ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Самойлов Виктор Максимович

ФИЛЬТРАЦИЯ ЛОЖНЫХ СРАБАТЫВАНИЙ В СТАТИЧЕСКИХ АНАЛИЗАТОРАХ ДЛЯ C/C++ С ПОМОЩЬЮ СИМВОЛЬНОЙ МАШИНЫ KLEE

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
ООО «Техкомпания Хуавей»,
инженер ключевых проектов
Е.К. Куликов

Руководитель
д-р. физ.-мат. наук, проф.
Б.А. Новиков

Санкт-Петербург 2023

Оглавление

Статический анализ — это важнейшая техника, используемая для выявления потенциальных ошибок и уязвимостей в программном коде. Несмотря на его важность, ложные срабатывания часто могут стать серьезной проблемой, приводя к ненужной трате времени и ресурсов, поскольку разработчики исследуют и устраняют проблемы, которых на самом деле нет. В данной работе предложен подход к фильтрации ложных срабатываний в отчетах статического анализа для C/C++ с использованием символьной виртуальной машины KLEE. Данный метод направляет символьное исполнение вдоль трассировок ошибок, что позволяет находить пути исполнения, подтверждающие истинные предупреждения. Разработанный инструмент устроен таким образом, чтобы не зависеть от анализатора, поскольку он принимает отчеты SARIF и использует специальную структуру данных для эффективной обработки нескольких трассировок ошибок одновременно. Кроме того, мы ввели систему оценки уверенности, чтобы показать пользователям, насколько мы уверены, что данное предупреждение является ложным срабатыванием. Использование данного подхода должно значительно сократить время и усилия, необходимые для обработки предупреждений о потенциальных ошибках и уязвимостях в программном коде.

Ключевые слова: символьное исполнение, статический анализ, ложные срабатывания, KLEE, SARIF

Static analysis is a crucial technique used to detect potential errors and vulnerabilities in software code. Despite its importance, false positives can often become a serious issue, leading to unnecessary waste of time and resources as developers investigate and address issues that don't actually exist. In this work, an approach is proposed to filter out false positives in static analysis reports for C/C++ using the symbolic virtual machine KLEE. This method directs symbolic execution along error traces, enabling the discovery of execution paths that confirm true warnings. Our tool is designed to be independent of the analyzer, as it takes SARIF reports and utilizes a specialized data structure for efficient processing of multiple error traces simultaneously. Additionally, we introduced a confidence assessment system to indicate to users how confident we are that a given warning is a false positive. The use of this approach should significantly reduce the time and effort required to handle warnings about potential errors and vulnerabilities in software code.

Keywords: symbolic execution, static analysis, false positives, KLEE, SARIF

Введение

Актуальность и релевантные работы

Статический анализ — это метод тестирования программного обеспечения, который предполагает анализ исходного кода без фактического выполнения программы [17]. Целью статического анализа является обнаружение потенциальных дефектов, уязвимостей безопасности и других типов проблем, которые могут быть найдены в коде до того, как программа будет скомпилирована и запущена.

Инструменты статического анализа могут выполнять широкий спектр проверок кода, включая синтаксический анализ, анализ потока управления, анализ потока данных и проверку типов. Эти проверки могут помочь выявить такие проблемы, как переполнение буфера, разыменование нулевого указателя, условия гонки, уязвимости в системе безопасности и другие [18].

Статический анализ является важной частью процесса разработки программного обеспечения, поскольку он может помочь разработчикам выявить проблемы на ранней стадии, до того как их устранение станет более сложным и дорогостоящим [9]. Выявляя проблемы на ранней стадии, разработчики могут сократить количество времени и усилий, необходимых для их устранения, и обеспечить более высокую безопасность и надежность программного обеспечения. Кроме того, статический анализ может помочь разработчикам обеспечить соблюдение стандартов написания кода и лучших практик, что может повысить общее качество кода.

Однако статический анализ может давать ложные срабатывания, то есть проблемы, о которых сообщает инструмент, но которые на самом деле не присутствуют в коде. Ложные срабатывания могут возникать, когда инструмент не имеет достаточного контекста, чтобы точно определить, является ли проблема реальной или нет. Например, инструмент статического анализа может отметить фрагмент кода как уязвимый для SQL-инъекций, даже если входные данные правильно

проверены и код на самом деле не является уязвимым.

Ложные срабатывания представляют собой серьезную проблему для разработчиков, поскольку они могут отнимать ценное время и ресурсы. Когда инструмент статического анализа сообщает о большом количестве ложных срабатываний, разработчикам бывает трудно отличить реальные проблемы от несущественных. Это может привести к деморализации и скептическому отношению к инструменту, что может заставить разработчиков игнорировать его результаты [21]. Фильтрация ложных срабатываний может эффективно выявить ложные проблемы, позволяя разработчикам сосредоточиться на настоящих ошибках, способствуя более эффективному и безопасному процессу разработки программного обеспечения. Сейчас существует несколько методов для борьбы с ложными срабатываниями.

Некоторые статические анализаторы [22] могут присваивать каждому результату уровень уверенности: категорию или число, которые говорят о том, насколько инструмент уверен в том, что данное предупреждение является верным. Разработчик может отранжировать результаты, но в дальнейшем всё равно будет проверять их вручную.

Существуют подходы, которые обучают модели для классификации результатов на ложные и истинные [3, 8]. Однако они зачастую зависят от конкретного инструмента или проекта и для использования их с другим инструментом/проектом нужны новые размеченные данные, что может быть неудобно при создании нового проекта или использовании нового инструмента.

