Федеральное государственное бюджетное учреждение науки

Институт системного программирования им. В.П. Иванникова

Российской академии наук

На правах рукописи

Асланян Айк 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»

Диссертация

на соискание ученой степени

кандидата физико-математических наук

Научный руководитель:

к. ф.-м. н.
Курмангалеев Шамиль Фаимович

Москва 2019

Оглавление

Введение

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 и 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, потока данных и помеченных данных в исполняемом коде.
4. Разработать методы поиска клонов (синтаксически похожих фрагментов) исполняемого кода и 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 алгоритмы и методы для исполняемых файлов x86, x86-64, ARM.

Научная новизна:

1. Предложены и разработаны методы анализа значений и 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 [1], двойного освобождения памяти [2], переполнения буфера [3], форматных строк [4], внедрения команд [5] 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 в пригодных для анализа исполняемых файлах размером несколько десятков мегабайт. Разработанные методы статического анализа обеспечивают высокую точность и могут использоваться в жизненном цикле разработки безопасного ПО в целях повышения его надежности и безопасности. 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 г.
2. Международная Ершовская конференция по информатике PSI–2017, Москва, Россия, 27-29 июня 2017 г.
3. 11-я международная конференция CSIT 2017 Ереван, Армения, 25-29 сентября, 2017 г.
4. 60-я Научная конференция МФТИ, Москва, Россия, 20-25 ноября 2017 г.
5. Открытая конференция ИСП РАН им. В.П. Иванникова, Москва, Россия, от 30 ноября до 1 декабря 2017 г.
6. 12-я годичная научная конференция Российско-Армянского университета, Ереван, Армения, 4-8 декабря 2017 г.

7. Открытая конференция ИСП РАН им. В.П. Иванникова, Москва, Россия, 22-23 ноября 2018 г.

Публикации:

По теме диссертации опубликовано 6 научных работ, в том числе, 3 научные статьи [6] [7] [8] в рецензируемых журналах, входящих в перечень рекомендованных ВАК РФ. Работа [7] индексирована в Web of Science.

В работах [6] и [9] представлен метод поиска клонов исполняемого файла на основе графов зависимостей программы. В работе [6] личный вклад автора заключается в разработке алгоритмов разделения графов зависимостей программы на подграфы. В статье [9] автором представлен алгоритм поиска наибольших общих подграфов двух графов зависимостей программы. В работе [7] 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 [8] автором описывается платформа межпроцедурного анализа исполняемых файлов. В коллективных работах [10] и [11] описаны методы поиска дефектов использования памяти после освобождения, двойного освобождения памяти и форматных строк. В работе [10] личный вклад автора заключается в разработке алгоритма поиска дефектов форматных строк. Личный вклад автора в статье [11] 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 приложения. Полный объем диссертации составляет 118 страниц, включая 11 рисунков и 22 таблицы. Список литературы содержит 89 наименований.

В первой главе приводится обзор работ, которые имеют отношение к теме диссертации. Рассматриваются современные методы статического анализа исполняемого кода, методы поиска клонов исполняемого кода, а также методы сравнений исполняемых файлов.

Вторая глава посвящена описанию предлагаемого метода для межпроцедурного, контекстно-чувствительного статического анализа исполняемых файлов. Описывается метод анализа значений, модель памяти и анализ потока данных.

В третьей главе рассматриваются методы поиска клонов исполняемого кода и сравнения двух версий исполняемых файлов для автоматического поиска и определения характера изменений в новых версиях программ.

В четвертой главе описываются 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.

В заключении содержатся выводы и направления дальнейшего развития разработанных методов.

Глава 1. Обзор существующих работ

1.1 Подходы к анализу исполняемого кода

В настоящее время используются два подхода к анализу исполняемого кода: статический и динамический анализ. Статический анализ исследуемой программы проводится без ее реального выполнения и включает в себя методы анализа потока управления, анализа потока данных, абстрактной интрерпретации, а также методы, использующие символьное выполнение. Статические анализаторы используют промежуточные представления программы: абстрактные синтаксические деревья, граф потока управления, граф зависимостей программы, граф зависимостей системы, граф вызовов функций и т.д.

Динамический подход позволяет проводить анализ вовремя или после выполнения программ. Динамические анализаторы нуждаются в информации о входных данных и их эффективность напрямую зависит от количества и качества этих данных. Направления динамического анализа включают в себя динамическое символьное выполнение, фаззинг, отладку и профилирование.

Каждый подход имеет свои преимущества и недостатки. Статический анализ исследует все возможные пути выполнения и все значения переменных. Таким образом, он способен обнаруживать дефекты, которые могут долгое время не выявляться динамическим анализатором (он может выявить их только в том случае, если исполняемый путь прошел через точку дефекта при некоторых значениях переменных). Статические анализаторы (в отличие от динамических), как правило, не требуют информацию о входных данных. Кроме того, динамические анализаторы нуждаются в использовании эмуляторов или аппаратуры, которая имеет архитектуру анализируемого исполняемого файла. Однако статические анализаторы допускают

ложные срабатывания (20%-70% [12]) из-за неполного восстановления информации о программе (например, вызовы виртуальных функций) или невозможности обработки и использования всей информации, полученной во время анализа.

Несмотря на свои недостатки, статический анализ кода широко используется в жизненном цикле разработки программ. С помощью статических анализаторов компании, разрабатывающие ПО, могут найти дефекты в своем коде и существенно повысить надежность своей продукции.

1.2 Обзор существующих методов статического анализа исполняемого кода программ

Г. Балакришнан и Т.Репс [13] разработали метод анализа интервалов значений, используемый в инструменте CodeSurfer/x86 для анализа исполняемого кода архитектуры x86. Сначала исполняемый файл дизассемблируется с использованием IDA Pro [14]. Инструмент использует следующую информацию, восстановленную с помощью IDA Pro: графы потока управления, границы функций, вызовы к функциям стандартных библиотек (идентифицируются с использованием алгоритма под названием Fast Library Identification and Recognition Technology − FLIRT [15]), статически известные адреса памяти и смещения. Авторы создали плагин Connector для IDA Pro, создающий структуры данных для представления информации, полученной от IDAPro. Основываясь на структурах данных в Connector, разрабатывался алгоритм для анализа интервалов значений, который дополняет и исправляет информацию, восстановленную IDAPro. CodeSurfer улучшает графы потока управления и граф вызовов функций, учитывая косвенные переходы. С использованием этой информации строится собственная коллекция промежуточных представлений, состоящая из абстрактных синтаксических деревьев, графов потока

управления, графа вызовов функций и графа зависимостей системы (ГЗС) [16]. ГЗС состоит из всех графов зависимостей программы (ГЗП) каждой функции. Вершинам ГЗП соответствуют инструкции в программе, а ребрам − зависимости по данным и по управлению между инструкциями. ГЗП соединены между собой межпроцедурными ребрами, которым соответствуют зависимости потока управления между вызовами функций, а также зависимости данных между фактическими параметрами и формальными параметрами / возвращаемыми значениями.

На основе разработанной абстрактной модели памяти восстанавливается информация о глобальных переменных, локальных переменных, об указателях, структурах, массивах, объектах из классов (и подобъектах из подклассов), о косвенных переходах и косвенных вызовах через указатели функций. В проекте также реализованы декомпилятор и алгоритм анализа алиасов на основе абстрактной интерпретации [17].

В работе [18] Дж. Киндер разработал и реализовал платформу для анализа исполняемого кода архитектуры x86. Основные черты системы:

- Использование промежуточного языка. Сложные машинные инструкции транслируются в последовательность инструкций промежуточного языка. Во время трансляции вызовы функций и возвращение из функций заменяются переходом без условия. Это позволяет преодолеть некоторые запутывающие преобразования кода.
- Конструирование потока управления. Для оптимального решения косвенных ветвлений предлагается интегрированный метод анализа потока управления и потока данных на основе абстрактной интерпретации.
- На основе модели памяти (из работы [13]) проводится анализ интервалов значений. Значения присваиваются регистрам и областям памяти, причем каждое из них помечено идентификатором региона, который служит

символьным базовым адресом. Таким образом, указатели на область глобальной памяти, стек и кучу могут быть идентифицированы, и предполагается, что они не перекрываются. Количество значений ограничивается для каждой переменной для каждого местоположения, что обеспечивает сходимость алгоритма.

- Дизассемблирование по требованию. Вместо того, чтобы пытаться дизассемблировать как можно больше инструкций, дизассемблируется только одна инструкция за раз. Это происходит во время абстрактной интерпретации: транслируется только инструкция, соответствующая следующему этапу выполнения. Такой метод позволяет справляться с перекрывающимися инструкциями, так как не требуется фиксированное представление, которое однозначно сопоставляет каждый байт одной команде. Вместо этого одни и те же байты можно интерпретировать как разные инструкции в зависимости от контекста выполнения.

Разработанные методы были реализованы в инструменте Jakstab. Платформа может анализировать исполняемые файлы Windows и Linux архитектуры x86.

BAP [19] дает возможность анализа исполняемых файлов архитектуры x86 и ARM. Сначала исполняемый файл дизассемблируется на основе линейного алгоритма. Следующим шагом является получение промежуточного представления, которое не имеет побочных эффектов и не зависит от архитектуры. Промежуточное представление приводится в форму статического единственного присваивания [20]. На полученном представлении проводятся анализ и оптимизации. Строятся DEF-USE, USE-DEF цепочки, удаляется мертвый код (с учетом только регистров и флагов, работа с памятью не поддерживается).

Другая работа, посвященная анализу исполняемого кода, – BitBlaze [21]. Система состоит из трех компонентов: Vine, TEMU и Rudder. Vine является

компонентом статического анализа, TEMU – динамического, а Rudder – конкретного и символического анализа, который объединяет статический и динамический анализ. Инструмент выполняет следующие шаги:

1. Получается ассемблер с помощью дизассемблера IDA Pro;
2. Ассемблер транслируется в промежуточное представление VEX IL;
3. VEX IL транслируется на VINE IL.

Авторы адаптировали анализ интервалов значений из [13] для представления VINE IL. На основе этого анализа проводятся следующие виды анализа:

- восстановление косвенных вызовов и переходов в графе потока управления;
- генерация графов зависимостей программы;
- анализ потока данных:
    - распространение констант;
    - удаление мертвого кода;
    - анализ активных переменных.
- генерация C кода из VINE IL.

В [22] представлена платформа поиска критических дефектов с использованием промежуточного представления REIL [23]. Дизассемблирование происходит с помощью IDA Pro, а полученная информация передается в Binnavi [24] для генерации REIL представления. Следующим шагом является трансляция REIL в eREIL. eREIL представление является расширением представления REIL, к которому добавлены новые инструкции. Авторы адаптировали анализ интервалов значений из [13] для представления eREIL. Платформа основана на межпроцедурном анализе. Посредством анализа интервалов значений внутрипроцедурный анализ собирает результаты всех значений программы, кроме локальных переменных. В платформе реализованы анализ помеченных данных, а также анализ зависимостей по данным. На

основе анализа помеченных данных авторы реализовали поиск дефектов форматной строки и некоторых типов переполнения буфера.

В статье [25] описан статический метод поиска дефектов форматной строки в исполняемых файлах x86. Разработанный алгоритм проводит анализ помеченных данных и находит простые случаи проявления дефектов форматной строки внутри одной функции.

В работе [26] описывается метод поиска дефектов для исполняемого кода x86 на основе анализа помеченных данных, который проводится методом статического символьного выполнения.

В статье [27] описывается метод статического анализа поиска дефектов использования памяти после освобождения в исполняемом коде программ архитектуры x86. Метод основан на межпроцедурном анализе доступных выражений. Для каждого базового блока определяется, какие выражения убиваются и какие генерируются. Также обрабатывается влияние вызовов функций на доступные выражения. Если выражение не доступно, но все равно используется, то выдается предупреждение о дефекте.

В [28] авторы представляют инструмент GUEB, который находит дефекты использования памяти после освобождения и двойного освобождения в исполняемом коде. Сначала инструмент отслеживает операции получения и удаления памяти в куче, а также передачу этой информации. Также инструмент учитывает алиасы, которые реализованы на основе анализа значений. Далее полученная информация используется для поиска дефектов. На последнем этапе происходит получение подграфов ГПУ, которые показывают, в какой точке программы освобождалась память и какие инструкции передали информацию.

1.3 Обзор методов поиска клонов исполняемого кода

Для быстрого решения проблем разработчики программного обеспечения часто копируют и вставляют некоторую часть кода программы. Однако это не только может привести к различным ошибкам, но и увеличить размер исходного и исполняемого кода. Согласно исследованиям [29] [30], до 20% кода являются клонами (похожими фрагментами). Существует ряд методов для поиска похожих частей (клонов) исходного кода [31] [32] [33] [34] [35] [36] [37]. Надо отметить, что компилятор путем копирования некоторых частей тоже может создать клоны исполняемого кода, которых нет в исходном. Обнаружение клонов исполняемого кода используется для поиска вредоносных программ, семантических ошибок, нарушения авторских прав и т. д.

Клоны исполняемого кода делятся на три типа. Первый тип – фрагменты кода, которые полностью совпадают. Второй – фрагменты кода, которые могут отличаться типами и значениями данных, а также именами регистров. Третий тип – фрагменты кода, которые могут отличаться типами, значениями данных и именами регистров, а также некоторыми инструкциями, которые могут присутствовать или отсутствовать в конкретном фрагменте.

На рисунке 1 приведены примеры клонов кода в ассемблерной форме (для архитектуры x86). Клон первого типа совпадает с конкретным фрагментом. Клон второго типа отличается от конкретного фрагмента распределением регистра *ecx* вместо *eax*. Клон третьего типа отличается от конкретного фрагмента распределением регистра *ecx* вместо *eax* и отсутствием одной инструкции (*imul eax, ebp+var_4]*).

Фрагмент кода	Клон типа 1	Клон типа 2	Клон типа 3

public main
main proc near
 	
var_4= dword ptr -4
argc= dword ptr 8
argv= dword ptr 0Ch
envp= dword ptr 10h
 	
push ebp
mov ebp, esp
mov [ebp+var_4], 5
mov eax,[ebp+var_4]
imul eax,[ebp+var_4]
leave
retn
main endp
	public main
	main proc near
	
	var_4= dword ptr -4
	argc= dword ptr 8
	argv= dword ptr 0Ch
	envp= dword ptr 10h
	
	push ebp
	mov ebp, esp
	mov [ebp+var_4], 5
	mov eax,[ebp+var_4]
	imul eax,[ebp+var_4]
	leave
	retn
	main endp
		public main
		main proc near
		
		var_4= dword ptr -4
		argc= dword ptr 8
		argv= dword ptr 0Ch
		envp= dword ptr 10h
		
		push ebp
		mov ebp, esp
		mov [ebp+var_4],**10**
		mov **ecx**,[ebp+var_4]
		imul **ecx**,[ebp+var_4]
		leave
		retn
		main endp
			public main
			main proc near
			
			var_1= dword ptr -4
			argc= dword ptr 8
			argv= dword ptr 0Ch
			envp= dword ptr 10h
			
			push ebp
			mov ebp, esp
			mov [ebp+var_1],**15**
			mov **ecx**,[ebp+var_1]
			
			leave
			retn
			main endp

Рисунок 1. Примеры типов клонов для архитектуры x86 (соответствующий ассемблер)

Существует несколько подходов поиска клонов исполняемого кода.

- **Текстовый подход.**

Инструменты на основе текстового подхода для обнаружения клонов исполняемого кода рассматривают фрагмент исполняемого кода как последовательность байтов или строк кода ассемблера и сравнивают каждую пару фрагментов кода для поиска идентичных последовательностей.

Джанг и Брумли [38] предложили алгоритм на основе отпечатков для кластеризации вредоносных программ. Алгоритм называется BitShred и использует фильтры Блума [39]. Кроме того, BitShred использовался для обнаружения ошибок, которые возникали в результате некорректного использования скопированного фрагмента кода, однако такой подход не привел к решению данной задачи. Алгоритм работы BitShred состоит из трех этапов. На первом этапе BitShred делит весь исполняемый код на так называемые n-граммы. Затем для повышения масштабируемости и эффективности хранения BitShred использует фильтр Блума,

который создан из всех фрагментов исполняемого файла для представления отпечатка файла. Далее BitShred вычисляет сходство между двумя отпечатками − размер пересечения наборов фрагментов делится на размер объединения. В случае сравнения двух фильтры Блума ов используется следующая формула:

$$J(A,B) = S\ (BF_A \wedge BF_B) / S\ (BF_A \vee BF_B)$$

$S(BF_A)$ − это счетчик установленных битов в BF_A, а $S(BF_B)$ − это счетчик установленных битов в BF_B. Наконец, фрагменты, имеющие более высокую оценку сходства, группируются вместе. Алгоритмы, основанные на этом подходе, находят клоны только первого типа.

- **Подход, основанный на токенах**.

Инструменты поиска клонов исполняемого кода, которые основаны на токенах, дизассемблируют исполняемый файл и полученный ассемблер разбивают на коды операций и операнды. Типы кодов операций и операндов могут быть обобщены или отфильтрованы, и полученная последовательность будет проверена в целях обнаружения дубликатов кода.

А. Шульман [40] предложил систему для поиска клонов функций в исполняемом файле. Это первая работа, которая находит клоны на уровне функций. Основная идея реализованной системы − создать хеш каждой функции в анализируемом исполняемом файле и сохранить ее в базе данных. Идентичные хеш-значения указывают на наличие клонов в коде. Хеш-коды вычисляются на основе opstring’ов с учетом следующих моментов:

1. В хеше используется только код операции инструкции, а не операнда;
2. Коды операций преобразуются в их мнемонику. Например, кодом операции 51 является инструкция push ecx. Opstring’гу вместо числового значения присваивается push;
3. Метки местоположения добавляются в opstring для более точного результата;

4. Вызовы системных функций нормализируются (без учета формата ASCII или UNICODE)

В результате получаются opstring'и, например, loc,[MessageBox],cmp,jnz,ret. Для opstring'ов вычисляется «md5» хеш [41].

Алгоритм, предложенный в [42], находит клоны исполняемого кода для классификации вредоносных программ. Для этого рассматриваются отдельные части фрагментов и, если эти части имеют похожие части, фрагменты считаются клонами. Целью проекта стало создание моделей вредоносного ПО, основанных на особенностях их исполняемого кода и поиск таких моделей в тестируемых файлах. Были построены модели для обработки программ посредством переупорядочения кода (переупорядочение команд или блоков) . Для реализации использовались n-граммы и их перестановки, которые авторы назвали n-пермы. Это позволяет утверждать, что полученные перестановки могут включать в себя переупорядочение команд, блоков или подпрограмм.

