

Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 22.Б15-мм

Разработка санитайзеров для динамического анализа кода с целью выявления уязвимостей

Лановая Александра Юрьевна

Отчёт по производственной практике
в форме «Производственное задание»

Научный руководитель:
к. ф.-м. н., доц. Е. К. Куликов

Консультант:
начальник группы разработки в области информационной безопасности ООО «СТЦ»,
М. Д. Шалагин

Санкт-Петербург
2025

Оглавление

Введение	3
1. Постановка задачи	6
2. Обзор	7
2.1. Рассматриваемые уязвимости	7
2.2. Подходы к поиску уязвимостей	8
2.3. Символьные предикаты безопасности	10
2.4. Структура анализатора	11
3. Реализация	14
3.1. Обнаружение целочисленных ошибок	14
3.2. Обнаружение деления на ноль	17
4. Тестирование и эксперимент	21
Заключение	24
Список литературы	25

Введение

Современное программное обеспечение становится все более сложным, что приводит к увеличению числа потенциальных уязвимостей. Ручное обнаружение этих уязвимостей является трудоемким и не всегда эффективным процессом. Поэтому автоматизированные методы анализа кода, такие как динамический анализ [16], приобретают все большую популярность. Динамический анализ позволяет выявлять ошибки во время выполнения программы, предоставляя ценную информацию о ее поведении. Одним из перспективных подходов в динамическом анализе является сочетание символьного-конкретного (concolic) исполнения [19, 25] и фаззинга [9], позволяющее достичь высокого уровня покрытия кода и обнаружить трудноуловимые дефекты [12]. Однако существующие инструменты динамического анализа не всегда обеспечивают достаточную точность и эффективность в обнаружении специфических классов уязвимостей [2].

Несмотря на развитие методов динамического анализа, остается актуальной проблема разработки специализированных инструментов для выявления конкретных типов уязвимостей с высокой точностью. Зачастую универсальные анализаторы генерируют большое количество ложных срабатываний [8], что затрудняет процесс анализа и требует дополнительных ресурсов на проверку [1]. Недостаточная точность анализа может привести к тому, что реальные уязвимости останутся незамеченными.

В рамках данной работы рассматривается проблема повышения точности и эффективности динамического анализа кода с помощью специализированных инструментов. Разрабатываемый анализатор предназначен для поиска уязвимостей в коде и сочетает метод concolic-исполнения с фаззингом для генерации тестовых сценариев [10, 17]. Анализатор поддерживает две архитектуры: x86 и ARM64. В качестве символьного движка используется Triton¹. Сгенерированные сценарии анализируются с помощью специально разработанных санитайзеров, что позволяет

¹<https://github.com/JonathanSalwan/Triton/tree/master> (дата обращения: 2025-01-01)

с высокой точностью выявлять уязвимости и минимизировать количество ложных срабатываний [6]. Санитайзеры, как инструменты анализа, изначально появились в ответ на необходимость автоматизированного выявления ошибок времени выполнения, таких как переполнения буфера, утечки памяти или некорректные арифметические операции.

Инструмент уже обладает функциональностью для обнаружения ошибок памяти, включая CWE-121: Stack-based Buffer Overflow², CWE-122: Heap-based Buffer Overflow³, CWE-124: Buffer Underwrite⁴, CWE-126: Buffer Over-read⁵, CWE-127: Buffer Overwrite⁶. Однако для расширения возможностей по выявлению других критически важных уязвимостей необходимо разработать два новых санитайзера.

Один из них предназначен для детектирования целочисленных ошибок, таких как переполнение (Integer Overflow), недополнение (Integer Underflow), неожиданное расширение знака (Unexpected Sign Extension) и ошибки преобразования между знаковыми и беззнаковыми типами (Signed to Unsigned Conversion Error). Второй санитайзер сосредоточен на обнаружении ошибок деления на ноль (Division by Zero).

Эти уязвимости выбраны из-за их высокой значимости для безопасности и надёжности программного обеспечения. Деление на ноль может вызвать аварийное завершение программы, отказ в обслуживании или другие непредсказуемые последствия. Ошибки с целыми числами, в частности переполнение, активно эксплуатируются злоумышленниками для обхода проверок и выполнения произвольного кода. Таким образом, выявление данных ошибок на этапе разработки существенно снижает риск атак, таких как эскалация привилегий, нарушение целостности данных и компрометация системы.