Ещё одним подходом является направленное символьное исполнение, которое пытается проверить существует ли путь, проходящий через трассировку ошибки из отчёта статического анализатора. Условие истинности предупреждения — наличие такого пути. Если путь за отведённое время найти не удалось, то данное предупреждение объявляется ложным. Однако большинство текущих реализаций [1, 7, 19] данного подхода тесно связаны с конкретным инструментом и никак не информируют пользователя о том, что путь мог быть не найден из-за недостатка ресурсов (например, недостаточного временного лимита), а

также обрабатывают все предупреждения по отдельности. Целью данной работы является реализация подхода направленного символьного исполнения, лишённая данных проблем.

Входным форматом для результатов статического анализа был выбран SARIF. Основой для реализации данной работы стал KLEE — символьная виртуальная машина, интерпретирущая LLVM[1] IR (intermediate representation — промежуточное представление), форк которого активно разрабатывается в компании Huawei.

Цель и задачи

Данная работа ставит целью добавление режима направленного символьного исполнения в KLEE, который будет как можно меньше зависеть от конкретного статического анализатора, обрабатывать все предупреждения одновременно и выводить степень уверенности в случае, когда мы объявляем предупреждение анализатора ложным срабатыванием.
Для этого ставятся следующие задачи:

- Транслировать результаты статического анализатора на язык LLVM IR
- Реализовать структуру для хранения и работы с транслированными трассировками ошибок
- Направить символьное исполнение по трассировкам ошибок
- Реализовать систему оценки уверенности в случае ложных срабатываний
- Протестировать инструмент на отчётах различных статических анализаторов

[1]LLVM — платформа для создания компиляторов и наборов инструментов для них

Достигнутые результаты

В рамках данной работы в KLEE был добавлен режим направленного символьного исполнения. Был реализован модуль, транслирующий предупреждения из SARIF отчёта в трассировку целей с помощью отладочной информации. Также были реализованы опциональные проверка на проходимость трассировки и умный выбор стартовой функции для трассировки, основанные на модуле подсчёта расстояний, уже реализованном в KLEE.

Для хранения и работы с трассами был реализован персистентный бор целей, в котором узлы связываются множеством из целей. Он позволяет эффективно обрабатывать множество трассировок одновременно.

С помощью компоненты целевого поиска, уже имеющейся в KLEE, было реализовано направление символьного исполнения вдоль трассировок.

Также была добавлена система оценки уверенности для ложных срабатываний, основанная на количестве изученных путей, приводящих к интересующей ошибке.

Была проведена оценка инструмента на двух группах программ — Juliet Test Suite и открытых проектах на C/C++.

Разработанный инструмент используется инженерами компании Huawei.

Структура работы

В главе 1 представлен обзор существующих подходов к фильтрации ложных срабатываний и их реализаций, а также рассмотрены системы символьного исполнения для C/C++.

В главе 2 представлено описание LLVM IR, SARIF формата и его трансляция на язык LLVM IR с помощью отладочной информации.

В главе 3 представлено описание первого и итогового вариантов реализации структуры данных для хранения и работы с трассировками целей.

В главе 4 представлено описание направления символьного исполнения вдоль трассировок целей.

В главе 5 представлены описание алгоритма расчёта степени уверенности для предупреждения и его реализация в рамках данной работы.

В главе 6 представлены постановка и результаты экспериментов на двух группах программ.

В заключении данная работа была проанализирована и дальнейшие направления исследований в данной области были представлены.

1. Обзор литературы

Ложные срабатывания — важная проблема в области статического анализа программного обеспечения. Инженерам безопасности очень трудозатратно проверять отчёты статических анализаторов, в которых подавляющее большинство ошибок ложные: они тратят время впустую, что может привести к демотивации и игнорированию отчётов анализаторов вовсе. Далее рассматриваются различные подходы по работе с ложными срабатываниями.

Также в данной главе представлены современные системы символьного исполнения для C/C++, которые рассматривались, как возможная основа для данной работы.

1.1. Подходы по борьбе с ложными срабатываниями

Степень уверенности и классификаторы

Одним из самых простов способов помочь разработчикам разбираться с ложными срабатываниями является добавление уровня уверенности для предупреждения со стороны статического анализатора, как сделано, например, в PVS-Studio [22]. Все предупреждения классифицируются на три группы, в зависимости от того, насколько анализатор уверен в своей диагностике:

- Высокая степень уверенности — требуется проверка
- Средняя степень уверенности — анализатор не уверен, но можно посмотреть после всех проверок с высокой степенью уверенности
- Низкая степень уверенности — анализатор не уверен совсем и смотреть не советуется

Для каждого типа ошибки реализованы свои эвристики, на основе которых анализатор определяет уровень уверенности. Однако данный способ реализован не во всех статических анализаторах и помогает только

в приоритизации порядка, в котором инженеры безопасности будут отсматривать предупреждения и вручную их проверять.