Вначале алгоритм разбивает исполняемый файл на токены для преобразования входной программы в последовательность кодов операций. Затем из этой последовательности извлекаются n-граммы и n-пермы. После этого создается матрица, которая показывает количество появления характеристик (то есть, сколько раз конкретный n-грамм или n-perm встречается в фрагментах). Полученная матрица характеризует сходство фрагментов кода.

Алгоритмы, основанные на этом подходе, находят клоны первого и второго типа.

- **Метрический подход.**

В работе [43] разработана платформа для поиска клонов исполняемого кода на основе метрик. Для поиска выполняются следующие шаги:

1. Дизассемблирование исполняемого файла с использованием IDA Pro;

2. Разбивание кода ассемблера на перекрывающиеся фрагменты кода, которые состоят из последующих инструкций функции;
3. Нормализация последовательностей команд посредством абстракции информации о ячейках памяти и регистрах;
4. Вычисление метрик для нормализованных последовательностей;
5. Сравнение метрик и получение групп клонов.

Авторы предлагают два метода поиска клонов исполняемого кода: поиск точных клонов (нормализованные последовательности совпадают) и поиск неточных клонов (некоторые члены нормализованных последовательностей не совпадают). Первый метод использует хеш для каждой области кода, и фрагменты кода считаются клонами, если хеш-коды совпадают. Второй метод извлекает набор характеристик из нормализованной последовательности и ищет другие нормализованные последовательности с тем же набором характеристик. Метод подсчитывает количество вхождений каждой характеристики для создания характеризующего вектора для каждой нормализованной последовательности. Затем используется локально-чувствительное хеширование (LSH) [44] для каждого характеризующего вектора и вычисляются расстояния для кластеризации клонов.
Основываясь на этой работе, М. Фархади и др. [45] создали систему для обнаружения клонов вредоносного кода в программах.
Брусчи и др. создали систему [46] нормализации ассемблерного кода для обнаружения вредоносных программ. Авторы разработали прототип на основе декомпилятора Boomerang [47]. Использовались следующие шаги нормализации:

1. Распространение значений;
2. Удаление мертвого кода;
3. Оптимизация графа потока управления путем удаления искусственно добавленных и бесполезных ребер.

После нормализации авторы вычисляют метрики из 7 пунктов:

1. Количество узлов в графе потока управления;
2. Количество ребер в графе потока управления;
3. Количество прямых вызовов;
4. Количество косвенных вызовов;
5. Количество прямых переходов;
6. Количество косвенных переходов;
7. Количество условных переходов.

Эта метрика используется для сравнения фрагментов кода. Сходство определяется на основе евклидова расстояния.

Алгоритмы, основанные на этом подходе, находят клоны первого, второго и третьего типа.

- **Подход, основанный на поведении программы.**

В работе [48] авторы разработали систему под названием REANIMATOR, которая позволяет идентифицировать скрытые функциональности во вредоносных программах. В основе системы лежит тот факт, что динамический анализ вредоносных программ фиксирует выполнение вредоносного ПО и сообщает о его поведении. Основная идея работы − запустить большой набор вредоносных программ и определить их динамически различные функциональные возможности. Как только они будут идентифицированы, функции можно будет обнаружить в новых вредоносных программах. Работа системы REANIMATOR осуществляется в три этапа. Первые два отвечают за создание функциональной модели для разных моделей поведения. На третьем этапе построенные модели используются для проверки скрытых поведений.

Алгоритмы, основанные на этом подходе, находят клоны первого, второго и третьего типа.

1.4 Обзор методов сравнения исполняемых файлов

Методы сравнения исполняемых файлов используются для сравнения двух версий одной программы. Результат анализа, как правило, представляет собой сопоставление фрагментов первого и второго файла.

Существует несколько методов для сравнения исполняемых файлов. Одним из первых предложенных методов является BMAT [49]. Метод состоит из двух этапов. На первом этапе сопоставляются функции. Сопоставление происходит на основе имен и типов аргументов. Далее, используя хеш-коды, временно сопоставляются функции, у которых имена изменились только на несколько символов. После процедуры сопоставления внутри каждой пары функций BMAT сопоставляет базовые блоки. Если сопоставляются большой процент базовых блоков, то временное сопоставление сохраняется, в противном случае – удаляется. Сопоставление базовых блоков происходит на основы хеширования, в котором учитываются адреса, коды операций и операнды инструкций, а также информация о потоке управления.

Т.Дюллиен и др. предложили алгоритм [50] [51] [52] сравнения исполняемых файлов − BinDiff, который основан на сопоставлении узлов и ребер графов потока управления и графов вызовов. Сначала исполняемые файлы дизассемблируются с помощью IDA Pro. На основе восстановленных графов и ассемблера происходит сопоставление функций. Алгоритм основан на вычислении хешей на графах потока управления и графах вызовов. Ребрам графов (и потока управления, и вызовов) сопоставляется вектор с пятью элементами:

1. $z1$ − номер ребра при топологическом обходе графа;
2. $z2$ − количество входных ребер начальной вершины ребра;
3. $z3$ − количество выходных ребер начальной вершины ребра;
4. $z4$ − количество входных ребер конечной вершины ребра;
5. $z5$ − количество выходных ребер конечной вершины ребра.

После получения всех векторов считаются MD-index хеши для графа G следующим образом:

$$emb(vector) = z1 + z2\sqrt{2} + z3\sqrt{3} + z4\sqrt{5} + z5\sqrt{7}$$

$$MD - index(G) = \sum_{for\ all\ vector} \frac{1}{\sqrt{emb(vector)}}$$

Сопоставление функций происходит посредством вычисления следующих хешей:

- Стандартный хеш-код для последовательности байтов конкретной функции;
- MD-index конкретной функции;
- MD-index хеш-коды начальной и конечной функции ребра графа вызовов;
- MD-index хеш-коды для функций, которые вызывают конкретную функцию;
- MD-index хеш-коды для функций, которые вызываются конкретной функцией;
- Хеш-код, основанный на произведении небольших простых чисел (такой хеш игнорирует переупорядочение инструкций, вызванных компилятором).

Существует ряд работ, которые нацелены на усовершенствование BinDiff.

Первая из них – работа И. Брионеса и А.Гомеса [53]. Её цель − оптимизация реализации BinDiff в целях сокращения времени сопоставления функций для баз данных вредоносных программ. В работе [54] авторы используют метод BinDiff для сопоставления базовых блоков, а затем − венгерский алгоритм [55] для дальнейшего сопоставления оставшихся базовых блоков.

Инструмент DarunGrim2 [56] основан на методе отпечатков. Основная идея − создать уникальный отпечаток для каждого базового блока, объединить отпечатки и использовать полученный результат для сопоставления базовых блоков. Отпечатки генерируются на основе кодов операций базовых блоков простейшими зависимостями по данным (учитываются только регистры).

В работе [57] автор сравнивает исполняемые файлы на основе расстояния редактирования графов [58]. Разработанный алгоритм строит графы базовых блоков и применяет расстояние редактирования графов для поиска близости двух функций.

Расстояние редактирования строк [59] используется в работе [60]. Алгоритм кодирует графы потока управления как строки и для пары строк применяется расстояние редактирования строк для сопоставления функций.

BinHash [61] − другой алгоритм сопоставления функций, основанный на построении хешей. Авторы предлагают схему, которая анализирует семантику функций и строит семантические хеш-коды. Для каждой функции рассматриваются некоторые особенности и на их основе считаются хеш-коды. Аналогичный метод используется в [62]. В работе авторы представили инструмент BinJuice, который за несколько этапов нормализует инструкции базового блока для извлечения семантического хеша. Семантически похожие базовые блоки затем идентифицируются посредством сравнений хешей.

Другие исследования основаны на символическом выполнении и SMT-решателях. BinHunt [63] пытается найти семантические различия между исполняемой программой и ее новой версией. Сначала алгоритм находит наибольший общий подграф двух графов вызовов, после чего сопоставляются базовые блоки. Чтобы определить, насколько похожи два базовых блока, проверяются все возможные пары уравнений. Д. Минг и другие предложили iBinHunt [64], который улучшает алгоритм BinHunt путем добавления анализа помеченных данных. В алгоритме рассматриваются только те уравнения, в которых переменные зависят из пользовательского ввода.

1.5 Обзор методов анализа характера изменений программ между версиями

Как говорится в главе 1.3, согласно исследованиям [29] [30], до 20% кода являются клонами (похожими фрагментами). Иногда в новых версиях программ исправляется ошибка, но в клонах фрагмента с ошибкой исправление не происходит. Необработанные клоны кода представляют собой скрытые ошибки, которые необходимо быстро обнаружить. Существует несколько работ, посвященных поиску клонов кода с ошибкой.

В статьях [65], [66], [67] описывается метод поиска неисправленных клонов кода на основе измененного фрагмента. В работах применяется ряд алгоритмов, которые выполняют следующие шаги:

1. Получение измененного фрагмента с помощью svn [68], git [69];
2. Извлечение фрагмента кода в старой измененной версии;
3. Нормализация кода (удаление комментариев и ненужных пробелов, преобразование всех символов в нижний регистр);
4. Поиск клонов фрагмента кода с ошибкой.

Описанные работы отличаются только некоторыми оптимизациями и методами поиска клонов кода. Например, в работе [65] алгоритм после нормализации удаляет мертвый код и проводит поиск клонов кода на основе фильтры Блума ов [39]. В работах [66] и [67] поиск клонов кода основан на хешировании.

SPAIN [70] представляет собой систему анализа изменений, которая автоматически идентифицирует изменения безопасности и суммирует шаблоны изменений и соответствующие им шаблоны дефектов. В частности, если даны исходные и исправленные версии исполняемых файлов, SPAIN находит функции, которые были изменены в исходном исполняемом файле с помощью инструмента BinDiff [50]. Затем инструмент обнаруживает измененные последовательности базовых блоков для каждой исправленной функции для фиксации изменений на

уровне функции. Такие измененные последовательности базовых блоков могут содержать изменения, исправляющие ошибки. Для этого рассматриваются изменения, которые исправляют ошибки двойного освобождения памяти, целочисленного переполнения, использования памяти после освобождения, использования нулевого указателя и переполнения буфера. Далее проводится анализ помеченных данных и если переменные измененных фрагментов помечаются, то выдается предупреждение аналитику.

1.6 Выводы

В разработке программного обеспечения широко используется статический анализ кода. Более того, статический анализ исполняемого кода имеет тенденцию развития.

В главе 1.2 описаны современные методы статического анализа исполняемых файлов архитектур x86 и ARM. Все методы имеют ограничения по масштабируемости, так как не поддерживают аннотации для функций. Приведенные в данной диссертационной работе методы анализа программ являются архитектурно независимыми. Более того, предложенный метод включает в себя межпроцедурный, контекстно-чувствительный и потоко-чувствительный анализ исполняемых файлов.

В главе 1.3.1 описаны современные методы поиска клонов исполняемого кода. Текстовый подход позволяет находить только первый тип клонов. Подходы, основанные на токенах, находят клоны первого и второго типов. Подходы, основанные на метриках, могут обнаружить клоны первого, второго и третьего типов, однако у них наблюдается низкий процент истинных срабатываний. Все описанные методы поиска клонов не учитывают поток данных, что приводит к низкому проценту истинных срабатываний. Более того, эти методы не справляются с переупорядочением инструкций. Подходы, основанные на поведении программы, находят клоны первого, второго и третьего типов, однако при этом они требуют

большой вычислительной мощности. Предложенный в диссертации метод поиска клонов исполняемого кода находит клоны с большой точностью (больше 90%), может анализировать исполняемые файлы размером в десятки мегабайт за несколько часов и легко справляется с переупорядочением инструкций.

В главе 1.4.1 описаны современные подходы к сравнению исполняемых файлов. Все представленные работы имеют недостаток: они не учитывают поток данных, что приводит к снижению качества полученных результатов. В диссертации предложены методы для сравнения исполняемых файлов, которые основаны 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.

Доступны результаты только двух инструментов, выполняющих поиск дефектов использования памяти после освобождения, двойного освобождения памяти, форматных строк и переполнения буфера в исполняемом коде. Это GUEB [28] и LoongChecker [22]. Однако у них наблюдается большое количество неправильных срабатываний (больше 90%). Программная реализация и результаты более продвинутых 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).

В диссертации предложен также метод поиска 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.

Глава 2. Межпроцедурный, контекстно-чувствительный статический анализ исполняемых файлов

Вторая глава посвящена методам статического анализа исполняемых файлов. Разработаны и реализованы методы анализа значений, анализа помеченных данных, достигающих определений, DEF-USE, USE-DEF цепочек, трансформация удаления мертвого кода, анализ динамической памяти (отслеживание выделения и удаления 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 исполняемых файлов за несколько часов);
- Возможность расширения функционала платформы.

На рисунке 2 представлена общая архитектура анализа. Первым шагом является получение кода ассемблера из исполняемого файла. Дизассемблер создает инструкции языка ассемблера, используя исполняемый код в качестве входных данных.

В инструменте используется дизассемблер IDA Pro [14], 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 [24], который транслирует ассемблер в представлении REIL

Рисунок 2. Архитектура статического анализа исполняемых файлов

(Reverse Engineering Intermediate Language). REIL представление является промежуточным языком низкого уровня, который можно использовать для написания независимых от платформы алгоритмов анализа. Он состоит из 17 инструкций, причем каждая из них вычисляет не более одного результата и не имеет побочных эффектов (установки флагов и т.д.). REIL представление создано для виртуального процессора с неограниченной памятью и неограниченным количеством регистров, которые назначаются символами t0, t1, t2 и т. д. Регистры целевой машины также доступны в REIL.

С использованием REIL представления реализованы межпроцедурный анализ, внутрипроцедурный анализ, детекторы дефектов, поиск клонов исполняемого кода, сравнение двух версий исполняемых файлов для автоматического поиска и определения характера изменений в новых версиях программ и поиск неисправленных фрагментов в новой версии программ.

2.1 Межпроцедурный анализ

После генерации REIL представления граф вызовов функций приводится в ациклическую форму.

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 [71].
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.

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

Доказательство. Предположим, после всех шагов алгоритма в графе остался ориентированный цикл $v_1, v_2, ..., v_k$. Обозначим $K_1, K_2, ..., K_p$ все сильно связанные компоненты начального графа вызовов.Так как из произвольного узла каждого сильно связанного компонента существует путь ко всем вершинам того же компонента, то после всех шагов 2.1, 2.2, 2.3, 2.4, очевидно, что $K_1, K_2, ..., K_p$ больше не будет иметь циклов. Получается, что вершины цикла принадлежат нескольким сильно связанным компонентам – $K_{i'}$ и $K_{j'}$, $v_i \in K_{i'}$, $v_j \in K_{j'}$. Тогда получится, что из любой вершины $K_{i'}$ существует путь к любой вершине $K_{j'}$, так как существует путь из любой вершины $K_{i'}$ в v_i и существует путь из v_i в v_j (если $i<j$, то $v_i, v_{i+1},..., v_j$, если $i>j$, то $v_i, v_{i+1},..., v_k, v_1, ..., v_j$), и из v_j к вершине из $K_{j'}$. Также из любой вершины $K_{j'}$ существует путь к любой вершине $K_{i'}$, так как существует путь из любой вершины $K_{j'}$ в v_j и существует путь из v_j в v_i (если $j<i$, то $v_j, v_{j+1},..., v_i$, если $j>j$, то $v_j, v_{j+1},..., v_k, v_1, ..., v_i$), и из v_i к вершине из $K_{i'}$. Полученное противоречит тому, что $K_{i'}$ и $K_{j'}$ являются разными сильно связанными компонентами начального графа вызовов функций. ■

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

Доказательство. Временная сложность алгоритма Тарьяна составляет *O(V+E).* Обозначим $K_1, K_2, ..., K_p$ все сильно связанные компоненты начального графа вызовов, количество вершин компонента K_i обозначим через V_i, а E_i – количество ребер. Для каждого компонента проводится поиск в глубину, сложность которого является $O(V_i + E_i)$, что эквивалентно существованию m_i такого, что сложность равна $m_i*(V_i + E_i)$. Временная сложность поиска в глубину для всех компонентов будет:

$$\sum_{i=1}^{p} m_i(V_i + E_i)$$

Обазначим $m_{max} = \max_i m_i$.

$$\sum_{i=1}^{p} m_i(V_i + E_i) \le \sum_{i=1}^{p} m_{max}(V_i + E_i) \le m_{max} \sum_{i=1}^{p} (V_i + E_i) \le m_{max} \sum_{i=1}^{p} (E_i + E_i)$$

$$= 2m_{max} \sum_{i=1}^{p} E_i \le 2m_{max}E$$

То есть временная сложность поиска в глубину для всех компонентов будет $O(E)$. Общая временная сложность алгоритма будет $O(V+E)+O(E)=O(V+E)$. ■

Далее проводится разбиение вершин графа вызовов на группы (рисунок 3) по следующему принципу: в первой группе находятся вершины, у которых нет исходящих ребер. Во второй – те вершины, все предшественники которых находятся в первой группе. Таким образом, в каждую следующую группу попадают те вершины, у которых все выходящие вершины уже рассмотрены и принадлежат предыдущим группам. Так как в графе вызовов больше нет ориентированных циклов, то через конечное количество шагов алгоритм завершится, и каждая вершина попадет в некоторую группу. Сформулируем более формально:

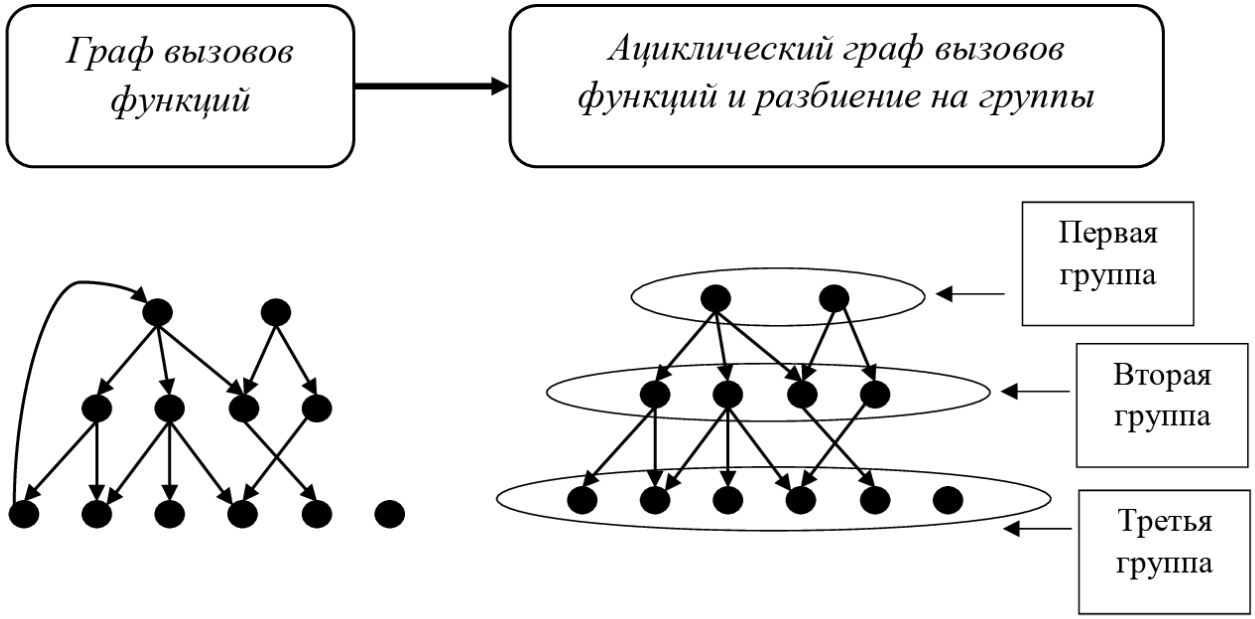

Рисунок 3. Разбиение вершин графа вызовов на группы

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 исходящих 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.