Данная работа посвящена разработке двух санитайзеров для динамического анализатора кода, которые позволят более точно и эффективно выявлять критические уязвимости, улучшая тем самым безопас-

²<https://cwe.mitre.org/data/definitions/121.html> (дата обращения: 2025-01-01)

³<https://cwe.mitre.org/data/definitions/122.html> (дата обращения: 2025-01-01)

⁴<https://cwe.mitre.org/data/definitions/124.html> (дата обращения: 2025-01-01)

⁵<https://cwe.mitre.org/data/definitions/126.html> (дата обращения: 2025-01-01)

⁶<https://cwe.mitre.org/data/definitions/127.html> (дата обращения: 2025-01-01)

ность разрабатываемого программного обеспечения.

1. Постановка задачи

Цель работы заключается в разработке санитайзеров, интегрированных в динамический анализатор, для точного и эффективного выявления классов уязвимостей:

- CWE-369: Divide by Zero.⁷
- CWE-190: Integer Overflow.⁸
- CWE-191: Integer Underflow.⁹
- CWE-194: Unexpected Sign Extension.¹⁰
- CWE-195: Signed to Unsigned Conversion Error.¹¹

Для её выполнения были поставлены следующие задачи:

1. Рассмотреть уже существующие подходы поиска уязвимостей;
2. Реализовать санитайзеры для поиска уязвимостей CWE-369 и CWE-190ых;
3. Интегрировать санитайзеры в существующий анализатор кода, учитывая поддержку нескольких архитектур;
4. Протестировать и собрать статистику работы санитайзеров на Juliet Test Suite¹², который представляет собой набор разнообразных тестовых случаев для оценки инструментов анализа;

⁷<https://cwe.mitre.org/data/definitions/369.html> (дата обращения: 2024-12-12)

⁸<https://cwe.mitre.org/data/definitions/190.html> (дата обращения: 2024-12-12)

⁹<https://cwe.mitre.org/data/definitions/191.html> (дата обращения: 2024-12-12)

¹⁰<https://cwe.mitre.org/data/definitions/194.html> (дата обращения: 2024-12-12)

¹¹<https://cwe.mitre.org/data/definitions/195.html> (дата обращения: 2024-12-12)

¹²<https://github.com/ispras/juliet-dynamic> (дата обращения: 2024-12-12)

2. Обзор

2.1. Рассматриваемые уязвимости

Начнем с описания выбранных CWE (Common Weakness Enumeration) и их влияния в реальных приложениях. Рассмотрим основные уязвимости: их суть и примеры случаев, где такие проблемы стали критическими.

CWE-369: Divide by Zero

Эта уязвимость возникает, когда программа пытается выполнить деление на ноль, что может привести к критическим последствиям, таким как исключения, сбой системы или утечки данных. Проблема особенно опасна в системах реального времени, таких как авиационные или космические системы, где каждый сбой может привести к серьезным последствиям. Пример: В 1997 году на американском крейсере USS Yorktown, оснащенный системой управления Aegis, произошел серьезный сбой [7] в системе управления данными, вызванный ошибкой деления на ноль. Программа попыталась обработать запрос, в котором одно из входных значений оказалось равным нулю. В результате операционная система вышла из строя, нарушив работу всей системы корабля, включая двигатели. Судно оказалось обездвиженным посреди океана на несколько часов, пока инженеры вручную не перезагрузили систему.

CWE-190: Integer Overflow

Integer Overflow происходит, когда результат арифметической операции превышает диапазон допустимых значений для типа данных. Эта уязвимость обычно может привести к неопределенному поведению и, как следствие, к сбоям. Когда вычисленный результат используется для выделения ресурсов, эта уязвимость может привести к выделению слишком большого (или слишком малого) количества ресурсов, что потенциально может вызвать сбой, если продукт запрашивает больше ресурсов, чем может быть предоставлено. Это часто используется в атаках переполнения буфера. Пример: в 1996 году ракета-носитель

Ariane 5 взорвалась через 40 секунд после запуска из-за переполнения 16-битного целого числа [13] в программе навигации.

CWE-191: Integer Underflow

Integer Underflow — это ситуация, когда результат арифметической операции становится меньше минимально возможного значения. Это может привести к неправильным вычислениям и обходу проверок безопасности. Например, ошибки в обработке индексов массивов при отрицательных значениях могут позволить злоумышленникам читать или записывать данные за пределами разрешенных областей памяти.

CWE-194: Unexpected Sign Extension

Эта уязвимость возникает, когда данные, сохраненные в меньшем типе данных, расширяются неожиданным образом, что приводит к изменению их значения. Если старший бит числа, интерпретируемого как знаковый, равен 1, то при расширении этот бит будет скопирован во все старшие биты большего числа. Это может использоваться в атаке типа «sign mismatch», где злоумышленник манипулирует знаковыми данными для выполнения вредоносных действий.