Другим подходом является обучение моделей для классификации предупреждений на истинные и ложные. Kwangkeun et al. [8] в своей работе классифицировали потенциальные дефекты, найденные Airac — статическим анализатор, который был разработан для нахождения всевозможных переполнений буффера в программах на C. В векторе характеристик предупреждения использовались данные, которые были получены из Airac, такие как:

- местоположение переполнения буфера
- размер буфера
- значение индекса, который обращается к буферу, вызывая переполнение
- синтаксический контекст, например, произошло ли переполнение буфера внутри цикла
- семантический контекст, например, применялись ли какие-то приближения статическим анализатором

Однако не все статические анализаторы могут обеспечить такую подробную информацию об ошибке, что затрудняет обобщение подхода на другие инструменты статического анализа.

Cheirdari et al. [3] предложили алгоритм классификации SAPI. В нём участвует четыре основные характеристики предупреждения статического анализатора — тип ошибки в CWE[2], местоположение ошибки, метод или функция, в котором произошла ошибка и персональный идентификатор автора ошибки. Для каждого предупреждения для специальных комбинаций данных характеристик считается вероятность по формуле: $\frac{TF}{TF+FF}$, где TF — количество истинных предупреждений с данной комбинацией и FF — количество ложных предупреждений с данной комбинацией. Далее вероятность предупреждения считается

[2]CWE (Common Weakness Enumeration) — система категорий программных уязвимостей

как специальное усреднение вероятности используемых комбинаций, и если она больше какого-то числа, то предупреждение считается истинным. Проблема данного подхода в том, что модель надо будет переобучать при смене проекта или введении в проект новых разработчиков.

Несмотря на то, что классификаторы показывают хорошую производительность в определении ложных срабатываний, они привязаны к проекту и при попытке использовать их на новом проекте их нужно будет обучать заново для чего понадобятся новый размеченный людьми датасет истинных и ложных предупреждений на данном проекте.

Направленное символьное исполнение

Символьное исполнение [11] — это метод тестирования программного обеспечения, который анализирует код, выполняя его не с конкретными, а с символьными переменными, обозначающими произвольное значение, которое тип переменной может принять. Если рассматривать граф потока исполнения программы, то символьное исполнение пытается исследовать все пути в этом графе. Каждый путь в таком графе можно описать текущими значениями символьных переменных, которые могут меняться, например, в результате операции присваивания, и булевым условием, накладывающим ограничения на значения символьных переменных, удовлетворение которого является условием достижимости пути. Это условие может меняться, например, в результате операции ветвления при булевом условии ϕ: тогда символьное исполнение также ветвится и в одной ветви в условие пути добавляется ϕ, а в другой — $\neg\phi$. При достижении листа в графе или возникновении ошибки вызывается решатель на текущем условии пути, который либо говорит о недостижимости пути, либо возвращает конкретные значения входных символьных переменных, удовлетворяющих условию. Если запустить программу с этими входными данными, то она пройдёт тот же путь исполнения. Таким образом результатом символьного исполнения можно считать достижимые пути и значения входов, необходимых для их прохождения.

Давайте рассмотрим пример символьного исполнения одного из пу-

тей, приведённой ниже программы на рисунке 1. В начале исполнения у нас есть только входная символьная переменная `x` и никаких ограничений на неё. Далее мы приходим в `if` на 2 строке и тут исполнение ветвится, в данном примере мы выбираем `then` ветвь, поэтому добавляем $x \geq 0$ в условие пути. В строке 3 у нас появляется новая символьная переменная `y`, но условие пути не меняется. Далее снова происходит ветвение, мы снова выбираем `then` ветвь и потому добавляем $y < 0$ в условие пути. В строке 5 мы приходим в `assert`, который является листом в нашем графе потока исполнения, а потому вызываем решатель, который возвращает нам значение для входной `x` — INT_MAX. И действительно из-за переполнения, `y` станет равным INT_MIN, то есть меньше 0.

Статические анализаторы для некоторых ошибок указывают трассировку (массив местоположений в исходном коде), которая приводит к ошибке. Местоположение ошибки отдельно есть всегда и её можно считать трассировкой размера 1. Направленное символьное исполнение заключается в том, что мы принимаем трассировки предупреждений и пытаемся направить символьное исполнение только по тем путям в графе потока исполнения программы, которые могут пройти по хотя бы одной из трассировок. Если мы находим путь, проходящий по трассировке предупреждения, то это предупреждение можно считать истинным.

Существует несколько реализаций данного подхода. Одним из таких исследований является TASMAN [19], целью которого является удаление ложных срабатываний из отчета FlowDroid для Android. Однако

```
1 int foo(int x) {
2     if (x >= 0) {
3         int y = x + 1;
4         if (y < 0) {
5             assert(0);
6         }
7     }
8     return 0;
9 }
```

Символьные переменные	Условия пути
x	–
x	$x \geq 0$
x, $y = x + 1$	$x \geq 0$
x, $y = x + 1$	$x \geq 0$ && $y < 0$
solve($x \geq 0$ && $x + 1 < 0$)	

Рис. 1: Пример работы символьного исполнения

он построен поверх FlowDroid, что является серьезным недостатком, поскольку он тесно связан с этим инструментом статического анализа. Следовательно, мы не можем использовать этот инструмент с другими анализаторами. Основное отличие TASMAN от нашего подхода заключается в том, что в нем используется обратное символьное выполнение, которое начинает выполнение с точки ошибки и пытается достичь исходного состояния. Обратное символьное выполнение часто включает более сложные операции решения ограничений, поскольку оно предполагает работу в обратном направлении по программе для поиска удовлетворяющего входа, который приводит к состоянию ошибки. Это может быть сложнее, чем прямое символьное выполнение, которое может использовать более простые операции решения ограничений для исследования пространства входных данных из начального состояния.