Доказательство. На шаге 1 алгоритм делает *V* операций, чтобы найти все вершины, у которых нет исходящих ребер. На шаге 2 для каждой вершины алгоритм выполняет столько операций, сколько конкретная вершина имеет исходящих ребер.

Количество операций для всех вершин будет не больше, чем E. Получается, что временная сложность алгоритма 2 составляет $O(V+E)$ ■

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 доступнао тело. Это значит, что функции динамически связанных библиотек, не анализируются (для них доступны только аннотации). Функции, соответствующие каждой группе, анализируются параллельно.

Теорема 4. Вычислительная сложность межпроцедурного анализа составляет $O(V+E)$, где V – количество вершин графа вызовов функций, а E – количество ребер.

Доказательство очевидно и следует из теорем 2 и 3**Ошибка! Источник ссылки не найден.**

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 удаляет динамическую память, на которую показывает первый аргумент. При анализе конкретной функции используются аннотации всех вызываемых ею функций.

2.1.1. Аннотации функций

Для достижения масштабируемости используются аннотации для функций. Каждая аннотация содержит информацию о некотором свойстве функции. Архитектура инструмента позволяет легко добавлять аннотации для реализации новых видов анализа. 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 контолируется пользователем (например, функция gets, fgets);
- copy_arguments_annotation – функция копирует информации из одного аргумента в другой.

Аннотации для функций формируются следующим способом:

1. Аннотации для функций стандартных библиотек получены из инструмента Svace [12] подробнее в [72] раздел 3.3.3.
2. Аннотации для других функций считаются в процессе внутрипроцедурного анализа на основе аннотаций всех функций, которые они вызывают.

Очевидно, что если нет рекурсивных вызовов, то при анализе конкретной функции аннотации всех вызывающихся 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 считается, что вызывающие функции не имеют аннотаций.

Надо отметить, что для всех аннотаций имеется поддержка работы со структурами. Например, если функция принимает структуру как аргумент и удаляет только одно поле, то в аннотации информация сохраняется точно так же и при использовании других полей структуры не имеет влияния. Это достигается использованием разработанной модели памяти.

Внутрипроцедурный анализ для каждой функции запускается только один раз, что позволяет достичь масштабируемости с учетом количества всех функций. Разбиение вершин графа вызовов на группы дает преимущество анализировать функции в каждой отдельной группе параллельно.

2.1.2. Контекстно-чувствительность

Контекстно-чувствительность анализа достигается использованием аннотаций функций. Так как все аннотации являются параметризованными, то все виды внутрипроцедурного анализа при обработке инструкций вызова функции используют аннотации и заменяют формальные аргументы функции на конкретные значения. Контекстно-чувствительность значительно повышает точность разработанных видов анализа и детекторов дефектов.

2.2. Внутрипроцедурный анализ

Внутрипроцедурный анализ проводится на основе REIL представления функции. Во время анализа кода при обработке инструкций вызова функций используются аннотации этой функции, и изменение контента происходит с учетом реальных параметров и соглашений о вызовах. Этот процесс обеспечивает контекстно-чувствительность анализа. В текущий момент в платформе разработаны и реализованы анализ значений, анализ достигающих определений, построение DEF-USE и USE-DEF цепочек, трансформация для удаления мертвого кода, анализ помеченных данных и анализ динамической памяти. Архитектура внутрипроцедурного анализа позволяет легко расширять множество видов анализа (рисунок 4) и добавлять плагины. Реализован программный интерфейс пользователя, который предоставляет возможность написать новые виды анализа и детекторы дефектов.

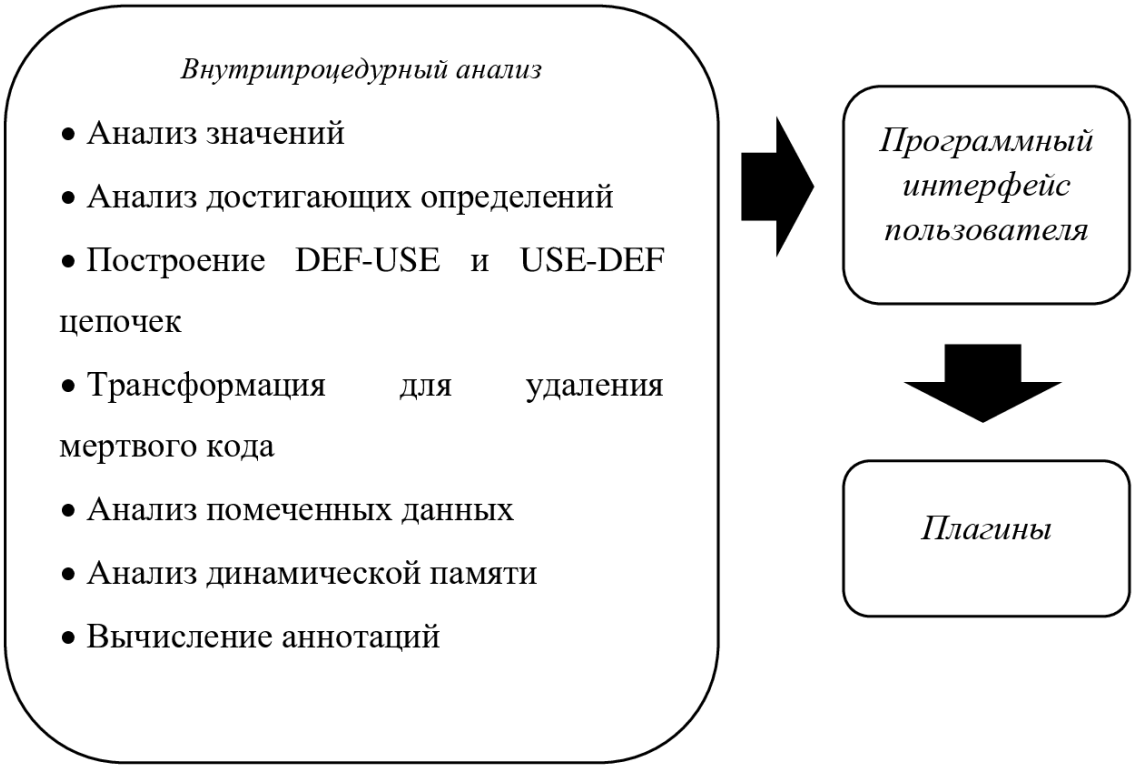

Рисунок 4. Архитектура внутрипроцедурного анализа

2.2.1. Анализ значений

Анализ значений [73] используется для отслеживания значений в регистрах и в ячейках памяти. Регистры целевой архитектуры и временные регистры, а также все ячейки памяти, которые используются в программе, условно называются переменными. Во время анализа все переменные получают значения. Во всех точках программы сохраняются значения всех переменных. Для значений областей памяти была разработана и реализована модель памяти, который моделирует память в стеке, в куче и в статической области.

Анализ значений проводится на основе итеративного алгоритма анализа потока данных. Для этого определяется полурешетка: задаются начальные значения для всех переменных и определяются передаточные функции. Все остальные виды анализа базируются на анализе значений.

В разработанном анализе значений существуют несколько типов символьных значений: целое число, регистр целевой архитектуры, временный регистр REIL, область памяти и специальные значения bottom и top. Значение bottom присваивается переменным, значение которых неизвестно (в полурешетке представляет нижний элемент), а значение top присваивается переменным, которые могут иметь любое значение (в полурешетке представляет верхний элемент). В рисунке 5 представлена диаграмма полурешетки анализа значений.

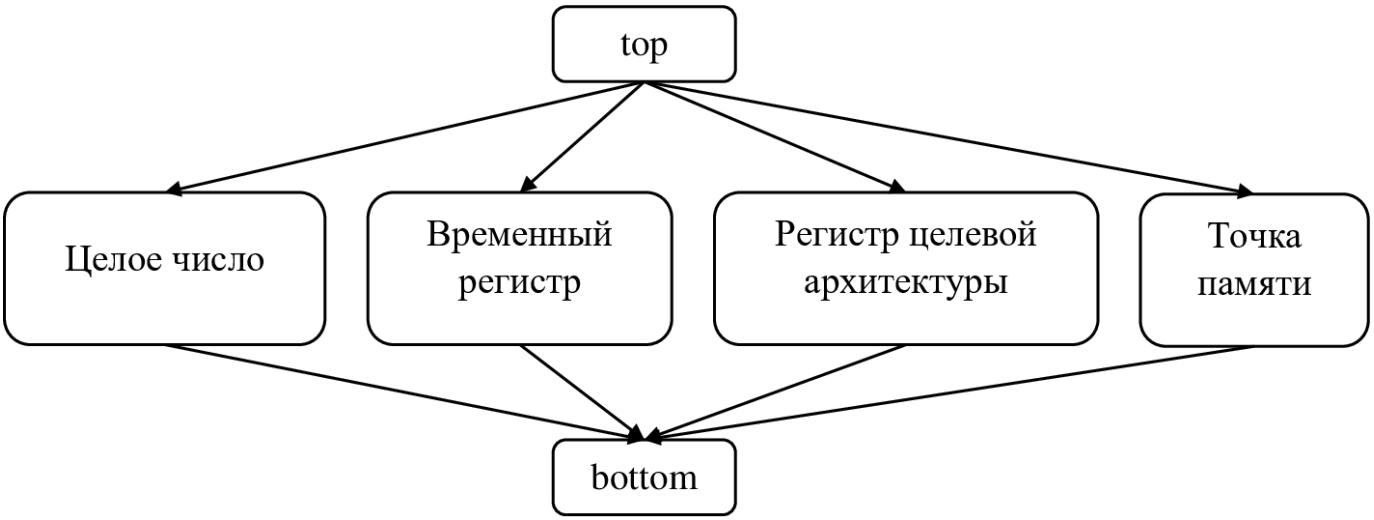

Рисунок 5. Диаграмма полурешетки анализа значений

Простой моделью памяти является рассмотрение памяти как массива байт. Запись (чтение) в этой тривиальной модели памяти 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 −П$stack$, 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).

Важно отметить, что не все части формулы обязательно используются для моделирования конкретной ячейки памяти.

Анализ значений основан на итеративном алгоритме потока данных. Определяется полурешетка, передаточные функции и начальные значения. Верхним элементом полурешетки является top, а нижним элементом – bottom. Начальным значениям всех переменных присваивается bottom. Если тип операндов является временным регистром или регистром конкретной архитектуры, то в таблице типом будет регистр. Исключением являются аргументы функции и начало стека, которым присваиваются символьные значения. Для всех инструкций REIL определяются передаточные функции. Ниже представлены все передаточные функции для одного значения каждого операнда. Если операнды имеют более одного значения, то представленные функции применяются для всех пар значений операндов.

Передаточная функция анализа значений для инструкции add t1,t2,t3

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
top	*top*	*Любое значение*	*-*	*top*	*top*
Любое значение	*-*	*top*	*top*	*top*	*top*
bottom	*bottom*	*Любое значение*	*-*	*bottom*	*bottom*
Любое значение	*-*	*bottom*	*bottom*	*bottom*	*bottom*
C1	*Целое число*	*C2*	*Целое число*	*C1+C2*	*Целое число*
**(reg + constants_array)+ constant*	*Точка памяти*	*C*	*Целое число*	**(reg+constants_array)+(constant+C)*	*Точка памяти*
C	*Целое число*	**(reg + constants_array)+ constant*	*Точка памяти*	**(reg+constants_array)+(constant+C)*	*Точка памяти*
В остальных случаях				*top*	*top*

Передаточная функция анализа значений для инструкции and t1, t2, t3

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
top	*top*	*Любое значение*	*-*	*top*	*top*
Любое значение	*-*	*top*	*top*	*top*	*top*
bottom	*bottom*	*Любое значение*	*-*	*bottom*	*bottom*
Любое значение	*-*	*bottom*	*bottom*	*bottom*	*bottom*
C1	*Целое число*	*C2*	*Целое число*	*C1 and C2*	*Целое число*
В остальных случаях				*top*	*top*

Передаточная функция анализа значений для инструкции bisz t1, ,t3

Значение t1	***Тип t1***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
top	*top*	*top*	*top*
bottom	*bottom*	*bottom*	*bottom*
0	*Целое число*	*0*	*Целое число*
C1 (!=0)	*Целое число*	*1*	*Целое число*
В остальных случаях		*top*	*top*

Передаточная функция анализа значений для инструкции bsh *t1, t2, t3*

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
top	*top*	*Любое значение*	*-*	*top*	*top*
Любое значение	*-*	*top*	*top*	*top*	*top*
bottom	*bottom*	*Любое значение*	*-*	*bottom*	*bottom*
Любое значение	*-*	*bottom*	*bottom*	*bottom*	*bottom*
C1	*Целое число*	*C2*	*Целое число*	*C1<<C2*	*Целое число*
В остальных случаях				*top*	*top*

Передаточная функция анализа значений для инструкции div *t1, t2, t3*

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
top	*top*	*Любое значение*	*-*	*top*	*top*
Любое значение	*-*	*top*	*top*	*top*	*top*

bottom	*bottom*	*Любое значение*	*-*	*bottom*	*bottom*
Любое значение	*-*	*bottom*	*bottom*	*bottom*	*bottom*
C1	*Целое число*	*C2*	*Целое число*	*C1/C2*	*Целое число*
В остальных случаях				*top*	*top*

Передаточная функция анализа значений для инструкции ldm t1, , t3

Значение t1	***Тип t1***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
top	*top*	*top*	*top*
bottom	*bottom*	*bottom*	*bottom*
**(reg + constants_array)+ constant*	*Точка памяти*	*Значение точки памяти*	*-*
reg	*Регистр*	*Значение регистра*	*-*
В остальных случаях		*top*	*top*

Передаточная функция анализа значений для инструкции mod t1, t2, t3

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***

top	*top*	*Любое значение*	*-*	*top*	*top*
Любое значение	*-*	*top*	*top*	*top*	*top*
bottom	*bottom*	*Любое значение*	*-*	*bottom*	*bottom*
Любое значение	*-*	*bottom*	*bottom*	*bottom*	*bottom*
C1	*Целое число*	*C2*	*Целое число*	*C1%C2*	*Целое число*
В остальных случаях				*top*	*top*

Передаточная функция анализа значений для инструкции mul t1, t2, t3

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
top	*top*	*Любое значение*	*-*	*top*	*top*
Любое значение	*-*	*top*	*top*	*top*	*top*
bottom	*bottom*	*Любое значение*	*-*	*bottom*	*bottom*
Любое значение	*-*	*bottom*	*bottom*	*bottom*	*bottom*

C1	*Целое число*	*C2*	*Целое число*	*C1*C2*	*Целое число*
В остальных случаях				*top*	*top*

Передаточная функция анализа значений для инструкции or *t1, t2, t3*

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
top	*top*	*Любое значение*	*-*	*top*	*top*
Любое значение	*-*	*top*	*top*	*top*	*top*
bottom	*bottom*	*Любое значение*	*-*	*bottom*	*bottom*
Любое значение	*-*	*bottom*	*bottom*	*bottom*	*bottom*
C1	*Целое число*	*C2*	*Целое число*	*C1 or C2*	*Целое число*
В остальных случаях				*top*	*top*

Передаточная функция анализа значений для инструкции stm *t1, ,t3*

Значение t1	***Тип t1***	***Значение точки памяти после инструкции***	***Тип точки памяти после инструкции***
top	*top*	*top*	*top*
bottom	*bottom*	*bottom*	*bottom*

C1	*Целое число*	*C1*	*Целое число*
reg	*Регистр*	*reg*	*Регистр*
**(reg + constants_array) + constant*	*Точка памяти*	**(reg + constants_array)+ constant*	*Точка памяти*
В остальных случаях		*top*	*top*

Передаточная функция анализа значений для инструкции str t1, ,t3

Значение t1	***Тип t1***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
top	*top*	*top*	*top*
bottom	*bottom*	*bottom*	*bottom*
C1	*Целое число*	*C1*	*Целое число*
reg	*Регистр*	*reg*	*Регистр*
**(reg + constants_array)+ constant*	*Точка памяти*	**(reg + constants_array) + constant*	*Точка памяти*
В остальных случаях		*top*	*top*

Передаточная функция анализа значений для инструкции sub t1,t2,t3

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***

top	*top*	*Любое значение*	*-*	*top*	*top*
Любое значение	*-*	*top*	*top*	*top*	*top*
bottom	*bottom*	*Любое значение*	*-*	*bottom*	*bottom*
Любое значение	*-*	*bottom*	*bottom*	*bottom*	*bottom*
C1	*Целое число*	*C2*	*Целое число*	*C1-C2*	*Целое число*
**(reg + constants_array)+ constant*	*Точка памяти*	*C*	*Целое число*	**(reg+constants_array)+(constant-C)*	*Точка памяти*
В остальных случаях				*top*	*top*

Передаточная функция анализа значений для инструкции undef, , t3

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***
-	*-*	*-*	*-*	*bottom*	*bottom*

Передаточная функция анализа значений для инструкции xor t1, t2, t3

Значение t1	***Тип t1***	***Значение t2***	***Тип t2***	***Значение t3 после инструкции***	***Тип t3 после инструкции***

top	*top*	*Любое значение*	-	*top*	*top*
Любое значение	-	*top*	*top*	*top*	*top*
bottom	*bottom*	*Любое значение*	-	*bottom*	*bottom*
Любое значение	-	*bottom*	*bottom*	*bottom*	*bottom*
C1	*Целое число*	*C2*	*Целое число*	*C1 xor C2*	*Целое число*
В остальных случаях				*top*	*top*

Для всех остальных инструкций передаточные функции ничего не делают. Результатом сбора значений данных для переменной является объединение всех прежних значений переменной.