CWE-195: Signed to Unsigned Conversion Error

Проблема возникает, когда знаковое значение преобразуется в беззнаковое, что приводит к некорректной интерпретации данных. Пример: в 2008 году ошибка в драйвере Windows [24] позволила злоумышленникам получать доступ к запрещенным областям памяти, приводя к эскалации привилегий.

2.2. Подходы к поиску уязвимостей

Обнаружение уязвимостей в приложениях является ключевым этапом обеспечения их безопасности. Существуют различные подходы к анализу кода и поведения программ, позволяющие выявлять уязвимости на разных стадиях разработки и эксплуатации. Одним из таких методов является статический анализ безопасности кода (SAST) [14]. Этот подход предполагает исследование исходного или скомпилированного кода без его выполнения. Инструменты SAST сканируют текст

программы, выявляя уязвимости на основе заданных правил или паттернов. Они также могут применять символичный анализ, чтобы моделировать выполнение кода и исследовать все возможные пути его выполнения. Метод SAST хорошо интегрируется в процесс разработки и позволяет находить проблемы на ранних этапах, что снижает стоимость их устранения. Однако SAST часто генерирует большое количество ложных срабатываний и не способен выявлять уязвимости, которые проявляются только во время выполнения программы [23].

Другим популярным методом является динамический анализ безопасности приложений (DAST) [18], который анализирует поведение программы в процессе ее выполнения. Этот подход предполагает эмуляцию атак, где специально сформированные запросы направляются к приложению для проверки его реакции на некорректные данные. Ответы приложения анализируются, чтобы выявить уязвимости, такие как SQL-инъекции или межсайтовый скриптинг (XSS). DAST не требует доступа к исходному коду и подходит для тестирования готовых приложений в реальных условиях. Однако данный метод имеет ограниченную глубину анализа [5], зависящую от охвата тестов, и сложности в локализации проблем в коде.

Интерактивный анализ безопасности приложений (IAST) [20] представляет собой комбинированный подход, объединяющий преимущества SAST и DAST. Инструменты IAST работают во время выполнения приложения и собирают информацию о его поведении изнутри. Для этого в приложение внедряются агенты, которые мониторят выполнение кода и анализируют движение данных через систему. Это позволяет более точно идентифицировать уязвимости, такие как утечка данных или неправильная обработка пользовательского ввода. Подход IAST обеспечивает высокую точность и снижает количество ложных срабатываний, однако его настройка и интеграция могут быть сложными.

Отдельного упоминания заслуживает метод фаззинга, или фазз-тестирования [10], который основан на подаче случайных или полуслучайных данных на вход программы. Приложение выполняется с этими данными, а результаты анализа позволяют выявить нестандарт-

ные проблемы, которые трудно обнаружить другими методами. Фаззинг особенно эффективен для поиска ошибок, связанных с обработкой исключительных ситуаций. Его слабая сторона заключается в сложности интерпретации результатов и зависимости от качества генерации входных данных.

Помимо перечисленных методов, существуют и другие подходы. Например, анализ состава программного обеспечения (SCA) [4], который проверяет внешние библиотеки и зависимости, а также другие компоненты, входящие в состав приложения (например, описанные в SBOM), на наличие известных уязвимостей. Технологии защиты приложений в режиме реального времени (RASP) [20], которые работают внутри приложения и предотвращают атаки на этапе их реализации. Дополнительно, в последние годы активно развиваются подходы, основанные на использовании больших языковых моделей (LLM, Large Language Models) [3]. LLM, такие как GPT или BERT, позволяют анализировать исходный код, документацию и текстовые описания для выявления уязвимостей. Эти модели применяются для автоматической генерации тестовых данных. LLM способны учитывать контекст программы, что делает их полезными для обнаружения сложных и нестандартных проблем.

Все перечисленные подходы дополняют друг друга, создавая комплексную систему обнаружения и устранения уязвимостей, что делает приложения более защищенными на всех этапах их жизненного цикла. Интеграция традиционных методов с передовыми технологиями, такими как LLM, открывает новые горизонты для автоматизации анализа кода и повышения уровня его безопасности.

2.3. Символьные предикаты безопасности

Предикаты безопасности [22] представляют собой дополнительные условия к предикату пути, которые помогают выявлять ошибки в программном коде. Для создания такого предиката во время символьного анализа программы изучаются определенные инструкции и потенциа-

но опасные участки кода. Если выявляется опасный участок, формируется предикат безопасности, который включает в себя систему уравнений и неравенств, описывающих символьные переменные и постоянные значения. После этого предикат безопасности объединяется с предикатом пути, образуя систему уравнений и неравенств, передаваемую на вход SMT-решателю [15]. Решение данной системы позволяет определить набор входных данных, обеспечивающих прохождение программы по исследуемому пути и проявление конкретной ошибки. Таким образом, выполнение программы с этими входными данными позволяет воспроизвести обнаруженные ошибки.

