На правах рукописи

Айк Асланян Каренович

Методы статического анализа для поиска дефектов в исполняемом коде программ

Специальность 05.13.11 —
«Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей»

Автореферат
диссертации на соискание ученой степени
кандидата физико-математических наук

Москва 2019

Работа выполнена в Федеральном государственном бюджетном учреждении науки Институт системного программирования им. В.П. Иванникова Российской академии наук.

Научный руководитель: **Курмангалеев Шамиль Фаимович**, кандидат физико-математических наук

Официальные оппоненты:

Ильин Вячеслав Анатольевич, доктор физико-математических наук, старший научный сотрудник, начальник отдела Курчатовского комплекса НБИКС-природоподобных технологий Национального исследовательского центра «Курчатовский институт»

Волконский Владимир Юрьевич, кандидат технических наук, начальник отделения «Системы программирования» Публичного акционерного общества «Институт электронных управляющих машин им. И. С. Брука»

Ведущая организация:

Федеральное государственное учреждение «Федеральный научный центр Научно-исследовательский институт системных исследований Российской академии наук»

Защита состоится 14 марта 2019 г. в 15 часов на заседании диссертационного совета Д 002.087.01 при Федеральном государственном бюджетном учреждении науки Институт системного программирования им. В.П. Иванникова Российской академии наук по адресу: 109004, Москва, ул. Александра Солженицына, д. 25.

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

Автореферат разослан «___»_____ 2019 г.

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

Общая характеристика работы

Актуальность.

Разработчики программного обеспечения часто совершают ошибки, которые могут привести к сбоям в работе програмного продукта. Исправить их можно на любом этапе жизненного цикла ПО. Вместе с тем, стоит учитывать, что обнаружение и исправление ошибок на поздних этапах разработки могут существенно увеличить затраты и издержки, а ошибки, проявляющиеся на этапе эксплуатации могут представлять опасность жизни и здоровью людей. Поэтому в жизненном цикле разработки ПО широко используются различные инструменты анализа программного кода для обнаружения дефектов.

Одним из подходов к проблеме своевременного обнаружения дефектов является статический анализ, заключающийся в исследовании кода без выполнения программы. С помощью анализа синтаксиса, семантики, а также потоков управления и данных статический анализ позволяет находить ошибки, которые трудно или невозможно обнаружить экспертным методом в силу размера современных программных комплексов.

Большинство существующих инструментов статического анализа работают с исходным кодом программы – Svace, Coverity, Klocwork, HP Fortify, IBM AppScan. Между тем, анализа исходного кода часто недостаточно. Причиной этого может служить использование сторонних бинарных библиотек, недоказуемость правильности всех оптимизаций компиляторов. Ошибки могут возникать из-за неточностей в описании интерфейсов библиотек или при их неправильном использовании. Агрессивные компиляторные оптимизации могут привести к дефектам, которые отсутствуют в исходном коде, а также в силу учета компилятором особенностей целевой платформы дефект может проявляться только в бинарных сборках для некоторых архитектур.

Таким образом, требуется инструмент статического анализа бинарного кода, позволяющий искать дефекты в уже готовой программе и в библиотеках. Такой инструмент должен обладать следующими возможностями: межпроцедурный анализ, чувствительность к потоку управления, чувствительность к потоку данных и чувствительность к путям выполнения. Кроме того, качественный анализатор должен масштабироваться для анализа больших файлов размером в десятки мегабайт за несколько часов, а также обладать высокой точностью (количество правильных срабатываний − больше 50%) и расширяемостью для поиска новых типов дефектов.

Цель работы − исследование и разработка методов статического анализа исполняемого кода для поиска дефектов. Методы должны быть архитектурно независимыми, обладать высокой точностью и масштабируемостью для анализа исполняемых файлов размером в десятки мегабайт (несколько миллионов строк кода).

Основные задачи:

1. Исследовать существующие методы анализа исполняемого кода.
2. Разработать архитектуру статического анализа исполняемого кода для поиска дефектов.
3. Разработать методы анализа значений, потоков данных и помеченных данных в исполняемом коде.
4. Разработать методы поиска клонов (синтаксически похожих фрагментов) исполняемого кода и методы сравнения двух версий исполняемых файлов для автоматического поиска и определения характера изменений в новых версиях программ.
5. Разработать методы поиска дефектов в исполняемом коде.
6. Реализовать разработанные методы для анализа исполняемых файлов архитектур x86, x86-64, ARM.

Научная новизна:

1. Предложены и разработаны методы анализа значений и анализа помеченных данных, которые позволяют проводить межпроцедурный, чувствительный к контексту и чувствительный к потоку данных анализ исполняемых файлов.
2. Предложен и разработан метод поиска клонов исполняемого кода на основе семантического подхода, который позволяет находить измененные фрагменты кода с заданной точностью.
3. Предложен и разработан метод сравнения двух версий исполняемых файлов для автоматического поиска и определения характера изменений в новых версиях программ.
4. Предложены и разработаны методы поиска дефектов использования памяти после освобождения, двойного освобождения памяти, переполнения буфера, форматных строк, внедрения команд и поиска неисправленных фрагментов в новой версии исполняемого файла.

Теоретическая и практическая значимость работы.

Теоретическая значимость заключается в разработке архитектуры системы анализа, методов и алгоритмов поиска клонов исполняемого кода, а также поиска дефектов. Разработанные методы пригодны для анализа исполняемых файлов размером в несколько десятков мегабайт, обеспечивают

высокую точность и могут использоваться в жизненном цикле разработки безопасного ПО, что позволит повысить его надежность и безопасность. Эффективность методов подтверждена результатами анализа на тестовых наборах и на реальных программах. Реализованные инструменты используются в ИСП РАН.

Методология и методы исследования.

Результаты диссертационной работы получены на базе использования методов абстрактной интерпретации и теории решеток. Математическую основу исследования составляют теория графов и алгебра логики.

