов из marked(P) и формирующий множество Clusters, реализовать этот алгоритм для выбранной системы контроля версий и языка программирования.

Теперь необходимо выяснить, существуют ли алгоритмы, способные обнаруживать клоны кода, удовлетворяющие определению 2, и, если подобных средств нет, какие способы представления исходного кода могут позволить обнаруживать такие клоны.

3. Обзор способов представления исходного кода программы

Для того, чтобы производить поиск клонов исходного кода, недостаточно выбрать только определение клонов – требуется также выбрать подходящий способ представления исходного кода программы, с которым и будет работать алгоритм. В большом количестве работ способы представления исходного кода делятся на следующие категории **[2], [5], [22]:**

- последовательность символов, то есть текст **[8], [9], [12],**
- последовательность лексем **[3], [8], [9], [11], [14],**
- абстрактное синтаксическое дерево **[4], [10], [13], [15],**
- числовые характеристики кода **[6], [16],**
- граф зависимостей программы **[7], [17], [18];**

Рассмотрим каждый способ представления текста программы и некоторые средства, использующие эти способы. При рассмотрении представлений кода учитывались следующие критерии:

- способ представления исходного кода позволяет обнаруживать клоны кода, удовлетворяющие определению 2,
- для указанного способа представления кода существуют бесплатные средства, способные сформировать это представление для произвольного фрагмента кода;

Кроме того, от выбора способа представления исходного кода программы зависит скорость формирования выбранного представления, скорость сравнения фрагментов в этом представлении и удобство работы с этим представлением внутри программы.

3.1. Последовательность символов

Текстовое представление исходного кода (или же последовательность символом) не предполагает никакого отличного от текста внутрипрограммного представления кода. Детекторы клонов работающие на этом уровне **[8, 12]** производят поиск посимвольно совпадающих фрагментов текста и, вообще говоря, применимы не только к языкам программирования, но и к любым текстам в принципе, поскольку не учитывают абсолютно никаких фактов, известных о языке.

При необходимости, возможна нормализация текста перед поиском клонов, то есть. приведение текстов к некоторому нормальному виду – например, удаление незначащих

пробелов, комментариев и скобок. Нормализация позволяет уменьшить число ложных срабатываний, однако требует использования некоторого подобия лексического анализатора и сужает область применения текстового детектора клонов.

Можно вводить и другие правила нормализации или учитывать некоторую лексическую информацию, но это практически превращает текстовый детектор клонов в урезанную версию детектора, утилизирующего лексическое представление.

Очевидно, что текстовое представление исходного кода не позволит находить клоны кода, удовлетворяющие определению 2. Однако, стоит отметить, что применение даже такого простого представления всё-таки может дать полезную информацию, поскольку далеко не всегда при копировании и вставке происходит хотя бы смена имён переменных, а исходный код сразу находится в этом представлении и не нужно применять дополнительных преобразований.

3.2. Последовательность лексем

Для представления исходного кода программы в этом виде необходимо путём разбора преобразовать текст программы в последовательность лексических единиц (так называемых "*лексем*"), в которой затем при помощи лексических детекторов клонов кода производится поиск сходных подпоследовательностей, отличающихся, может быть, некоторым набором лексем **[3], [7], [8], [9], [11], [14]**.

Данный подход позволяет обнаруживать клоны, отличающиеся в идентификаторах, константах, и в произвольных лексемах, но ограничить спектр клонов можно только введя ограничения на количество отличающихся лексем и может быть тип таких лексем.

Один из самых распространённых и простых способов поиска клонов – так называемая параметризация, при которой определённый тип идентификаторов и констант заменяется на специальные лексемы-заполнители. Подобная схема реализована в средстве Dup **[8]**, **[9]**, которое способно искать клоны кода, отличающиеся в именах переменных и числовых константах, учитывая при этом, что было заменено, дабы избежать конфликтов, когда одной лексеме из первого фрагмента должно соответствовать сразу несколько лексем из другого. В средстве CCFinder **[14]** алгоритм работы сходен с Dup, но оно дополнительно способно работать с именами классов и функций, аналогично сохраняя информацию о соответствии между лексемами фрагментов во избежание конфликтов соответствия.

Необычным способом утилизирует лексическое представление исходного кода программы модификация алгоритма сбора данных CloSpan (Closed Sequential Pattern Mining), реализованная в средстве CP-Miner **[3]**, **[11]**. CP-Miner уже может находить клоны кода, отличающиеся друг от друга вставкой или удалением некоторого числа лексем, где максимальное число вставок/удалений является параметром алгоритма.

К сожалению, и это представление не позволяет обнаруживать клоны, удовлетворяющие определению 2, так как на уровне последовательности лексем невозможно выделить синтаксические единицы исходного кода или проверять отличие между клонами в виде замены таковых единиц на другие.

3.3. Абстрактные синтаксические деревья

Как уже было сказано в разделе 2, абстрактное синтаксическое дерево – это конечное, размеченное, ориентированное дерево, где внутренним вершинам соответствуют операторы языка программирования, а листьям – их операнды.

АСД используются для представления программы после построения дерева разбора, но до генерации структуры данных, используемой как внутреннее представление компилятора или интерпретатора программы.

Поскольку дерево сохраняет всю информацию и синтаксической структуре фрагмента исходного кода программы, то появляется возможность игнорировать бесполезные клоны кода, например, такие, которые представляют собой конец тела одной функции и начало другой.

Разнообразие средств поиска клонов, основанных на использовании АСД, больше, чем в случае с представлением текста программы в виде последовательности лексем.

Самый простой подход заключается в выделении схожих участков или наоборот различных участков фрагментов кода, сходство или отличие которых было обнаружено на уровне АСД. К таким средствам относится утилита cdiff **[15]**. Такой способ очевидно не подходит под определение клонов 2.

Ещё одним подходом является использование хеш-функций. В данном случае для каждого поддерева вычисляется значение некоторой хеш-функции, затем производится поиск одинаково помеченных поддеревьев в АСД всей программы. В средстве CloneDR **[10]** хеш-функция была выбрана так, чтобы игнорировать имена переменных и константы.

Однако для соответствия такого подхода определению 2 необходимо было бы разработать хеш-функцию, инвариантную относительно замены *произвольного* поддерева, и, к сожалению, хеш-функции также невозможно использовать для поиска клонов, удовлетворяющих определению 2.

Последним способом поиска клонов на уровне АСД является использование шаблонов – таких АСД, в которых в листьях находятся специальные метки-заполнители, означающие, что на место этой метки может быть подставлено поддерево.

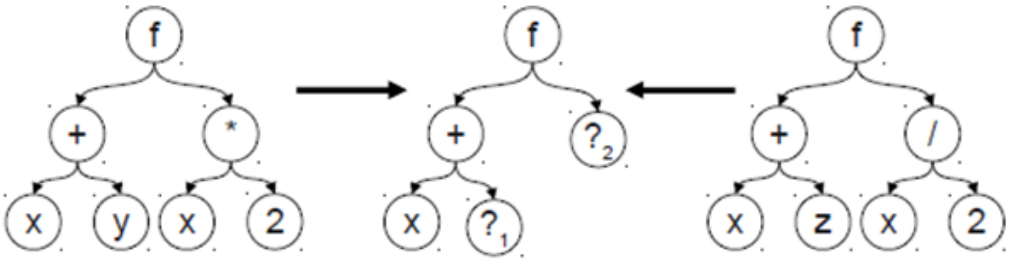

Рисунок 4. Пример общего шаблона двух деревьев

Общим шаблоном двух деревьев при этом подходе называется такой шаблон, что из него путём замены меток-заполнителей на полноценные поддеревья можно получить и одно, и другое дерево. Клонами же называются два дерева, имеющие такой общий шаблон, что он максимально отражает структуры обоих деревьев (то есть любой другой шаблон двух деревьев будет шаблоном и этого общего шаблона) и удовлетворяет ограничениям на глубину и размер подставляемых на место меток-заполнителей поддеревьев, которые являются параметрами алгоритма. Рисунок 4 иллюстрирует ситуацию, когда два фрагмента кода имеют общий шаблон и являются клонами, согласно данному методу поиска и определению 2. На основе данного подхода созданы средства Asta **[13]** и CloneDigger **[2]**.

Алгоритм, реализованный в средстве Asta позволяет находить клоны кода, удовлетворяющие определению 2, но состоящие только из отдельных операторов, а не последовательностей операторов. CloneDigger использует сходное определение клонов и позволяет находить клоны, удовлетворяющие определению 2, но само средство имеет другую направленность – поиск клонов кода максимального размера, а также реализовано только для поиска клонов кода в языках Python и Java.

Кроме того, к сожалению, не для всех языков программирования существуют бесплатные средства, способные создать АСД для произвольного фрагмента исходного кода.

3.4. Числовые характеристики

Данный подход, рассмотренный в работах **[6]**, **[16]**, предполагает вычисление для анализируемых фрагментов (функций, классов или целых файлов) числовых характеристик, таких как количество символов, строк, лексем, обращений к файлам, операций ввода/вывода, вызовов других функций и др., которые затем записываются в последовательность в определённом порядке.

После вычисления метрик кода для фрагментов, последовательности сравниваются, и, соответственно, те фрагменты, метрики которых совпали, считаются клонами.

Использование данного подхода является более производительным, чем поиск клонов кода на уровне абстрактных синтаксических деревьев или графов зависимостей программ (ГЗП), но одновременно и менее точным и не позволяет обнаруживать клоны, отличающиеся, например, в подвыражениях. Кроме того, зачастую для вычисления метрик сначала всё равно строятся АСД, графы зависимостей или хотя бы используется лексический анализатор, а использование числовых характеристик имеет смысл только при работе с достаточно большими фрагментами кода и при использовании нескольких метрик. В противном случае велика вероятность возникновения большого количества ложноположительных результатов.

Использование только числовых характеристик не подходит для поиска клонов, отвечающих определению 2, поскольку это потребовало бы использования набора метрик, инвариантного относительно замены произвольного поддерева. При использовании вкупе с другими представлениями этот подход не даёт значимых преимуществ и может привести к увеличению числа ложных результатов, учитывая, что размер фрагмента - параметр.

3.5. Граф зависимостей программы

Граф зависимостей программы (ГЗП) – это абстрактное представление программы, хранящее информацию о зависимостях по данным и управлению. Этот подход рассмотрен в работах **[7], [17], [18]**.

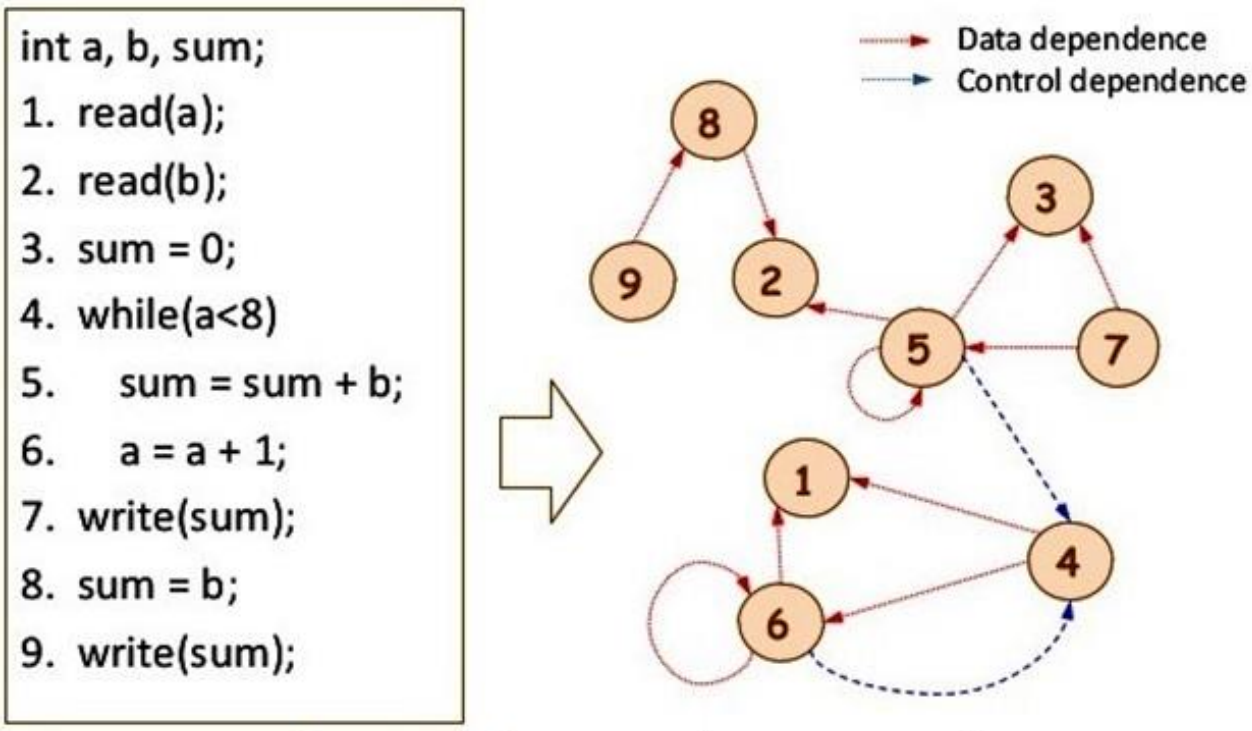

Рисунок 5. Пример графа зависимостей

Клонами с точки зрения ГЗП являются семантически схожие фрагменты кода. Поиск клонов на этом уровне требует проверки изоморфности подграфов, что является NP-полной задачей, поэтому применяются эвристические алгоритмы (то есть такие, у которых нет строгого обоснования, но в большинстве случаев дающие приемлемое решение) **[7]**.