2.4. Структура анализатора

Анализатор представляет собой эмулятор, который принимает на вход бинарные файлы. Такой подход обеспечивает независимость инструмента от исходного кода программы и языка программирования. Однако в текущей реализации анализатор ориентирован на программы, написанные на C/C++, так как обертки реализованы только для функций из стандартной библиотеки `libc`. На выходе анализатор формирует отчет, содержащий потенциально уязвимые места программы и соответствующие входные данные, которые позволяют эксплуатировать найденные уязвимости.

Основой работы анализатора является эмуляция выполнения программы на уровне инструкций ассемблера. Эмулятор декодирует бинарный файл в инструкции и последовательно исполняет их. При встрече инструкции перехода (например, `jmp`) происходит ветвление, что позволяет анализатору исследовать различные пути выполнения. В текущей реализации поддерживаются две архитектуры: ARM64 и x86.

Ключевой принцип работы анализатора основан на совмещении `concolic` исполнения и фаззинга. В архитектуре анализатора выделяются три основные сущности – **Executor**, **Mutator** и **Fuzzer**. Действия этих сущностей организованы в цикле:

1. **Executor** запускает эмуляцию с некоторым входным значением

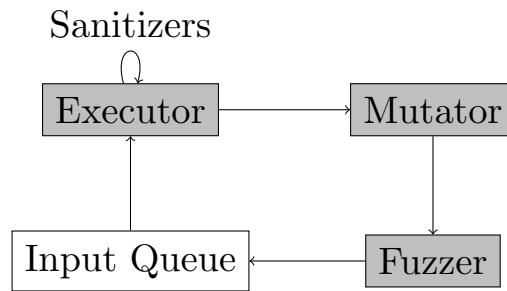


Рис. 1: Схема взаимодействия компонентов анализатора

из очереди (в первом цикле используется predetermined значение). В процессе эмуляции **Executor** выполняет инструкции программы и параллельно запускает санитайзеры для проверки на наличие уязвимостей. Возможны три сценария:

- **Тривиальная проверка:** Если санитайзер обнаруживает потенциальную проблему с помощью простой проверки при конкретном значении входных данных, предикат безопасности для символьной проверки не составляется, а текущее значение, с которым выполнялась эмуляция, сохраняется для дальнейшего анализа и воспроизведения эксплоита.
 - **Символьная проверка:** Если тривиальная проверка не выявила проблем, формируется предикат безопасности, описывающий условия возникновения уязвимости. Этот предикат передается SMT-решателю. Если решатель находит решение (SAT), это означает, что существует входной набор данных, приводящий к уязвимости. Это значение сохраняется для дальнейшего анализа и воспроизведения эксплоита.
 - **Ни одна из проверок не сработала:** Если решатель возвращает UNSAT, это означает, что в данном контексте уязвимость отсутствует.
2. **Mutator** генерирует новые входные данные на основе текущего значения и информации о ветвлении программы.
 3. **Fuzzer** управляет процессом фаззинга, добавляет сгенерированные **Mutator**'ом данные в очередь на исполнение **Executor**'ом и

отслеживает покрытие, достигнутое этими данными, принимая решения по улучшению покрытия кода.

3. Реализация

3.1. Обнаружение целочисленных ошибок

Рассматриваемые целочисленные ошибки находятся по одному и тому же шаблону, отличаются только сами предикаты безопасности, поэтому для их обнаружения применён единый санитайзер. Рассмотрим общий принцип его работы на примере ошибки CWE-190: Integer Overflow.

Целочисленное переполнение — это распространённая ошибка программирования, которая может приводить к непредсказуемому поведению программы и возникновению уязвимостей в системе. Она возникает, когда результат арифметической операции превышает диапазон значений, который может быть представлен используемым типом данных. Для решения данной проблемы был разработан подход к обнаружению целочисленных переполнений в бинарном коде с использованием динамического анализатора кода. Основной акцент сделан на анализе критических участков программы, чтобы минимизировать избыточный анализ и повысить эффективность.

Цель метода состоит в обнаружении целочисленных переполнений только в тех участках программы, где они могут привести к критическим ошибкам. Для этого выделяются два ключевых понятия:

- Источник (source) — инструкция, где может произойти переполнение (например, сложение, умножение или битовый сдвиг).
- Сток (sink) — место в коде, где использование результата переполнения может привести к ошибкам (например, аргументы функций, адреса памяти, условные переходы).

Предикат безопасности для источника проверяется только тогда, когда найден сток ошибки, использующий этот источник.