Другое исследование — DyTa [7], которое состоит из состоит из двух этапов: статического и динамического. На статическом этапе генерируются потенциальные дефекты с использованием методов статического анализа, а на динамическом этапе эти дефекты подтверждаются с помощью символьного выполнения. Однако DyTa — это единый инструмент, который обрабатывает оба этапа, что затрудняет замену инструментов в конвейере. Кроме того, он поддерживает только C#.

Существует реализация для C/C++ [1]. Во-первых, в ней используется собственный закрытый символьный интерпретатор и неизвестный статический анализатор, что затрудняет прямое сравнение наших подходов. Кроме того, в их исследовании предлагается отображать исходный код в машинный, что может существенно усложнить реализацию и ограничить поддержку сложных языковых конструкций в трассировках. Во-вторых, данный проект использует динамическое символьное исполнение, что затрудняет направленный анализ кода.

Данный подход хорош тем, что может предложить конкретные входы для истинных предупреждений и разработчикам не надо будет делать этого вручную. Также для данного подхода не нужны размеченные датасеты. Однако текущие реализации тесно связаны с каким-то конкретным инструментом, обрабатывают предупреждения по одному,

что может быть не продуктивно, если трассировки имеют одинаковый префикс, а также не говорят никакой дополнительной информации в случае ложного срабатывания, кроме той, что из-за различных ограничений символьное исполнение не смогло найти подходящего пути.

1.2. Системы символьного исполнения

Символьное выполнение кода на языке C/C++ может быть осуществлено различными методами, которые можно разделить на две основные группы: классическое символьное и динамическое символьное исполнение. Классическое символьное исполнение подразумевает интерпретацию входных данных в виде промежуточного представления или байткода. В этом методе все возможные пути кода исследуются одновременно путем разветвления выполнения в каждой точке, где доступны несколько путей. С другой стороны, динамическое символьное исполнение начинается с определенного ввода и выполняет программу конкретно, чтобы собрать ограничения для этого пути. Затем ограничения ветвления инвертируются для создания новых входных данных. Этот метод требует модификации или инструментирования двоичного кода, что может оказаться сложной задачей.

Существует несколько динамических систем символьного исполнения, включая QSYM [14], которая использует машинный код и поэтому требует обработки каждой инструкции для каждой архитектуры процессора. SymCC [13], с другой стороны, использует байткод LLVM [12], который проще реализовать, поскольку он содержит всего 62 инструкции и не зависит от архитектуры. SymCC внедряет код, отслеживающий символьные вычисления, в виде разделяемой библиотеки с определенным интерфейсом, реализация которой называется бэкендом. В ней уже реализовано несколько бэкендов, и она может быть расширена вашей собственной реализацией символьного бэкенда.

Angr [16] — это символьный интерпретатор, который исполняет байткод VEX и поднимает машинный код в IR, что может быть неточно. Кроме того, Angr реализован на языке Python, что может привести к

более медленной производительности символьного исполнения по сравнению с другими системами.

KLEE [2] является одной из самых популярных символьных виртуальных машин и интерпретирует LLVM IR, который можно получить непосредственно из исходного кода. Он имеет большое сообщество и поддержку, что облегчает его разработку. Его форк, разрабатывающийся компанией Huawei, уже включает модуль для вычисления расстояний между базовыми блоками на основе графа потока исполнения, что необходимо для направления символьного исполнения.

1.3. Выводы

По результатам изучения подходов к фильтрации ложных срабатываний были сделаны следующие выводы:

- Уровень уверенности для предупреждений со стороны статических анализаторов упрощает работу с отчётами, но только в ранжировании предупреждений для просмотра и далеко не все статические анализаторы его поддерживают.

- Классификаторы фильтрует большое количество ложных срабатываний, однако они тесно связаны с характеристиками, которые предоставляет конкретный статический анализатор или которые свойственны конкретному проекту, поэтому для их работы с новым проектом/статическим анализатором необходимы новые размеченные данные, которых например может не быть, если проект только начинается.

- Направленное символьное исполнение не обладает выше перечисленными недостатками, а также для обнаруженных истинных предупреждений предоставляет конкретные значения входных данных, запуск с которыми воспроизводит дефект. Однако текущие реализации данного подхода обладают рядом недостатков такими как привязка к конкретному статическому анализатору, обра-

ботка предупреждений по одному и отсутствие дополнительной информации для пользователя при ложном срабатывании.

По результатам сравнения современных систем символьного исполнения для C/C++ выбор был сделан в пользу KLEE, так как он обладает модульной архитектурой и хорошо расширяем, большим сообществом и поддержкой, а также там уже реализованы модули необходимые для направленного символьного исполнения, связанные с подсчётом расстояний. Также направление символьного исполнения в целом проще в символьных виртуальных машинах, чем в динамических, что еще больше поддерживает решение использовать KLEE в данной работе.

2. Перевод отчёта статического анализатора на язык LLVM IR

В данной главе представлено описание выбранного входного формата SARIF [15], LLVM IR, процесс отображения местоположений в исходном коде в соответствующие элементы LLVM IR. Также в данной главе описан выбор функции, из которой нужно начать символьное исполнение для проверки предупреждения и опциональная проверка на проходимость трассировки.