Положения, выносимые на защиту:

1. Методы анализа значений и помеченных данных, позволяющие проводить межпроцедурный, чувствительный к контексту и чувствительный к потоку данных анализ исполняемых файлов.
2. Методы поиска клонов исполняемого кода и сравнения двух версий исполняемых файлов для автоматического поиска и определения характера изменений в новых версиях программ.
3. Методы поиска дефектов использования памяти после освобождения, двойного освобождения памяти, переполнения буфера, форматных строк, внедрения команд и поиска неисправленных фрагментов в новой версии исполняемого файла.

Апробация работы.

Основные результаты работы обсуждались на конференциях:

1. Открытая конференция ИСП РАН, Москва, Россия, 1-2 декабря 2016 года.
2. Международная Ершовская конференция по информатике PSI-2017, Москва, Россия, 27-29 июня 2017 года.
3. 11-я Международная конференция CSIT 2017, Ереван, Армения, 25-29 сентября 2017 года.
4. 60-я Научная конференция МФТИ, Москва, Россия, 20-25 ноября 2017 года.
5. Открытая конференция ИСП РАН им. В.П. Иванникова, Москва, Россия, 30 ноября-1 декабря 2017 года.
6. 12-я Годичная научная конференция Российско-Армянского университета, Ереван, Армения, 4-8 декабря 2017 года.
7. Открытая конференция ИСП РАН им. В.П. Иванникова, Москва, Россия, 22-23 ноября 2018 года.

Публикации

По теме диссертации опубликовано 6 научных работ, в том числе, 3 научные статьи [1] [2] [3] в рецензируемых журналах, входящих в перечень рекомендованных ВАК РФ. Работа [2] индексирована в Web of Science.

В работах [1] и [4] представлен метод поиска клонов исполняемого файла на основе графов зависимостей программы. В работе [1] личный вклад автора заключается в разработке алгоритмов разделения графов зависимостей программы на подграфы. В статье [4] автором представлен алгоритм поиска наибольших общих подграфов двух графов зависимостей программы. В работе [2] представлен метод сравнения двух исполняемых файлов, личный вклад автора заключается в разработке алгоритма сопоставления функций на основе анализа графа зависимостей программы и графа вызовов функций. В статье [3] автором описывается платформа межпроцедурного анализа исполняемых файлов. В коллективных работах [5] и [6] описаны методы поиска дефектов использования памяти после освобождения, двойного освобождения памяти и форматных строк. В работе [5] личный вклад автора заключается в разработке алгоритма поиска дефектов форматных строк. Личный вклад автора в статье [6] заключается в разработке алгоритмов поиска дефектов использования памяти после освобождения и двойного освобождения памяти на основе графов зависимостей системы.

Личный вклад.

Все представленные в диссертации результаты получены лично автором.

Объем и структура диссертации.

Диссертация состоит из введения, четырех глав, заключения и одного приложения. Полный объем диссертации составляет 118 страниц, включая 11 рисунков и 22 таблицы. Список литературы содержит 89 наименований.

Краткое содержание работы

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

В первой главе приводится обзор методов анализа кода и обзор работ, которые имеют отношение к теме диссертации.

В разделе 1.1 описаны существующие подходы к анализу кода − статический и динамический, а также преимущества и недостатки каждого из них.

В разделе 1.2 приводится обзор современных методов статического анализа. Существующие методы восстанавливают ассемблер, графы потока управления, графы вызовов функций, косвенные переходы и косвенные вызовы. На основе восстановленных данных проводится анализ интервалов значений, а также анализ потока данных.

В разделе 1.3 описаны подходы к поиску клонов исполняемого кода: текстовый метод, а также методы, основанные на токенах, метриках и поведении программы.

В разделе 1.4 описаны подходы к сравнению исполняемых файлов. Существует несколько таких подходов: основанные на метриках, на хешировании, на сравнении графов потока управления и на символьном выполнении.

В разделе 1.5 описаны методы анализа исходного кода, которые используются для изучения характера изменений программ между версиями.

В разделе 1.6 приводятся следующие выводы:

1. Все существующие разработки предназначены для статического анализа исполняемых файлов архитектуры x86 и ARM и имеют ограничения по масштабируемости, так как не поддерживают использование в анализе аннотации для функций. Методы, разработанные в рамках данной диссертационной работы, не зависят от архитектуры исполняемого файла. Более того, предложенный метод включает в себя межпроцедурный, контекстно-чувствительный и потоко-чувствительный анализ исполняемых файлов.

2. Все существующие методы поиска клонов и сравнения исполняемых файлов не учитывают поток данных, что приводит к низкому проценту истинных срабатываний. Более того, эти методы не справляются с переупорядочением инструкций. Предложенный в диссертации метод поиска клонов исполняемого кода находит клоны с большой точностью (более 90%), может анализировать исполняемые файлы размером в десятки мегабайт за несколько часов и является устойчивым к переупорядочению инструкций. Для сравнения исполняемых файлов предложены методы, основанные на графах зависимостей программы и графах вызовов функций. На основе предложенных алгоритмов сравнения файлов разработан новый метод автоматического поиска и определения характера изменений в новых версиях программ.

3. Что касается сторонних инструментов, выполняющих поиск дефектов использования памяти после освобождения, двойного освобождения памяти, форматных строк и переполнения буфера в исполняемом коде, то тут

доступны результаты только двух из них: GUEB и LoongChecker. При этом у обоих наблюдается большое количество неправильных срабатываний (больше 90%). Программная реализация и результаты более эффективных методов являются закрытыми, в связи с чем сравнение с ними невозможно. Разработанные в диссертационной работе методы позволяют достигать числа истинных срабатываний на уровне 60% (в среднем).

Разработан также метод поиска потенциально неисправленных фрагментов в новых версиях программ, созданный с использованием предложенных в работе методов поиска клонов исполняемого кода и автоматического поиска и определения характера изменений в новых версиях.