Использование ГЗП позволяет находить клоны, отличающиеся пробелами, комментариями, идентификаторами, константами, значениями переменных и вставкой или удалением некоторого числа строк, но не предполагает отличий в подвыражениях.

Кроме того, не для всех языков программирования существуют бесплатные средства, способные производить ГЗП для произвольного фрагмента исходного кода.

3.6. Итоги обзора

Как видно, для поиска клонов, соответствующих определению, подходит только представление исходного кода в виде абстрактных синтаксических деревьев. В наибольшей степени определение клонов кода 2 похоже на определение, данное в работе **[2]**, посвящённой средству CloneDigger, но отличие состоит в формате фрагментов, поскольку в работе **[2]** рассматриваются линейные последовательности, по сути пропускающие составные операторы, а в нашем случае в фрагмент может входить даже неполная часть такого оператора, и рассматриваются непрерывные последовательности. Кроме того, целью

CloneDigger является обнаружение клонов, представляющих из себя линейные последовательности, наибольшего размера по тексту программы, а в рамках данной выпускной квалификационной работы необходимо производить поиск только клонов заданного размера некоторых исходных, оригинальных, фрагментов. Определения из работ, посвящённых Asta **[12]** и CloneDR **[10]**, также похожи на определение 2, но CloneDR обнаруживает клоны, представляющие собой последовательности операторов, но отличающиеся только в именах и значениях констант, а Asta обнаруживает клоны, отличающиеся в подвыражениях, но только для отдельных операторов.

В начале раздела 3 были сформулированы критерии проведённого обзора. Итоги обзора представлены в таблице 1.

	Позволяет обнаруживать клоны, удовлетворяющие определению 2	Есть средства получения представления для произвольного фрагмента кода
Текст	-	+
Лексемы	-	+
АСД	+	±
Характеристики	-	+
ГЗП	-	±

Таблица 1. Итоги обзора способов представления исходного кода программы.

Для деревьев и графов во втором столбце указан ±, поскольку для всех языков можно получить указанное представление произвольного фрагмента кода.

4. Обзор средств статического анализа исходного кода программы

Перейдём к выбору способа получения информации о том, клоны кода каких фрагментов необходимо обнаружить при анализе репозитория.

В современных реалиях разработки сложность программ постоянно растёт, помимо ожидаемого увеличения объёма исходного кода усложняются и алгоритмы. В исследовании **[19]** было показано, что плотность ошибок неизбежно растёт с ростом размеров программного обеспечения. Многие ошибки, как уже упоминалось ранее, могут приводить к катастрофическим последствиям - от некорректного функционирования приложения, до исполнения вредоносного кода, получения излишних прав доступа и утечки конфиденциальной информации. В этой связи при разработке ПО особенное внимание уделяется предотвращению, поиску и устранению ошибок.

Для поиска ошибок используют различные подходы **[20]**:

- ручной анализ исходного кода или работы приложения экспертами в области поиска ошибок и уязвимостей - т.н. аудит и экспертиза,
- статический анализ исходного кода,
- динамические методы верификации,
- формальные методы верификации,
- смешанные стратегии, например, тестирование на основе моделей или проверка формальных свойств,

Ранее уже оговаривалось, что использоваться будет статический анализатор. Опишу здесь причину использования именно статического анализа.

Экспертиза, очевидно, неприменима, поскольку целью выпускной работы является создание средства автоматизации поиска клонов и ошибок в исходном коде. Разумеется, эксперт может быть источником дополнительной информации о проблемных фрагментах кода, то есть источником появления меток в исходной версии, но данная возможность уже предусмотрена условием поставленной передо мной задачи.

Также из-за того, что средство должно работать только с исходным кодом и явно указывать на местоположение клона в тексте программы, нет возможности использовать формальные, динамические и смешанные методы верификации, так как нацелены они на

проверку некоторых формальных свойств или проверку работоспособности частей ПО или моделей путём выполнения программы. Эти методы верификации ПО обладают большой ценностью и могут быть автоматизированы, но к сожалению, не применимы в рамках поставленной задачи.

Итак, остался только один метод - статический анализ. Статические анализаторы исходного кода осуществляют поиск ошибок в программе без её фактического запуска, на основании только исходного кода. При этом, обычно, статические анализаторы анализируют все пути исполнения независимо от вероятности их выполнения. На редко исполняемых путях статические анализаторы способны находить ошибки, которые часто остаются незамеченными во время тестирования. Ещё одним неоспоримым преимуществом статических анализаторов является то, что они предоставляют диагностическую информацию об ошибке - её местоположение в тексте программы и информацию о том, почему именно этот фрагмент исходного кода приводит к некорректной работе.

В данном обзоре будут рассмотрены средства статического анализа, нацеленные на работу с программами, написанными на языке C, поскольку именно для него предполагается реализация средства. К анализаторам предъявляются следующие требования:

1. работа только с файлами исходного кода программы - то есть отсутствие необходимости в дополнительных входных данных, таких как список свойств, которым должен удовлетворять текст, или предварительном изменении исходного кода так, чтобы его смог обрабатывать анализатор,

2. доступность для использования в рамках выпускной квалификационной работы – то есть наличие полноценной, не ознакомительной версии ПО, удовлетворяющей остальным критериям,

3. возможность вывода результата анализа в командную строку или в файл (поскольку предполагается автоматизация процесса поиска клонов проблемных фрагментов, а в качестве входных данных предполагается использовать только исходный код),

4. возможность запуска процесса анализа при помощи команд консоли (также связано с автоматизацией),

5. рассматриваемые средства должны быть способны находить следующие наиболее часто встречающиеся ошибки **[62], [63], [64], [65], [66]**:

- деление на ноль,
- выход за границу массива,
- переполнение буфера,
- разыменование нулевого указателя,
- переполнение при арифметических операциях,
- чтение неинициализированных переменных,
- ошибки, связанные с логическими операциями и операциями сдвига,
- ошибки преобразования типов,
- некорректное использование операций освобождения памяти,
- обращение к освобождённой памяти;

Единственным критерием участия средства в обзоре является его принадлежность к классу статических анализаторов, однако подробно будет описана только часть рассмотренных анализаторов, а остальные будут только упомянуты в итогах, поскольку аналогично неподходящим средствам из основной части анализаторов не удовлетворяют некоторым из пяти требований обзора и не могут быть использованы в реализации алгоритма. Кроме того, они обладают аналогичными достоинствами и недостатками и их включение в обзор практически привело бы к повторению описания уже рассмотренных средств статического анализа.

4.1. Astrée Static Analyzer

Astrée – это коммерческий статический анализатор программ, доказывающий отсутствие ошибок времени выполнения в написанных на языке Си (в стандарте C99) приложениях, для которых безопасность выполнения критична. Помимо указанных в критерии дефектов данное средство обнаруживает проблемы, возникающие при параллельном выполнении двух нитей, и нарушение заданных пользователем утверждений **[42]**.

Разработчики средства утверждают о полноте анализа своего средства (то есть о том, что Astrée найдёт все возможные ошибки времени выполнения), а также о том, что Astrée может находить в точности ноль ложноположительных результатов. Кроме того, Astrée можно запускать в пакетном режиме и экспортировать результат работы любым способом для дальнейшей обработки. Помимо этого, данное средство не требует (хотя и может использовать) внесения дополнительных аннотаций в исходный код и способно работать полностью автоматически.

Средство является проприетарным, но доступна пробная 30-дневная версия для 64-битных ОС Windows или Linux.

4.2. BLAST

BLAST (Berkeley Lazy Abstraction Software Verification Tool) – программа, предназначенная для проверки моделей для языка Си. Задача, решаемая данным инструментом – проверка того, что программа удовлетворяет заданным поведенческим требованиям к ней **[31]**. Средство является бесплатным и открытым и распространяется по лицензии Apache 2.0.

Средство предназначено для ОС Linux и требует внесения непосредственно в исходный код изменений, поскольку работа BLAST основывается на использовании механизмов *assert* и *lock/unlock* – проверки равенства выражения нулю и реализации семафоров соответственно.

4.3. Coverity

Coverity Code Advisor – одно из самых известных и дорогостоящих средств статического анализа исходного кода на C, C++, C#, Java и JavaScript. Данное средство не обладает свойством полноты анализа, а лишь стремится найти как можно большее число ошибок, и по сравнению с некоторыми другими средствами (не все из которых бесплатны) статического анализа имеет меньше ложноположительных срабатываний **[33]**.

Coverity обладает внушительным списком обнаруживаемых дефектов согласно классификации CWE или OWASP, с которым можно ознакомиться на официальном сайте продукта. Кроме дефектов, указанных в критерии 5, Coverity обнаруживает проблемы, возникающие при параллельной работе нескольких процессов, и нарушение заданных пользователем свойств.

Средство является проприетарным, а стоимость лицензии назначается для каждого конкретного проекта. Как и в случае с многим проприетарным ПО, можно получить пробную тридцатидневную версию Coverity Code Advisor, оставив заявку на официальном сайте.

Существует бесплатная SaaS-версия (Software as a Service, то есть программное обеспечение в качестве сервиса) – Coverity Scan, не привязанная к какой-либо платформе, но анализировать разрешается только открытые проекты.

Использование данного сервиса требует регистрации через веб-интерфейс, локальной сборки проекта специальной утилитой Coverity Build Tool, его загрузки в сервис и последующего ожидания модерации. Только после этого можно будет просмотреть отчёты анализа.

4.4. PVS-Studio

PVS-Studio – закрытый проприетарный статический анализатор исходного кода на C, C++ и C#, выполняющий широкий спектр проверок на ошибки, поиск опечаток и ошибок копирования-вставки. Средство работает как на ОС Windows, так и на Linux, может быть интегрирован в Microsoft Visual Studio или использоваться как самостоятельная утилита **[29, 30]**.

PVS-Studio может быть использована только совместно с компилятором или IDE (Integrated Development Environment, интегрированная среда разработки), поскольку проверяет препроцессированные файлы. Также PVS-Studio может быть запущена из командной строки и может сохранять результат проверки в файл.

Кроме дефектов в критерии, список диагностик PVS-Studio с официального сайта включает следующие дефекты (и по утверждению самих разработчиков является неполным):

- неопределённое/неуточняемое поведение,
- опечатки,
- ошибки копирования-вставки,
- диагностики, созданные по запросу пользователей;

Для PVS-Studio доступна бесплатная версия с ограниченной функциональностью и количеством просматриваемых результатов анализа (ограниченное число, после расходования которого программа сама предложить заполнить форму с персональными данными и связаться с разработчиками), так же в рамках обучения использованию PVS-Studio можно получить бесплатный доступ к пробной версии, если связаться с разработчиками и получить временный ключ.

До 2015 года существовал проект CppCat, созданный в качестве альтернативы PVS-Studio для маленьких команд и индивидуальных разработчиков, но он был признан

убыточным и не востребованным на рынке и на данный момент не развивается, не поддерживается и не доступен для загрузки.

4.5. Clang Static Analyzer

Clang — это фронтенд-транслятор для языков C, C++, Objective-C, Objective-C++ для UNIX-подобных ОС, являющийся частью проекта LLVM (Low Level Virtual Machine). Комбинация Clang и LLVM представляет собой полноценный компилятор и предоставляет набор инструментов, позволяющих полностью заменить GCC. Благодаря архитектуре, основанной на библиотеках, Clang (как и LLVM) легко встраивается в другие приложения **[32]**.

Clang Static Analyzer – бесплатный кроссплатформенный статический анализатор с открытым исходным кодом, часть Clang. Он доступен либо в качестве части IDE Xcode или в качестве утилиты scan-build и проводит статический анализ как часть процесса сборки. Второй способ позволяет автоматизировать запуск анализатора и выводить результат проверки в виде HTML-файла.