Метод реализует следующий алгоритм:

1. Выделение источников переполнений:

- Анализируются арифметические инструкции.
- Проверяется, являются ли операнды символьными (переменными, а не фиксированными значениями).
- Строятся предикаты безопасности для проверки знакового и беззнакового переполнения с использованием соответствующих флагов процессора (CF и OF).

2. Проверка использования результата переполнения в стоке:

- Сопоставляются абстрактные синтаксические деревья (AST) источника и стока. Если дерево источника является поддеревом дерева стока, это указывает на использование переполненного значения.

3. Определение знаковости операций:

- С помощью обратного слайсинга (анализ предшествующих инструкций, влияющих на текущую) анализируется предыстория выполнения программы, чтобы определить, является ли операция знаковой или беззнаковой.

Определение знаковости арифметических операций является важной частью метода, так как тип операции (знаковая или беззнаковая) влияет на вероятность переполнения. Для этого используется алгоритм обратного слайсинга [21], описанный ниже:

Вход: sink — узел AST (абстрактного синтаксического дерева), содержащий переполненное значение, call stack — стек вызовов на момент ошибки в стоке, path predicate — предикат пути (ограничения пути выполнения до стока).

Выход: True, если операция знаковая, False, если операция беззнаковая, None, если знаковость невозможно определить.

Шаги алгоритма:

1. Инициализировать множество переменных: $\text{vars} \leftarrow \text{used variables}(\text{sink})$ — переменные, связанные с узлом sink.

2. Выполнить итеративный анализ предикатов пути в обратном порядке. Для каждого предиката pr из $reversed(path\ predicate)$:

- Проверить, используют ли переменные из $vars$ текущий предикат, то есть существует ли пересечение между $vars$ и переменными, используемыми в pr : $if\ vars \cap used\ variables(pr) \neq \emptyset$
- Если пересечение найдено:
 - (a) Извлечь инструкцию, соответствующую предикату: $inst \leftarrow instruction(pr)$.
 - (b) Определить место вызова (call site) функции, содержащей $inst$: $s \leftarrow callsite(function(inst))$.
 - (c) Если $inst$ является инструкцией ветвления (branch) и её место вызова присутствует в стеке вызовов ($s \in call\ stack$), то:
 - Если инструкция знаковая ($s/jns/jg/jge/jl/jle$), вернуть True.
 - Если инструкция беззнаковая ($ja/jae/jb/jbe$), вернуть False.

3. Если знаковость остаётся неоднозначной (например, инструкция jz), вернуть None.

Некоторые стандартные функции требуют дополнительного анализа из-за их критичности для безопасности:

- Функции выделения памяти ($malloc$, $calloc$):
 - Для этих функций создаются дополнительные предикаты безопасности, чтобы проверить, не привело ли целочисленное переполнение к выделению некорректного объёма памяти. Например, переполненное значение должно быть меньше, чем ожидаемый размер памяти для текущего выполнения, но при этом не равным нулю. Это позволяет избежать выделения некорректного объёма памяти, который может быть

успешно выделен, если переполненное значение окажется в пределах допустимого диапазона.

- Если переполненное значение больше доступной памяти, функция возвращает NULL, что предотвращает ошибки, связанные с выделением.
- Функции копирования (memcpy, strncpy):
 - Для таких функций проверяется, превышает ли переполненное значение допустимый размер копирования, который используется в текущем выполнении программы.
 - Если переполненное значение больше допустимого размера, это может привести к выходу за пределы буфера, что критично для обеспечения безопасности программы.

Для операций с типами данных, размер которых меньше sizeof(int), учитывается автоматическое расширение типов. Оригинальная инструкция не может вызвать переполнение напрямую, поэтому создаются дополнительные предикаты безопасности, которые учитывают корректный размер операндов.

Данный метод для обнаружения целочисленных переполнений в бинарном коде представляет собой эффективный подход, фокусирующийся на критических участках программы. Выделение источников и стоков, использование обратного слайсинга для определения знаковости операций, а также анализ функций стандартной библиотеки позволяют минимизировать избыточные проверки и концентрироваться на потенциально опасных местах. Этот подход обеспечивает баланс между точностью анализа и его производительностью, что делает его полезным инструментом для разработки безопасного программного обеспечения.

3.2. Обнаружение деления на ноль

Деление на ноль является одной из самых критических ошибок, способных привести к неожиданному поведению программы или её аварий-

ному завершению. Для предотвращения этой ошибки в процессе исполнения используется подход, который заключается в построении предикатов безопасности. Каждый раз при выполнении инструкции деления осуществляется проверка, не равен ли делитель нулю. Эта проверка позволяет выявить потенциально опасные участки кода и предотвратить их использование в реальных условиях.