Во второй главе описывается общая архитектура предлагаемого инструмента. Приводится описание используемой модели памяти, описываются применения абстрактной интерпретации и анализа потока данных для анализа исполняемых файлов. Исполняемый код транслируется в промежуточное представление, на котором проводятся: анализ значений и достигающих определений, построение DEF-USE и USE-DEF цепочек, трансформация для удаления мертвого кода, а также анализ помеченных данных и динамической памяти.

Предлагаемая архитектура инструмента разработана с учетом следующих требований:

- независимость от целевой архитектуры;
- межпроцедурный, контекстно-чувствительный, чувствительный к потоку данных и потоку управления анализ;
- масштабируемость (анализ десятков мегабайт исполняемого кода за несколько часов);
- возможность расширения функционала платформы.

На рисунке 1 представлена общая архитектура анализа. Первым шагом является получение ассемблерного кода из исполняемого файла. Дизассемблер создает инструкции языка ассемблера, используя исполняемый код в качестве входных данных. Используется дизассемблер IDA Pro, поскольку он поддерживает множество форматов исполняемых файлов, автоматически восстанавливает графы потока управления и граф вызовов функций. Дизассемблер также восстанавливает соглашения о вызовах. Полученная информация передается инструменту Binnavi, который транслирует ассемблер в представление REIL (Reverse Engineering Intermediate Language). REIL представление является промежуточным языком низкого уровня, который можно использовать для написания независимых от платформы алгоритмов анализа. Он имеет только 17 инструкций, каждая из которых вычисляет не

более одного результата и не имеет побочных эффектов (установки флагов и т.д.). Все алгоритмы реализованы с использованием REIL представления.

В разделе 2.1 представлена архитектура межпроцедурного анализа, который основан на аннотациях функций и обеспечивает контекстно-чувствительность.

После генерации REIL представления граф вызовов функций приводится в ациклическую форму с помощью алгоритма 1.

Алгоритм 1. В качестве входа алгоритм принимает граф вызовов функций и возвращает ациклический граф вызовов. Алгоритм выполняет следующие шаги:

1. Определение сильно связанных компонентов на основе алгоритма Тарьяна.

2. Для каждого сильно связанного компонента проводится обход в глубину для удаления ребер циклов. Сначала все вершины помечены как «не рассмотренные».

2.1. Выбирается произвольная вершина из сильно связанного компонента и вставляется в стек;

2.2. Если стек не пуст, то извлекается узел из вершины стека и отмечается как «рассмотренный». Если стек пуст, то алгоритм переходит к шагу 2.

2.3. Для всех «рассмотренных» потомков узла удаляются их соединяющие ребра.

2.4. Остальные потомки добавляются в стек и алгоритм переходит к шагу 2.2.

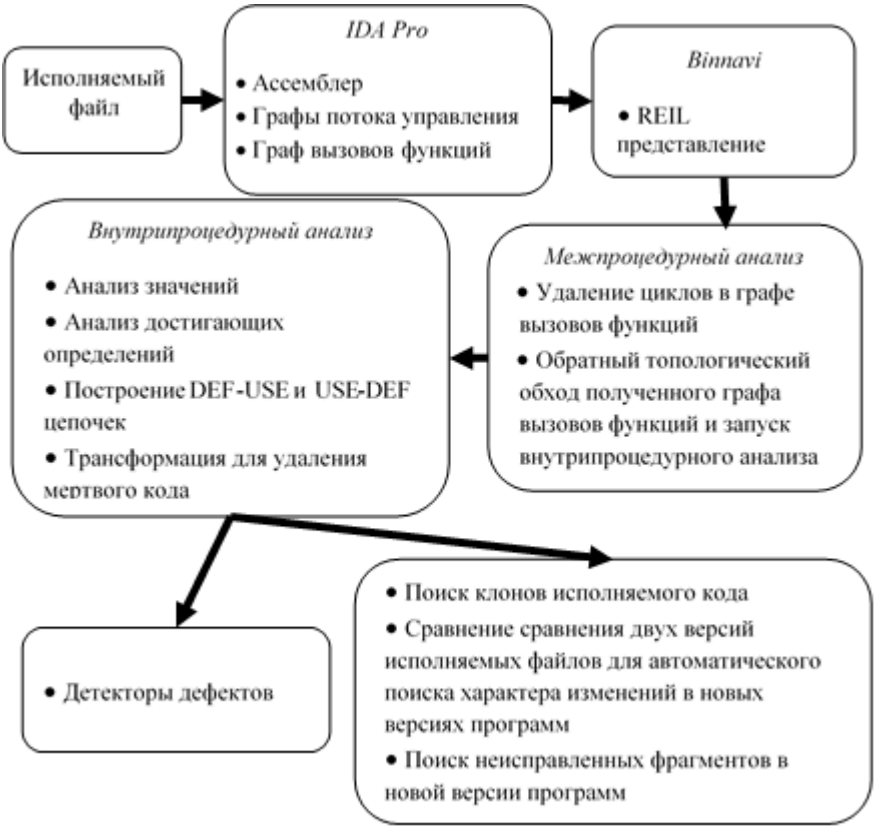

Рисунок 1. Архитектура статического анализа исполняемых файлов

Теорема 1. Граф вызовов функций, который возвращает алгоритм 1, является ациклическим.

Теорема 2. Временная сложность алгоритма 1 составляет $O(V+E)$, где $V-$ количество вершин графа вызовов функций, а $E-$ количество ребер.

Алгоритм 2. В качестве входа алгоритм принимает ациклический граф вызовов и возвращает множество групп вызовов. Алгоритм выполняет следующие шаги:

1. В первой группе находятся те вершины, у которых нет выходящих ребер. Они помечаются как «рассмотренные». Номер группы обозначен через *groupNumber*, который изначально равен 2 (*groupNumber=2)*.

2. В группу *groupNumber* добавляются те вершины, которые являются потомками вершин из группы *(groupNumber-1),* и все предшественники

помечены как «рассмотренные». Если нет потомков для вершин из группы *(groupNumber-1)*, то алгоритм заканчивает работу. В противном случае к *groupNumber* добавляется 1 и повторяется шаг 2.