Сходимость алгоритма обеспечивается путем ограничения количества вычисленных значений. Алгоритм завершается, если не было изменений после итерации или если для переменной было вычислено заданное количество значений.

2.2.2. Анализ помеченных данных

Анализ помеченных данных [74] отслеживает переменные (регистры и точки памяти), которые могут контролироваться пользовательским управляемым вводом. Данный анализ проводится после анализа значений и использует полученные значения для переменных. Алгоритм анализа помеченных данных основан на итеративном алгоритме потока данных. Определяется полурешетка, передаточные функции и начальные значения. Верхним элементом полурешетки является taint, а

нижним элементом – bottom. Начальным значениям всех переменных присваивается bottom. Исключением являются аргументы функции main, которым присваивается taint. Для всех инструкций REIL определяются передаточные функции для отслеживания помеченных данных.

Передаточные функции анализа памеченных данных для инструкций add t1,t2,t3, and t1,t2,t3, bsh t1,t2,t3, div t1,t2,t3, mod t1,t2,t3, mul t1,t2,t3, or t1,t2,t3, sub t1,t2,t3, xor t1,t2,t3

Метка для значения t1	***Метка для значения t2***	***Метка для значения t3 после инструкции***
not_taint	*not_taint*	*not_taint*
taint	*not_taint*	*taint*
not_taint	*taint*	*taint*

Передаточные функции анализа памеченных данных для инструкций bisz t1,t3, stm t1,t3

Метка для значения t1	***Метка для значения t3 после инструкции***
not_taint	*not_taint*
taint	*taint*

Передаточная функция анализа памеченных данных для инструкции ldm t1,t3

Метка для значения t1	***Метка для значения переменной, на которую указывает t1***	***Метка для значения t3 после инструкции***
taint	*not_taint*	*taint*
taint	*taint*	*taint*

not_taint	*taint*	*taint*
not_taint	*not_taint*	*not_taint*

Передаточная функция анализа памеченных данных для инструкции undef, t3

Метка для значения t3 после инструкции
not_taint

Особо нужно отметить передаточную функцию для инструкций jcc, которая использует аннотации argument_untrusted_annotation и return_untrusted_annotation.

Для инструкции jcc t1, t3 (выполняет условный переход в адрес t3, если t1 не равно 0) передаточная функция определяется следующим образом:

- *Если jcc является переходом на функцию и вызываемая функция имеет аннотацию argument_untrusted_annotation, то помечаются все фактические аргументы, для которых соответствующий формальный аргумент имеется в аннотации;*
- *Если jcc является переходом на функцию и вызываемая функция имеет аннотацию return_untrusted_annotation, то помечается переменная, в которой записывается возвращаемый результат вызываемой функции и аннотации;*
- *Если jcc является переходом на функцию и вызываемая функция имеет аннотацию copy_arguments_annotation, то для аргументов помечаются переменные, в которые копируются значения;*
- *В остальных случаях ничего не помечается.*

Результатом сбора значений и помеченных данных для переменной является not_taint, если все прежние значения not_taint, в противном случае – taint.

2.2.3. Анализ достигающих определений

Анализ достигающих определений ([75], раздел 9.2.4) для всех точек программы дает информацию обо всех переменных, определение которых достигло конкретной точки программы (т.е. не переопределялось). Анализ проводится после анализа значений и использует полученные значения для переменных, при этом он учитывает как регистры, так и точки памяти. Алгоритм анализа достигающих определений основан на итеративном алгоритме потока данных. Определяется полурешетка, передаточные функции и начальные значения. Передаточные функции реализованы следующим способом:

1. Вначале удаляются все определения, которые достигли точки до конкретной инструкции;
2. Потом нужно получить значение переменной, которую определяет конкретная инструкция, и сохранить ее.

Результатом сбора значений и помеченных данных для переменной является объединение всех достигающих определений. Итеративный алгоритм сходится, так как полурешетка монотонна [75].

2.2.4. Построение DEF-USE, USE-DEF цепочек

DEF-USE цепочка [75] – это структура данных, которая содержит информацию обо всех переменных касательно того, в какой инструкции конкретная переменная определилась и в каких используется. USE-DEF цепочка, наоборот, содержит информацию о том, в какой инструкции используется конкретная переменная и в каких она определилась.

Построение DEF-USE и USE-DEF цепочек основано на анализе достигающих определений. Алгоритм построения DEF-USE цепочек для всех инструкций функции вычисляет достигающие определения, и для переменных, которые используются в конкретной инструкции, находятся их определяющие инструкции из множества

достигающих определений. Алгоритм построения USE-DEF цепочек для всех инструкций вычисляет достигающие определения, и для переменной, которая определяется в конкретной инструкции, находятся их использующие инструкции из множества достигающих определений.

2.2.5. Трансформация удаления мертвого кода

Трансформация удаления мертвого кода [76] устраняет все инструкции, которые не влияют на работу функции. При трансформации ассемблера на REIL представление создается много инструкций, результат которых не используется. Удаление мертвого кода имеет несколько преимуществ: оно сокращает размер функции и позволяет другим видам анализа избегать обработки ненужных инструкций, тем самым сокращая время остальных видов анализа. После трансформации создается новая функция и новый граф потока управления. Трансформация реализована с использованием двухпроходного алгоритма Mark – Sweep [77].

2.2.6. Анализ динамической памяти

Анализ динамической памяти отслеживает выделение и удаление памяти в куче. Алгоритм рассматривает два класса значений: отслеживание точек памяти, которым соответствуют буферы в динамической памяти, и отслеживание переменных, которые указывают на буферы в динамической памяти. Использование двух классов значений позволяет более точно отследить выделение и удаление памяти, что реально влияет на качество работы детекторов дефектов использования памяти после освобождения и двойного освобождения. Для проведения анализа используется итеративный алгоритм потока данных.

Точки памяти в процессе анализа получают три значения:

1. bottom – точка памяти не указывает на динамическую память;

2. freed_memory – точка памяти указывает на динамическую память, которая освободилась;
3. not_freed_memory − точка памяти указывает на динамическую память, которая доступна.

Второй класс значений приписывается к переменным:

1. bottom – переменная не указывает на динамическую память;
2. dangling_pointer – переменная указывает на динамическую память, которая освободилась;
3. not_dangling_pointer − переменная указывает на динамическую память, которая доступна.

Передаточные функции для инструкций, кроме jcc, никаких операций не делают. Передаточная функция для инструкций jcc использует аннотации argument_free_annotation и allocate_memory_annotation. Она определяется следующим образом:

- Если *jcc* является переходом на функцию и вызываемая функция имеет аннотацию *argument_free_annotation,* то все фактические аргументы, которым соответствует формальный аргумент, участвующий в аннотации, получают значение *dangling_pointer*, а точки памяти, на которые указывают эти аргументы, получают значение *freed_memory;*
- Если *jcc* является переходом на функцию и вызываемая функция имеет аннотацию *allocate_memory_annotation,* то все фактические аргументы, которым соответствует формальный аргумент, участвующий в аннотации, получают значение *not_dangling_pointer,* а точки памяти, на которые указывают эти аргументы, получают значение *not_freed_memory.*

Результатом сбора значений для точек памяти является:

- *Bottom* (если все прежние значения являются *bottom);*

- *freed_memory* (если существует одно прежнее значение, равное *freed_memory*);
- *not_freed_memory* (в противном случае).

Результатом сбора значений для переменных, которые указывают на динамическую память, является:

- *Bottom* (если все прежние значения являются *bottom*);
- *dangling_pointer* (если существует одно прежнее значение, равное *dangling_pointer*);
- *not_dangling_pointer* (в противном случае).

2.2.7. Вычисление аннотаций функции

Последним этапом внутрипроцедурного анализа является вычисление аннотаций функций. Аннотации для функций стандартных библиотек получены из инструмента Svace [72]. Кроме функций стандартных библиотек, аннотации получают только те функции, у которых доступен код. Аннотации конкретных функций вычисляются на основе всех аннотаций всех вызывающихся функций.

Функция получает аннотацию argument_free_annotation, если функция удаляет память, на которую указывает ее аргумент. Для отслеживания аргумента конкретной функции используется анализ значений. Все аргументы анализируемой функции получают символьные значения – *(arg1), *(arg2), … ,*(arg_n), где n количество аргументов. Рассматриваются все вызывающие функции, которые имеют аннотацию argument_free_annotation, и если значение фактического аргумента одной из этих функций имеет значение *(reg+constants_array)+constant и reg равно arg_i, где $1 \leq i \leq n$, то к множеству аннотаций анализируемой функции добавляется argument_free_annotation и информация *(reg+constants_array)+constant. Эта часть сохраняется, потому что аргумент может быть структурой, и если только некоторые поля удаляются, то на другие поля это не влияет.

Схожий алгоритм используется при вычислении аннотаций argument_format_string_annotation, argument_buffer_overflow_annotation, argument_command_injection_annotation и argument_untrusted_annotation, только рассматриваются вызывающие функции, которые имеют аннотацию argument_format_string_annotation, argument_buffer_overflow_annotation, argument_command_injection_annotation, argument_untrusted_annotation соответственно.

При вычислении аннотации argument_dereference_annotation для функций используется вышеописанный алгоритм, а также в анализируемой функции рассматриваются все инструкции, которые разыменовывают аргумент (аргумент разыменовывается, если является первым операндом инструкции ldm или третьим операндом инструкции stm).

Аннотация return_untrusted_annotation для функций задается, если функция возвращает значение, которое может контролироваться пользовательским вводом. Для этого используется анализ помеченных данных. Рассматриваются все точки функции после инструкций возврата, и если в одной из этих точек значение возвращаемого значения функции является tainted, то к множеству аннотаций анализируемой функции добавляется аннотация return_untrusted_annotation.

Вычисление аннотации copy_arguments_annotation для функции происходит следующим образом. Если аргументы передаются функции, которая имеет аннотацию copy_arguments_annotation, то к множеству аннотаций анализируемой функции добавляется аннотация copy_arguments_annotation и отмечается, информация каких аргументов копируется и в другие аргументы.

2.2.8. Результаты

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 понять из таблицы, php имеет больший размер, чем clam, однако время работы анализа на этом проекте меньше. Такие результаты связаны с тем, что в проекте clam размеры функций в среднем намного больше, чем в php. Поэтому параллельная обработка функций в php срабатывает намного эффективней.

Проект	**Архитектура**	**Размер**	**Время работы анализа**
lepton 1.0	x86	5 МБ	19 мин. 21 сек.
clam 0.100.2	x86	18 МБ	4 ч. 20 мин.
dba 2.4.1	x86	312 КБ	1 мин. 30 сек.
pbs_server 2.4.8	x86	320 КБ	2 мин. 12 сек.
httpd 0.5.0	x86	6.4 МБ	6 мин. 31 сек.
iwconfig 26	x86	44 КБ	18 сек.
mkfs 1.1.12	x86	56 КБ	21 сек.
pswdb 2.4.1	x86	300 КБ	45 сек.
shar 4.2.1	x86	44 КБ	21 сек.
hsolinkcontrol 1.0.118	x86	28 КБ	2 сек.
mkfs 1.1.12	x64	56 КБ	18 сек.
libtorque 2.0.0	x64	892 КБ	47 сек.
alsa_in 1.1.3	x64	28 КБ	9 сек.

alsa_out 1.1.3	x64	28 КБ	9 сек.
pbs_server 2.4.8	x64	320 КБ	3 мин. 2 сек.
php 7.0.5	x64	29 МБ	3 ч. 12 мин.
jasper 1.900.1	arm	478 КБ	20 мин. 20 сек.
hsolinkcontrol 1.0.118	arm	22 КБ	3 сек.
openssl 1.0.1s	arm	624 КБ	6 мин. 30 сек.
htget 0.1	arm	27 КБ	31 сек.
acpid 2.0.9	arm	152 КБ	39 сек.

Таблица 1. Время работы анализа

Результаты работы трансформации удаления мертвого кода представлены в таблице 2.

Проект	**Архитектура**	**Общее количество инструкций**	**Количество инструкций после трансформации**	**Процент удаленного кода**
dba 2.4.1	x86	27265	19931	26.9%
httpd 0.5.0	x86	41030	35688	13%
jasper 1.900.1	x86	27265	19933	26.9%
alsa_in 1.1.3	x86	547	471	13.9%
alsa_out 1.1.3	x86	529	457	13.6%
hsolinkcontrol 1.0.118	x86	146	136	7%
hsolinkcontrol 1.0.118	x64	191	157	17.8%
mkfs 1.1.12	x64	2336	1798	23%

libtorque 2.0.0	x64	8879	7196	18.9%
alsa_in 1.1.3	x64	836	727	13%
alsa_out 1.1.3	x64	818	713	12.8%
jasper 1.900.1	arm	28402	23387	17.6%
openssl 1.0.1s	arm	41528	33246	20%
htget 0.1	arm	1053	897	14.8%
hsolinkcontrol 1.0.118	arm	180	154	14.4%
acpid 2.0.9	arm	2239	1857	17%

Таблица 2. Результаты работы трансформации удаления мертвого кода

Глава 3. Сравнение исполняемых файлов и анализ измененных участков кода

3.1 Поиск клонов исполняемого кода

В главе 1 определены типы клонов исполняемого кода. В этой главе предлагается модель инструмента поиска клонов исполняемого кода, который удовлетворяет следующим требованиям:

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

Существует ряд проблем в области точного обнаружения похожих и разных областей исполняемого кода:

- последовательности кодов операций могут оставаться идентичными, но распределение регистров может быть изменено в зависимости от доступности во время компиляции;
- если инструкции не зависят от других инструкций, они могут быть переупорядочены из-за оптимизаций.

Архитектура инструмента представлена на рисунке 6. Инструмент может анализировать исполняемые файлы архитектур x86, x86-64, ARM. Инструмент получает два аргумента: минимальное количество инструкций для клонов и минимальный процент сходства двух фрагментов.

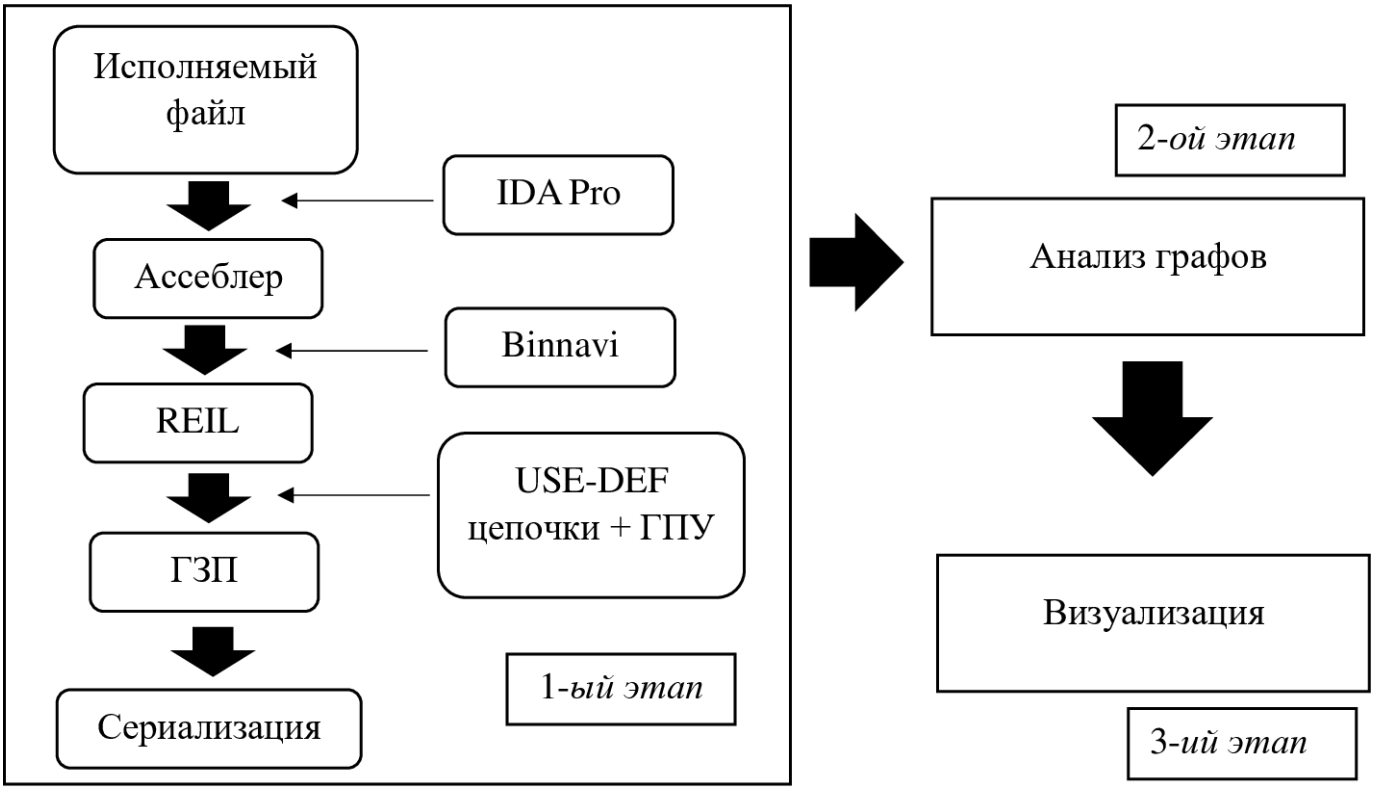

Рисунок 6 . Архитектура инструмента поиска клонов исполняемого кода.

Работа инструмента делится на три этапа:

Первый этап – генерация графов зависимостей программы (ГЗП) для функций (рисунок 7). Сначала исполняемый файл дизассемблируется с помощью IDA Pro и транслируется в REIL представление с помощью Binnavi. Генерация ГЗП происходит на основе графа потока управления, который восстанавливает IDA Pro дизассемблер, и разработанного анализа USE-DEF цепочек. Узлам ГЗП соответствуют инструкции REIL, а ребрам – зависимости по данным и по управлению. Весь процесс генерации ГЗП распараллелен.