2.1. SARIF

Обычно у каждого статического анализатора свой собственный формат с разным уровнем подробности сведений о предупреждении. Для поддержки нового инструмента статического анализа необходимо добавлять не только парсер нового формата, но и логику перевода предупреждений в промежуточное представление символьного интерпретатора, что очень неудобно. Поэтому в данной работе входным форматом был выбран SARIF — универсальный json[3] формат для результатов статического анализа, который довольно молод, но быстро набирает популярность и уже поддерживается многими популярными инструментами [4, 5, 10]. Пример отчёта статического анализатора в формате SARIF можно найти в приложении 1. Его основными элементами является идентификатор инструмента, который создал данный отчёт и массив предупреждений. Каждое предупреждение состоит из идентификатора ошибки, сообщения об ошибке, местоположения в исходном коде, в котором происходит ошибка и опциональной трассировки, которая является массивом местоположений в исходном коде и приводит к ошибке. Местоположение в исходном коде состоит из идентификатора файла, обязательной строки в этом файле и опциональными начальной и конечной колонками в этой строке. Идентификатор ошибки может быть произвольной строкой и зависит от конкретного инструмента, по-

[3]Формат JSON — www.json.org/json-en.html

этому для поддержки нового инструмента, необходимо лишь добавить отображение из идентификаторов ошибок данного статического анализатора во множество ошибок внутреннего перечисления: это сделано потому, что одна ошибка статического анализатора может соответствовать множеству ошибок из нашего перечисления, например, ошибка USE_AFTER_FREE в Infer может быть как обычный $UseAfterFree$, так и $DoubleFree$. Для парсинга SARIF-файла использовалась C++ библиотека nlohmann/json[4].

2.2. LLVM IR в KLEE

Рассмотрим основные элементы LLVM IR на оригинальном LLVM IR на примере с рисунка 2. Это:

- Функции: в данном примере *abs* — набор блоков и входных параметров
- Блоки: это набор инструкций с идентификатором, в данном примере 1, 6, 8, 11
- Инструкции: чуть больше 60 различных примитивных операций

У каждого следующего элемента иерархии есть родительская ссылка на предыдущего, т.е. каждая инструкция находится только в одном блоке и обладает ссылкой на него, то же с блоком и функцией. Но KLEE в процессе инициализации может модифицировать оригинальный байткод в ходе оптимизации и инструментирования: удалить/добавить инструкции (например, *br* в строке 10 модифицированного LLVM IR отсутствует в оригинальном), разделить блок на несколько (1 разделился на $1, 4, 5$, а 11 на $13, 15$). $!i$ — является идентификатором отладочной информации, которая состоит из имени файла, номеров строки и колонки и можно заметить, что в модифицированном байткоде возможна ситуация, при которой новые инструкции из одного блока разделяют

[4]Библиотека JSON для C++: github.com/nlohmann/json

```
1 int abs(int x) {
2     if (x >= 0) {
3         return x;
4     }
5     return -x;
6 }
```

```
1 define dso_local i32 @abs(i32 %0) #0 {
2 1:
3   %2 = alloca i32, align 4
4   %3 = alloca i32, align 4
5   store i32 %0, i32* %3, align 4
6   call void @llvm.dbg.declare, !13
7   %4 = load i32, i32* %3, align 4, !14
8   %5 = icmp sge i32 %4, 0, !16
9   br i1 %5, label %6, label %8, !17
10
11 6:
12   %7 = load i32, i32* %3, align 4, !18
13   store i32 %7, i32* %2, align 4, !20
14   br label %11, !dbg !20
15
16 8:
17   %9 = load i32, i32* %3, align 4, !21
18   %10 = sub nsw i32 0, %9, !22
19   store i32 %10, i32* %2, align 4, !23
20   br label %11, !dbg !23
21
22 11:
23   %12 = load i32, i32* %2, align 4, !24
24   ret i32 %12, !24
25 }
```

```
1 define dso_local i32 @abs(i32 %0) #0 {
2 1:
3   %2 = alloca i32, align 4
4   %3 = alloca i32, align 4
5   store i32 %0, i32* %3, align 4
6   br label %4, !12
7
8 4:
9   call void @llvm.dbg.declare, !12
10  br label %5, !14
11
12 5:
13  %6 = load i32, i32* %3, align 4, !14
14  %7 = icmp sge i32 %6, 0, !16
15  br i1 %7, label %8, label %10, !17
16
17 8:
18  %9 = load i32, i32* %3, align 4, !18
19  store i32 %9, i32* %2, align 4, !20
20  br label %13, !dbg !20
21
22 10:
            ;
23  %11 = load i32, i32* %3, align 4, !21
24  %12 = sub nsw i32 0, %11, !22
25  store i32 %12, i32* %2, align 4, !23
26  br label %13, !dbg !23
27
28 13:
29  %14 = load i32, i32* %2, align 4, !24
30  br label %15, !24
31
32 15:
33  ret i32 %14, !24
34 }
```

Рис. 2: C++, оригинальный и модифицированный LLVM IR программы

одинаковую отладочную информацию со старыми из другого (*br* в строке 10 и *load* в строке 13 имеют одинаковую отладочную информацию !14).

2.3. Отображение местоположения исходного кода в набор LLVM IR блоков

Для того, чтобы во время символьного исполнения в KLEE понимать прошли ли то или иное местоположение исходного кода из трассировки ошибки, необходимо отобразить его в множество соответствующих ему блоков LLVM IR. Для этого вначале перебираются все функции входной программы и выбираются те, что находятся в одном с файле с местоположением исходного кода. Путь файла функции берётся из её отладочной информации, однако нельзя просто сравнить значения имён двух путей на равенство, так как сборка байткод файла и стати-

ческий анализ могли быть произведены на разных машинах, поэтому проверяются только простые имена файлов: возможно в рассмотрение будут взяты какие-то лишние функции, но так точно не будут упущены те функции, что имелись ввиду.