Теорема 3. Временная сложность алгоритма 2 составляет O(V+E), где V − количество вершин графа вызовов функций, а E − количество ребер.

После разбиения вершин графа вызовов функций на группы проводится их обход. Запуск внутрипроцедурного анализа начинается с вершин первой группы. Каждая последующая группа рассматривается в том случае, если проанализированы функции, соответствующие всем вершинам предыдущей. Надо отметить, что анализируются только те функции, для которых доступна имплементация. Это значит, что функции библиотек, которые прикреплены динамически, не анализируются (для них доступны только аннотации).

После завершения анализа для каждой функции сохраняются только так называемые аннотации – информация, которая содержит специфические особенности функции. Например, функция возвращает значение, которое контролируется пользователем (например, функция gets в C/C++), или функция освобождает динамически выделенную память, на которую показывает первый аргумент. При анализе конкретной функции используются аннотации всех вызываемых ею функций. Очевидно, что если нет рекурсивных вызовов, то аннотации всех вызываемых функций доступны. При рекурсивных вызовах некоторые ребра удаляются из графа вызовов и некоторые аннотации вызываемых функций могут быть недоступны при анализе функции. В таких случаях условно считается, что вызывающие функции не имеют аннотаций. Платформа предоставляет возможность добавления новых аннотаций.

Для проведения анализа используются несколько аннотаций:

- argument_free_annotation – функция удаляет память, на которую указывает ее аргумент (например, функция free);
- allocate_memory_annotation – функция выделяет память в куче и возвращает (например, функция malloc);
- argument_dereference_annotation – функция разыменовывает аргумент (например, функция strcpy);
- argument_format_string_annotation – аргумент функции является функцией стандартной библиотеки, чей аргумент – форматная строка (например, функции printf, fprintf, sprintf, vsprintf). Или же аргумент передается функции, которая имеет аннотацию (argument_format_string_annotation);

- argument_buffer_overflow_annotation – аргумент функции является функцией стандартной библиотеки, чей аргумент – буфер, в котором записывается информация (например, функции strcpy, memcpy, sprintf). Или же аргумент передается функции, которая имеет аннотацию argument_buffer_overflow_annotation;
- argument_command_injection_annotation − аргумент функции является функцией стандартной библиотеки, которая вызывает команду (например, функция system), или аргумент передается функции, которая имеет аннотацию argument_command_injection_annotation;
- argument_untrusted_annotation – функция записывает в аргумент информацию, которая контролируется пользователем (например, функции getline, gets);
- return_untrusted_annotation − функция возвращает информацию, которая контролируется пользователем (например, функция gets, fgets)
- copy_arguments_annotation – функция копирует информацию из одного аргумента в другой.

Внутрипроцедурный анализ для каждой функции запускается только один раз, что позволяет обеспечить масштабируемость с учетом количества всех функций. Разбиение вершин графа вызовов на группы дает возможность проводить параллельный анализ функции в каждой отдельной взятой группе.

В разделе 2.2 описывается внутрипроцедурный анализ для каждой функции. Анализ проводится на основе REIL представления функции. Во время анализа кода и обработки инструкций вызова функций используются аннотации этой функции, и изменение контента происходит с учетом реальных параметров и соглашения о вызовах. Данный процесс обеспечивает контекстно-чувствительность анализа. Разработаны и реализованы: анализ значений, анализ достигающих определений, построение DEF-USE и USE-DEF цепочек, трансформация для удаления мертвого кода, анализ помеченных данных, а также анализ работы с динамической памятью. Архитектура внутрипроцедурного анализа позволяет легко расширять количество видов анализа и добавлять плагины.

В разделе 2.2.1 описывается метод для анализа значений, который используется для отслеживания значений в регистрах и в ячейках памяти. Все регистры целевой архитектуры и временные регистры, а также все ячейки памяти, которые используются в программе, условно называются переменными. Во время анализа все они получают значения, которые сохраняются во всех точках программы. Для значений точек памяти была

разработана и реализована модель, которая моделирует память в стеке, в куче и в статической области.

Анализ значений проводится на основе итеративного алгоритма анализа потока данных. Все остальные виды анализа базируются на анализе значений.

Простой моделью памяти является рассмотрение памяти как массива байт. Запись (чтение) в этой тривиальной модели обрабатывается как запись (чтение) в соответствующий элемент массива. Однако у такой простой модели есть некоторые недостатки. В частности, невозможно определить конкретные значения адресов для определенных блоков памяти (например, выделенных из кучи через функцию malloc). Более того, последовательность вызовов функций может меняться во время каждого запуска программы, что в целом приводит к неоднозначности значений памяти.

Требуется разделять сегменты памяти для корректного проведения анализа. Для моделирования памяти программы используется следующая конструкция: $*(reg + constants\_array) + constant$, где reg является регистром, $constants\_array$ представляет собой массив из константных значений, а $constant$ является константой. $constants\_array$ и $constant$ играют роль смещения, при этом $constants\_array$ обеспечивает возможность моделировать иерархическую память, а reg имеет базовое символьное значение.

- **Моделирование памяти в стеке.** Поскольку невозможно определить точное значение начала стека функции на основе статического анализа, модель ссылается на локальные переменные по смещению с начала стека текущей функции. В связи с этим используется символ для начального адреса стека анализируемой функции − $stack$, и все локальные переменные моделируются относительно него. Например, в архитектуре x86 после инструкции $mov\ eax, esp + 4$ значение eax будет $stack + 4$, а после инструкции $mov\ ebx, [esp + 8]$ значение ebx будет $*(stack + \{8\})$.
- **Моделирование памяти в куче.** Для моделирования памяти в куче используется символ $heap$ и в $constants\_array$ вставляется значение адреса инструкции, который вызывает функцию для получения динамической памяти. Например, после обработки инструкции $call\ malloc$, чей адрес равен $0xFFFFFFFF$ (архитектура x86 и cdecl соглашение о вызовах), eax будет иметь значение $*(heap + \{0xFFFFFFFF\})$.
- **Моделирование статической памяти.** Обращение к статическим и глобальным переменным происходит по адресу без смещения (значение $constants\_array$) или со смещением ($constant$).