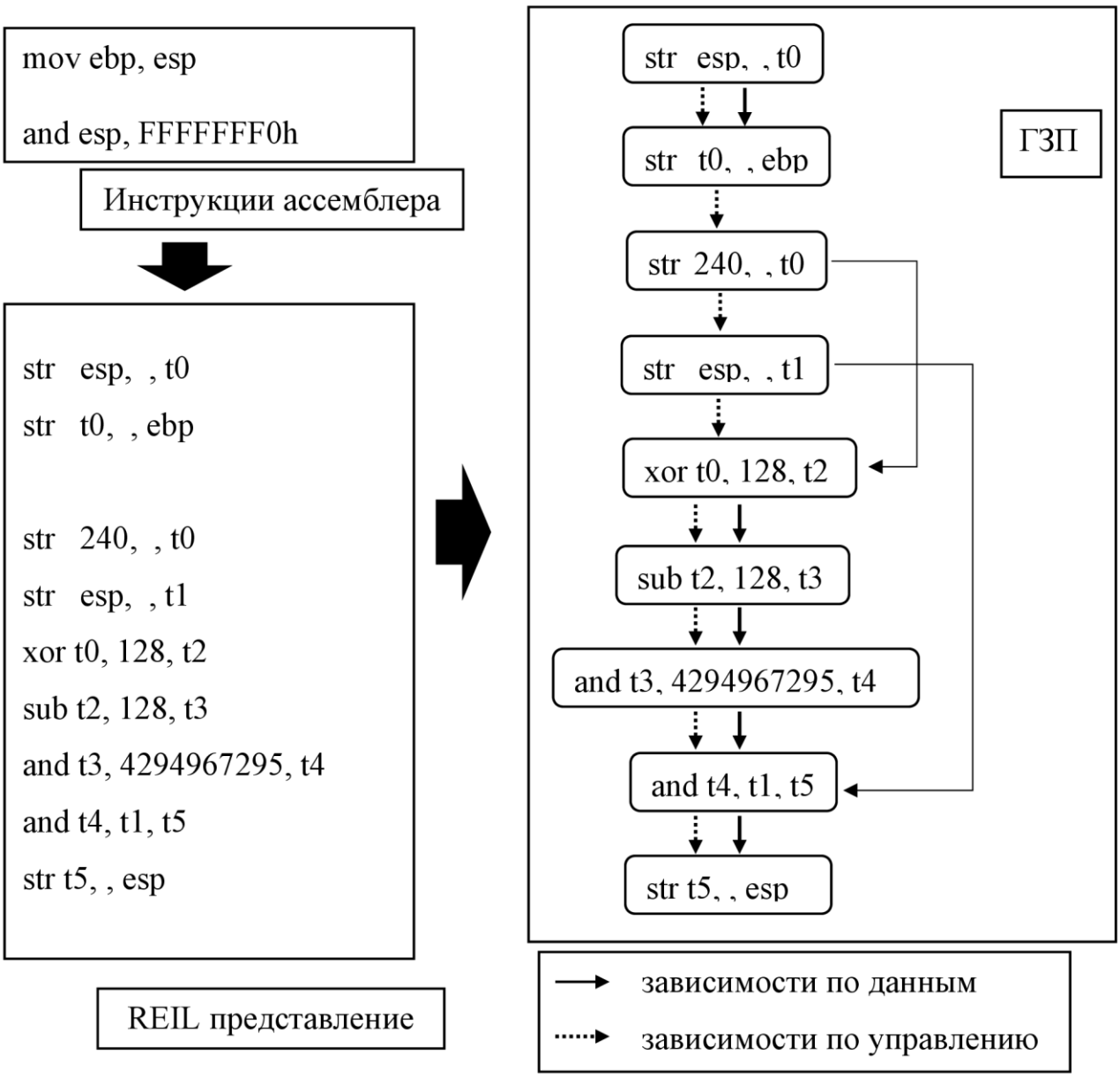

Рисунок 7. Пример ГЗП

В конце первого этапа все графы сериализуются. Промежуточная сериализация на первом этапе и десериализация на втором дает два преимущества. Первое: для некоторых проектов работа с созданными графами может потребовать большое количество оперативной памяти. Сериализация графов позволяет извлекать и сравнивать графы парами, тем самым экономя память. Второе: возможность

многократного применения второго этапа с различными значениями для аргументов (минимальное количество инструкций для клонов и минимальный процент сходства).

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 (Приложение 1) и, 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 (рисунок 8).

Преимуществом предлагаемого метода является то, что он основан на семантическом подходе, который позволяет достичь высокой точности и масштабируемости.

Функция	Клон функции
1) push ebp 2) mov ebp, esp 3) mov edx, ss: [ebp + arg 4] 4) mov eax, 10 5) cmp edx, eax 6) jbe 134513709 7) mov eax, -1 8) jmp 134513727 9) mov eax, ss: [ebp + arg 4] 10) mov edx, ss: [ebp + arg 0] 11) mov ds: [myArray + eax * 4], edx 12) mov eax, 0 13) pop ebp 14) retn []	1) push ebp 2) mov ebp, esp 3) mov edx, ss: [ebp + arg 4] 4) mov eax, 10 5) cmp edx, eax 6) jbe 134513715 cmp ss: [ebp + arg 4], byte 0 jns 134513715 7) mov eax, -1 8) jmp 134513733 9) mov eax, ss: [ebp + arg 4] 10) mov edx, ss: [ebp + arg 0] 11) mov ds: [myArray + eax * 4], edx 12) mov eax, 0 13) pop ebp 14) retn []

Рисунок 8. Визуализация клонов исполняемого кода

Результаты приведены в таблице 3 (исполняемые файлы получены компиляцией исходного кода с флагами -O0 –fno-inline) и в таблице 4 (исполняемые файлы получены компиляцией исходного кода с флагами –O3 –fno-inline), где минимальное количество инструкций для клонов равно 30 и минимальный процент сходства равен 90% (функции для каждого исполняемого файла сравнивались между собой).

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 3. Результаты поиска клонов кода между функциями одного исполняемого файла

Проект	Размер	Количество найденных клонов	Время работы инструмента
libxml2.so 2.9.3	1.8 МБ	437	1 мин. 25 сек.
libfreetype.so 2.6.5	740 КБ	73	36 сек.
openssl 1.1.0	672 КБ	71	26 сек.
d8 5.0	20 МБ	19453	17 мин. 26 сек.

Таблица 4. Результаты поиска клонов кода между функциями одного исполняемого файла

В таблице 5 приведены результаты сравнения функций двух последующих версий программы в целях оценки корректности работы инструмента. Инструмент проанализировал пары функций, которые имеют одно и то же имя в старой и в новой версиях программы. Поскольку одна и та же функция, скорее всего, изменится с несколькими инструкциями в новой версии, то они должны быть клонами. Если функция не изменяется, то они должны быть клонами первого или второго типа.

Имя программы	Версии		Размер (МБ)		Количество функций с одинаковыми именами	Время анализа	Количество найденных клонов (МП = 50%)	Количество найденных клонов (МП = 90%)
	старая	новая	старая	новая				
gcc	4.9.0	5.4.0	3.2	3.4	1041	50 сек.	956	662
git	2.6.0	2.9.5	9.4	9.8	3257	1 мин. 15 сек.	3098	2627
libxml2	2.9.2	2.9.3	5.4	5.4	2583	20 сек.	2577	2532
openssl	101f	101r	2.8	2.9	5393	2 мин. 26 сек.	5385	5358
php	7.0.5	7.0.6	29	29.1	8284	1 мин. 30 сек.	8282	8247
python	3.5	3.6	12	12	3864	1 мин. 36 сек.	3375	2081

Таблица 5. Результаты поиска клонов кода между функциями двух последовательных версий программы

3.2 Сравнение исполняемых файлов

Цель сравнения двух исполняемых файлов – выявление похожих фрагментов кода. Предлагаемый метод сначала сопоставляет множество функций первого файла с функциями второго файла. Сравнение исполняемых файлов используется для нахождения статически скомпонованных библиотек старых версий, которые имеют дефекты, для восстановления отладочной информации исполняемых файлов и т. д. Трудности, которые возникают при нахождении клонов исполняемого кода, также встречаются при сравнении исполняемых файлов:

- последовательности кодов операций могут оставаться идентичными, но распределение регистров может быть изменено в зависимости от доступности во время компиляции;

- если инструкции не зависят от других инструкций, они могут быть переупорядочены из-за оптимизаций;
- разные компиляторы или разные версии одного компилятора могут генерировать разный исполняемый код.

Предложенный метод решает вышеописанные проблемы. Инструмент может анализировать исполняемые файлы архитектур x86, x86-64, ARM. Архитектура предложенного метода состоит из двух этапов (рисунок 9).

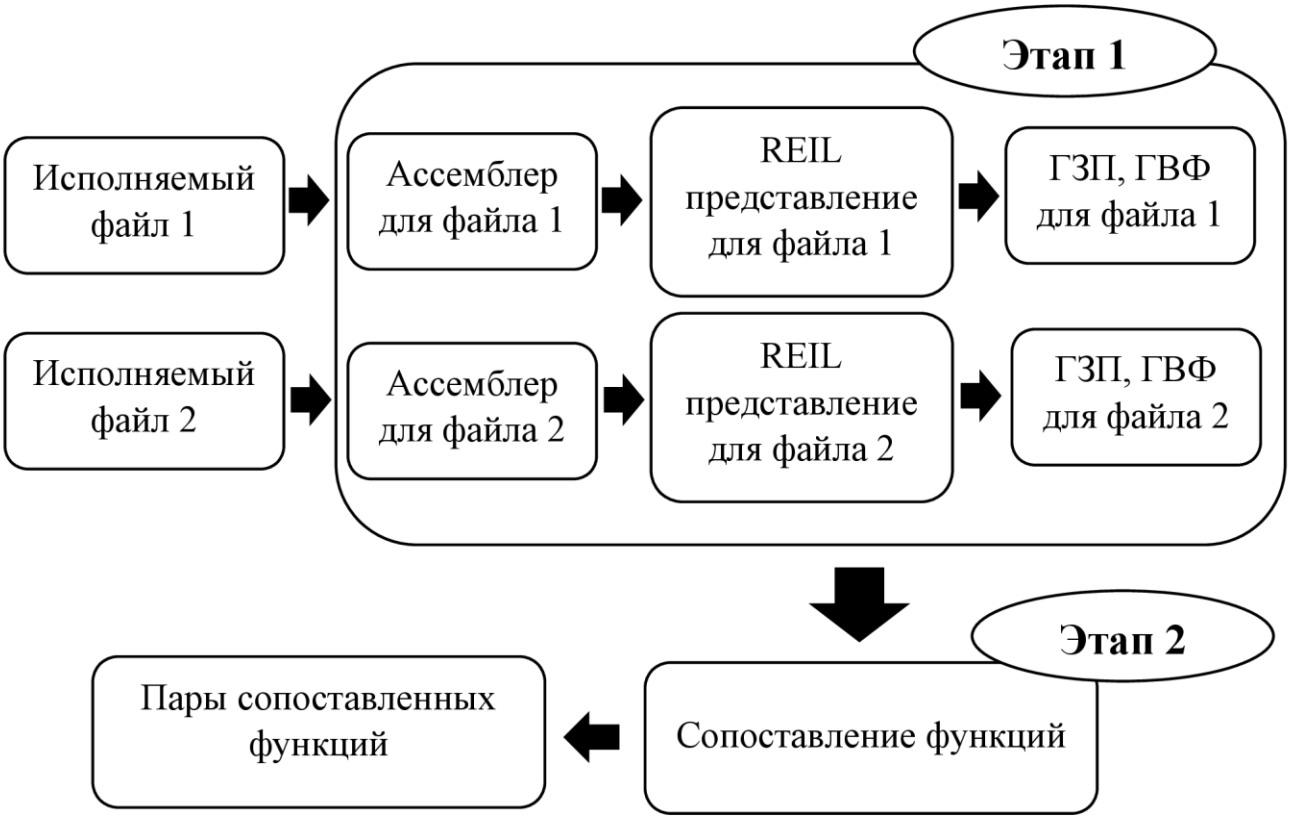

Рисунок 9. Архитектура инструмента сравнения исполняемых файлов

На первом этапе генерируются ГЗП и граф вызовов функций для каждого исполняемого файла. Генерация ГЗП происходит как описано в разделе 3.1. Графы вызовов восстанавливаются с помощью дизассемблера.

На втором этапе происходит сопоставление функций с учетом полученных ГЗП и графов вызовов. Для сопоставления функций разработано три алгоритма:

1. Алгоритм, основанный на эвристиках;
2. Алгоритм, основанный на нахождении наибольшего общего подграфа двух ГЗП;
3. Объединенный алгоритм (сначала часть функций сопоставляются на основе эвристик, потом на основе наибольшего общего подграфа двух ГЗП).

3.2.1 Алгоритм, основанный на эвристиках.

Сопоставление функций на основе метрик происходит следующим образом. При сопоставлении функций на основе эвристик 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 по определенной последовательности.

1. Сопоставление функций на основе хеширования ассемблерного кода (используется std::hash).
2. Сопоставление ребер графа вызовов (сопоставляются сразу две функции) на основе хеширования ГЗП по MD-индексу [50].
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 (используется std::hash).
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 5. Временная сложность алгоритма на основе эвристик – $O(k^2(v_1 log_2 v_1 + v_2 log_2 v_2 + \ldots + v_k log_2 v_k))$, где k – количество функций, а $v_1, v_2, \ldots, v_k$ – количество вершин графов ГПУ.

Доказательство. Для вычисления хешей ассемблерного кода для всех функций проводятся $O(\mathrm{v_1} + \mathrm{v_2} + \ldots + \mathrm{v_k})$ вычислений. Для сопоставления по этому хешу проводятся $O(k * (\mathrm{v_1} + \mathrm{v_2} + \ldots + \mathrm{v_k}))$ вычислений. Для вычисления хешей ребра графа вызовов функций на основе хеширования ГЗП по MD-индексу проводятся $O(\mathrm{v_1} + \mathrm{v_2} + \ldots + \mathrm{v_k})$ вычислений. Для сопоставления по этому хешу проводятся $O(k^2 * (\mathrm{v_1} + \mathrm{v_2} + \ldots + \mathrm{v_k}))$ вычислений. Для вычисления хешей ребра графа вызовов функций на основе хеширования MD-индекса соседних вершин начальной и конечной вершин ребра проводятся $O(\mathrm{v_1} + \mathrm{v_2} + \ldots + \mathrm{v_k})$ вычислений. Для сопоставления по этому хешу проводятся $O(k^2 * (\mathrm{v_1} + \mathrm{v_2} + \ldots + \mathrm{v_k}))$ вычислений. Для вычисления хешей функций на основе MD-индекса ГЗП проводятся $O(\mathrm{v_1} + \mathrm{v_2} + \ldots + \mathrm{v_k})$ вычислений. Для сопоставления по этому хешу проводятся $O(k * (\mathrm{v_1} + \mathrm{v_2} + \ldots + \mathrm{v_k}))$ вычислений. Для вычисления хешей функций, описанных в 5-м пункте, проводятся $O(\mathrm{v_1 log_2\, v_1} + \mathrm{v_2 log_2\, v_2} + \ldots + \mathrm{v_k log_2\, v_k})$ вычислений. Для сопоставления по этому хешу проводятся $O(k(\mathrm{v_1 log_2\, v_1} + \mathrm{v_2 log_2\, v_2} + \ldots + \mathrm{v_k log_2\, v_k}))$ вычислений. Для вычисления хешей функций описанный на 6-ом пункте, проводятся. $O(\mathrm{v_1} + \mathrm{v_2} +$

$... + v_k$) вычислений. Для сопоставления по этому хешу проводятся $O(k * (v_1 + v_2 + ... + v_k))$ вычислений.

Общая сложность будет:

$O(k * (v_1 + v_2 + ... + v_k)) + O(k(v_1 \log_2 v_1 + v_2 \log_2 v_2 + \ldots + v_k \log_2 v_k)) + O(k^2 * (v_1 + v_2 + ... + v_k)) = O(k(v_1 \log_2 v_1 + v_2 \log_2 v_2 + \ldots + v_k \log_2 v_k)) + O(k^2 * (v_1 + v_2 + ... + v_k))$, что не больше чем

$$O(k^2(v_1 \log_2 v_1 + v_2 \log_2 v_2 + \ldots + v_k \log_2 v_k))\blacksquare$$

Результаты работы алгоритма приведены в таблице 6.

Проекты	**Версии**		**Размеры (МБ)**		**Время работы (секунды)**	**Количество функций**		**Количество сопоставленных функций**
	старая	***новая***	***старая***	***новая***		***старая***	***новая***	
python	3.5.1	3.5.2	12	12	50	3944	3951	3849
python	3.5.2	3.6.3	12	12	43	3944	4107	2614
php	7.0.5	7.0.6	29	29	92	8287	8292	8078
php	7.0.6	7.0.24	29	29	86	8292	8342	5808
libxml2	2.9.2	2.9.3	5.4	5.4	22	2584	2603	2556
openssl	1.0.1r	1.0.1s	2.8	2.9	47	5395	5430	5376
openssl	1.0.1f	1.0.1s	2.2	2.9	34	5414	5430	5170
rsync	3.0.9	3.1.1	1.6	1.8	6	599	636	399
gcc	4.9.0	5.4.0	3.2	3.5	8	1094	1145	931
git	2.6.0	2.9.5	9.4	9.8	21	3335	3471	3153

Таблица 6. Результаты сопоставления функций на основе эвристик

3.2.2 Алгоритм, основанный на графах.

Алгоритм, основанный на графах, сопоставляет функции по следующим шагам:

1. Сопоставляются все пары вершин графов вызовов функций из первого и второго исполняемого файла. Если их ГЗП изоморфны (вызывается процедура Tracebasedslice, Приложение 1), 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 [55] 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 6. 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 $v_1, v_2, \dots, v_k$ − количество инструкций функций первого исполняемого файла и l − количество функций во втором исполняемом файле.

Доказательство. На шаге 1 проводятся $O\left(\sum_{i=0}^{k}\sum_{j=0}^{l} {v_1}^4 \, log_2 {v_1}^3\right) = O\left(l * \sum_{i=0}^{k} {v_1}^4 \, log_2 {v_1}^3\right)$ вычислений. На шагах 2-4 венгерский алгоритм вызывается не больше чем $k * l$ раз, а сложность алгоритма не превышает $O((max(k,l))^3)$. Сложность вызова процедуры Tracebasedslice в шагах 2-4 равна $O\left(l *\right.$

$\sum_{i=0}^{k} {v_1}^4 \, log_2 \, {v_1}^3\big)$. Общая сложность будет – $O\big(l * \sum_{i=0}^{k} {v_1}^4 \, log_2 \, {v_1}^3\big) + O((max(k,l))^3)$, что не больше чем $O\big(l * \sum_{i=0}^{k} {v_1}^4 \, log_2 \, {v_1}^3\big) + O((k+l)^3)$.