Для целых чисел деление на ноль является тривиальной ошибкой, которую можно обнаружить при помощи проверки значения делителя. Реализуется два подхода проверки:

- **Тривиальная проверка:** В момент выполнения инструкции деления проверяется, равен ли делитель нулю. Если делитель равен нулю, уязвимость считается обнаруженной.
- **Символьная проверка:** Если тривиальная проверка не сработала и делитель является символьным, то в момент выполнения инструкции деления строится предикат безопасности, выражающий условие равенства делителя нулю. Если этот предикат выполним (т.е. существует набор входных данных, при котором делитель равен нулю), уязвимость считается обнаруженной.

Дополнительно учитываются особенности в обработке инструкций деления в разных архитектурах:

- В x86 для инструкций `div` и `idiv` делитель указывается как первый операнд.
- В ARM для инструкций `SDIV` и `UDIV` делитель указывается как третий операнд.

Для чисел с плавающей точкой проблема деления на ноль имеет свои особенности. Хотя в стандарте IEEE 754 деление на ноль не вызывает аппаратных исключений, может возникнуть неопределенное поведение при приведении результата к целочисленному типу после деления на ноль. Примером является код на листинге 1.

```

void CWE369_Divide_by_Zero__float_fscanf_01_bad()
{
    float data;
    /* Initialize data */
    data = 0.0F;
    /* POTENTIAL FLAW: Use a value input
    from the console using fscanf() */
    fscanf (stdin, "%f", &data);
    {
        /* POTENTIAL FLAW: Possibly divide by zero */
        int result = (int)(100.0 / data);
        printIntLine(result);
    }
}

```

Листинг 1: Пример кода из Juliet Test Suite

Для обнаружения подобных ошибок используются следующие методы проверки:

- **Тривиальная проверка:** В момент выполнения инструкции деления проверяется, меньше ли абсолютное значение делителя $\epsilon = 10^{-6}$. Если это так, уязвимость считается обнаруженной.
- **Символьная проверка:** Если тривиальная проверка не сработала и делитель является символьным, то в момент выполнения инструкции деления строится предикат безопасности, выражающий условие того, что абсолютное значение делителя меньше $\epsilon = 10^{-6}$. Если этот предикат выполним (т.е. существует набор входных данных, при котором абсолютное значение делителя меньше $\epsilon = 10^{-6}$), уязвимость считается обнаруженной.

Важно обнаружить не только сам факт деления, но и проверить, является ли следующая инструкция приведением результата к целочисленному типу. Здесь используется похожий подход с источником и стоком, как в обнаружении целочисленного переполнения, но гораздо

проще. Предикат безопасности составляется на инструкции деления и решается на следующей инструкции, если она является инструкцией приведения числа с плавающей точкой к целочисленному типу. То есть в данном случае источником является инструкция деления, а стоком — следующая за делением инструкция приведения.

Дополнительно учитываются особенности в обработке инструкций деления в разных архитектурах:

- В x86 для деления используется инструкция `divss` (или `divsd`), где делитель указывается вторым операндом, для приведения числа с плавающей точкой к целому типу используется инструкция `cvtss2si` или `cvtsd2si` (преобразование `float` в `int`).
- В ARM для деления используется инструкция `FDIV`, где делитель указывается как третий операнд, для приведения числа с плавающей точкой к целому типу используется инструкция `FCVTZS` (конвертация `float` в целое с округлением к нулю).

Предложенный подход к обнаружению деления на ноль позволяет эффективно выявлять потенциальные ошибки в коде как для целых чисел, так и для чисел с плавающей точкой. Учет архитектурных особенностей и использование предикатов безопасности обеспечивает надежность анализа и помогает предотвратить возникновение критических ошибок в программном обеспечении.

4. Тестирование и эксперимент

Для тестирования и сбора статистики использовалась Juliet Test Suite — коллекция тестовых случаев, разработанная для проверки возможностей инструментов анализа безопасности программного обеспечения.

Juliet Test Suite был выбран для тестирования разработанных санитайзеров благодаря его статусу общепризнанного стандарта в области анализа безопасности программного обеспечения. Этот тестовый набор включает тысячи сценариев, покрывающих широкий спектр уязвимостей, описанных в базе CWE. Juliet обеспечивает воспроизводимость экспериментов, что позволяет объективно сравнивать результаты различных инструментов. А также предоставляет удобные возможности для оценки точности и производительности анализа.

Juliet Test Suite использовался многими исследовательскими группами и компаниями для оценки различных инструментов статического и динамического анализа безопасности. Например, в исследованиях, проведенных в области проверки инструментов безопасности, таких как Clang Static Analyzer [11] и других популярных решениях, показано, что Juliet обеспечивает высокую степень покрытия уязвимостей и является надежным инструментом для оценки эффективности анализаторов. Среди известных проектов, использующих Juliet, можно выделить SANS Institute, MITRE Corporation, а также ряд крупных компаний в области кибербезопасности, которые используют Juliet для тестирования своих собственных инструментов.