В разделах 2.2.2-2.2.6 описываются виды анализа и трансформации для полученного REIL представления и графов потока управления. На основе анализа значений реализован анализ потока данных: анализ достигающих определений, трансформация для удаления мертвого кода, анализ помеченных данных и динамической памяти. Для проведения всех перечисленных видов анализа применяется итеративный алгоритм потока данных. Определяются полурешетки, множества передаточных функций и присваиваются начальные значения переменным. Построение DEF-USE и USE-DEF цепочек основано на анализе достигающих определений.

В разделе 2.2.7 описывается, как вычисляются аннотации для функций.

Платформа предоставляет программный интерфейс для работы со всеми реализованными видами анализа, что позволяет реализовывать его новые виды.

Все реализованные алгоритмы протестированы на реальных проектах. В таблице 1 приведено время работы всех описанных видов анализа для проектов lepton, php и clam. Тесты проводились на машине с процессором core i5, 4 ядер и 16 ГБ ОЗУ.

Как можно увидеть в таблице, php имеет больший размер, чем clam, однако время работы анализа на этом проекте меньше. Такой результат связан с тем, что в проекте clam размеры функций в среднем намного больше, чем в php. Поэтому параллельная обработка функций в php работает намного эффективней.

Исполняемый файл	**Архитектура**	**Размер**	**Время работы всех видов анализа**
lepton	x86	5 МБ	19 мин. 21 сек.
php	x64	29 МБ	3 ч. 12 мин.
clam	x86	18 МБ	4 ч. 20 мин.

Таблица 1. Время работы анализа

Третья глава посвящена поиску клонов исполняемого кода, сравнению исполняемых файлов, а также анализу характера изменений между двумя версиями программы.

В разделе 3.1 описывается метод поиска клонов исполняемого кода, которые делятся на три типа. Первый тип – фрагменты кода, которые полностью совпадают. Второй тип – фрагменты, которые могут отличаться значениями данных и именами регистров. Третий тип – фрагменты кода, которые могут отличаться значениями данных, именами регистров, а также

некоторыми инструкциями (в конкретном фрагменте могут присутствовать или отсутствовать некоторые инструкции).

Предлагаемая модель инструмента для поиска клонов исполняемого кода была разработана с учетом следующих требований:

- поиск всех типов клонов;
- независимость от целевой архитектуры;
- масштабируемость (размер анализируемых программ может достигать десятков мегабайт);
- большой процент истинных срабатываний (> 90%).

Инструмент получает на вход два аргумента: минимальное количество инструкций для клонов и минимальный процент сходства. Работа инструмента делится на три этапа.

Первый этап – генерация графов зависимостей программы (ГЗП) для функций. Сначала исполняемый файл дизассемблируется с помощью IDA Pro и транслируется в REIL представление с помощью Binnavi. Генерация ГЗП происходит на основе графа потока управления и разработанного анализа USE-DEF цепочек. Узлам ГЗП соответствуют инструкции REIL, а ребрам – зависимости по данным и по управлению. В конце первого этапа все графы сериализуются.

На втором этапе происходит десериализация и анализ графов. После загрузки ГЗП они разделяются на единицы сравнения (ЕС), которые представляют собой подграфы ГЗП (или весь ГЗП) и рассматриваются как потенциальные клоны друг друга. Так как графы изначально генерируются для функций, то для сравнения кода в рамках одной функции необходимо разделение графов. Разработаны два метода для решения этой задачи: разделение по базовым блокам исполняемого кода и разделение по слабо связанным компонентам ГЗП.

Поиск клонов исполняемого кода происходит на основе полученных ЕС. Для пар ЕС эвристическим алгоритмом считается максимальный общий подграф. Если он соответствует критериям минимального количества инструкций для клонов и минимального процента сходства, то фрагменты кода считаются клонами.

На третьем этапе визуализируются клоны исполняемого кода и соответствующие им ГЗП. Преимуществом предлагаемого метода является то, что он основан на семантическом подходе, который позволяет достичь высоких уровней точности и масштабируемости.

Результаты приведены в таблице 2 (исполняемые файлы получены компиляцией исходного кода с флагами -O0 –fno-inline), где минимальное

количество инструкций для клонов равно 30, а минимальный процент сходства равен 90%. Функции для каждого исполняемого файла сравнивались друг с другом.

Проект	**Размер**	**Количество найденных клонов**	**Время работы инструмента**
libxml2.so 2.9.3	1.8 МБ	2123	1 мин. 46 сек.
libfreetype.so 2.6.5	816 КБ	270	57 сек.
openssl 1.1.0	764 КБ	40	35 сек.
d8 5.0	33 МБ	40397	26 мин. 52 сек.

Таблица 2. Результаты поиска клонов исполняемого файла

В разделе 3.2 представлен метод сравнения двух исполняемых файлов. Метод состоит из двух этапов.

На первом этапе генерируются ГЗП (см. раздел 3.1). На втором этапе происходит сопоставление функций с учетом полученных ГЗП и графов вызовов. Для сопоставления функций разработаны 3 алгоритма: на основе эвристик, на основе нахождения наибольшего общего подграфа двух ГЗП и объединенный алгоритм.

Сопоставление функций на основе метрик происходит следующим образом. Для каждой функции считается ряд эвристик. Если конкретная эвристика для некоторой функции из первого исполняемого файла совпадает с эвристикой функции из второго исполняемого файла и не совпадает с эвристиками других функций, то две функции сопоставляются. Если функции идентичны по рассмотренной эвристике, но существуют другие функции, которые имеют равную эвристику, то рассматривается следующая эвристика и процесс может продолжаться до рассмотрения всех эвристик в определенной последовательности.