Кроме дефектов, указанных в начале обзора, Clang Static Analyzer обнаруживает такие ошибки, как:

- ошибки в логике вызова функций,
- передаче функции в качестве параметров, которые должны быть ненулевыми, нулевых указателей,
- проблемы, связанные со смещением (offset),
- использование небезопасных функций, таких как gets, strcpy, memcpy;

4.6. CppCheck

CppCheck – это бесплатный открытый кроссплатформенный статический анализатор кода на C и C++, распространяемый по лицензии GPLv3. Главными целями проекта CppCheck является обнаружение ошибок, пропускаемых компиляторами, и снижение числа ложноположительных результатов вкупе с высокой скоростью проверки **[36]**.

Данное средство доступно для ОС Windows и UNIX-подобных операционных систем, может быть интегрировано со многими IDE, такими как CLion, Code::Blocks, Microsoft Visual Studio и Eclipse, а также с системами контроля версий, например Git, Tortoise SVN, Mercurial. CppCheck обладает своим GUI (graphical user interface, графический

интерфейс пользователя), но может быть запущен из командной строки. Результат проверки может быть выведен в стандартный поток вывода или сохранён в файл.

Спектр обнаруживаемых ошибок не уступает дорогостоящим проприетарным средствам и помимо указанных в критерии включает в себя:

- возможность утечки памяти,
- использование небезопасных функций,
- проверка пользовательских свойств и др;

4.7. PC-Lint и FlexeLint

PC-Lint и FlexeLint – коммерческие статические анализаторы исходного кода для ОС Windows и UNIX-подобных операционных систем (Linux, Mac OS X, Solaris, SunOS) соответственно, последние версии которых были выпущены в сентябре 2008 года **[35]**.

Эти средства представляют собой утилиты командной строки, результатом работы которых является список подозрительных или откровенно ошибочных фрагментов исходного кода. PC-Lint находит очень большое количество результатов, больша́я часть которых является ложноположительной, однако могут быть обнаружены и действительные проблемы, которые пропустили более новые и сложные анализаторы, такие как PVS-Studio или Microsoft /analyze **[21]**. Борьба же с ложноположительными и ненужными результатами анализа является трудоёмким процессом, требующим наличия опыта работы с данным средством для его тонкой настройки.

Множество находимых дефектов очень широко и содержит в себе все вышеописанные проблемы, которые находят другие средства статического анализа.

4.8. Polyspace Bug Finder

Polyspace Bug Finder – коммерческое средство статического анализа для программ на C, C++ и Ada, которое при помощи абстрактной интерпретации находит ошибки времени исполнения либо доказывает их отсутствие в исходном коде, а также проверяет соответствие исходного кода стандартам программирования **[39]**.

Данный анализатор имеет свой собственный пользовательский интерфейс, может быть интегрирован в IDE Eclipse, но его также можно использовать полностью из командной строки. Как и большинство других анализаторов, можно ознакомиться с

пробной версией Polyspace Bug Finder после оставления заявки на официальном сайте производителя.

Эта программа обрабатывает большой спектр дефектов в исходном коде ПО, помимо удовлетворяющих критерию 5:

- утечки памяти,
- дефекты, относящиеся к безопасности ПО, параллельной обработке данных и др.

4.9. KlocWork

KlocWork – это проприетарное средство статического анализа, нацеленное на обнаружение проблем безопасности и надёжности в больших, “промышленных” программах, написанных на C, C++, C# и Java **[48]**.

Средство представляет собой множество плагинов для интеграции в IDE, отвечающих за статический анализ исходного кода на предмет дефектов, подсчёт метрик, рефакторинг кода и многое другое. KlocWork доступен для всех распространённых ОС, помимо использования его вместе с какой-либо IDE предлагается утилита kwcheck для работы из командной строки. Средство является проприетарным, а получение пробных версий доступно только организациям. Существует возможность отправки своего кода для анализа производителю KlocWork, для чего нужно оформить заявку и ждать ответа представителя компании.

Поскольку KlocWork является мощным, промышленным средством анализа исходного кода, список обнаруживаемым им дефектов обширен и включается в себя все проблемы, обнаруживаемые и другими средствами статического анализа.

4.10. Splint

Splint – бесплатная утилита с открытым исходным кодом для статического анализа исходного кода программ на C на предмет угроз безопасности и ошибок программирования. Анализатор предназначен для любых UNIX-подобных систем, оперируется из командной строки и по сути своей представляет современную версию утилиты для поиска подозрительного кода **lint**, появившейся в седьмой версии ОС Unix в 1979 году. Сообществом были разработаны GUI и версии для других ОС, однако последний стабильный релиз состоялся в 2007 году **[50]**.

В документации к данному средству указано, что оно способно проводить и более серьёзные проверки, чем соответствующие критерию 5, пользовательские проверки, а результат проверок может быть улучшен. Это достигается при помощи внесения в проверяемый исходный код комментариев-аннотаций, выражающих предположения об определённых функциях, переменных, типах и параметрах.

4.11. Итоги обзора

В начале раздела 4 были сформулированы критерии проведённого обзора. Итоги обзора представлены в таблице 2.

	Работа только с файлами	Доступность	Текстовый вывод	Управление из командной строки	Поиск ошибок из критерия 5
Astrée	+	-	+	+	+
BLAST	-	+	+	+	-
Coverity	-	-	+	+	+
PVS-Studio	-	-	+	+	+
Clang	-	+	+	+	+
CppCheck	+	+	+	+	+
PC-Lint	+	-	+	+	+
Polyspace	+	-	+	+	+
KlocWork	+	-	+	+	+
Splint	±	+	+	+	+

Таблица 2. Итоги обзора средств статического анализа.

Как видно из итоговой таблицы, для поставленной в рамках ВКР задачи подходят только CppCheck и Splint, однако Splint является более старым средством, и для его успешной работы желательно вносить аннотации в исходный код, в то время как CppCheck

– активно развивающийся проект, сильными сторонами которого являются скорость работы и работа по снижению числа ложных результатов.

Остальные средства либо не работают только с файлами исходного кода, что может послужить источником проблем при автоматизации, либо доступны в лучшем случае в виде пробной версии.

Помимо представленных в итоговой таблице средств было рассмотрено ещё двенадцать статических анализаторов (lint, Parasoft C/C++ Test, LDRA Testbed & TBvision, QA-C, Cantata, Coccinelle, CppDepend, ÉCLAIR, Fluctuat, Frama-C, SLAM, Sparse), но их использование невозможно по аналогичным причинам с теми средствами, что уже есть в таблице, и соответственно их полноценное описание в обзоре практически привело бы к повторению описания уже ранее описанного средства, поскольку достоинства и недостатки у них аналогичны:

- средство является проприетарным и кроме того может не иметь возможность работы через командную строку,
- спектр находимых дефектов отличается от приведённого в критериях в меньшую сторону – например средство Fluctuat нацелено только на поиск ошибок работы с числами с плавающей точкой;

5. Алгоритм анализа репозитория

Поскольку в качестве внутрипрограммного представления было выбрано абстрактное синтаксическое дерево, а определение 2 параметризовано, сначала необходимо определить расстояние между фрагментами исходного кода.

5.1. Наиболее специализированный общий шаблон

В работах **[2]** и **[13]** было предложено использовать понятие шаблона (pattern) $\mathbf{T_p}$ – абстрактного синтаксического дерева, в листьях которого вместо операндов стоят специальные метки-заполнители, означающие, что вместо этой метки может быть подставлено поддерево. $\mathbf{Y(T_p) = \{ ?_1, \ldots, ?_k\}}$ – множество меток-заполнителей шаблона. АСД является шаблоном, множество Y() которого не содержит меток.

Подстановкой назовём множество $\mathbf{S = \{ (?_1 , T_s^1), \ldots, (?_k, T_s^k) \}}$ пар **(метка, поддерево)**, устанавливающих соответствие между меткой-заполнителем в некотором дереве и поддеревом, которое следует подставить на место этой метки. $\mathbf{S(?_i)}$ возвращает поддерево $\mathbf{T_s^i}$, соответствующее метке $\mathbf{?_i}$. Результатом $\mathbf{S(T_p)}$ применения подстановки **S** к дереву $\mathbf{T_p}$ является дерево $\mathbf{T_p'}$, полученное путём замены меток-заполнителей $\mathbf{?_i}$ из множества $\mathbf{Y(T_p)}$ на соответствующие им поддеревья $\mathbf{S(?_i)}$.

Например, результат применения подстановки $S = \{(?_1, +(a,b)), (?_2, 3)\}$ к шаблону $T_p = -(?_1, ?_2)$ это дерево $S(T_p) = -(+(a,b), 3)$.

Общим шаблоном (common pattern) двух деревьев T_1 и T_2 называется такой шаблон T_{cp}, что существуют такие подстановки S_1 и S_2, что $S_1(T_{cp}) = T_1$ и $S_2(T_{cp}) = T_2$. Повторно приведём рисунок 4, содержащий пример общего шаблона:

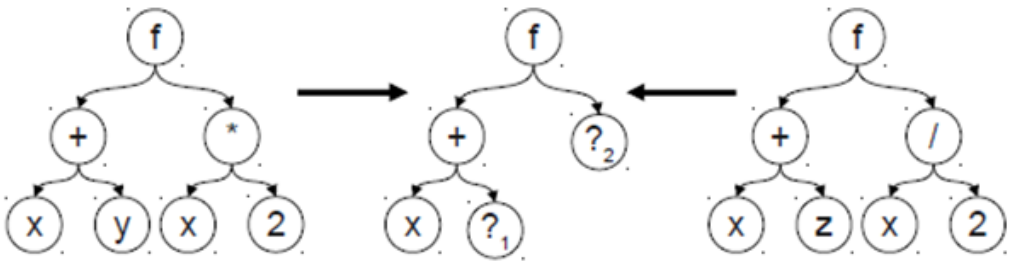

Рисунок 4. Пример общего шаблона двух деревьев

Определение 3. Наиболее специализированным общим шаблоном (most specialized common pattern, НСОШ, **[2]**, **[13]**, **[25]**, **[26]**) двух деревьев T_1 и T_2 называется такой общий

шаблон этих деревьев $\mathbf{T_{mscp} = MSCP(T_1, T_2)}$, что для любого другого общего шаблона этих деревьев T_{cp} существует такая подстановка S, что $S(T_{cp}) = T_{mscp}$. Практически, наиболее специализированный общий шаблон двух деревьев максимально отражает структуру этих двух деревьев одновременно из всех их общих шаблонов.

Очевидно, что для любых двух деревьев их НСОШ существует и определён с точностью до переименования меток-заполнителей. На рисунке 4 приведён пример не только общего шаблона двух деревьев, но и их НСОШ, так как, например, шаблон $T_{cp} = f(?_1, ?_2)$ тоже является их общим шаблоном.

5.2. Расстояние между фрагментами исходного кода

Пусть $\mathbf{c_l}$ – число листьев с операндами в дереве, а $\mathbf{c_p}$ – число листьев с метками-заполнителями. Тогда размер дерева T определим, как $\mathbf{|T| = 0.5 * c_p + c_l}$. По этому определению размер АСД равен числу вхождений имён переменных и констант в программу и используется потому, что оно инвариантно относительно способа представления фрагмента исходного кода в виде абстрактного синтаксического дерева **[2]**.

Коэффициент **0.5** используется потому, что шаблон тоже является деревом и в определении расстояния, которое последует далее, благодаря такому определению размера дерева, расстояние между деревьями будет равно 0, только если АСД полностью идентичны или если два шаблона отличаются только переименованием меток заполнителей. На практике, поскольку в алгоритме будет вычисляться размер подставляемых на место меток-заполнителей поддеревьев, в которых не будет новых заполнителей, можно отказаться от учёта $\mathbf{c_p}$ и определить размер дерева $\mathbf{|T| = c_l}$.

Определение 4 [2, 13]. Пусть $\mathbf{T_{mscp} = MSCP(T_1, T_2)}$ – наиболее специализированный общий шаблон двух деревьев $\mathbf{T_1}$ и $\mathbf{T_2}$, соответствующих фрагментах исходного кода F_1 и F_2, $\mathbf{Y = Y(T_{mscp})}$ – множество меток-заполнителей в T_{mscp}, а подстановки S_1 и S_2 таковы, что $\mathbf{S_i(T_{mscp}) = T_i, i = 1,2}$. Тогда расстоянием между фрагментами F_1 и F_2 назовём число, равное суммарному размеру подставляемых S_1 и S_2 в T_{mscp} поддеревьев, из которого вычтено число меток-заполнителей в T_{mscp}. То есть расстояние между фрагментами F_1 и F_2 равно $\mathbf{\rho(F_1, F_2) = \sum_{p \in Y}|S_{1(p)}| + \sum_{p \in Y}|S_{2(p)}| - |Y|}$.

Например, для рисунка 4 имеем S_{Left} = { $(?_1, y)$, $(?_2, *(x, 2))$ }, S_{Right} = { $(?_1, z)$, $(?_2, /(x, 2))$ }. Тогда $\rho(F_{Left}, F_{Right}) = (1 + 2) + (1 + 2) – 2 = 4$.