Для оценки был использован адаптированный вариант Juliet Test Suite [22], предназначенный для инструментов динамического анализа. Тестовые случаи организованы по 118 различным типам уязвимостей (CWE) и представляют собой фрагменты кода, которые можно скомпилировать и запустить. Каждый тест ориентирован на один конкретный тип уязвимости, хотя может содержать и другие, не связанные с ней, дефекты.

Каждый исходный файл Juliet содержит два теста: положительный

и отрицательный. Положительный тест содержит ошибку, в то время как отрицательный обрабатывает возможные уязвимости с помощью дополнительных проверок. Эти тесты реализуются в функциях `_bad` и `_good` соответственно. Существуют также «bad-only» тесты, содержащие только ошибочные конструкции, но в данной работе они пропущены.

Система тестирования оценивает количество истинно положительных (TP) и истинно отрицательных (TN) срабатываний санитайзеров, далее вычисляется показатель истинно положительных срабатываний (TPR) = TP / P , где P – количество положительных тестов, и показатель истинно отрицательных случаев (TNR) = TN / N , где N – количество отрицательных тестов, а также точность (ACC) = $(TP + TN) / (P + N)$, где количество положительных и отрицательных тестов одинаково ($P = N$).

Результаты тестирования разработанных санитайзеров на основе Juliet Test Suite, организованные по различным типам уязвимостей (CWE) представлены в таблице 1.

	$P = N$	TNR	TPR	ACC
CWE-190: Integer Overflow	1848	98,63%	93,54%	96,09%
CWE-191: Integer Underflow	1740	98,67%	93,49%	96,08%
CWE-194: Unexpected Sign Extension	548	100%	100%	100%
CWE-195: Signed to Unsigned Conversation Error	548	99,94%	100%	99,97%
CWE-369: Divide by Zero	582	100%	100%	100%

Таблица 1: Таблица с результатами эксперимента

Ошибки false positive и false negative в CWE190/CWE191 возникают из-за того, что тесты содержат арифметику больших чисел, для которых нужен отдельный предикат. На 32-битной архитектуре тип `int64_t` выполняет арифметические операции с большими числами, которые не поддерживаются предикатами на данный момент. Другая проблема возникает из-за вычитания `int64_t` на 32-битной архитектуре: некоторые оптимизации компилятора заменяют вычитание `int64_t` на сложение, например, заменяя «`sub eax, 1`» на «`add eax, 0xffffffff`», что также пока что не поддерживается. Также есть некорректные результаты в тестах,

содержащих возведение в квадрат типа short, что требует дальнейшего исследования.

Несмотря на некоторые несовершенства в некоторых типах уязвимостей, результаты тестирования санитайзеров показывают, что их эффективность близка к идеальной. Например, показатели TPR и TNR для большинства типов уязвимостей в наборе тестов (например, CWE-194, CWE-369) приближаются к 100%. Это указывает на то, что санитайзеры высокоэффективны в обнаружении соответствующих уязвимостей и минимизации ложных срабатываний.

Заключение

В ходе данной работы основная цель была достигнута — разработаны санитайзеры, они интегрированы в динамический анализатор для точного и эффективного выявления классов уязвимостей:

- CWE-369: Divide by Zero
- CWE-190: Integer Overflow
- CWE-191: Integer Underflow
- CWE-194: Unexpected Sign Extension
- CWE-195: Signed to Unsigned Conversion Error

Результаты:

1. Рассмотрены уже существующие подходы поиска уязвимостей;
2. Реализованы санитайзеры для поиска уязвимостей CWE-369 и CWE-190ых;
3. Санитайзеры интегрированы в существующий анализатор кода, учтена поддержка нескольких архитектур;
4. Санитайзеры протестированы, собрана статистика их работа на Juliet Test Suite;

Код находится под договором о неразглашении, поэтому не может быть предоставлен.