1. Сопоставление функций на основе хеширования ассемблерного кода.
2. Сопоставление ребер графа вызовов (сопоставляются сразу две функции) на основе хеширования ГЗП по MD-индексу.
3. Сопоставление ребер графа вызовов (сопоставляются сразу две функции) на основе хеширования MD-индекса соседних вершин начальной и конечной вершин ребра.
4. Сопоставление функций на основе хеширования вершин ГЗП, считанных на ГЗП на основе MD-индекса.
5. Сопоставление функций на основе хеширования вершин ГЗП. Эвристика считает хеш на основе зависимостей по данным между инструкциями, группирует их и считает конечный хеш.

6. Сопоставление функций на основе специального хеша ГЗП. Каждому коду операции сопоставляется простое число. После этого считается произведение этих хешей для всех инструкций. Чтобы избежать переполнения, после каждого произведения учитывается только остаток деления на 2^{64}.

Теорема 4. Временная сложность алгоритма на основе эвристик равна $O(k^2(v_1 log_2 v_1 + v_2 log_2 v_2 + \ldots + v_k log_2 v_k))$, где k − количество функций, а $v_1, v_2, \ldots, v_k$ − количество вершин графов ГПУ.

Основанный на графах алгоритм сопоставляет функции по следующим шагам:

1. Сопоставляются все пары вершин графов вызовов функций из первого и второго исполняемого файла. Если их ГЗП изоморфны, то вершины графов вызовов функций сопоставляются. Если ни одна пара не сопоставилась, то сопоставляется пара вершин, функции которых похожи больше, чем функции других пар.
2. Рассматриваются предшественники (преемники) для каждой сопоставленной пары вершин: P1 и P2 (S1 и S2). Для всех пар вершин из P1 и P2 (S1 и S2) вычисляется наибольший общий подграф для их ГЗП и строится матрица из сопоставленных частей. Процесс обнаружения общего подграфа распараллеливается для каждой пары ГЗП, чтобы обеспечить масштабируемость.
3. Применяется венгерский алгоритм для нахождения лучшего соответствия ГЗП функций из P1(S1) и ГЗП функций из P2(S2).
4. Шаги 2-4 повторяются до рассмотрения всех сопоставленных функций.

Теорема 5. Временная сложность алгоритма, основанного на графах − $O\left(l * \sum_{i=0}^{k} {v_1}^4 \log_2 {v_1}^3\right) + O((k+l)^3)$, где k − количество функций в первом исполняемом файле, а $v_1, v_2, \ldots, v_k$ − количество инструкций функций первого исполняемого файла и l − количество функций во втором исполняемом файле.

Эвристический алгоритм сопоставляет функции быстрее, однако значительное количество функций не сопоставляются. Алгоритм, использующий поиск наибольшего общего подграфа двух ГЗП, работает медленно, но при этом происходит сопоставление всех возможных функций. С учетом достоинств и недостатков данных двух методов был предложен объединенный алгоритм: сначала сопоставление происходит на основе эвристик, далее используется алгоритм, основанный на графах.

Результаты работы инструмента показаны в таблице 3 (использовался объединенный алгоритм). Все тесты выполнены на процессоре i5 3,3 ГГц с 4 физическими ядрами. Ручной анализ результатов показал, что точность инструмента превышает 95%.

Проект	Версии		Размер (МБ)		Время работы (секунды)	Количество функций		Количество сопоставленных функций
	старая	*новая*	*старая*	*новая*		*старая*	*новая*	
python	3.5.1	3.5.2	12	12	55	3944	3951	3944
python	3.5.2	3.6.3	12	12	243	3944	4107	3943
php	7.0.5	7.0.6	29	29	99	8287	8292	8287
php	7.0.6	7.0.24	29	29	355	8292	8342	8292
libxml2	2.9.2	2.9.3	5.4	5.4	20	2584	2603	2584
openssl	1.0.1r	1.0.1s	2.8	2.9	47	5395	5430	5395
openssl	1.0.1f	1.0.1s	2.2	2.9	48	5414	5430	5414
rsync	3.0.9	3.1.1	1.6	1.8	8	599	636	599
gcc	4.9.0	5.4.0	3.2	3.5	12	1094	1145	1094
git	2.6.0	2.9.5	9.4	9.8	32	3335	3471	3334

Таблица 3. Результаты работы инструмента сопоставления функций двух исполняемых файлов

В разделе 3.3 описывается метод, который определяет характер изменений между двумя версиями программ. Сначала исполняемые файлы сравниваются методом, который описан в разделе 3.2. Результатом сравнения является сопоставление функций и инструкций, которое позволяет определить измененные фрагменты кода. Цель алгоритма – обнаружить те изменения кода, которые могут повлиять на корректность работы программ (например, добавление проверки). Алгоритм определяет ряд изменений, которые не являются результатом рефакторинга. Отслеживаются следующие изменения:

1. Добавился новый базовый блок в исполняемом коде.
2. Добавилась новая инструкция возврата из функции.
3. Изменились аргументы функции.
4. Изменилась вызывающая функция.
5. Добавилась новая инструкция выхода из цикла.
6. Добавилась новая инструкция для перехода в начало цикла.

В таблице 4 приведены результаты работы инструмента:

Проект	Время работы	Количество найденных точек	Количество правильных срабатываний
Carbonate	2 сек.	3	2
CGC_Board	3 сек.	3	3
Diary_Parser	1 сек.	6	5
CGC_Planet_Markup_ Language_Parser	3 сек.	14	13

Таблица 4. Результаты работы инструмента анализа характера изменений между версиями программ

Четвертая глава посвящена описанию программного инструмента для нахождения дефектов использования памяти после освобождения (ИПО), двойного освобождения памяти (ДО), форматных строк (ФС), переполнений буфера (ПБ) и внедрения команд (ВК).

В разделе 4.1 описываются два метода нахождения дефектов ИПО и ДО. Первый метод основан на построении графов зависимостей системы (ГЗС), а второй – на аннотациях функций.