Мощность множества меток вычитается из суммарного размера поддеревьев затем, чтобы не совпадали расстояния между парами фрагментов, суммарные размеры подставляемых в АСД поддеревьев для которых совпадают, но не равны числа меток в их НСОШ. Например, рассмотрим две пары фрагментов, высота АСД которых равна 3, а число листьев равно 8. Пусть в первой паре отличие состоит в замене двух поддеревьев высоты 2 (то есть их НСОШ имеет высоту 1), а во второй – 4 поддеревьев высоты 1 (НСОШ высоты 2). Тогда для первой пары расстояние будет равно 4 + 4 – 2 = 6, а для второй 2 + 2 + 2 + 2 – 4 = 4. Очевидно, что чем меньше высота НСОШ по сравнению с исходными АСД, тем труднее отобразить одним шаблоном структуру сразу двух деревьев, а значит и расстояние между фрагментами должно быть больше.

Определённое таким образом расстояние отражает структуру фрагментов и чувствительно к замене больших поддеревьев или изменению порядка поддеревьев в деревьях, соответствующих фрагментам.

5.3. Алгоритм

Опишем предлагаемый алгоритм поиска клонов в репозитории проекта. Будем использовать обозначения, введённые в пункте 2.1, посвящённом формальной постановке. При начале работы алгоритма известны только множество версий R – весь репозиторий проекта, и множество версий P, при помощи которых необходимо провести инициализацию.

1. В первую очередь необходимо задать число $C \geq 0$, называемое контекстом, которое хранит информацию о том, фрагменты какого размера будут обрабатывать в программе.
2. Затем задаётся ограничение L на расстояние между фрагментами ($\rho(F_1, F_2) \leq L$) числом или формулой.
3. На следующем этапе запускается процесс обработки статическим анализатором версий из множества P, после чего мы получим информацию о положении фрагментов в файлах версий, заключающуюся в номере строки, в которой находится центр фрагмента, в виде множества marked(P).
4. На этапе формирования кластеров клонов кода для каждого элемента из множества marked(P) будет считан фрагмент исходного кода размера 2*C + 1 – по C строк перед и после центральной, помеченной строки из соответствующего файла и версии, после чего будет выполнена простая однопроходная инициализация – очередной

фрагмент проверяется на принадлежность существующим кластерам при помощи формирования НСОШ с первым элементом кластера (оригиналом кластера) и вычисления расстояния между текущим фрагментом и оригиналом, после чего информация о его положении в репозитории в формате (версия, файл, положение в файле) либо добавляется в существующий кластер (если расстояние удовлетворяет ограничению L), либо инициализирует новый.

5. Далее, для всех версий V_i из R, таких что $i > p_1, i \neq p_2, \ldots, p_m$, необходимо для каждого файла текущей версии, считывая скользящим окном размера 2*C + 1 фрагменты, проверять принадлежность фрагментов исходного кода уже инициализированным кластерам. В том случае, если расстояние между оригиналом кластера и текущим фрагментом удовлетворяет ограничению L, то в кластер записывается информация о найденном клоне в формате (версия, файл, положение в файле) и окно сдвигается на 2*C + 1 строку вперёд, а если фрагмент не подходит ни в один кластер, то окно сдвигается на 1 строку.

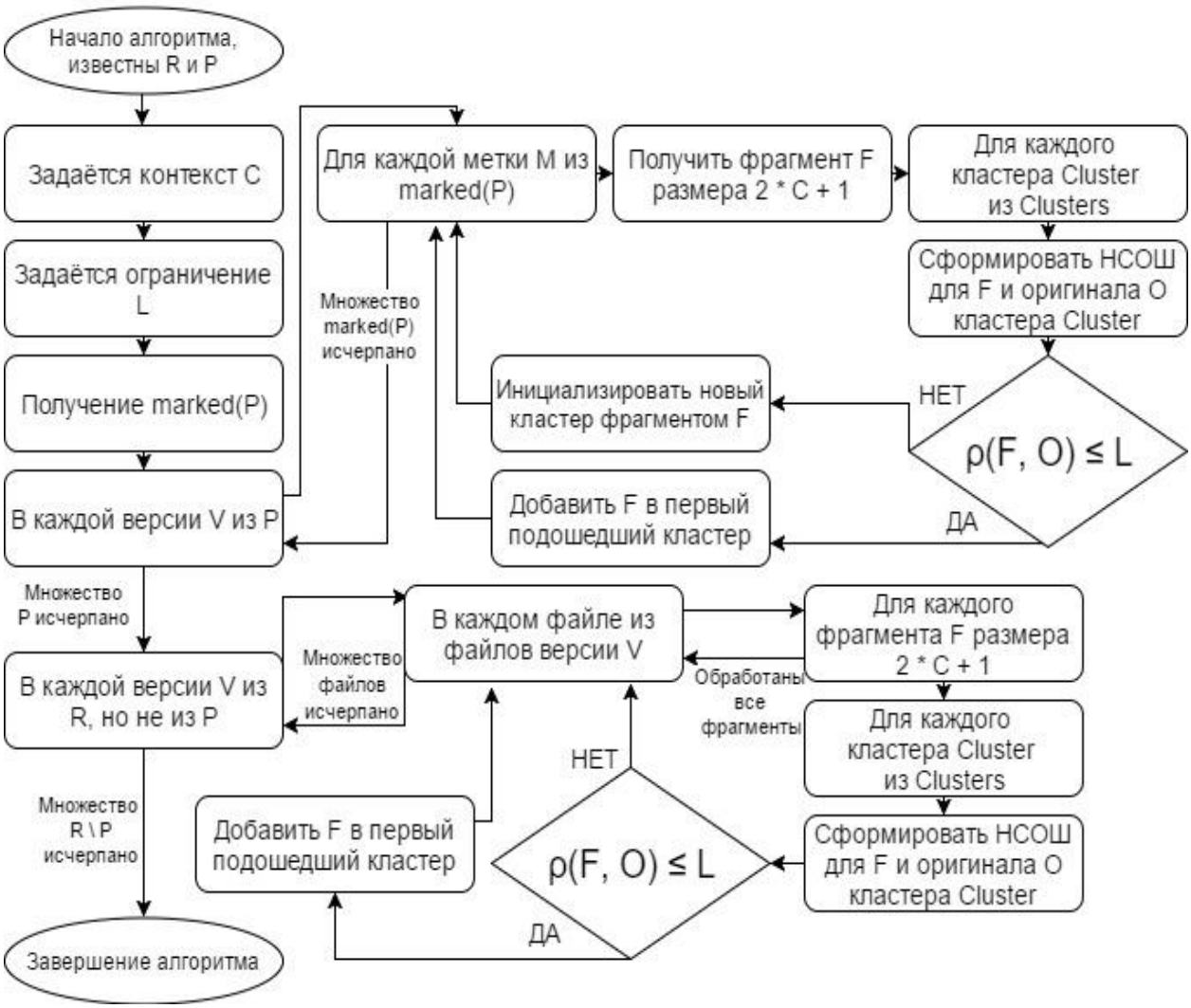

Рисунок 6. Схема работы алгоритма.

На рисунке 6 представлена схема работы предложенного алгоритма.

По завершении работы алгоритма, мы получим множество кластеров, в каждом из которых, упорядоченно согласно возрастанию номера версии, будет содержаться информация о положении в репозитории фрагментов исходного кода, расстояние по определению 4 от которых до оригинала кластера удовлетворяет ограничению L.

6. Описание реализации

Рассмотрим подробности реализации предложенного алгоритма.

Программа была написана на языке C++ стандарта 2011 года с использованием библиотеки Boost для получения списка путей к расположению файлов с подходящим расширением в директории (и её поддиректориях всех уровней вложенности) и Linux-специфичных системных вызовов для выполнения команд Git и запуска внешних по отношению к программе на C приложений. Также, часть проекта была реализована на Python3 с использованием PyCParser и AnyTree для получения фрагмента кода в виде абстрактного синтаксического дерева и перевода его в внутрипрограммное представление для последующего сравнения фрагментов соответственно. Язык Python был выбран потому, что написание модуля сравнения на этом языке позволяло значительно сократить время разработки программы, реализующей алгоритм, предложенный в разделе 5 **[53], [67]**.

В качестве системы управления версиями, как уже было упомянуто, был выбран Git, поскольку это легковесная система, хранящая версии только в виде изменений по сравнению с предыдущей версией, и для хранения даже достаточно больших репозиториев не требуется большого количества свободного места в памяти компьютера. Кроме того, это очень популярная система и на сетевых ресурсах для хранения открытых репозиториев, таких, как GitHub, уже размещено огромное количество проектов на C, на которых можно провести экспериментальное исследование.

Для статического анализа исходного кода используется утилита CppCheck, причины выбора которой подробно описаны в разделе 3.

Для анализа были выбраны проекты на языке C в силу большого количества уже существующих репозиториев, что предоставляет широкий простор для выбора проектов, участвующих в экспериментальном исследовании, более высокой сложности для синтаксического разбора по сравнению с некоторыми другими языками, и наличием сложностей, общих для многих других языков.

В силу выбора языка C в рамках ВКР решалась в том числе и проблема формирования АСД для *произвольного* фрагмента исходного кода, поскольку не было обнаружено готовых средств, обладающих таким функционалом. Препятствиями к разбору произвольных фрагментов кода являются, например, возможность нарушения баланса фигурных скобок в фрагменте, означающих границы составного оператора, возможность

наличия пользовательских типов данных и существования разных стилей оформления исходного кода.

6.1. Функционирование реализации

Реализация соответствует алгоритму из раздела 5 и работает следующим образом:

1. После запуска программы с корректными параметрами командной строки формируются списки версий-инициализаторов и версий, которые необходимо проверить,
2. Происходит инициализация множеств (кластеров) клонов кода при помощи запуска CppCheck для всех файлов с подходящим расширением (.c) во всех версиях-инициализаторах и добавления в множества фрагментов, центры которых пометил утилита, размера 2 * C + 1. После считывания фрагмента, проверяется баланс фигурных скобок и наличие пользовательских типов, формируется окончательный вид фрагмента, который затем препроцессируется и отправляется модулю на Python. При добавлении фрагмента в кластер возможно два случая:
    a. Фрагмент является клоном первого элемента одного из существующих кластеров (т.н. оригиналов этих кластеров) и добавляется в этот кластер,
    b. Фрагмент не является клоном ни одного из оригиналов кластеров и инициализирует новый кластер;
3. Запускается процесс поиска клонов кода оригиналов созданных кластеров в оставшихся версиях. Для этого аналогично шагу 2 в каждом файле считываются фрагменты подходящего размера, и сравниваются с оригиналами существующих кластеров. Аналогично шагу 2 возможно два случая:
    a. Фрагмент является клоном оригинала кластера и добавляет в этот кластер,
    b. Фрагмент не является клоном ни одного из оригиналов кластеров и пропускается;
4. В части реализации, написанной на Python, для двух сравниваемых фрагментов получаются их представления в формате абстрактных синтаксических деревьев, после чего по определению 4 вычисляется расстояние между фрагментами и сверяется с критерием, по которому

принимается решение, является ли фрагменты клонами. Ответ возвращается модулю на C++,

5. Получив решение модуля сравнения фрагмент добавляется в кластер, в случае положительного ответа, и происходит переход к следующему фрагменту кода.
6. После анализа всех версий при помощи GraphViz производится визуализация кластеров;

6.2. Взаимодействие модулей программы

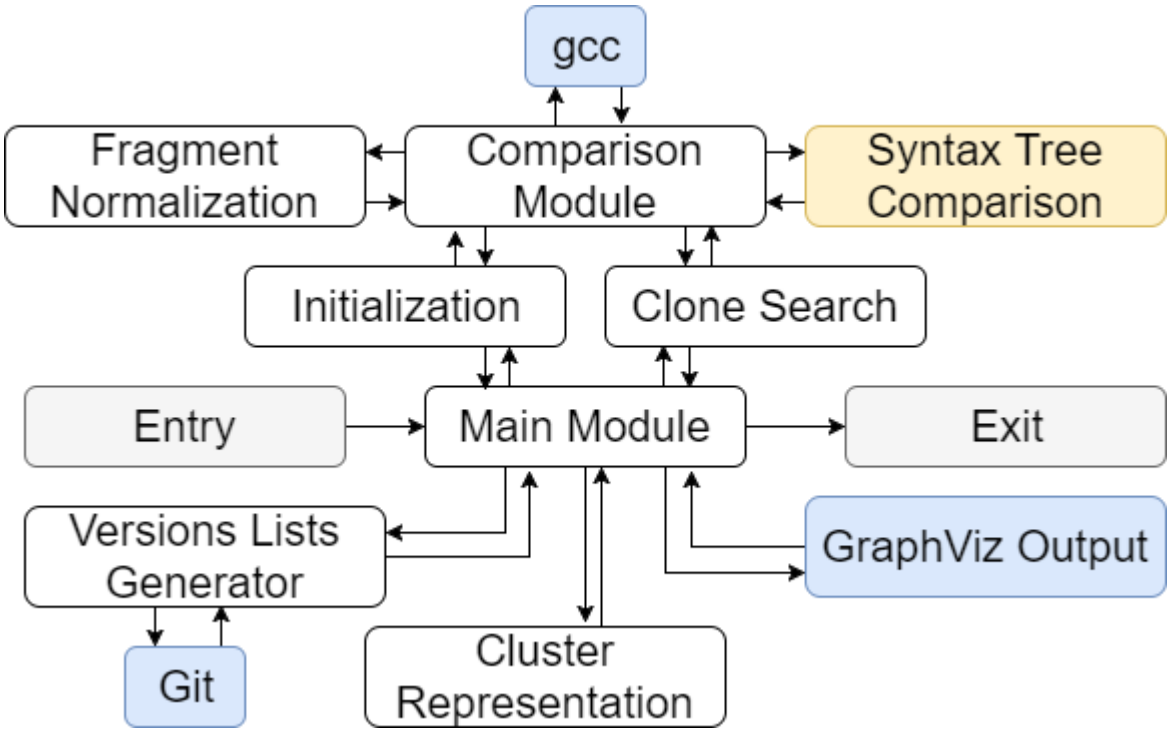

Рисунок 7. Схема взаимодействия модулей программы.