Результаты работы алгоритма приведены в таблице 7.

Проекты	**Версии**		**Размеры (МБ)**		**Время работы (секунды)**	**Количество функций**		**Количество сопоставленных функций**
	старая	***новая***	***старая***	***новая***		***старая***	***новая***	
python	3.5.1	3.5.2	12	12	122	3944	3951	3944
python	3.5.2	3.6.3	12	12	791	3944	4107	3943
php	7.0.5	7.0.6	29	29	120	8287	8292	8287
php	7.0.6	7.0.24	29	29	924	8292	8342	8292
libxml2	2.9.2	2.9.3	5.4	5.4	23	2584	2603	2584
openssl	1.0.1r	1.0.1s	2.8	2.9	53	5395	5430	5395
openssl	1.0.1f	1.0.1s	2.2	2.9	187	5414	5430	5414
rsync	3.0.9	3.1.1	1.6	1.8	125	599	636	599
gcc	4.9.0	5.4.0	3.2	3.5	82	1094	1145	1094
git	2.6.0	2.9.5	9.4	9.8	80	3335	3471	3334

Таблица 7. Результаты сопоставления функций на основе наибольшего общего подграфа пар ГЗП

3.2.3 Объединенный алгоритм

Как следует из таблиц 6 и 7, эвристический алгоритм сопоставляет функции быстрее, однако значительное количество функций не сопоставляется. 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 , потом сохраняются те сопоставления, которые выявляют клоны первого, второго или третьего типа, похожие более чем 50%. Далее осуществляет работа алгоритма, описанного в разделе 3.2.3. Результаты приведены в таблице 8. Исполняемые файлы были получены компиляцией исходного кода с помощью компиляторов gcc/g++ 7.2.

Проекты	**Версии**		**Размеры (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 8. Результаты объединенного алгоритма

В таблице 9 представлено сравнение с инструментом BinDiff [50]. BinDiff сначала сопоставляет функции по именам. Между тем, иногда эта информация недоступна. Для сравнения с BinDiff имена функций в исполняемых файлах были изменены. После этого был произведен запуск предложенного алгоритма и BinDiff. Далее посредством восстановления имен пояснялось, если две функции сопоставились и имеют одинаковые имена, то сопоставление считается правильным; если же сопоставились функции с различными именами, то неправильным.

Проекты	**Версии**		**Результаты BinDiff**		**Результаты объединенного алгоритма**		**Общая часть**
	старая	***новая***	***Сопоставленные функции***	***Неправильные срабатывания***	***Сопоставленные функции***	***Неправильные срабатывания***	
python	3.5.1	3.5.2	3931	36	3944	8	3895
python	3.5.2	3.6.3	3884	903	3943	489	2981
php	7.0.5	7.0.6	8287	16	8287	9	8271
php	7.0.6	7.0.24	8065	1224	8292	842	6840
libxml2	2.9.2	2.9.3	2581	4	2584	3	2577
openssl	1.0.1r	1.0.1s	5395	6	5395	16	5373
openssl	1.0.1f	1.0.1s	5413	108	5414	57	5305
rsync	3.0.9	3.1.1	569	148	599	79	420
gcc	4.9.0	5.4.0	1068	108	1094	79	960

git	2.6.0	2.9.5	3335	288	3334	168	3046

Таблица 9. Сравнения результатов с BinDiff

В таблицах 10, 11, 12, 13 представлены результаты сравнения исполняемых файлов, которые были получены с помощью компиляции одной версии программы разными версиями компиляторов или разными компиляторами, но одинаковыми флагами компиляции. Как результат выдается процент правильно сопоставленных функций.

Компиляторы		**Количество функций**		**Количество сопоставленных функций**	**Количество правильных срабатываний**	**Процент правильных срабатываний**
gcc 4.8	gcc 5.4	3712	3692	3692	3264	88.5%
gcc 4.8	gcc 7.2	3712	3711	3711	3686	99,3%
gcc 5.4	gcc 7.2	3692	3711	3692	3271	88.8%
clang 5.0	gcc 4.8	3713	3712	3712	3679	99.1%
clang 5.0	gcc 5.4	3713	3692	3692	3282	88.9%
clang 5.0	gcc 7.2	3713	3711	3711	3679	99.1%

Таблица 10. Сравнение исполняемых файлов python 2.7.10, полученных с помощью разных компиляторов или разных версий одного компилятора

Компиляторы		**Количество функций**		**Количество сопоставленных функций**	**Количество правильных срабатываний**	**Процент правильных срабатываний**
gcc 4.8	gcc 5.4	5436	5429	5429	4538	83.6%
gcc 4.8	gcc 7.2	5436	5415	5415	4719	87.1%
gcc 5.4	gcc 7.2	5429	5415	5415	4830	89.2%
clang 5.0	gcc 4.8	5045	5436	5045	3496	69.3%
clang 5.0	gcc 5.4	5045	5429	5045	3379	67%
clang 5.0	gcc 7.2	5045	5415	5045	3395	67.3%

Таблица 11. Сравнение исполняемых файлов openssl 1.0.1f, полученных с помощью разных компиляторов или разных версий одного компилятора

Компиляторы		**Количество функций**		**Количество сопоставленных функций**	**Количество правильных срабатываний**	**Процент правильных срабатываний**
gcc 4.8	gcc 5.4	10299	10124	10123	9370	92.6%
gcc 4.8	gcc 7.2	10299	10299	10299	10258	99.6%
gcc 5.4	gcc 7.2	10124	10299	10124	9397	92.8%

clang 5.0	gcc 4.8	10297	10299	10297	10266	99.7%
clang 5.0	gcc 5.4	10297	10124	10123	9358	92.4%
clang 5.0	gcc 7.2	10297	10299	10297	10265	99.7%

Таблица 12. Сравнение исполняемых файлов postgresql 9.5.3, полученных с помощью разных компиляторов или разных версий одного компилятора

Компиляторы		**Количество функций**		**Количество сопоставленных функций**	**Количество правильных срабатываний**	**Процент правильных срабатываний**
gcc 4.8	gcc 5.4	8866	8856	8856	7917	89.4%
gcc 4.8	gcc 7.2	8866	8865	8865	8861	99.6%
gcc 5.4	gcc 7.2	8856	8865	8856	7921	89.5%
clang 5.0	gcc 4.8	8864	8866	8864	8849	99.8%
clang 5.0	gcc 5.4	8864	8856	8856	7900	89.2%
clang 5.0	gcc 7.2	8864	8865	8864	8850	99.8%

Таблица 13. Сравнение исполняемых файлов php 7.1.10, полученных с помощью разных компиляторов или разных версий одного компилятора

В таблице 14 представлены результаты сравнения исполняемых файлов, которые были получены компиляцией openssl версии 1.0.1f, компилятором gcc версии

5.4 и разными флагами компиляции. Как результат выдается процент правильно сопоставленных функций. Из таблицы очевидно, что исполняемые файлы, которые получились оптимизациями –O0 и –O3, сопоставились с низким процентом правильных срабатываний. Причина этого − сильно различные инструкции.

Флаги оптимизаций		**Количество функций**		**Количество сопоставленных функций**	**Количество правильных срабатываний**	**Процент правильных срабатываний**
-O0	-O3	6134	5429	4939	888	18%
-O2	-O3	5598	5429	5429	4695	86.5%

Таблица 14. Сравнение исполняемых файлов, компилированных из openssl 1.0.1f разными флагами оптимизаций

3.3 Анализ характера изменений в новых версиях исполняемых файлов

Целью данного раздела является поиск измененных участков кода в новой версии программы, разбиение их на группы по изменению семантики и обнаружение всех неизмененных участков в новой версии. Инструмент также может использоваться другими инструментами для анализа только измененного фрагмента программы.

Сначала исполняемые файлы сравниваются по методу, описанному в разделе 3.2. В качестве результата сравнение предоставляет сопоставленные функции и инструкции функций, что позволяет понять, какой фрагмент кода добавился и какой удалился. Цель алгоритма – найти те добавления и удаления кода, которые могут повлиять на корректность программ (например, добавление проверки). Алгоритм определяет ряд изменений, которые не являются результатом рефакторинга.

Отслеживаются следующие изменения:

1. Добавился новый базовый блок в исполняемом коде. Предупреждение выдается, если все инструкции исполняемого кода базового блока в новой версии не сопоставились с инструкциями старой. В таком случае считается, что добавился новый базовый блок. Если в исходном коде добавилась проверка или цикл, то после трансляции в исполняемый код добавляется новый базовый блок.
2. Добавилась новая инструкция возврата из функции. Предупреждение выдается, если добавленный участок кода добавляет новый путь в конец функции. В таком случае считается, что добавилась инструкция возврата из функции.
3. Изменились аргументы функции. Предупреждение выдается, если от добавленных инструкций существует путь по потоку данных до инструкций, которые обрабатывают аргументы функции.
4. Изменилась вызывающая функция. Предупреждение выдается, если в сопоставленных инструкциях вызова функций в новой версии вызывается другая функция. Так как иногда имена функций могут изменяться в ходе рефакторинга, то алгоритм указывает изменение вызова функции на основе сопоставленных функций.
5. Добавилась новая инструкция выхода из цикла.
6. Добавилась новая инструкция для перехода в начало цикла.

В таблице 15 приведены результаты работы инструмента на некоторых тестах из DARPA Cyber Challenge [79]:

Проект	**Время работы (секунды)**	**Количество найденных изменений**	**Количество правильных срабатываний**	**Процент правильных срабатываний**
Carbonate	2	3	1	66%
CGC_Board	3	3	0	100%

Diary_Parser	1	6	1	84%
CGC_Planet_Markup_Language_Parser	3	14	1	93%
3D_Image_Toolkit	1	2	0	100%
ASCII_Content_Server	1	5	0	100%
basic_messaging	2	1	1	0%
CGC_Video_Format_Parser_and_Viewer	1	6	3	50%
Differ	1	3	1	67%
Dive_Logger	2	3	0	100%
Enslavednode_chat	1	3	1	67%
Estadio	2	2	0	100%

Таблица 15. Результаты на некоторых тестах из DARPA Cyber Challenge

В среднем доля правильных срабатываний на тестах DARPA Cyber Challenge составляет 77%.

В таблице 16 приведены результаты на тестовом наборе Corebench [80].

Проект	**Коммиты git**		**Количество найденных изменений**	**Количество правильных срабатываний**	**Процент правильных срабатываний**
	старая	**новая**			
find	244453b8	f7197f3a	4	2	50%
find	756b47b1	24e2271e	3	2	67%
find	a53a2585	ff248a20	6	5	83%

find	aca12907	b445af98	1	1	100%
grep	02f1daa1	074842d3	2	2	100%
grep	074842d3	2be0c659	6	6	100%
grep	0b027cfb	54d55bba	2	1	50%
grep	2665746b	7aa698d3	12	12	100%
grep	3a4b5484	db9d6340	3	1	33%
grep	90cc2ba2	3220317a	1	0	0%
grep	a1ea9506	5fa8c7c9	1	1	100%
grep	c1cb19fe	8f08d8e2	2	1	50%
grep	c2b9a4fe	6d952bee	6	6	100%
grep	e1d437a5	c1cb19fe	0	0	100%
grep	e99afd62	c96b0f2c	1	1	100%
grep	ee9c7844	3c3bdace	2	2	100%
make	036760a9	5acda13a	4	1	25%
make	0afbbf85	d65b267e	15	15	100%
make	3aa2aa7e	0a81d50d	9	5	55%
make	3cc351de	0afbbf85	90	90	100%
make	64055348	88f1bc8b	2	2	100%
make	87ac68fe	40a49f24	5	3	60%

make	97f106fa	fc644b4c	9	7	77%
make	c3188c6f	3b1432d8	4	3	75%
make	c3524b83	fd30db12	3	3	100%
make	d584d0c1	4923580e	5	4	80%

Таблица 16. Результаты на тестовом наборе Corebench

В среднем доля правильных срабатываний на тестовом наборе Corebench составляет 73.3%.

Глава 4. Детекторы дефектов

После внутрипроцедурного анализа, описанного в главе 2, запускаются детекторы дефектов для каждой анализируемой функции (с использованием полученных результатов и аннотаций функций). Реализованы детекторы использования памяти после освобождения (ИПО), двойного освобождения памяти (ДО), форматных строк (ФС), переполнения буфера (ПБ) и внедрения команд (ВК).

4.1 Поиск дефектов использования памяти после освобождения и двойного освобождения памяти

Поиск дефектов ИПО и ДО реализован двумя методами:

1. На основе ГЗС.
2. На основе аннотаций.

4.1.1 На основе ГЗС

Алгоритм поиска дефектов ИПО и ДО, основанный на ГЗС (рисунок 10), сначала строит части ГЗС, после чего анализ проводится на полученных частях. Использование кусков ГЗС является результатом того, что даже для нескольких килобайт исполняемых файлов целый ГЗС может быть слишком большим (более 20 гигабайт) для оперативной памяти. Для получения кусков ГЗС на графе вызовов строятся все пути, концом которых является некоторая библиотеческая 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

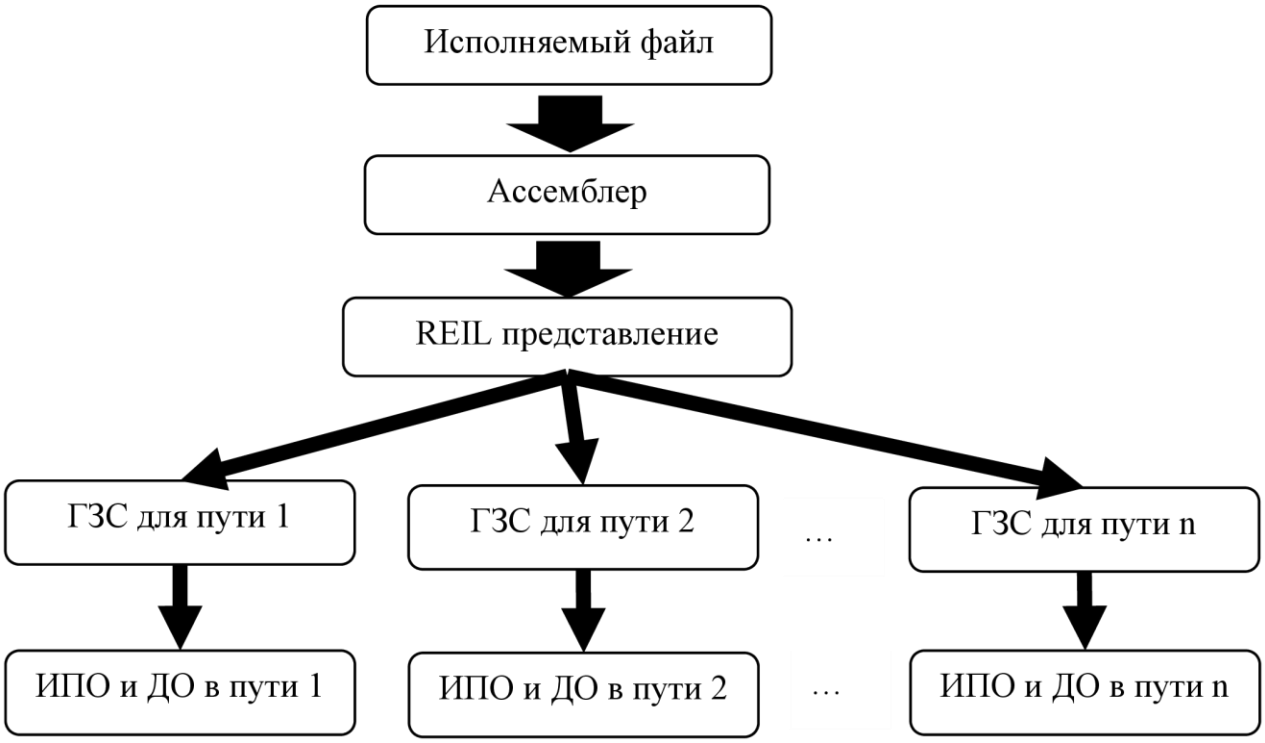

Рисунок 10. Архитектура алгоритма нахождения дефектов ИПО и ДО

управлению. Надо отметить, что ГЗС могут иметь много общих частей – функций. Таким образом, для каждой функции ГЗП генерируется один раз (как в разделе 3.1) и используется в алгоритме генерации ГЗС. Поскольку все полученные пути не зависят друг от друга, то весь процесс генерации параллелен. При нехватке оперативной памяти ГЗС сериализируются.

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 также предусмотрен простой анализ алиасов, который пытается найти все алиасы для данного указателя внутри функции, используя 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 17 представлены результаты анализа. Результаты всех тестов получены на сервере Intel Xeon 2.3ГГц с 20 ядрами. Представлена время анализа при использовании 20 ядер. Архитектура исполняемых файлов является x86.

Проект	**Архитектура**	**Размер**	**Время анализа**	**Количество ИПО и ДО**	**Процент правильных срабатываний**
jasper 1.900.1	x86	1 MB	166 сек.	3	100%
gifcolor 5.1.2	x86	50 KB	10 сек.	1	100%
libtiff 4.0.3	x86	1 MB	100 сек.	12	67%

gnome-nettool 3.8.1	x86	336 KB	48 сек.	1	100%
openslp 1.2.1	x86	700 KB	40 сек.	4	50%
libssh 0.5.2	x86	700 KB	99 сек.	19	79%
accel-pppd 1.10.0	x86	232 КБ	44 сек.	8	50%

Таблица 17. Результаты поиска ИПО и ДО на основе ГЗС

В таблице 18 представлены результаты сравнения с инструментом GUEB [28]. Использовались результаты GUEB из статьи.

Проект	**Время анализа GUEB**	**Найденные ИПО и ДО GUEB**	**Процент правильных срабатываний GUEB**	**Найденные ИПО и ДО**	**Процент правильных срабатываний**
gnome-nettool 3.8.1	16 сек.	4	25%	1	100%
gifcolor 5.1.2	21 сек.	15	6%	1	100%
jasper 1.900.1	4 мин. 23 сек.	255	1.2%	3	100%
accel-pppd 1.10.0	5 мин. 5 сек.	35	11.4%	8	50%

Таблица 18. Сравнение с GUEB

4.1.2 Поиск дефектов использования памяти после освобождения и двойного освобождения на основе аннотаций

После внутрипроцедурного анализа (Глава 2) запускаются детекторы нахождения ИПО и ДО.