Далее перебираются все блоки функции и оставляются только те, для которых номер строки местоположения исходного файла находится между номерами строк первой и последней инструкции в блоке.

Как было отмечено в прошлой подглаве, из-за модификации байт-кода могут добавиться новые инструкции с той же отладочной информацией, что и старые инструкции из других блоков. Из-за этого могут быть выбраны лишние блоки, поэтому перед началом процесса отображения сохраняется множество четвёрок (имя файла, номер строки, номер колонки, код операции) по инструкциям и их отладочной информации из оригинального LLVM IR. Теперь для того, чтобы окончательно выбрать блок, нужно проверить, что хотя бы одна инструкция из этого блока соответствует местоположению исходного кода, содержится в оригинальном LLVM IR и не является отладочной инструкцией. Условием соответствия местоположению исходного кода является равенство номеров строк в отладочной информации и местоположении исходного кода. Если есть номер стартовой колонки, то также равенство ей номера колонки из отладочной информации инструкции, а если также есть номер конечной колонки, то номер колонки из отладочной информации должен находиться между номерами стартовой колонки и конечной колонки.

2.4. Выбор стартовой функции и проверка на проходимость трассировки

Для того, чтобы понять откуда начинать символьное исполнение нужно выбрать стартовую функцию для каждого предупреждения. Очевидное решение, которое сейчас используется по умолчанию, — это взять функцию, которая соответствует блокам отображения первого элемента трассировки ошибки. Однако это не всегда корректно: в при-

мере на рисунке 3 выберется `helper`, как стартовая функция, так как первое местоположение исходного файла находится в данной функции, а из неё невозможно достичь функции, в которой происходит ошибка — разыменование нулевого указателя. Поэтому я добавил возможность умного выбора стартовой функции с помощью флага `--smart-resolve-entry-function`, которая основана на использовании реализованного в KLEE модуля подсчёта расстояний между блоками и функциями:

- Введём переменные `res` и `cur`, в которых хранятся функции
- Изначально в переменной `res` находится функция соответствующая первому элементу из трассировки
- Начинаем цикл со второго элемента трассировки
- Записываем функцию текущего элемента трассировки в переменную `cur`
- Если из `res` до `cur` ненулевое расстояние, то идём дальше
- Иначе проверяем, если из `cur` до `res` ненулевое расстояние, то записываем `cur` в `res`
- Иначе сообщаем пользователю об ошибке

В конце концов, выберется функция, из которой достижимы все элементы трассировки, либо пользователю будет сообщено, что такой нет. Этот режим является опциональным, так как подсчёт расстояний между функциями требует дополнительных ресурсов и такая проблема отсутствует во множестве статических анализаторов.

Также возможна ситуация, когда трассировка ошибки непроходима для символьного исполнения, то есть из предыдущей локаций синтаксически недостижима следующая: статические анализаторы могут делать это, чтобы лучше обьяснить ошибки пользователям. В таких случаях не хочется объявлять предупреждение ложным срабатыванием, а хочется сообщить о том, что данное предупреждение инструмент

```c
void *helper() {
    return NULL;
}

int doSmth(char *arg) {
    char **dev = NULL;
    dev = helper();
    dev[0] = arg;
    return 0;
}
```

```
{
    "trace": [
        {
            "file": "test.c",
            "line": 2
        },
        {
            "file": "test.c",
            "line": 8
        }
    ]
}
```

Рис. 3: C++ программы и трассировка ошибки

проверить не можем и его надо смотреть вручную. Эта проверка добавлена с помощью флага `--check-traversability` и также основана на использовании блока подсчёта расстояний. Смотрим на множества блоков, в которые отобразились текущее и следующее местоположение исходного кода. Нужно, чтобы хотя бы из одного блока текущего можно было достичь блока следующего. Если функции блоков совпадают, то условием достижимости одного блока из другого является ненулевое расстояние между ними, иначе ненулевое прямое или обратное расстояние между функциями блоков.

2.5. Выводы

Чтобы решить проблему необходимости писать много нового кода для поддержки нового инструмента статического анализа, был использован универсальный формат для результатов статического анализа — SARIF. Результатом парсинга SARIF является массив предупреждений, каждое из которых состоит из трассировки ошибки и множества типов ошибок, которые ожидаются в конце прохождения трассировки. Трассировка ошибки состоит из местоположений исходного кода, для каждого уникального местоположения получено отображение в множество LLVM IR блоков. Также были реализованы опциональные умный выбор стартовой функции и проверка трассировки на проходимость, основанные на использовании модуля подсчёта расстояний в KLEE.

3. Хранение и работа с транслированными трассировками ошибок

В данной главе подробно описана структура данных для хранения и работы с транслированными трассировками ошибок, включая её внутреннюю реализацию и интерфейс. Также в данной главе представлен первый вариант внутренней реализации, который был признан неудачным.