Рассмотрим схему взаимодействия модулей программы, приведённую выше на рисунке 6. Синим отмечены сторонние утилиты:

- Git – выбранная система управления версиями, используемая в том числе для получения списков-версий инициализаторов и версий, которые необходимо проанализировать,
- GraphViz – утилита для генерации графов по входным текстовым файлам специального формата, к которому приводится содержимое кластеров,
- gcc – используется для препроцессирования исходного кода, что необходимо для получения представления в виде абстрактного синтаксического дерева,
- CppCheck – утилита статического анализа, позволяющая найти в файлах с исходным кодом в версиях-инициализаторах положение проблемных фрагментов;

Опишем работу каждого модуля:

1) Main Module:

Данный модуль является главным в программе и запускает процессы инициализации и поиска клонов кода. В первую очередь главный модуль считывает аргументы командной строки и завершает работу программы в том случае, если параметры были заданы некорректно. После чего заполняют списки версий-инициализаторов и версий, которые необходимо проанализировать, обращаясь к модулю Versions Lists Generator. Используя полученные списки версий производится инициализация, посредством обращения к модулю Initialization, и поиск клонов кода, обращаясь к модулю Clone Search. После завершения анализа главный модуль обращается к модулю подготовке содержимого кластеров к визуализации Cluster Representation, затем к модулю визуализации GraphViz Output и завершает работу программы.

2) Versions Lists Generator:

Данный модуль генерирует списки версий, которые необходимо использовать при инициализации и поиске клонов кода на основании информации, полученной от Main Module, в виду файла со списком уникальных идентификаторов версий-инициализаторов.

3) Fragment Normalization:

Данный модуль производит нормализацию фрагмента исходного кода, то есть при необходимости восстанавливает баланс фигурных скобок, определяет наличие в нём пользовательских типов данных и добавляет их псевдоопределения в фрагмент, а также добавляет.

4) Syntax Tree Comparison:

Данный модуль написан на Python3 и производит сравнение двух фрагментов исходного кода путём формирования их абстрактных синтаксических деревьев, их наиболее специализированного общего шаблона, вычисления расстояния между этими деревьями согласно определению 4 и проверки того, удовлетворяет ли найденное расстоянию критерию отбора клонов кода.

5) Comparison Module:

Данный модуль производит подготовку двух фрагментов к сравнению – их нормализацию, обращаясь к модулю Fragment Normalization, препроцессирование, передачу модулю сравнения фрагментов Syntax Tree Comparison.

6) Initialization:

Данный модуль производит инициализацию непересекающихся множеств клонов кода (кластеров) на основании списка версий-инициализаторов при помощи утилиты CppCheck. Для каждой версии из списка все файлы исходного кода анализируются CppCheck, а результат, указывающий на положение ошибок в исходном коде, сохраняется. После этого в исходном коде локализуется каждый фрагмент размера 2 * C + 1 (где C – размер контекста – один из параметров алгоритма), на который указал CppCheck, и добавляется в соответствующий кластер, если является клоном уже добавленного фрагмента, или инициализирует новый кластер. Проверка того, является ли обрабатываемый в данный момент фрагмент кода клоном, осуществляется путём обращения к модулю Comparison Module.

7) Clone Search:

Данный модуль производит поиск клонов кода тех фрагментов, которые были добавлены в кластеры на этапе инициализации, в файлах с исходным кодом версий из списка версий, которые необходимо проанализировать, полученного от главного модуля. Для этого все фрагменты размера 2 * C + 1 (где C – размер контекста – один из параметров алгоритма) сравниваются с существующими кластерами, путём обращения к модулю Comparison Module, и добавляются к кластеру только в том случае, если модуль сравнения дал положительный ответ.

8) Cluster Representation:

В этом модуле производится запись содержимого кластеров в текстовые файлы специального формата, для последующей передачи их модулю GraphViz Output и генерации изображений, представляющих содержимое кластеров, при помощи утилиты GraphViz.

9) GraphViz Output:

Данный модуль производит вызов утилиты GraphViz для каждого файла, представляющего кластер, созданного модулем Cluster Representation.

Работа каждого модуля описана при помощи псевдокода в приложении А.

7. Экспериментальное исследование.

В данном разделе будут изложены цели экспериментального исследования и кратко описаны проведённые эксперименты.

7.1. Цель исследования

Целью исследования является выяснение способности реализации предложенного алгоритма обнаруживать ошибки, для чего будет проведено сравнения количества и качества результата работы с поверсионным применением утилиты CppCheck, а также оценка зависимости времени выполнения анализа от объёма входных данных.

Прежде, чем перейти к описанию проведённых экспериментов, следует заметить, что при написании части реализации, отвечающей за формирование АСД сравниваемых фрагментов исходного кода, получения НСОШ и вычисление расстояния между фрагментами, был выбран язык программирования Python, в силу его удобства для реализации нужного функционала. Однако, благодаря общемировой статистике, собранной за всё время использования языка Python, известно, что реализация алгоритмов на языке Python значительно уступает в скорости реализации этих алгоритмов на C++ **[52]**.

Помимо этого, использование модуля на Python требует предварительного приведения фрагмента к нормализованному виду и препроцессирования, что также несколько замедляет работу средства, но не сравнимо с вкладом Python-модуля.

Соответственно, в связи с использованием медленной реализации части алгоритма, сравнение времени работы реализации предложенного алгоритма с поверсионным применением утилиты CppCheck не является целесообразным.

7.2. Подготовленный репозиторий.

В качестве первого исследуемого проекта использовался специально созданный небольшой тестовый репозиторий, содержащий 10 версий. В файлах исходного кода данного репозитория содержались специально внесённые ошибки, такие как выход за границы массива, деление на 0, использование неинициализированных переменных. Данный проект не обладал никаким реальным функционалом, так как целью его анализа являлось исследование работоспособности созданного средства. Соответственно, можно было обеспечить бóльшую плотность ошибок в исходном коде программы, чем это

наблюдается в реальных проектах. Были подвергнуты анализа 2 последние версии, 5 последних версий и все 10 версий.

Контекст при анализе был равен 1, то есть сравнивались фрагменты размера не больше 3 операторов, а условие, накладываемое на расстояние было взято так, чтобы клонами считались только фрагменты, отличающиеся в операндах.

При анализе репозитория CppCheck обнаружил для 2, 5 и 10 версий 13, 31 и 41 ошибку соответственно, а реализация алгоритма во всех случаях дополнительно обнаруживала 2 ошибки, пропущенные CppCheck, так как данное средство не обнаружило деление на 0, когда значение 0 содержит переменная, и выход за границу массива, когда адресация производится при помощи переменной.

7.3. Реализация алгоритма, предложенного в рамках ВКР.

Для этого эксперимента использовалась копия репозитория написанной в рамках ВКР программы. Несмотря на то, что она написана на C++, созданная реализация может производить поиск клонов, игнорируя незнакомые конструкции.

Поскольку на момент эксперимента репозиторий содержал 56 версий с большим количеством исходного кода, было решено провести эксперимент с 2 и 7 последними версиями.

Контекст был равен 5, то есть размер сравниваемых фрагментов не превышал 11 операторов, в качестве ограничения на расстояние между фрагментами при сравнении было выбрано среднее геометрическое размеров сравниваемых фрагментов с точки зрения АСД.

При этом на протяжении последних 12 версий в репозитории присутствовал файл с примером исходного кода, предназначенный для тестирования и отладки, содержавший две ошибки. И CppCheck, и разработанное средство нашли одинаковое число ошибок – 4 и 14 для 2 и 7 версий соответственно.

7.4. Cranium.

В данном случае рассматривался открытый репозиторий библиотеки искусственных нейронных сетей с прямой связью **Cranium**, загруженной с ресурса GitHub **[54]**. Репозиторий содержал всего 49 объёмных версий и было проведено два эксперимента - с последними 2 версиями и последними 5 версиями.

Размер контекста составлял, то есть размер сравниваемых фрагментов не превышал 11 операторов, в качестве ограничения на расстояние между фрагментами было выбрано среднее геометрическое размеров АСД, соответствующих сравниваемым фрагментам.

При этом CppCheck и разработанное средство обнаружили 28 и 69 ошибок для 2 и 5 версий соответственно.

7.5. CCWT.

В этом эксперименте производился анализ репозитория библиотеки комплексного непрерывного вейвлет-преобразования **CCWT (Complex Continuous Wavelet Transformation)**, загруженного с ресурса GitHub **[55]**. Репозиторий содержал всего 33 версии, анализ был проведён для 2,5 и 10 последних версий.

При проведении эксперимента размер контекста составлял 5 операторов, в качестве ограничения, накладываемого на расстояние, было выбрано среднее геометрическое размеров сравниваемых фрагментов.

Оба средства нашли совпадающие множества ошибок – 2, 5 и 10 ошибок для 2, 5 и 10 последних версий соответственно.

7.6. Tiny JPEG.

Для данного эксперимента с ресурса GitHub был загружен репозиторий проекта **Tiny JPEG**, реализующего базовый алгоритм кодирования JPEG **[56]**. Репозиторий содержал 42 версии, анализ был проведён для 7 и 10 последних версий.

После ознакомления с примерами исходного кода из этого репозитория размер контекста был выбран равным 3, в качестве ограничения на вычисленное расстояние между фрагментами снова было выбрано среднее геометрическое размеров сравниваемых фрагментов.

Оба средства обнаружили одинаковые ошибки – 1 и 4 ошибки для 7 и 10 последних версий репозитория.

7.7. LibGOST 15.

В данном эксперименте был рассмотрен репозиторий проекта, реализующего алгоритм блочного шифрования “Кузнечик”, **libgost15**, загруженный с ресурса GitHub **[57]**.

Репозиторий содержал 80 версий с большим количеством исходного кода, поэтому анализу подверглись только 2, 5 и 7 последних версий.

Размер контекста в данном эксперимента был равен 3, расстояние между сравниваемыми фрагментами ограничивалось сверху средним геометрическим размеров деревьев, соответствующих фрагментам.

При этом CppCheck обнаружил 2, 5 и 7 уязвимых мест в исходном коде репозитория для 2, 5 и 10 версий соответственно, а разработанная программа – дополнительно обнаружила 6, 10 и 22 ложноположительных результата. Ложноположительные результаты были найдены потому, что при формировании фрагмента перед сравнением не требуется, чтобы его размер был строго равен 2 * C + 1, где C – контекст, - в случае невозможности достичь нужного размера фрагмент может быть меньше. В связи с этим на этапе реализации множества клонов кода были проинициализированы фрагментами единичного размера, содержащими только оператор возврата. Соответственно, из-за слишком маленького размера оригинального фрагмента многие сравниваемые с ним фрагменты исходного кода могли считаться его клонами.

7.8. C-INI-Parser.

В данном эксперименте был рассмотрен репозиторий синтаксического анализатора INI-файлов C-INI-Parser, загруженный с ресурса GitHub **[61]**. Репозиторий содержал 7 версий, анализу подверглись 2 и 6 последних версий, так как первая версия была инициализирующей и не содержала файлов с исходным кодом.

Контекст был равен 5, расстояние между сравниваемыми фрагментами было ограничено сверху средним геометрическим размеров сравниваемых деревьев.

Оба средства обнаружили равное число ошибок – 4 для 2 версий и 12 для 6 версий.

7.9. Итоги экспериментального исследования.