При использовании первого метода строятся части целого ГЗС и анализ проводится на полученных частях. Использование кусков ГЗС обусловлено тем, что даже для программ размером несколько килобайт ГЗС может быть слишком большим (более 20 гигабайт) для оперативной памяти. Для получения кусков ГЗС на графе вызовов строятся все пути, концом которых является функция, освобождающая память. Длина путей в графе вызовов ограничивается аналитиком. Узлам ГЗС соответствуют инструкции REIL представления, а ребрам – межпроцедурные и внутрипроцедурные зависимости по данным и зависимости по управлению.

Анализ для обнаружения дефектов ИПО и ДО начинается с листьев ГЗС, чему соответствует функция, которая освобождает динамическую память. Алгоритм находит указатель, который передается такой функции. В алгоритме проводится также анализ алиасов, цель которого − найти все алиасы для данного указателя внутри функции с учетом зависимости по данным. Существует несколько возможных случаев обработки найденного освобожденного указателя:

1. Освобожденный указатель определяется в функции, которая содержит вызов освобождения памяти.
2. Освобожденный указатель является глобальной переменной.

3. Освобожденный указатель является аргументом анализируемой функции.

Первый и второй случай алгоритм обрабатывает путем обхода ГЗС по зависимостям по управлению для нахождения всех путей (если они существуют): от инструкции, в которой освобождается переменная (или ее алиас), в инструкцию, в которой она используется (или повторно освобождается). Затем, используя зависимости по данным, алгоритм проверяет переопределение указателя на каждом найденном пути. Если хотя бы на одном пути нет переопределения, алгоритм сообщает о дефектах ИПО или ДО.

В третьем случае, когда освобожденный указатель передается функции в качестве аргумента, функция получает аннотацию argument_free_annotation. Далее рассматривается функция, которая ее вызывает. В таблице 5 представлены результаты анализа.

Проект	**Архитектура**	**Размер**	**Время анализа**	**Количество ИПО и ДО**	**Процент правильных срабатываний**
jasper 1.900.1	x32	1 MB	166 сек.	3	100%
gifcolor 5.1.2	x32	50 KB	10 сек.	1	100%
libtiff 4.0.3	x32	1 MB	100 сек.	12	67%
gnome-nettool 3.8.1	x32	336 KB	48 сек.	1	100%
openslp 1.2.1	x32	700 KB	40 сек.	4	50%
libssh 0.5.2	x32	700 KB	99 сек.	19	79%
accel-ppd 1.10.0	x32	232 КБ	44 сек.	8	50%

Таблица 5. Результаты поиска ИПО и ДО на основе ГЗС

В таблице 6 представлены результаты сравнения с инструментом GUEB.

Проект	**Время анализа GUEB**	**Найденные ИПО и ДО GUEB**	**Процент правильных срабатываний GUEB**	**Найденные ИПО и ДО**	**Процент правильных срабатываний**
gnome-nettool	16 сек.	4	25%	1	100%
gifcolor	21 сек.	15	6%	1	100%
jasper	4 мин. 23 сек.	255	1.2%	3	100%
accel-ppd	5 мин. 5 сек.	35	11.4%	8	50%

Таблица 6. Сравнение с GUEB

Второй метод нахождения ИПО и ДО дефектов основан на аннотациях функций и анализе динамической памяти. Алгоритм совершает следующие шаги для всех точек программы:

1. Получает все переменные, указывающие на освобожденную память (они помечены как dangling_pointer, а соответствующие точки памяти − freed_memory).
2. Рассматривает все вызовы функций, которые имеют аннотацию argument_dereference_annotation, и получает ее фактические аргументы.
3. Для всех фактических аргументов, помеченных как dangling_pointer и указывающих на память, которая помечена как freed_memory, выдается дефект ИПО.

Алгоритм поиска дефектов ДО отличается от алгоритма поиска ИПО только вторым пунктом: рассматриваются все вызовы функций, которые имеют аннотацию argument_free_annotation.

Результаты представлены в таблице 7.

Проект	**Архитектура**	**Размер**	**Время анализа**	**Количество найденных дефектов**	**Процент правильных срабатываний**
accel_pppd 1.10.0	x86	232 КБ	3 мин.	4	100%
gnome-nettool 3.8.1	x86	336 КБ	1 мин. 40 сек.	1	100%
slpd 1.2.1	x86	128 КБ	50 сек.	1	100%
jasper 1.900.1	x86	980 КБ	11 мин. 41 сек.	1	100%
libtiff 4.0.3	x86	1 МБ	2 мин. 58 сек.	3	67%
accel_pppd 1.10.0	x64	244 КБ	4 мин. 1 сек.	1	100%
gnome-nettool 3.8.1	x64	436 КБ	1 мин. 50 сек.	3	67%
slpd 1.2.1	x64	128 КБ	3 мин. 17 сек.	1	100%
pbs_server 2.4.8	x64	1.6 МБ	11 мин. 48 сек.	1	100%
jasper 1.900.1	ARM	490 КБ	2 мин. 34 сек.	1	100%

Таблица 7. Результаты поиска ИПО и ДО

В разделе 4.2 описываются методы поиска дефектов ФС, ПБ и ВК. Надо отметить, что детектор поиска БО находит дефекты, которые являются

результатом неправильного использования библиотечных функций копирования и присвоения (strcpy, memcpy…).

Для поиска дефектов БО в анализе конкретной функции для всех точек программы осуществляются следующие шаги:

1. Получаются все переменные, значения которых могут контролироваться пользователем (помечены как taint).
2. Рассматриваются все вызовы функций, которые имеют аннотацию argument_buffer_overflow_annotation, и получаются их фактические аргументы.
3. Для всех фактических аргументов, помеченных как taint, выдается дефект ВО.

Алгоритм поиска дефектов ВК отличается от алгоритма поиска БО только вторым пунктом: рассматриваются все вызовы функций, имеющие аннотацию argument_command_injection_annotation, а при поиске дефектов ФС рассматриваются все вызовы функций, имеющие аннотацию argument_format_string_annotation.

Результаты представлены в таблице 8.