3.1. Класс цели и бор целей

Цель — это блок LLVM IR и множество типов ошибок, которые мы можем ожидать при символьном исполнении блока: оно может быть и пустым, если мы не ожидаем никаких ошибок. Трассировку можно воспринимать как слово, где вместо букв у нас есть местоположения исходного кода. В прошлой главе был описан процесс получения отображения в множество LLVM IR блоков для каждого местоположения исходного кода. Каждый элемент трассировки можно перевести во множество целей — воспользуемся отображением и для каждого блока создадим соответствующую цель: множество ошибок для всех пустое, а для целей последнего элемента трассировки (местоположения ошибки) берём множество ошибок из предупредения. Таким образом каждое предупреждение представлено в виде слова, где буквами являются множества целей (трассировка целей). Структурой данных для такого представления был выбран бор — древовидная структура данных, состоящая из узлов, которые связаны ключами, хранящимися на рёбрах. Ключом в случае обычных слов, например, могут выступать буквы.

3.2. Реализация бора целей

Первый вариант

В первом варианте реализации бора целей ключом, связывающим узлы выступали цели. При добавлении перебираются все элементы из

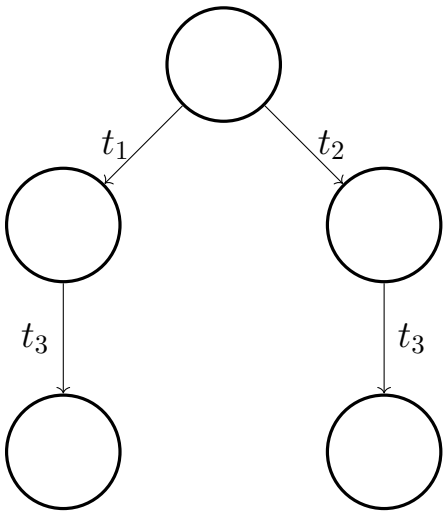

Рис. 4: Пример первого варианта бора целей

трассировки целей, на каждом уровне выбираем какой-то блок из множества и идём дальше. Пусть есть предупреждение с трассировкой, состоящей из двух местоположений исходного кода, первое из которых отобразилось в два блока LLVM IR, а второе в один. Тогда бор целей для него будет, как на рисунке 4. Сначала смотрим на множество целей для первого элемента трассировки — $\{t_1, t_2\}$, берём t_1 и спускаемся в соответствующий узел, далее смотрим на множество целей для второго элемента — $\{t_3\}$, берём единственный элемент и спускаемся в соответствующий узел, так как больше элементов в трассировке нет, то возвращаемся на первый уровень, берём t_2 и повторяем с ним.

Проблема данного подхода в том, что для трассировки с $c_1, c_2, ..., c_n$, где c_i — количество целей в множестве i-ого элемента трассировки мы создаём $c_1 * c_2 * ... * c_n$ листов в боре, что увеличивает общее количество узлов в боре и время работы, так как из-за потери информации о том, что t_1 и t_2 в примере были привязаны к одной локации, при достижении t_1 достижение t_2 будет продолжаться независимо, хотя это и не требуется.

Итоговый вариант

В итоговом варианте реализации бора целей ключом было выбрано множество целей. Пример можно увидеть на рисунке 5. В данной ре-

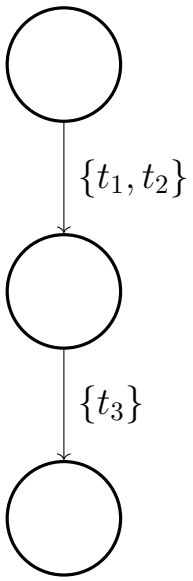

Рис. 5: Пример итогового варианта бора целей

ализации для каждого предупреждения существует ровно один лист. Каждое состояние символьного исполнения содержит свою копию бора целей, поэтому она персистентная, то есть при каждом изменении узла, создаётся и меняется его копия. Узел содержит отображение из множества целей в узел — `nodes` и отображение из цели в набор множеств целей, которые содержат эту цель и являются ключами в первом отображении — `targets`. Ключи второго отображения являются целями, которые состояние символьного исполнения пытается достичь в данный момент времени. Для дальнейшего направления символьного исполнения вдоль трассировок необходимо поддержать следующие операции:

- `stepTo(t: цель)` — при достижении `t` считаем все множества целей s_i, содержащие `t`, достигнутыми. Сохраняем узлы n_i, связанные с текущим по s_i и вызываем `remove(t)`. Далее объединяем текущий узел с n_i, где объединение узлов означает объединение отображений. Объединение отображений тривиально, если ключи не пересекаются, иначе при совпадение ключей `nodes` — рекурсивно вызываем объединение двух узлов, при совпадении `targets` — просто объединяем два множества.

- `block(t: цель)` — в данном методе `t` всегда является целью с ошибкой, то есть связывает узел n с листом l, и вызывается при её

достижении, чтобы другие состояния не тратили время на повторное подтверждение предупреждения связанного с l. В результате в n вызывается `remove(t)`.

- `remove(t: цель)` — удаляет `t` из данного узла. Сохраняем все множества целей s_i из `targets` по ключу `t` и удаляем `t` из этого отображения. Для каждого s_i проходимся по всем t_i из этого множества и смотрим есть ли t_i во множестве ключей `targets`. Если таких нет, то мы вызвали `remove(t_i)` для всех t_i и s_i можно удалить из `nodes`.

В прошлой главе для каждого предупреждения была выбрана стартовая функция, для каждой уникальной стартовой функции создаётся свой бор целей, в который добавляются соответствующие предупреждения. Далее для каждой стартовой функции создаётся состояние символьного исполнения, для которого все параметры функции являются символьными переменными и которое хранит соответствующий бор целей, эти состояния передаются, как начальные в символьном исполнении.