Список литературы

- [1] Automatic exploit generation / Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert et al. // Communications of the ACM. — 2014. — Vol. 57, no. 2. — P. 74–84.
- [2] Banabic Radu, Candea George, Guerraoui Rachid. [Automated vulnerability discovery](#) // 2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W). — 2011. — P. 188–193.
- [3] Comparison of static application security testing tools and large language models for repo-level vulnerability detection / Xin Zhou, Duc-Manh Tran, Thanh Le-Cong et al. // arXiv preprint arXiv:2407.16235. — 2024.
- [4] Cruz Dinis Barroqueiro, Almeida João Rafael, Oliveira José Luís. Open Source Solutions for Vulnerability Assessment: A Comparative Analysis // [IEEE Access](#). — 2023. — Vol. 11. — P. 100234–100255.
- [5] Dencheva Lyubka. Comparative analysis of Static application security testing (SAST) and Dynamic application security testing (DAST) by using open-source web application penetration testing tools : Ph.D. thesis / Lyubka Dencheva ; Dublin, National College of Ireland. — 2022.
- [6] Dowd Mark, McDonald John, Schuh Justin. The art of software security assessment: Identifying and preventing software vulnerabilities. — Pearson Education, 2006.
- [7] Error in USS Yorktown (CG-48) System. — URL: [https://en.wikipedia.org/wiki/USS_Yorktown_\(CG-48\)#Smart_ship_testbed](https://en.wikipedia.org/wiki/USS_Yorktown_(CG-48)#Smart_ship_testbed).
- [8] ["False negative - that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing](#) / Amit Seal Ami,

- Kevin Moran, Denys Poshyvanyk, Adwait Nadkarni // 2024 IEEE Symposium on Security and Privacy (SP). — 2024. — P. 3979–3997.
- [9] Fuzzing for software security testing and quality assurance / Ari Takannen, Jared D Demott, Charles Miller, Atte Kettunen. — Artech House, 2018.
 - [10] Fuzzing vulnerability discovery techniques: Survey, challenges and future directions / Craig Beaman, Michael Redbourne, J Darren Mumery, Saqib Hakak // Computers & Security. — 2022. — Vol. 120. — P. 102813.
 - [11] Jordan Cosme. Investigating Clang Static Analyzer’s False-positives and False-negatives : Ph.D. thesis / Cosme Jordan ; École Polytechnique Fédérale de Lausanne. — 2023.
 - [12] King James C. Symbolic execution and program testing // Commun. ACM. — 1976. — . — Vol. 19, no. 7. — P. 385–394. — URL: <https://doi.org/10.1145/360248.360252>.
 - [13] Le Lann G. [An analysis of the Ariane 5 flight 501 failure-a system engineering perspective](#) // Proceedings International Conference and Workshop on Engineering of Computer-Based Systems. — 1997. — P. 339–346.
 - [14] Li Jinfeng. Vulnerabilities mapping based on OWASP-SANS: a survey for static application security testing (SAST) // arXiv preprint arXiv:2004.03216. — 2020.
 - [15] Niemetz Aina, Preiner Mathias. Bitwuzla at the SMT-COMP 2020 // arXiv preprint arXiv:2006.01621. — 2020.
 - [16] Pan Yuanyuan. Interactive application security testing // 2019 International Conference on Smart Grid and Electrical Automation (ICS-GEA) / IEEE. — 2019. — P. 558–561.

- [17] Păsăreanu Corina S, Visser Willem. A survey of new trends in symbolic execution for software testing and analysis // International journal on software tools for technology transfer. — 2009. — Vol. 11. — P. 339–353.
- [18] Rôla João Pedro Marques. Dynamic security testing. — 2022.
- [19] Sen Koushik. [Concolic testing](#) // Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering. — ASE '07. — New York, NY, USA : Association for Computing Machinery, 2007. — P. 571–572. — URL: <https://doi.org/10.1145/1321631.1321746>.
- [20] Seth Aishwarya. Comparing Effectiveness and Efficiency of Interactive Application Security Testing (IAST) and Runtime Application Self-Protection (RASP) Tools. — North Carolina State University, 2022.
- [21] Sydr-Fuzz: Continuous hybrid fuzzing and dynamic analysis for security development lifecycle / Alexey Vishnyakov, Daniil Kuts, Vlada Logunova et al. // 2022 Ivannikov Ispras Open Conference (ISPRAS) / IEEE. — 2022. — P. 111–123.
- [22] Symbolic security predicates: hunt program weaknesses / Alexey Vishnyakov, Vlada Logunova, Eli Kobrin et al. // 2021 Ivannikov ISPRAS Open Conference (ISPRAS) / IEEE. — 2021. — P. 76–85.
- [23] Techniques of SAST Tools in the Early Stages of Secure Software Development: A Systematic Literature Review / Adrián Herrera Jerónimo, Patricia Martínez Moreno, José Antonio Vergara Camacho, Gerardo Contreras Vega // 2024 IEEE International Conference on Engineering Veracruz (ICEV) / IEEE. — 2024. — P. 1–8.
- [24] Vulnerability in Windows Kernel Could Allow Elevation of Privilege (941693). — URL: <https://learn.microsoft.com/en-us/security-updates/securitybulletins/2008/ms08-025#vulnerability-in-windows-kernel-could-allow-elevation-of-privilege>

- [25] A survey of symbolic execution techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.