Проект	**Архитектура**	**Размер**	**Время анализа**	**Количество найденных дефектов**	**Процент правильных срабатываний**
dba 2.4.1	x32	312 КБ	1 мин. 40 сек.	12	50%
pbs_server 2.4.8	x32	320 КБ	2 мин. 22 сек.	5	80%
httpd 0.5.0	x32	6.4 МБ	6 мин. 51 сек.	22	90.9%
mkfs 1.1.12	x32	56 КБ	25 сек.	9	100%
pswdb 2.4.1	x32	300 КБ	55 сек.	9	33%
hsolinkcontrol 1.0.118	x32	28 КБ	2 сек.	22	100%
mkfs 1.1.12	x64	56 КБ	19 сек.	7	100%
pbs_server 2.4.8	x64	320 КБ	3 мин. 20 сек.	4	75%
socat 1.7.1.3	arm	259 КБ	48 сек.	1	100%
hsolinkcontrol 1.0.118	arm	23 КБ	4 сек.	25	68%

Таблица 8. Результаты поиска ФС, БО и ВК

В таблице 9 представлены результаты сравнения детектора с LoongChecker.

Проект	**Размер** МБ	**Количество найденных дефектов**	**Процент правильных срабатываний**	**Loong Checker**	**Процент правильных срабатываний LoongChecker**
Serenity .exe	19.6	2	50%	8	12.5%
FoxPlayer.exe	33	2	100%	27	4%

Таблица 9. Результаты сравнения с LoongChecker

В разделе 4.3 описано применение клонов кода и автоматического поиска и определения характера изменений в новых версиях программ для поиска неисправленных версий. Изменения, описанные в разделе 3.3, рассматриваются как исправления в некотором фрагменте исполняемого кода, а старая версия фрагмента кода считается неисправленной. После обнаружения исправлений алгоритм находит клоны неисправленного фрагмента в новой версии исполняемого файла. Такие клоны могут нуждаться в исправлении, о чем выдается предупреждение аналитику. Разработаны 3 метода для нахождения неисправленных фрагментов в старой версии исполняемого файла:

1. Фрагментом неисправленного кода считается измененная функция. Для неисправленной функции находятся все клоны в новой версии (для третьего типа клонов аналитик задает минимальный процент схожести). Используется эвристический алгоритм нахождения клонов кода на основе наибольшего общего подграфа ГЗП.
2. Фрагментом неисправленного кода считается множество базовых блоков, построенных по следующему принципу: в множество вставляется базовый блок, в котором выдается предупреждение (при предупреждении о добавлении базового блока вставляется предыдущий базовый блок), потом проводится слайсинг базовых блоков вверх и вниз по потоку управления. Количество базовых блоков ограничивается аналитиком. Потом делается поиск полученного множества базовых блоков в новой версии исполняемого файла с учетом кодов операций и потока управления между базовыми блоками.
3. Фрагментом неисправленного кода считается множество инструкций, построенных по следующему принципу: в множество вставляются инструкции базового блока, в котором выдается предупреждение (при предупреждении о добавлении базового блока вставляются инструкции предыдущего базового блока). Потом проводится слайсинг вверх и вниз по потоку данных, после чего делается поиск полученного множества инструкций с учетом потока данных.

В **заключении** содержатся выводы и направления последующего развития разработанных методов, алгоритмов и их программных реализаций.

Стоит отметить следующие направления дальнейших исследований:

- Разработка новых типов детекторов дефектов.
- Поддержка символьного выполнения.
- Повышение точности разработанных алгоритмов посредством проведения чувствительного к путям анализа.

Основные результаты диссертационной работы:

1. Разработана архитектура независимого, масштабируемого и легко расширяемого статического анализа исполняемого кода.
2. Предложены и разработаны методы анализа значений и помеченных данных, позволяющие проводить межпроцедурный, чувствительный к контексту, к потоку данных и к потоку управления анализ.
3. Предложены и разработаны методы поиска клонов исполняемого кода и сравнения двух версий исполняемых файлов для автоматического поиска и определения характера изменений в новых версиях программ.
4. Предложены и разработаны методы поиска дефектов использования памяти после освобождения, двойного освобождения памяти, переполнения буфера, форматных строк и внедрения команд, а также поиска неисправленных фрагментов в новой версии исполняемого файла.

Публикации по теме диссертации:

1. А.К. Асланян, Ш.Ф. Курмангалеев, В.Г. Варданян, М.С. Арутюнян, С.С. Саргсян. Платформенно-независимый и масштабируемый инструмент поиска клонов бинарного кода. Труды ИСП РАН, том 28, вып. 5, стр. 215-226, 2016 г. DOI: 10.15514/ISPRAS-2016-28(5)-13.
2. H. Aslanyan, A. Avetisyan, M. Arutunian, G. Keropyan, S. Kurmangaleev and V. Vardanyan. Scalable Framework for Accurate Binary Code Comparison. Ivannikov ISPRAS Open Conference (ISPRAS), Moscow, 2017. 34-38 pp. DOI: 10.1109/ISPRAS.2017.00013.
3. А.К Асланян. Платформа межпроцедурного статического анализа бинарного кода. Труды Института системного программирования РАН, том 30, вып. 5, 2018, стр. 89-100. DOI: 10.15514/ISPRAS-2018-30(5)-5.
4. Hayk K. Aslanyan. Effective and Accurate Binary Clone Detection. Mathematical Problems of Computer Science 48, ISSN 0131-4645. 64-73 pp. 2017.
5. Hayk Aslanyan, Sergey Asryan, Jivan Hakobyan, Vahagn Vardanyan, Sevak Sargsyan, Shamil Kurmangaleev. Multiplatform Static Analysis Framework for Program Defects Detection. 11th International Conference on Computer Science and Information Technologies, 2017. 182-185 pp.
6. Grigor S. Keropyan, Vahagn G. Vardanyan, Hayk K. Aslanyan, Shamil F. Kurmangaleev and Sergey. S. Gaissaryan. Multiplatform Use-After-Free and Double-Free Detection in Binaries. Mathematical Problems of Computer Science 48, ISSN 0131-4645. 50-56 pp. 2017.