Для поиска дефектов ИПО в анализе конкретной функции для всех точек программы проводятся следующие шаги:

1. Получаются все переменные, которые указывают на освобожденную память (они помечены как dangling_pointer, а соответствующие точки памяти − freed_memory).
2. Рассматриваются все вызовы функций, которые имеют аннотацию argument_dereference_annotation и получаются ее фактические аргументы.
3. Для всех фактических аргументов, если они помечены как dangling_pointer и указывают на память, помеченную как freed_memory, выдается дефект ИПО.

Рассмотрение двух классов значений помогает фильтровать ложные срабатывания. Например, в коде на5 рисунке 11 нет дефекта использования памяти после освобождения.

```
int p = malloc(10);
if (condition) {
   free(p);
   p = malloc(10);
}
*p = 1;
printf(*p);
```

Рисунок 11. Пример кода, в котором нет дефекта 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 19. Анализ проводился на машине core i5, 4 ядер, 16 ГБ ОЗУ.

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%
libssh 0.5.2	x86	632 КБ	3 мин.	14	57%
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%
libssh 0.5.2	x64	324 КБ	3 мин. 50 сек.	13	53%
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%
tiff2pdf 4.0.3	x64	192 КБ	27 сек.	1	100%
jasper 1.900.1	arm	490 КБ	2 мин. 34 сек.	1	100%
iwconfig 26	arm	53 КБ	20 сек.	2	100%
socat 1.7.1.3	arm	259 КБ	1 мин. 15 сек.	8	50%

Таблица 19. Результаты поиска ИПО и ДО

4.1.3 Выводы

Каждый из описанных методов имеет достоинства и недостатки по сравнению с другими. Метод, основанный на ГЗС, потребляет много памяти и вычислительной мощности, но может работать паралельно для всех путей, и если анализ проводится на многопоточном сервере, то предпочтительнее использовать первый метод. Вместе с тем, второй метод отличается большей точностью, так как использует значения переменных на всех путях выполнения. Например, проводится фильтрация ложных срабатываний (рисунок 11).

Как свидетельствуют данные в таблицах 18 и 19, средний процент истинных срабатываний превышает 80%. Однако в больших проектах (например, firefox) процент истинных срабатываний меньше (~20%). Причина этого заключается в том, что анализ не является чувствительным к путям выполнения. Инструменты также тестировались на тестовом наборе Juliet [81]. Процент истинных срабатываний составил 40%. В 5% тестов инструмент не нашел дефектов из-за неполного анализа алиасов.

4.2 Поиск дефектов форматной строки, переполнения буфера, внедрения команд

Детекторы поиска дефектов ФС, БО и ВК вызываются после завершения всех этапов внутрипроцедурного анализа. 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 проводятся 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 ее фактические аргументы.
3. Для всех фактических аргументов, если они помечены как taint, выдается дефект ВО.

Алгоритм поиска дефектов ВК отличается от алгоритма поиска БО только вторым пунктом: рассматриваются все вызовы функций, которые имеют аннотацию argument_command_injection_annotation.

Алгоритм поиска дефектов ФС отличается от алгоритма поиска БО только вторым пунктом: рассматриваются все вызовы функций, которые имеют аннотацию argument_format_string_annotation, и при восстановлении аргументов функции используется не только информация из IDA Pro, но и значения аргумента функции, который является форматной строкой. Так как IDA Pro не восстанавливает все аргументы функций с переменным числом аргументов, то если значение этого аргумента доступно, считается количество форматных спецификаторов (%s, %d, …) и выдается как количество аргументов.

Результаты инструмента представлены в таблице 20.

Проект	**Архитектура**	**Размер**	**Время анализа**	**Количество найденных дефектов**	**Процент правильных срабатываний**
dba 2.4.1	x86	312 КБ	1 мин. 40 сек.	12	50%
pbs_server 2.4.8	x86	320 КБ	2 мин. 22 сек.	5	80%
httpd 0.5.0	x86	6.4 МБ	6 мин. 51 сек.	22	90.9%
iwconfig 26	x86	44 КБ	24 сек.	3	100%
mkfs 1.1.12	x86	56 КБ	25 сек.	9	100%
pswdb 2.4.1	x86	300 КБ	55 сек.	9	33%
shar 4.2.1	x86	44 КБ	29 сек.	5	40%
hsolinkcontrol 1.0.118	x86	28 КБ	2 сек.	22	100%
alsa_in 1.1.3	x86	28 КБ	8 сек.	2	100%
alsa_out 1.1.3	x86	28 КБ	6 сек.	2	100%
htget 0.1	x86	28 КБ	8 сек.	12	100%
htget 0.1	x64	28 КБ	11 сек.	12	100%
mkfs 1.1.12	x64	56 КБ	19 сек.	7	100%
libtorque 2.0.0	x64	892 КБ	57 сек.	12	100%
alsa_in 1.1.3	x64	28 КБ	10 сек.	2	100%

alsa_out 1.1.3	x64	28 КБ	10 сек.	2	100%
pbs_server 2.4.8	x64	320 КБ	3м. 20с.	4	75%
acpid 2.0.9	arm	155 КБ	13 сек.	2	100%
hsolinkcontrol 1.0.118	arm	23 КБ	4 сек.	25	68%
htget 0.1	arm	28.1 КБ	9 сек.	12	100%
tipxd 1.1.0	arm	19.3 КБ	2 сек.	1	100%
socat 1.7.1.3	arm	259 КБ	48 сек.	1	100%

Таблица 20. Результаты поиска дефектов ФС, БО и ВК

В таблице 21 представлены результаты сравнения детектора с LoongChecker [22]. Использовались результаты LoongChecker из статьи.

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

Таблица 21. Результаты сравнения с LoongChecker

4.2.1 Выводы

Как можно увидеть в таблице 20, в среднем, доля истинных срабатываний составляет более 80%. Но, так же, как и при поиске ИПО и ДО, в больших проектах (например, firefox) процент истинных срабатываний меньше (~20%). Основная

причина заключается в том, что анализ не является чувствительным к путям выполнения, и неизвестны размеры буферов в стеке. Инструмент запускался на тестовом наборе Juliet [81]. Доля истинных срабатываний равна 100%, но в 15% тестов инструмент не нашел дефектов из-за неполного анализа алиасов.

4.3 Поиск неисправленных частей в новых версиях исполняемых файлов

Автоматический поиск характера изменений в новой версии исполняемого файла и поиск клонов исполняемого кода используются для поиска неисправленных частей в новой версии исполняемого файла. Изменения, описанные в разделе 3.3, рассматриваются как исправления в некотором фрагменте исполняемого кода, а старая версия фрагмента кода считается неисправленной. После нахождения исправлений алгоритм находит клоны неисправленного фрагмента в новой версии исполняемого файла. Такие клоны могут нуждаться в исправлении, о чем выдается предупреждение аналитику. Точное определение неисправленного фрагмента является трудной задачей в связи с неясностью следующих моментов:

- сколько инструкций взять до и после измененных инструкций;
- имеет ли влияние последовательность инструкций;
- нужно ли учитывать семантику.

В целях решения перечисленных задач 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 аналитиком задается 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 добавления базового блока вставляется предыдущий базовый блок), потом рассматриваются соседние базовые блоки по потоку управления. Количество базовых блоков ограничивается аналитиком. Потом проводится 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, построенное по следующему принципу: в множество вставляются инструкции базового блока, в котором выдается предупреждение (при предупреждении добавления базового блока вставляются инструкции предыдущего базового блока). Потом рассматриваются соседние базовые блоки по потоку данных. Далее проводится поиск полученного множества инструкций с учетом потока данных.

В таблице 22 приведены тесты, для которых в новых версиях были обнаружены клоны неисправленного фрагмента. Такие фрагменты были исправлены в следующих версиях.

Проект	**Коммит git**		**Имя функции с исправленной ошибкой**	**Имя функции с неисправленной ошибкой**
	старая версия	**новая версия**		
Tcpdump	B534e304	d3aae719	jupiter_monitor_print	jupiter_mlfr_print
Tcpdump	C2ef6938	50a44b6b	ikev1_nonce_print	1. ikev1_hash_print 2. ikev1_sig_print

				3. ikev1_ke_print 4. ikev1_vid_print
libosip	41fd94e7	1178ef7b	osip_message_set_accept	1.osip_message_set_proxy_authorization 2. osip_message_set_alert_info 3.osip_message_set_proxy_authenticate 4. osip_message_set_error_info 5. osip_message_set_contact 6.osip_message_set_accept 7.osip_message_set_accept_encoding 8.osip_message_set_call_info 9.osip_message_set_content_encoding 10.osip_message_set_content_disposition 11.osip_message_set_accept_language 12.osip_message_set_record_route 13.osip_message_set_route 14.osip_message_set_allow 15.osip_message_set_proxy_

				authentication_info
libosip	79240bdd	a54f15b8	osip_www_authenticate_init	1. sdp_connection_init 2. osip_authorization_init 3.osip_authentication_info_init
libosip	80a955e7	03fe3a1c	osip_negotiation_sdp_build_offer	__osip_negotiation_sdp_build_offer

Таблица 22. Результаты, которые показывают обнаружение клонов неисправленного фрагмента

Заключение

Данная диссертация посвящена разработке методов статического анализа исполняемых файлов для поиска дефектов. Дополнительной целью работы являлись разработка и реализация программных средств, осуществляющих совместное применение этих методов для объемных исполняемых файлов. По итогам проведения исследований получены следующие результаты:

1. Разработана архитектура статического анализа исполняемого кода, который является архитектурно независимым, масштабируемым и легко расширяемым.
2. Предложены и разработаны методы анализа значений и помеченных данных, которые позволяют проводить межпроцедурный, чувствительный к контексту, к потоку данных и к потоку управления анализ. Анализ проводится на основе аннотаций функций, что позволяет достичь масштабируемости.
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 и поиска неисправленных фрагментов в новой версии исполняемого файла.

   Для дальнейших исследований можно отметить следующие направления:

   - поддержка детекторов для поиска других типов дефектов;
   - поддержка символьного выполнения;
   - повышение точности разработанных алгоритмов посредством проведения чувствительного к путям выполнения анализа;
   - поиск функционально похожих фрагментов исполняемого кода.

Литература

[1] «CWE-416: Use After Free,» [В Интернете]. Available: https://cwe.mitre.org/data/definitions/416.html.

[2] «CWE-415: Double Free,» [В Интернете]. Available: https://cwe.mitre.org/data/definitions/415.html.

[3] «CWE-120: Buffer Copy without Checking Size of Input,» [В Интернете]. Available: https://cwe.mitre.org/data/definitions/120.html.

[4] «CWE-134: Use of Externally-Controlled Format String,» [В Интернете]. Available: https://cwe.mitre.org/data/definitions/134.html.

[5] «CWE-78: Improper Neutralization of Special Elements used in an OS Command,» [В Интернете]. Available: https://cwe.mitre.org/data/definitions/78.html.

[6] А. Асланян, Ш. Курмангалеев, В. Варданян, М. Арутюнян и С. Саргсян, «Платформенно-независимый и масштабируемый инструмент поиска клонов бинарного кода,» *Труды ИСП РАН,* т. 28, № 5, pp. 215-226, 2016.

[7] H. Aslanyan, A. Avetisyan, M. Arutunian, G. Keropyan, S. Kurmangaleev и V. Vardanyan, «Scalable Framework for Accurate Binary Code Comparison,» в *2017 Ivannikov ISPRAS Open Conference (ISPRAS)*, Moscow, 2017.

[8] А. Асланян, «Платформа межпроцедурного статического анализа исполняемого кода,» *Труды Института системного программирования РАН,* т. 30, № 5, pp. 89-100, 2018.

[9] H. K. Aslanyan, «Effective and Accurate Binary Clone Detection,» *Mathematical Problems of Computer Science,* т. 48, pp. 64-73, 2017.

[10] H. Aslanyan, S. Asryan, J. Hakobyan, V. Vardanyan, S. Sargsyan и S. Kurmangaleev, «Multiplatform Static Analysis Framework for Programs Defects Detection,» в *CSIT Conference 2017*, Yerevan, Armenia, 2017.

[11] G. S. Keropyan, V. G. Vardanyan, H. K. Aslanyan, S. F. Kurmangaleev и S. S. Gaissaryan, «Multiplatform Use-After-Free and Double-Free Detection in Binaries,» *Mathematical Problems of Computer Science,* т. 48, pp. 50-56, 2017.

[12] V. P. Ivannikov, A. A. Belevantsev, A. E. Borodin, V. N. Ignatiev, D. M. Zhurikhin и A. I. Avetisyan, «Static analyzer Svace for finding defects in a source program code,» *Programming and Computer Software*, т. 40, № 5, pp. 265-275, 2014.

[13] G. Balakrishnan и T. Reps, «WYSINWYX: What You See Is Not What You eXecute,» *ACM Transactions on Programming Languages and Systems* , т. 32, № 6, pp. 1-84, 2010.

[14] [В Интернете]. Available: https://www.hex-rays.com/products/ida.

[15] «IDA F.L.I.R.T. Technology,» Hex-Rays, 27 05 2015. [В Интернете]. Available: https://www.hex-rays.com/products/ida/tech/flirt/index.shtml.

[16] J. Ferrante, K. Ottenstein и J. Warren, «The program dependence graph and its use in,» *Trans. on Prog. Lang. and Syst. (TOPLAS)*, pp. 319-349, 1987.

[17] P. Cousot и R. Cousot, «Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points,» в *Principles of Programming Languages (POPL)*, 1977.

[18] J. Kinder, Static analysis of x86 executables. Ph.D. thesis, Technische Universitat Darmstadt, 2010.

[19] D. Brumley , I. Jager , T. Avgerinos и E. J. Schwartz , «A Binary Analysis Platform,» *Lecture Notes in Computer Science*, т. 6806, 2011.

[20] B. Rosen, M. N. Wegman и K. F. Zadeck, «Global value numbers and redundant computations,» в *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1988.

[21] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam и P. Saxena, «BitBlaze: A New Approach to Computer Security via,» в *In Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[22] S. Cheng, J. Yang, J. Wang, J. Wang и F. Jiang, «LoongChecker: Practical Summary-Based Semi-simulation to Detect Vulnerability in Binary Code,» в *10th International Conference on Trust, Security and Privacy in Computing and Communications*, Changsha, 2011.

[23] [В Интернете]. Available: https://www.zynamics.com/binnavi/manual/html/reil_language.htm.

[24] [В Интернете]. Available: https://www.zynamics.com/binnavi.html.

[25] V. Ganapathy, S. A. Seshia, S. Jha, T. W. Reps и R. E. Bryant, «Automatic discovery of API-level exploits,» в *27th International Conference on Software Engineering*, Saint Louis, MO, USA, 2005.

[26] M. Cova, V. Felmetsger, G. Banks и G. Vigna, «Static Detection of Vulnerabilities in x86 Executables,» в *22nd Annual Computer Security Applications Conference*, 2006.

[27] D. Dewey, B. Reaves и P. Traynor, «Uncovering Use-After-Free Conditions in Compiled Code,» в *0th International Conference on Availability, Reliability and Security*, Toulouse, 2015.

[28] J. Feist, L. Mounier и ML. Potet, «Statically detecting use after free on binary code,» *Journal of Computer Virology and Hacking Techniques*, т. 10, № 3, pp. 211-217, 2014.

[29] B. Baker, «On finding duplication and near-duplication in large software systems,» в *Proceedings of the 2nd Working Conference on Reverse Engineering*, 1995.

[30] C. K. Roy и J. R. Cordy, «An empirical study of function clones in open source software systems,» в *Proceedings of the 15th Working Conference on Reverse Engineering*, 2008.

[31] S. Ducasse, M. Rieger и S. Demeyer, «A language independent approach for detecting duplicated code,» в *Proceedings of the 15th International Conference on Software Maintenance*, 1999.

[32] T. Kamiya, S. Kusumoto и K. Inoue, «CCFinder: A multilinguistic tokenbased code clone detection system for large scale source code,» в *IEEE Transactions on Software Engineering*, 2002.

[33] I. Baxter, A. Yahin, L. Moura и M. Anna, «Clone detection using abstract syntax trees,» в *Proceedings of the 14th IEEE International Conference on Software Maintenance*, 1998.

[34] R. Tairas и J. Gray, «Phoenix-based clone detection using suffix trees,» в *Proceedings of the 44th Annual Southeast Regional Conference*, 2006.

[35] L. Jiang, G. Misherghi, Z. Su и S. Glondu, «DECKARD : Scalable and accurate tree-based detection of code clones,» в *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[36] S. Sargsyan, S. Kurmangaleev, A. Baloian и H. Aslanyan, «Scalable and Accurate Clones Detection Based on Metrics for Dependence Graph,» *Mathematical Problems of Computer Science,* т. 42, pp. 54-62, 2014.

[37] S. Sargsyan, S. Kurmangaleev, A. Belevantsev, H. Aslanyan и A. Baloian , «Scalable tool for code clone detection based on semantic analysis of program,» *Trudy. ISP RAS,* т. 1, pp. 39-50, 2015.

[38] J. Jang и D. Brumley, «Bitshred: Fast, scalable code reuse detection in binary code,» *CyLab,* p. 28, 2009.

[39] B.H. Bloom, «Space/time trade-offs in hash coding with allowable errors,» *Communications of the ACM,* pp. 422-426, 1970.

[40] A. Schulman, «Finding binary clones with opstrings function digests: Part III,» *Dr. Dobb's Journal,* 2005.

[41] R. Rivest, «The MD5 Message-Digest Algorithm,» *RFC Editor,* 1992.

[42] M.E. Karim, A. Walenstein, A. Lakhotia и L. Parida, «Malware phylogeny generation using permutations of code,» *Computer Virology,* pp. 13-23, 2005.

[43] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan и Z. Su, «Detecting code clones in binary executables,» в *Proceedings of the 18th International Symposium on Software Testing and Analysis*, 2009.

[44] A. Andoni и P. Indyk, «Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions,» в *47th Annual IEEE Symposium,* 2006.

[45] M. R. Farhadi, B. C. M. Fung, P. Charland и M. Debbabi, «BinClone: Detecting Code Clones in Malware,» в *2014 Eighth International Conference on*, San Francisco, CA, 2014.

[46] D. Bruschi, L. Martignoni и M. Monga, «Code normalization for self-mutating malware,» *IEEE Security & Privacy*, pp. 46-54, 2007.

[47] [В Интернете]. Available: http://boomerang.sourceforge.net/.

[48] P. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel и S. Zanero, «Identifying dormant functionality in malware programs,» в *Securityand Privacy (SP), 2010 IEEE Symposium on*, 2010.