Подведём итоги экспериментального исследования. Напомним, что целью исследования являлось выяснение способности предложенного алгоритма обнаруживать ошибки по сравнению с поверсионным применением утилиты статического анализа CppCheck и оценка зависимости времени анализа репозитория от объёма входных данных, Сравнение скорости работы с утилитой CppCheck нецелесообразно в связи с медленной

реализацией части алгоритма, отвечающей за сравнение фрагментов исходного кода, на языке Python.

Нижеприведённые таблицы описывают результаты проведённых экспериментов.

Репозиторий	Число версий	Число строк	Число строк без инициализации	Полная версия	Версия без Python	CppCheck
Подготовленный репозиторий	1 + 1	66	35	18.201	0.031	0.067
Подготовленный репозиторий	2 + 3	138	85	46.006	0.073	0.140
Подготовленный репозиторий	2 + 8	187	145	77.448	0.128	0.239
Реализация ВКР	1 + 1	7159	3573	310.184	20.787	26.235
Реализация ВКР	2 + 5	24113	17234	1445.167	44.712	89.131
Cranium	1 + 1	3618	1809	1971.587	5.280	12.702
Cranium	2 + 3	9002	5427	6112.179	7.864	30.628
CCWT	1 + 1	1065	534	53.117	0.307	0.533
CCWT	1 + 4	2608	2094	204.014	0.713	1.248

CCWT	2 + 8	5178	4128	410.591	1.255	2.476
Tiny JPEG	2 + 5	8713	6231	510.065	4.071	13.468
Tiny JPEG	2 + 8	12310	9832	841.280	4.237	20.002
LibGOST 15	1 + 1	51872	25936	70.184	10.064	24.995
LibGOST 15	1 + 4	129680	103744	260.684	11.170	62.655
LibGOST 15	1 + 6	181553	155616	374.684	11.806	84.455
C-INI-Parser	1 + 1	3890	1645	156.509	3.264	4.645
C-INI-Parser	2 + 4	9834	6563	650.634	5.136	17.532

Таблица 3. Время работы средств (в секундах).

Таблица 3 описывает время выполнения анализа репозитория в секундах. В данной таблице для каждого репозитория указано число проанализированных версий в виде суммы числа версий-инициализаторов (левое слагаемое) и числа версий, в которых производился поиск клонов кода (правое слагаемое), суммарное число строк в файлах с исходным кодом в проверенных версиях, число строк в файлах с исходным кодом без учёта версий, при помощи которых проводилась инициализация, и время анализа разработанной программой и версией этой программы с отключенным модулем сравнения, написанным на языке Python.

Видно, что в связи реализацией модуля сравнения фрагментов исходного кода на языке Python, сравнение по скорости анализа с поверсионным анализом при помощи любого другого средства статического анализа нецелесообразно. Время работы CppCheck приведено исключительно для статистики и подтверждения того факта, что зависимость времени его работы от объёма обработанного исходного кода хуже линейной, что легко может быть проверено при помощи интерполяции.

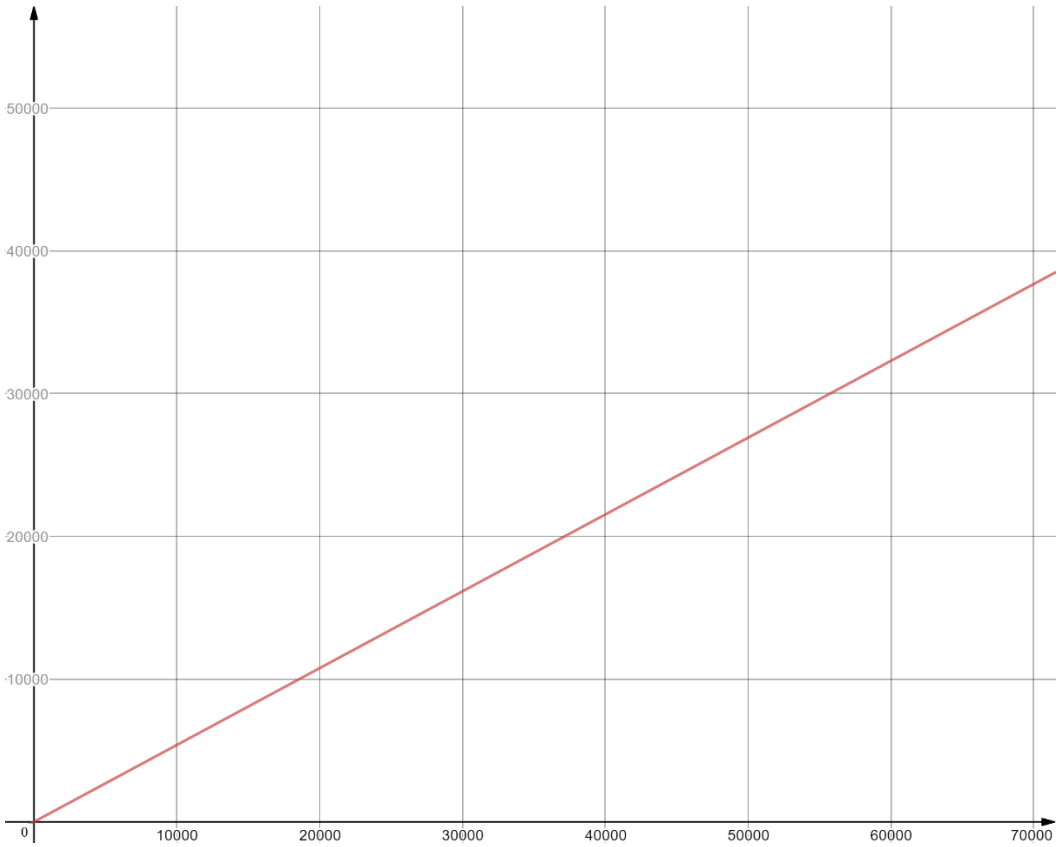

Рисунок 8. График зависимости времени работы от объёма данных для реализации предложенного алгоритма.

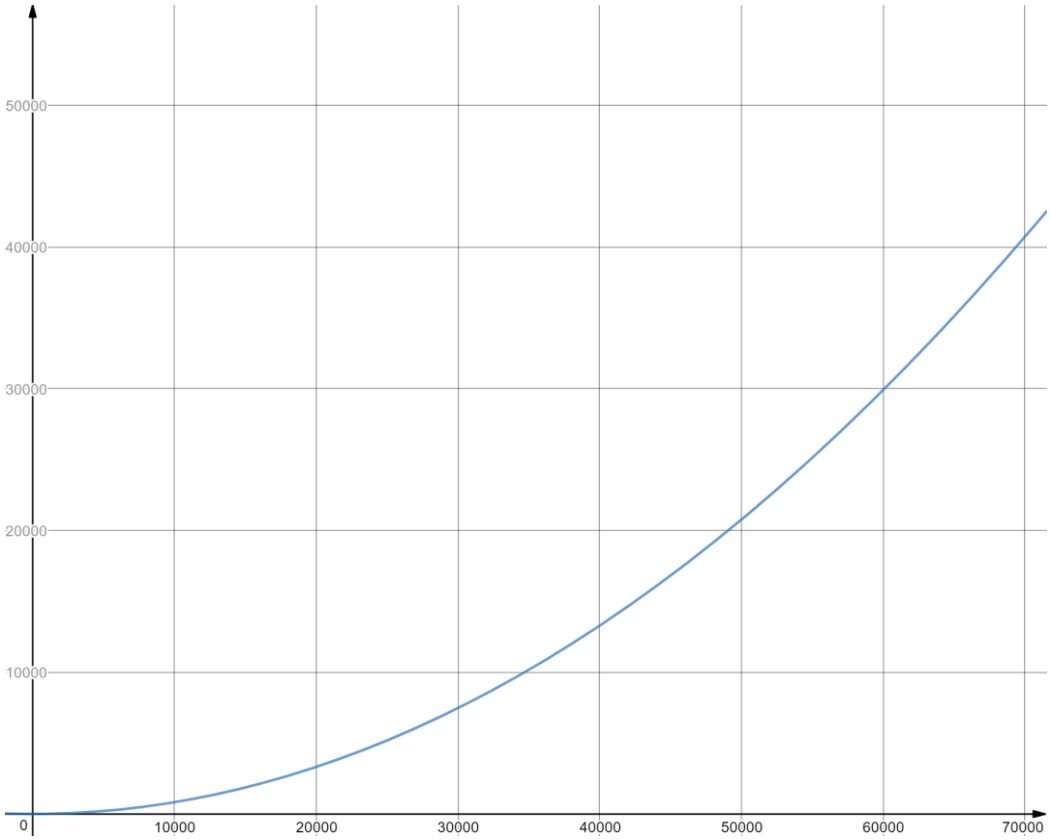

Рисунок 9. График зависимости времени работы от объёма данных для поверсионного применения утилиты CppCheck.

Также из таблицы 3 видно, что зависимость между объёмом исходного кода в анализируемых версиях, исключая версии-инициализаторы, и временем анализа линейна, а погрешность составляет не более 5%. В то же время, зависимость времени выполнения поверсионного анализа при помощи средства статического анализа, такого как CppCheck, от объёма исходного кода в проверяемых версиях хуже линейной. Соответственно, при объёме исходных данных, превышающих значение, при котором время выполнения анализа для предложенного в данной работе алгоритма и поверсионного анализа версий будет равно, предложенный алгоритм будет производить поиск ошибок быстрее средств статического анализа.

На примере подготовленного репозитория из раздела 7.2 данный факт проиллюстрирован на рисунках 8 и 9. Отдельно необходимо заметить, что график времени анализа на рисунке 8 представлен именно для реализации алгоритма и в большей степени предназначен для иллюстрации линейной зависимости времени анализа от объёма исходного кода, нежели для отражения реальной скорости работы алгоритма, то есть для оптимальной реализации предложенного в разделе 5 алгоритма точка пересечения времён работы будет достигаться значительно раньше.

Перейдём к сравнению количества действительных ошибок и ложноположительных результатов, находимых сравниваемыми средствами.

Репозиторий	Число версий	CppCheck	Алгоритм
Подготовленный репозиторий	1 + 1	13	13 + 2д.
Подготовленный репозиторий	2 + 3	31	31 + 2д.
Подготовленный репозиторий	2 + 8	41	41 + 2д.
Реализация ВКР	1 + 1	4	4
Реализация ВКР	2 + 5	14	14
Cranium	1 + 1	28	28

Cranium	2 + 3	69	69
CCWT	1 + 1	2	2
CCWT	1 + 4	5	5
CCWT	2 + 8	10	10
Tiny JPEG	2 + 5	1	1
Tiny JPEG	2 + 8	4	4
LibGOST 15	1 + 1	2	2 + 6 л-п.
LibGOST 15	1 + 4	5	5 + 10 л-п.
LibGOST 15	1 + 6	7	7 + 22 л-п.
C-INI-Parser	1 + 1	4	4
C-INI-Parser	2 + 4	12	12

Таблица 4. Количество найденных ошибок.

В таблице 4 приведены результаты анализа репозиториев посредством утилиты CppCheck и реализации предложенного в данной работе алгоритма, для каждой версии указано количество проанализированных версий. В случае, если в ячейках, соответствующих количеству результатов анализа, стоят равные числа, это означает что множества результатов сравниваемых средств совпали. В том случае, если в ячейке записана сумма чисел, это означает, что помимо некоторого общего множества результатов (без сокращения) были найдены другие – действительные ошибки (д.) или ложноположительные результаты (л-п.).

За счёт подходящего выбора размера контекста и ограничения на расстояние между фрагментами, разработанное средство находило все настоящие ошибки, и обнаружило

ложноположительные результаты только в одном репозитории. Оно также способно обнаруживать фрагменты, приводящие к ошибкам, но пропускаемые утилитой CppCheck.

Однако, предложенный алгоритм очень чувствителен к входным параметрам и при неудачном выборе инициализирующих версий, размера контекста или ограничения, накладываемого на расстояние между фрагментами, предложенный алгоритм может как обнаружить не все действительные ошибки, так и большое количество ложноположительных результатов. Кроме того, возможно задать такой набор входных параметров, что при большом времени анализа репозитория, средство найдёт не все действительные ошибки, а также обнаружит большое число ложноположительных результатов.

Подводя итог, можно сказать, что предложенный в данной работе алгоритм способен конкурировать с поверсионным применением статического анализатора, обнаруживая равное или большее число ошибок, и возможно даже превосходить его в скорости.

В силу использования поиска клонов кода, данный алгоритм также способен обнаруживать те дефекты, которые может пропустить обычный статический анализатор, а кроме того хорошо подходит для поиска плагиата исходного кода в том случае, если производится поиск клонов кода конкретных целевых фрагментов исходного кода фиксированного размера.

Улучшение предложенного в данной работе алгоритма может состоять в использовании всех возможностей систем управления версиями при анализе репозитория, например, замене чтения всех файлов с исходным кодом в каждой анализируемой версии на анализ только изменений в обрабатываемой версии по сравнению с предыдущей обработанной версией, что позволит в разы повысить быстродействие алгоритма. Также развитие может состоять в исследовании способов ограничения расстояния между фрагментами при сравнении их в качестве абстрактных синтаксических деревьев для достижения большей точности при поиске клонов кода. Кроме того, для повышения быстродействия алгоритма целесообразно рассмотреть возможности распараллеливания некоторых частей алгоритма и совмещения использования АСД с другими представления исходного кода, такими как последовательность лексем, что также позволит повысить точность при сравнении фрагментов небольшого размера.

В случае учёта предложенных улучшений при реализации алгоритма, описанного в разделе 5, и без использования медленных реализаций его частей, таких как сравнение

фрагментов на языке Python, можно будет достичь высокой скорости анализа вкупе с высокой точностью обнаружения клонов кода и уменьшением числа не обнаруживаемых и ложноположительных результатов.

8. Заключение

В ходе данной работы были выполнены следующие задачи:

- исследованы существующие подходы к определению клонов кода и способы представления исходного кода
- рассмотрены существующие средства статического анализа
- предложен алгоритм анализа произвольного репозитория проекта на предмет клонирования помеченных фрагментов
- написана реализация предложенного алгоритма
- проведено экспериментальное исследование написанной реализации

В работе было введено параметризованное определение клонов кода, использующее понятие абстрактных синтаксических деревьев и расстояния между ними. Благодаря использованию такого определения при поиске клонов возможно ограничивать множество тех фрагментов исходного кода, которые считаются клонами кода.

В ходе обзора способов представления исходного кода в алгоритме было выбрано представление в виде абстрактных синтаксических деревьев, поскольку только данное представление позволяет осуществлять поиск клонов кода, удовлетворяющих выбранному определению.

После выбора способа представления исходного кода была введено определение расстояния между фрагментами исходного кода, инвариантное относительно способа представления фрагментов в виде абстрактных синтаксических деревьев и чувствительное к изменению порядка выражений в сравниваемых фрагментах.

В ходе обзора средств статического анализа для использования при реализации алгоритма был сделан выбор в пользу средства CppCheck из-за его доступности, скорости проведения анализа и нацеленности на минимизацию количества ложноположительных результатов.

Предложенный алгоритм поиска клонов помеченных фрагментов кода, обнаруженных в инициализирующих версиях, во всех последующих версиях, использующий представление сравниваемых фрагментов в виде абстрактных синтаксических деревьев и введённое определение расстояния при принятии решения о том, являются ли сравниваемые фрагменты клонами, был реализован для Linux-подобных

операционных систем на языках программирования C++ стандарта 2011 года и Python 3.4 для проектов, написанных на языке C стандарта 1999 года, в системе управления версиями git.

После завершения реализации предложенного алгоритма было проведено экспериментальное исследование, показавшее линейную зависимость времени работы алгоритма от объёма анализируемого исходного кода и способность предложенного алгоритма обнаруживать равное с поверсионным применением статического анализатора число ошибок и в некоторых случаях превосходить его.

В завершение стоит отметить, что предложенный алгоритм способен производить поиск клонов любых целевых фрагментов кода, а не только фрагментов, содержащих операции, приводящие к ошибкам и уязвимостям, что может быть полезно при реорганизации исходного кода программы или при поиске плагиата.

Направления развития предложенного алгоритма состоят в анализе исключительно различий между обрабатываемыми версиями, исследовании способов ограничения расстояния между сравниваемыми фрагментам исходного кода, поиске частей алгоритма, которые могут быть подвергнуты распараллеливанию, и исследованию возможностей совмещения выбранного представления исходного кода с другими представлениями. Указанные улучшения позволят повысить скорость работы и точность алгоритма, в результате чего может быть создано полезное при поиске ошибок, уязвимостей, плагиата и при реорганизации кода средство для анализа исходного кода и исследования процесса разработки программ.

9. Литература

1. *Chacon S., Straub B.* Pro git. Second Edition – Apress, 2014. – Pp. 9-131.

2. *Булычев П. Е.* Алгоритмы вычисления отношений подобия в задачах верификации и реструктуризации программ //М.: МГУ им. МВ Ломоносова. – 2010.

3. *Ахин М. Х., Ицыксон В. М.* Обнаружение клонов исходного кода: теория и практика //Системное программирование. – 2010. – Т. 5. – №. 1. – С. 145-163.
4. *Зельцер Н. Г.* Поиск повторяющихся фрагментов исходного кода при автоматическом рефакторинге //Труды Института системного программирования РАН. – 2013. – Т. 25.
5. *Murakami H. et al.* Gapped code clone detection with lightweight source code analysis //Program Comprehension (ICPC), 2013 IEEE 21st International Conference on. – IEEE, 2013. – Pp. 93-102.
6. *Mayrand J., Leblanc C., Merlo E.* Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics //icsm. – 1996. – Т. 96. – P. 244.
7. *Саргсян С. и др*. Масштабируемый инструмент поиска клонов кода на основе семантического анализа программ //Труды Института системного программирования РАН. – 2015. – Т. 27. – №. 1.
8. *Baker B. S.* A program for identifying duplicated code //Computing Science and Statistics. – 1993. – P. 49.
9. *Baker B. S.* On finding duplication and near-duplication in large software systems //Reverse Engineering, 1995., Proceedings of 2nd Working Conference on. – IEEE, 1995. – Pp. 86-95.
10. *Baxter I. D. et al.* Clone detection using abstract syntax trees //Software Maintenance, 1998. Proceedings., International Conference on. – IEEE, 1998. – Pp. 368-377.
11. *Li Z. et al.* CP-Miner: Finding copy-paste and related bugs in large-scale software code //IEEE Transactions on software Engineering. – 2006. – Т. 32. – №. 3. – Pp. 176-192.
12. *Ducasse S., Rieger M., Demeyer S.* A language independent approach for detecting duplicated code //Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on. – IEEE, 1999. – Pp. 109-118.
13. *Evans W. S., Fraser C. W., Ma F.* Clone detection via structural abstraction //Software Quality Journal. – 2009. – Т. 17. – №. 4. – Pp. 309-330.

14. *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. – Pp. 654-670.
15. *Yang W.* Identifying syntactic differences between two programs //Software: Practice and Experience. – 1991. – Т. 21. – №. 7. – Pp. 739-755.
16. *Kontogiannis K. A. et al.* Pattern matching for clone and concept detection //Reverse engineering. – Springer US, 1996. – Pp. 77-108.
17. *Liu C. et al.* GPLAG: detection of software plagiarism by program dependence graph analysis //Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. – ACM, 2006. – Pp. 872-881.
18. *Komondoor R., Horwitz S.* Using slicing to identify duplication in source code //International Static Analysis Symposium. – Springer Berlin Heidelberg, 2001. – Pp. 40-56.
19. *Misra S., Bhavsar V.* Relationships between selected software measures and latent bug-density: Guidelines for improving quality //Computational Science and Its Applications—ICCSA 2003. – 2003. – Pp. 964-964.
20. *Кулямин В. В.* Методы верификации программного обеспечения //Институт системного программирования РАН. – 2008.
21. *Carmack J.* Static code analysis - [Электронный ресурс]. – Электрон. дан. - URL: https://www.viva64.com/en/a/0087/ (дата обращения: 01.05.2017).
22. *Roy C. K., Cordy J. R.* A survey on software clone detection research //Queen's School of Computing TR. – 2007. – Т. 541. – №. 115. – Pp. 64-68.
23. *Фаулер М.* Рефакторинг. Улучшение существующего кода //СПб.: Символ-Плюс,-2008.-423 с. – 2009.
24. *Ахо А. и др.* Компиляторы. Принципы, технологии, инструменты, 2-е издание //М.: Изд-во «Вильямс. – 2008.
25. *Plotkin G. D.* A note on inductive generalization //Machine intelligence. – 1970. – Т. 5. – №. 1. – Pp. 153-163.
26. *Reynolds J. C.* Transformational systems and the algebraic structure of atomic formulas //Machine intelligence. – 1970. – Т. 5. – Pp. 135-152.
27. *Бородин А. Е., Белеванцев А. А.* Статический анализатор Svace как коллекция анализаторов разных уровней сложности //Труды Института системного программирования РАН. – 2015. – Т. 27. – №. 6.
28. *Bessey A. et al.* A few billion lines of code later: using static analysis to find bugs in the real world //Communications of the ACM. – 2010. – Т. 53. – №. 2. – Pp. 66-75.

29. PVS-Studio: Static Code Analyzer for C, C++ and C# - [Электронный ресурс]. – Электрон. дан. - URL: https://www.viva64.com/en/pvs-studio/PVS-Studio/ (дата обращения: 01.05.2017).
30. PVS-Studio: Blog - [Электронный ресурс]. – Электрон. дан. - URL: https://www.viva64.com/en/b/ (дата обращения: 01.05.2017).
31. *Henzinger T. A. et al.* Software verification with BLAST //International SPIN Workshop on Model Checking of Software. – Springer Berlin Heidelberg, 2003. – Pp. 235-239.
32. LLVM: Clang Static Analyzer - [Электронный ресурс]. – Электрон. дан. - URL: http://clang-analyzer.llvm.org/ (дата обращения: 01.05.2017).
33. *Almossawi A., Lim K., Sinha T.* Analysis tool evaluation: Coverity prevent //Pittsburgh, PA: Carnegie Mellon University. – 2006.
34. Free-BSD 11.0: Unix & Linux Commands - man page for lint - [Электронный ресурс]. – Электрон. дан. - URL: http://www.unix.com/man-page/FreeBSD/1/lint/ (дата обращения: 01.05.2017).
35. Gimpel Software: PC-Lint/FlexeLint Product Information - [Электронный ресурс]. – Электрон. дан. - URL: http://www.gimpel.com/html/products.htm/ (дата обращения: 01.05.2017).
36. CppCheck: A tool for static C/C++ code analysis - [Электронный ресурс]. – Электрон. дан. - URL: http://cppcheck.sourceforge.net/ (дата обращения: 01.05.2017).
37. Parasoft C/C++ Test: Static Analysis for C/C++ - [Электронный ресурс]. – Электрон. дан. - URL: https://www.parasoft.com/product/static-analysis-cc/#cust_msgs/ (дата обращения: 01.05.2017).
38. LDRA: Testbed & TBvision - [Электронный ресурс]. – Электрон. дан. - URL: http://www.ldra.com/en/testbed-tbvision/ (дата обращения: 01.05.2017).
39. MathWorks: Polyspace Bug Finder - [Электронный ресурс]. – Электрон. дан. - URL: https://www.mathworks.com/products/polyspace-bug-finder.html/ (дата обращения: 01.05.2017).
40. PRQA: QA-C - [Электронный ресурс]. – Электрон. дан. - URL: http://www.programmingresearch.com/static-analysis-software/qac-qacpp-static-analyzers/ (дата обращения: 01.05.2017).
41. QA-Systems: Cantata - [Электронный ресурс]. – Электрон. дан. - URL: http://www.qa-systems.com/tools/cantata/ (дата обращения: 01.05.2017).
42. The Astrée Static Analyzer - [Электронный ресурс]. – Электрон. дан. - URL: http://www.astree.ens.fr/ (дата обращения: 01.05.2017)

43. Coccinelle: A Program Matching and Transformation Tool for Systems Code - [Электронный ресурс]. – Электрон. дан. - URL: http://coccinelle.lip6.fr/ (дата обращения: 01.05.2017).
44. CppDepend: C/C++ Static Analysis and Code Quality tool - [Электронный ресурс]. – Электрон. дан. - URL: http://www.cppdepend.com/ (дата обращения: 01.05.2017).
45. ÉCLAIR - [Электронный ресурс]. – Электрон. дан. - URL: http://bugseng.com/products/eclair/ (дата обращения: 01.05.2017).
46. Fluctuat - [Электронный ресурс]. – Электрон. дан. - URL: http://www.lix.polytechnique.fr/Labo/Sylvie.Putot/fluctuat.html/ (дата обращения: 01.05.2017).
47. Frama-C - [Электронный ресурс]. – Электрон. дан. - URL: http://frama-c.com/features.html/ (дата обращения: 01.05.2017).
48. KlocWork: Faster delivery of secure, reliable, and conformant code - [Электронный ресурс]. – Электрон. дан. - URL: https://www.klocwork.com/products-services/klocwork/ (дата обращения: 01.05.2017).
49. *Ball T. et al.* SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft //International Conference on Integrated Formal Methods. – Springer Berlin Heidelberg, 2004. – Pp. 1-20.
50. Splint Home Page - [Электронный ресурс]. – Электрон. дан. - URL: http://splint.org/ (дата обращения: 01.05.2017).
51. Sparse - [Электронный ресурс]. – Электрон. дан. - URL: https://sparse.wiki.kernel.org/index.php/Main_Page/ (дата обращения: 01.05.2017).
52. The Computer Language Benchmark Game: Python 3 vs C++ g++ - [Электронный ресурс]. – Электрон. Дан. – URL: http://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=python3&lang2=gpp/ (дата обращения: 09.05.2017)
53. Wolfram Blog – Code Length Measured in 14 Languages – [Электронный ресурс] – Электрон. дан. - URL: http://blog.wolfram.com/2012/11/14/code-length-measured-in-14-languages/ (дата обращения: 09.05.2017)
54. GitHub – 100/Cranium – [Электронный ресурс]. – Электрон. дан. – URL: https://github.com/100/Cranium/ (дата обращения: 09.05.2017)
55. GitHub - Lichtso/CCWT – [Электронный ресурс]. – Электрон. дан. - URL: https://github.com/Lichtso/CCWT/tree/master/ (дата обращения: 09.05.2017)
56. GitHub - serge-rgb/TinyJPEG – [Электронный ресурс]. – Электрон. дан. - URL: https://github.com/serge-rgb/TinyJPEG/ (дата обращения: 09.05.2017)