[49] Z. Wang, K. Pierce и S. McFarling, «BMAT - A Binary Matching Tool for Stale Profile Propagation,» *The Journal of Instruction-Level Parallelism (JILP)*, т. 2, 2000.

[50] T. Dullien и R. Rolles, «Graph-based comparison of executable objects,» *Symposium sur la Securite des Technologies de l'Information et des Communications,* 2005.

[51] T. Dullien, E. Carrera, S. Eppler и S. Porst, «Automated Attacker,» в *RTO-MP- IST-091*, 2010.

[52] T. Dullien, E. Carrera, S. Eppler и S. Porst, «Automated attacker correlation for malicious code,» DTIC Document, 2010.

[53] I. Briones и A. Gomez, «Graphs, entropy and grid computing: Automatic comparison of malware,» в *Proceedings of Virus Bulletin International Conference*, 2008.

[54] M. Bourquin, A. King и Ed. Robbins, «BinSlayer: Accurate Comparison of Binary Executables,» в *2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, 2013.

[55] Harold W. Kuhn, «The Hungarian Method for the assignment problem,» *Naval Research Logistics Quarterly,* т. 2, pp. 83-97, 1955.

[56] J. Oh, «ExploitSpotting: Locating Vulnerabilities Out Of Vendor Patches Automatically,» 2010.

[57] M. Gheorghescu, «An automated virus classification system,» в *Proceedings of Virus Bulletin International Conference,* 2005.

[58] A. Sanfeliu и K.-S. Fu, «A distance measure between attributed relational graphs for pattern recognition,» *IEEE Transactions on Systems, Man and Cybernetics,* p. 353–363, 1983.

[59] G. Navarro, «A guided tour to approximate string matching,» *ACM Computing Surveys,* pp. 31-88, 2001.

[60] S. Cesare и Y. Xiang, «Classification of malware using structured control flow,» *Australasian Symposium on Parallel and Distributed Computing,* т. 107, pp. 61-70, 2010.

[61] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines и P. Narasimhan, «Binary Function Clustering Using Semantic Hashes,» *IEEE ICMLA,* 2012.

[62] A. Lakhotia, M. D. Preda и R. Giacobazzi, «Fast Location of Similar Code Fragments Using Semantic 'Juice',» 2013.

[63] D. Gao, M. K. Reiter и D. Song, «BinHunt: Automatically Finding Semantic Differences in Binary Programs,» в *Info. and Comm. Security*, 2008.

[64] J. Ming, M. Pan и D. Gao, « iBinHunt: Binary Hunting with Inter-procedural Control Flow,» в *Kwon T., Lee MK., Kwon D. (eds) Information Security and Cryptology – ICISC 2012*, 2012.

[65] J. Jiyong, A. Abeer и B. David, «ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions,» в *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[66] Q. W. a. Y. C. Z. Liu, «VFDETECT: A vulnerable code clone detection system based on vulnerability fingerprint,» в *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference*, 2017.

[67] S. Kim, S. Woo, H. Lee и H. Oh, «VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery,» в *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[68] B. Collins-Sussman, B. W. Fitzpatrick и C. M. Pilato, «Version Control with Subversion».

[69] [В Интернете]. Available: https://git-scm.com/.

[70] Z. Xu, B. Chen, M. Chandramohan, Y. Liu и F. Song, «SPAIN: Security Patch Analysis for Binaries towards Understanding the Pain and Pills,» в *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Buenos Aires, Argentina, 2017.

[71] R. E. Tarjan, «Depth-first search and linear graph algorithms,» в *12th Annual Symposium on Switching and Automata Theory (swat 1971)*.

[72] А. А. Белеванцев, Многоуровневый статический анализ исходного кода для обеспечения качества программ. Диссертация д.ф.-м.н., Москва, 2018.

[73] P. Cuoq, F. Kirchner, N.Kosmatov, V. Prevosto, J. Signoles и B. Yakobowski, «Frama-C—a software analysis perspective,» *SEFM*, pp. 233-247, 2012.

[74] D. E. Denning, «A lattice model for secure information flow,» *Communication of the ACM*, 1976.

[75] A. V. Aho, M. S. Lam, U. D. Jeffrey и R. Sethi, Compilers: Principles, Techniques, and Tools, 1986.

[76] A. V. Aho, R. Sethi и J. D. Ullman, «A formal approach to code optimization,» в *Proceedings of a symposium on Compiler optimization*, 1970.

[77] R. Cytron, «Efficiently computing static single assignment form and the control dependence graph,» *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pp. 451-490, 1991.

[78] «Статический анализатор Svace. Промышленный поиск критических ошибок в безопасном цикле разработки программ,» [В Интернете]. Available: http://www.ispras.ru/technologies/svace/.

[79] «DARPA Cyber Challenge,» [В Интернете]. Available: http://archive.darpa.mil/cybergrandchallenge/about.html.

[80] «Corebench,» [В Интернете]. Available: https://www.comp.nus.edu.sg/~release/corebench/.

[81] [В Интернете]. Available: https://samate.nist.gov/SRD/testsuite.php.

[82] S. C. Misra и V. C. Bhavsar, «Relationships between selected software measures and latent bug-density: Guidelines for improving quality,» в *International Conference on Computational Science and its Applications, ICCSA*, Monreal, Canada, 2003.

[83] H. J. Boehm, «Threads cannot be implemented as a library,» в *Prog. Lang. Design and Implementation (PLDI)*, 2005.

[84] «Klocwork,» [В Интернете]. Available: http://www.klocwork.com.

[85] «IBM AppScan,» [В Интернете]. Available: http://www-03.ibm.com/software/products/en/appscan.

[86] «HP Fortify,» [В Интернете]. Available: http://www8.hp.com/us/en/software-solutions/static- code-analysis- sast.

[87] «Coverity,» [В Интернете]. Available: http://www.coverity.com.

[88] «ГОСТ 19.102-77 Единая система программной документации, МЕЖГОСУДАРСТВЕННЫЙ СТАНДАРТ,» [В Интернете].

[89] «Security Development Lifecycle (SDL),» Microsoft, [В Интернете]. Available: https://www.microsoft.com/en-us/sdl.

Список таблиц

Список рисунков

Приложение А. Эвристический алгоритм нахождения наибольшего общего подграфа двух ГЗП

Алгоритм нахождения наибольшего общего подграфа двух ГЗП используется для поиска клонов исполняемого кода, сравнения двух исполняемых файлов и нахождения характера изменений в новой версии программ. Алгоритм называется *Tracebasedslice*. Он использует несколько процедур, которые определены ниже.

Пусть *G1(V1,E1), G2(V2,E2)* является ГЗП, где *V1, V2* – вершины, а *E1, E2* – ребра. Пусть *X1* и *Y1* − произвольные множества вершин *G1,* а *X2* и *Y2* − произвольные множества вершин *G2*. Две вершины могут сопоставиться, если их коды операций, количество предшественников и преемников равны. Обозначим |*X1* | количество элементов множества.

Определение 1. *getPredecessors(X1,Y1)* процедура возвращает вершины из *X1,* которые не входят в *Y1* и являются предшественниками для вершин *Y1.*

Определение 2. *getSuccessors(X1,Y1)* процедура возвращает вершины из *X1*, которые не входят в *Y1* и являются преемниками для вершин *Y1.*

Определение 3. *sortVertices(X1)* процедура сортирует вершины *X1* по их кодам операций, количеству предшественников, количеству преемников и адресу в исполняемом файле. Используется сортировка слиянием.

Определение 4. *makeCorrespondence (X1, Y1)* процедура возвращает пары вершин из отсортированных множеств *X1* и *Y1*, которые могут сопоставляться. Процедура учитывает коды операций, количество предшественников, количество преемников.

Определение 5. *makeOneCorrespondence(X1, Y1)* процедура получает в качестве входных наборов вершины, которые не имеют предшественников, и возвращает пары вершин, которые еще не сопоставлены, но могут быть сопоставлены.

Определение 6. Для всех $m \in X1$ и $n \in X2$ *checkPredecessors(X1,X2,m,n)* условие выполняется, если предшественники m из *X1* и предшественники n из *X2* имеют одинаковый набор кодов операций.

Определение 7. Для всех $m \in X1$ и $n \in x2$ *checkSuccessors(X1,X2,m,n)* условие выполняется, если преемники m из *X1* и преемники n из *X2* имеют одинаковый набор кодов операций.

Определение 8. *inducedSubGraph(X1, G1)* процедура возвращает порожденный подграф из *X1* в *G1*.

Теорема 7.

1. Временная сложность процедуры *getPredecessors(X1,Y1)* - $O(|X1| * |Y1| * |V1|)$.
2. Временная сложность процедуры *getSuccessors(X1,Y1)* - $O(|X1| * |Y1| * |V1|)$.
3. Временная сложность процедуры *sortVertices(X1)* - $O(|X1| \log_2 |X1|)$.
4. Временная сложность процедуры *makeCorrespondence(X1,Y1)* - $O(|X1| + |Y1|)$.
5. Временная сложность процедуры *makeOneCorrespondence (X,Y)* - $O(|X| * |Y|)$
6. Временная сложность процедуры *checkPredecessors(X1,X2,m,n)* - $O(|V1|^2 + |V2|^2 + |V1| * |V2|)$.
7. Временная сложность процедуры *checkSuccessors(X1,X2,m,n)* - $O(|V1|^2 + |V2|^2 + |V1| * |V2|)$.

8. Временная сложность процедуры *inducedSubGraph(X1, G1)* - $O(|X1^2| * |E1|$

Доказательство. Для доказательства пункта 1 достаточно увидеть, что процедура для всех элементов из *X1* проводит поиск в *Y1,* и если элемент не входит в *Y1*, то проводится поиск в предшественниках вершин из *Y1,* количество которых меньше, чем $|V1|$. Получается временная сложность процедуры *getPredecessors(X,Y)* равна $O(|X1| * |Y1| * |V1|)$. Доказательство пункта 2 эквивалентно доказательству пункта 1. Для всех элементов из *X1* процедура проводит поиск в *Y1,* и если элемент не входит в *Y1*, то проводится поиск в преемниках вершин из *Y1,* количество которых меньше, чем $|V1|$. Получается временная сложность процедуры *getSuccessors (X1,Y1)* ровна $O(|X1| * |Y1| * |V1|)$.

Пункты 3, 4 и 5 очевидны. В процедуре *checkPredecessors(X1,X2,m,n)* рассматриваются все предшественники *m,* которые находятся в *X1* (сложность будет $O(|V1| * |X1|)$) и все предшественники *n,* которые находятся в *X2* (сложность будет $O(|V2| * |X2|)$). Далее проверяется идентичность полученных множеств (сложность будет $O(|V1| * |V2|)$, так как множества могут проверяться размерами $|V|$). Общая сложность процедуры будет - $O(|V1| * |X1| + |V2| * |X2| + |V1| * |V2|) = O(|V1|^2 + |V2|^2 + |V1| * |V2|)$. Аналогичным образом считается сложность процедуры *checkSuccessors(X1,X2,m,n)*.

В процедуре *inducedSubGraph(X1, G1)* выбираются две любые вершины из *X1* (сложность будет $O(|X1| * |X1 - 1|/2)$) и проверяется, существует соединяющее их ребро или нет. Общая сложность будет $O(|X1| * (|X1| - 1) * \frac{|E1|}{2}) = O(|X1|^2 * |E1|)$.

■

Ниже представлен псевдокод процедуры *Tracebasedslice.* В качестве входных данных процедура получает пары ГЗП − *G1 (V1, E1), G2 (V2, E2),* минимальное количество вершин – *minumumLength* (меньший результат не возвращается), и

минимальный процент схожести – *minimumPercentage*. Затем процедура возвращает наибольший общий подграф двух ГЗП.

Процедура *Tracebasedslice (G1, G2, minumumLength, minimumPercentage)*

1. *if* $|V1|==0$ or $|V2| ==0$ *or* $|V1| <$ *minumumLength or* $|V2| <$ *minumumLength or* $(2*|V1|)/(|V1|+|V2|) <$ *minimumPercentage or* $(2*|V2|)/(|V1|+|V2|) <$ *minimumPercentage*
2. *return* $\emptyset$
3. *matchedNodes1* $\subseteq V1$, *matchedNodes2* $\subseteq V2$
4. *matchedNodes1* $\leftarrow \emptyset$, *matchedNodes2* $\leftarrow \emptyset$
5. *noPredecessor1*$\leftarrow$*{n*$\in$*V1:n hasn't predecessor}*
6. *noPredecessor2*$\leftarrow$*{n*$\in$*V2:n hasn't predecessor}*
7. *continueMatching* $\leftarrow$ *true*
8. *while (continueMatching)*
9. *continueMatching*$\leftarrow$*false*
10. *tempMatching*$\subseteq V1 \times V2$
11. *tempMatching*$\leftarrow \emptyset$
12. *sortedPredecessors1*$\leftarrow$ *sortVertices* (*getPredecessors (V1, matchedNodes1))*
13. *sortedPredecessors2*$\leftarrow$ *sortVertices (getPredecessors (V2, matchedNodes2))*
14. *tempMatching*$\leftarrow$*makeCorrespondence(sortedPredecessors1, sortedPredecessors2)*
15. *sortedSuccessors1*$\leftarrow$ *sortVertices* (*getSuccessors (V1, matchedNodes1))*
16. *sortedSuccessors2*$\leftarrow$ *sortVertices (getSuccessors (V2, matchedNodes2))*
17. *tempMatching*$\leftarrow$*tempMatching* $\cup$ *makeCorrespondence(sortedSuccessors1, sortedSuccessors2)*
18. *if tempMatching is not empty*

19. *continueMatching*←*true*
20. *else*
21. *tempMatching*←*makeOneCorrespondence (noPredecessor1, noPredecessor2)*
22. *for all (v1, v2)* ∈ *tempMatching*
23. *if checkPredecessors*(*matchedNodes1*, *matchedNodes2, v1, v2*) *and checkSuccessors(matchedNodes1*, *matchedNodes2, v1, v2)*
24. *matchedNodes1* ←*matchedNodes1* ∪ *{v1}*
25. *matchedNodes2*←*matchedNodes2* ∪ *{v2}*
26. *subgraph1*← *inducedSubGraph(matchedNodes1, G1)*
27. *subgraph2*← *inducedSubGraph(matchedNodes2, G2)*
28. *count*← *vertices count of subgraph1*
29. *if count* < *minumumLength*
30. *return* ∅
31. *if (2*count)/(*|*V1*|+|*V2*|)*< *minimumPercentage*
32. *return* ∅
33. *return (subgraph1, subgraph2)*

Теорема 8. Временная сложность процедуры *Tracebasedslice* − $O(|V1|^4) * log_2|V1|^3$.

Доказательство. В 3-й и 4-й строках проводятся вычисления сложностью $O(|V1| + |V2|)$. Обозначим $min(|V1|, |V2|)$ минимальное из чисел $|V1|, |V2|$. Цикл в строке 6 может повторятся максимум $min(|V1|, |V2|)$ раз. В строках 10 и 13 поводятся вычисления сложностью $O(|V1|^2 * |matchedNodes1| * \log_2(|V1|^2$ $|matchedNodes1|))$ согласно теоремам 5.1 и 5.3. В строках 11 и 14 поводятся

вычисления сложностью $O(|V2|^2 * |matchedNodes2| * \log_2(|V2|^2$ $|matchedNodes2|))$ согласно теоремам 5.2 и 5.3. В строках 12 и 15 – $O(|V1| + |V2|)$. В строках 16-19 – $O(|noPredecessor1| + |noPredecessor2|)$. В строках 20-23 – $O(min(|V1|, |V2|) * (|V1|^2 + |V2|^2 + |V1| * |V2|))$, а в строках 24-25 – $O(|V1|^2 * |E1| + |V2|^2 * |E2|)$. В остальных строках проводятся вычисления сложностью $O(1)$. Общая сложность процедуры будет:

$$O(|V1| + |V2|) + min(|V1|, |V2|) * (O(|V1|^2 * |matchedNodes1| * \log_2(|V1|^2 * |matchedNodes1|)) + O(|V2|^2 * |matchedNodes2| * \log_2(|V2|^2 * |matchedNodes2|)) + O(|V1| + |V2|) + O(|noPredecessor1| + |noPredecessor2|) + O(min(|V1|, |V2|) * (|V1|^2 + |V2|^2 + |V1| * |V2|))) + O(|V1|^2 * |E1| + |V2|^2 * |E2|) + O(1)$$

В наихудшем случае $O(|E1|) = O(|V1|^2)$ и $O(|E2|) = O(|V2|^2)$. Общая сложность будет:

$$O(|V1| + |V2|) + min(|V1|, |V2|) * (O(|V1|^2 * |matchedNodes1| * \log_2(|V1|^2 * |matchedNodes1|)) + O(|V2|^2 * |matchedNodes2| * \log_2(|V2|^2 * |matchedNodes2|)) + O(|V1| + |V2|) + O(|noPredecessor1| + |noPredecessor2|) + O(min(|V1|, |V2|) * (|V1|^2 + |V2|^2 + |V1| * |V2|))) + O(|V1|^4 + |V2|^4) + O(1)$$

Так как $|noPredecessor1| < |V1|$ и $|noPredecessor2| < |V2|$, то можно упростить:

$$O(|V1| + |V2|) + min(|V1|, |V2|) * (O(|V1|^2 * |matchedNodes1| * \log_2(|V1|^2 * |matchedNodes1|)) + O(|V2|^2 * |matchedNodes2| * \log_2(|V2|^2 * |matchedNodes2|)) + O(|V1| + |V2|) + O(min(|V1|, |V2|) * (|V1|^2 + |V2|^2 + |V1| * |V2|))) + O(|V1|^4 + |V2|^4) + O(1)$$

Так как $|matchedNodes1| < |V1|$ и $|matchedNodes□| < |V2|$ и $|V1|$ и $|V2|$ больше 0, то можно упростить :

$$min(|V1|, |V2|) * (O(|V1|^3 * \log_2(|V1|^3)) + O(|V2|^3 * \log_2(|V2|^3) + O(min(|V1|, |V2|) * (|V1|^2 + |V2|^2 + |V1| * |V2|))) + O(|V1|^4 + |V2|^4)$$

Так как *(2*|V1|)/(|V1|+|V2|)< minimumPercentage or (2*|V2|)/(|V1|+|V2|)< minimumPercentage,* при вычислении строк 3-33, то $O(V1) = O(V2)$*:*

$$O(|V1|) * (O(|V1|^3 * \log_2|V1|^3) + O(|V1|^3))) + O(|V1|^4) = O(|V1|^4) * \log_2|V1|^3 \blacksquare$$