57. GitHub - aprelev/libgost15 – [Электронный ресурс]. – Электрон. дан - URL: https://github.com/aprelev/libgost15/ (дата обращения: 09.05.2017)
58. Git Version Control System – [Электронный ресурс] – Электрон. дан. – URL: https://git-scm.com/ (дата обращения: 09.05.2017)
59. Apache Subversion Version Control System – [Электронный ресурс] – Электрон. дан. – URL: https://subversion.apache.org/ (дата обращения: 09.05.2017)
60. Mercurial Source Control Management Tool – [] – Электрон. дан. – URL: https://www.mercurial-scm.org/ (дата обращения: 09.05.2017)
61. GitHub - taka-wang/c-ini-parser – [Электронный ресурс] – Электрон. дан – URL: https://github.com/taka-wang/c-ini-parser/ (дата обращения: 09.05.2017)
62. CWE – Top 25 Most Dangerous Software Errors – [Электронный ресурс] – Электрон. дан. - URL: http://cwe.mitre.org/top25/ (дата обращения: 09.05.2017)
63. CppStudio – Типичные ошибки программирования – [Электронный ресурс] – Электрон. дан. – URL: http://cppstudio.com/post/5142/ (дата обращения: 09.05.2017)
64. Common C Programming Errors – [Электронный ресурс] – Электрон. дан. – URL: http://www.drpaulcarter.com/cs/common-c-errors.php#2.7/ (дата обращения: 09.05.2017)
65. 10 Common Programming Mistakes – [Электронный ресурс] – Электрон. дан. – URL: http://alumni.cs.ucr.edu/~nxiao/cs10/errors.htm/ (дата обращения: 09.05.2017)
66. Common C/C++ Errors – [Электронный ресурс] – Электрон. дан. - URL: http://ace.cs.ohiou.edu/new_users/error.html/ (дата обращения: 09.05.2017)
67. Comparing Python to Other Languages – [Электронный ресурс] – Электрон. дан. - URL: https://www.python.org/doc/essays/comparisons/ (дата обращения: 09.05.2017)

Приложение А. Работа модулей реализации предложенного алгоритма.

В данном приложении при помощи псевдокода и комментариев описана работа модулей программы из раздела 6.2, реализующей предложенный в разделе 5 алгоритм поиска клонов кода изначально помеченных фрагментов исходного кода.

1) <u>Versions Lists Generator:</u>

```
#Данный модуль генерирует списки версий, которые необходимо
#использовать при инициализации и при поиске клонов.
def generate_to_check_List  ( curr_version, List, init_list ):
        children = children( curr_version )
        s = children.size()
        i = 0
        while ( i < s ):
                if not children[i] in List and not children[i] in init_List:
                        List.push_back( children[i] )
                        generate_to_check_List ( children[i], List )
                i += 1
        return

def generate_lists ( sha1_file, init_List, to_check_List ):
        for line in sha1_file:
        #строка содержит SHA-1 идентификатор, используемый при
        #инициализации
                init_List.push_back(line)
        if init_List.size() == 0:
                return
        current_version = init_List[0]
        generate_to_check_List( current_version, to_check_List, init_list ):
                return
```

2) <u>Fragment Normalization:</u>

```
#Данный модуль производит нормализацию фрагмента исходного кода –
#то есть при необходимости восстанавливает баланс фигурных скобок и
#определяет пользовательские типы данных и добавляет их
#псевдоопределения в фрагмент, чего достаточно для генерации дерева.
def normalize_fragment ( fragment ):
    l = count_left_braces( fragment )
    r = count_right_braces( fragment )
    #предполагается, что код написан в определённом стиле –
    #фигурные скобки, окружающие составные операторы и
    #структуры или объединения, расположены только в конце строк
    add_braces( fragment, l – r )
    #добавляет при необходимости фигурные скобки в зависимости
    #от значения ( l – r ) в начало или конец фрагмента
    typedefs = Ø
    for line in fragment:
        t = typedef( line )
        #t – пустая строка “” или имя пользовательского типа в
        #строке line
        if t != “”:
            typedefs.add( t )
    add_typedefs( fragment, typedefs )
    #к фрагменту добавляются необходимые пользовательские типы
    return
```

3) <u>Syntax Tree Comparison:</u>

```
#Данный модуль написан на Python3 и производит сравнение двух уже
#нормализованных и препроцессированных фрагментов кода согласно
#определению 4.
def compare_fragments_mscp ( fragment1, fragment2 ):
    ast1 = ast( fragment1 )
    ast2 = ast( fragment2 )
    mscp = common_pattern( ast1, ast2, substitution1, substitution2 )
    #создаётся наиболее специальный общий шаблон, который
    #используется для определения расстояния между фрагментами
```

```
    #по определению 4 (см. раздел 5)
    distance = 0
    #размер дерева соответствует используемому в определении 4
    #(см. раздел 5)
    for subtree in substitution1:
        distance += size( subtree )
    for subtree in substitution2:
        distance += size( subtree )
    distance -= substitution.size()
    return if_distance_is_good( distance )
    #возвращает 1, если фрагменты являются клонами,
    #0 – иначе
```

4) Comparison Module:

```
#Данный модуль производит подготовку к сравнению двух фрагментов –
#нормализацию, препроцессирование, передачу модулю сравнения деревьев.
def compare( fragment1, fragment2 ):
    match_centers_and_sizes( fragment1, fragment2 )
    #в случае необходимости фрагменты приводятся к одному
    #размеру, изменение размера будет учтено при добавлении в кластер
    normalize_fragment( fragment1 )
    normalize_fragment( fragment2 )
    write_to_file( fragment1, file1 )
    write_to_file( fragment2, file2 )
    execute("gcc –E –Ifake_libc_headers" + file1 + " –o " + file1_res )
    execute("gcc –E –Ifake_libc_headers" + file2 + " –o " + file2_res )
    return compare_fragments_mscp( file1_res, file2_res )
```

5) Initialization:

```
#Данный модуль производит инициализацию кластеров клонов кода,
#используя CppCheck.
def initialize ( Clusters, init_List ):
    for version in init_List:
        execute("git checkout " + version )
        file_List = files_in_current_dir()
```

```
        for file in file_List:
            defects = get_CppCheck_results(file)
            #возвращает список номеров строк, в которых находятся центры
            #отслеживаемых фрагментов в файле file версии version
            line = 0
            while not end_of_file(file):
                get_new_fragment( fragment, fragment_size )
                #line увеличивается на 1
                if line in defects:
                    found = 0
                    i = 0
                    while i < Clusters.size() and found == 0:
                        if compare( fragment, Clusters[i].original ) == 1:
                        #то есть фрагменты являются клонами
                            Clusters[i].add( fragment, line, file, version )
                            defects.erase( line )
                            found = 1
                        else:
                            i += 1
                    if found == 0:
                        new_cluster = Cluster( fragment, line, file, version )
                        Clusters.push_back( new_cluster )
    return
```

6) <u>Clone Search:</u>

```
#Данный модуль производит по исходному коду оставшихся версий
#фрагментов, являющихся клонами кода оригинальных фрагментов
#кластеров – Clusters[i].original.
def search ( Clusters, to_check_List, fragment_size ):
    for version in to_check_List:
        execute("git checkout " + version )
        file_List = files_in_current_dir()
        for file in file_List:
            line = 0
            while not end_of_file(file):
```

```
                    get_new_fragment( fragment, fragment_size )
                    #line увеличивается на 1
                     i = 0
                    found = 0
                    while i < Clusters.size() and found == 0:
                            if compare( fragment, Clusters[i].original ) == 1:
                            #то есть фрагменты являются клонами
                                    Clusters[i].add( fragment, line, file, version )
                                    found = 1
                            else:
                                    i += 1
        return
```

7) <u>Cluster Representation:</u>

В этом модуле функция output_of_clusters(Clusters, files, out_name) записывает в текстовые файлы описания кластеров для последующей обработки утилитой GraphViz. После работы этой функции и последующей работы GraphViz каждый кластер будет выглядеть следующим образом:

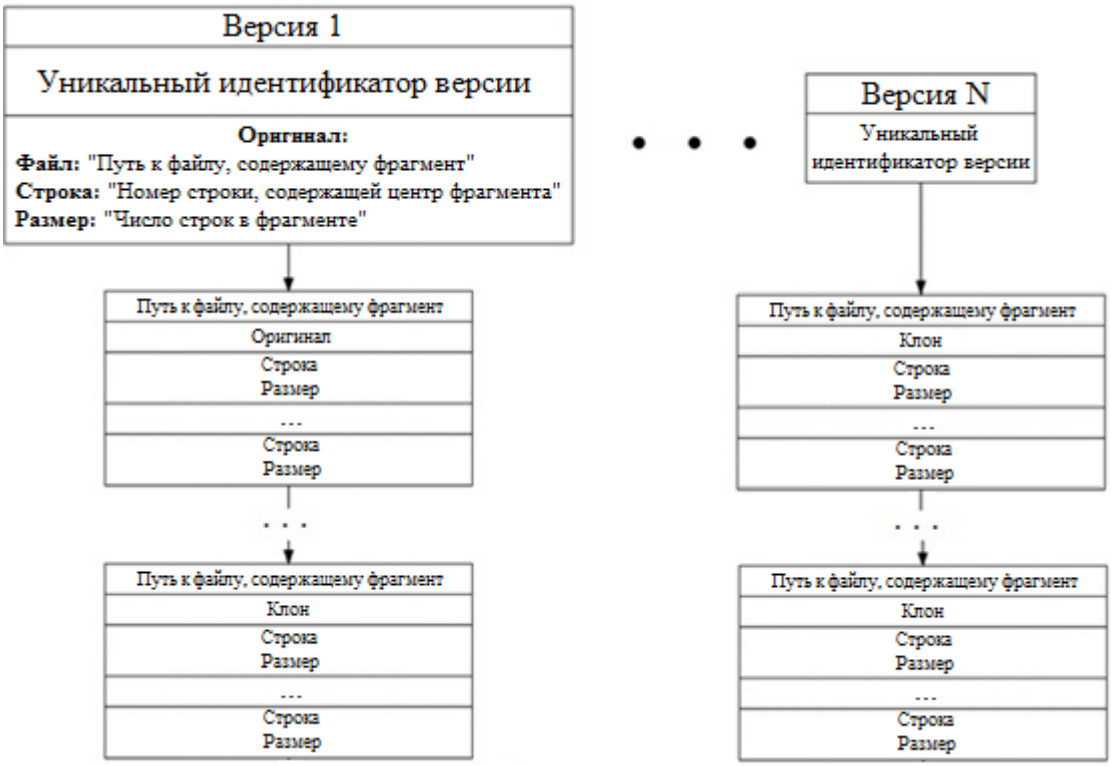

Рисунок 10. Пример представления кластера.

8) <u>Main Module:</u>

```
#Это основной модуль, который считывает аргументы командной
#строки, проверяет их корректность, запускает процесс генерации
#списков версий, инициализации, поиска клонов и вывода результата в
#виде изображения.
#если результат 1 – то были заданы некорректные параметры командной строки
if read_command_line_arguments( sha1_file, fragment_size, out_name ) == 1
        return
#файл с идентификаторами версий-инициализаторов
#размер контекста, окружающего помеченную строку
#базовое имя файлов вывода результата (отличаться будут номером кластера)
init_List = Ø
to_check_List = Ø
generate_lists( sha1_file, init_List, to_check_List )
Clusters = Ø
initialize( Clusters, init_List. fragment_size )
search( Clusters, to_check_List. fragment_size )
output_of_clusters( Clusters, files, out_name )
for i in range( 0, files.size() ):
        execute( “dot –Tpng “ + out_name + “.gv –o “ + out_name + “.png”)
        #генерирует изображение кластера в формате PNG
return
```