3.3. Выводы

Для удобства работы с предупреждениями они транслируются в трассировки целей, для хранения и работы с которыми был разработана структура данных — бор целей. Было разработана две версии, которые отличались тем, чем связываются узлы в боре. В первой — это были цели, но это версия была признана неэффективной, поэтому в итоговой — это множество целей. Также для данной структуры было поддержано условие персистентности, необходимое потому, что каждое состояние символьного исполнения хранит копию бора и может его модифицировать, и реализован необходимый для направления символьного исполнения интерфейс.

4. Направление символьного исполнения по трассировкам ошибок

В архитектуре KLEE есть модуль, отвечающий за выбор следующего для исследования символьного состояния — выборщик путей. Он обладает следующим интерфейсом:

- `update(states)` — получает массив состояний, состоящий из текущего состояния, новых состояний, удалённых состояний, на основе которых нужно обновить выборщик путей
- `next` — возвращает следующее состояние для исследования

С помощью данной компоненты можно направлять символьное исполнение. В данной главе представлено описание реализации выборщика путей, который направляет символьное состояние по трассировкам ошибок.

4.1. Реализация модуля направления символьного исполнения

В KLEE уже реализован целевой поиск для цели `t` — выборщик путей, который поддерживает ближайшее к `t` состояние: его стратегия выбора путей вдохновлена [6]. Мы хотим различать одинаковые цели, если они находятся в различных частях бора целей, то есть относятся к различным трассировкам, поэтому вместе с бором хранится история — массив целей, для которых вызывался `stepTo` на боре.

На рисунке 6 представлена общая архитектура выборщика путей, который был реализован в рамках данной работы: внутри хранится отображение из пары (история `h`, цель `t`) в компоненту целевого поиска для цели `t` и множество целей с ошибками, которых уже достигли (то есть финальные цели тех трассировок, которые уже подтверждены). Рассмотрим реализацию метода `update(states: массив состояний)` на примере с рисунка 6:

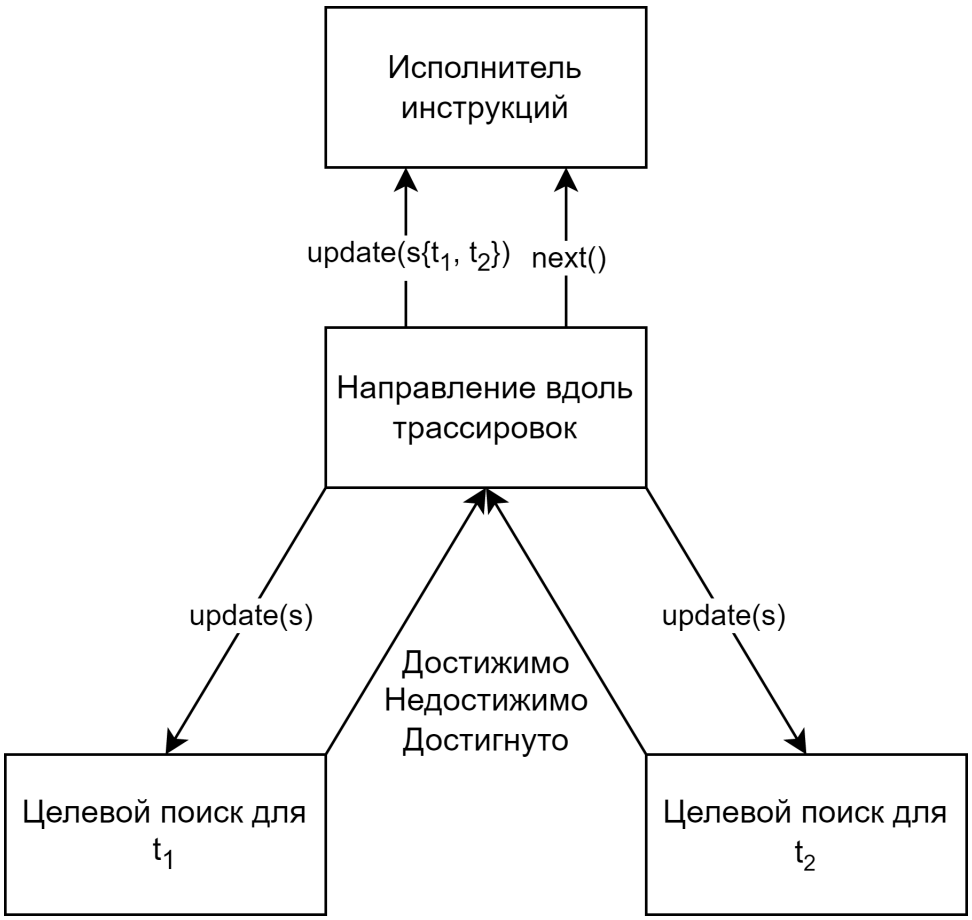

Рис. 6: Архитектура модуля направленного исполнения

- Для каждого состояния $s \in states$, для всех текущих целей s — t_i и его истории `h` обновляем соответствующие поисковые компоненты: если для пары (h, t_i) нет поисковой компоненты, то создаём новую. В нашем примере обновляются поисковые компоненты для t_1 и t_2, так как они являются текущими целями s.

- При достижении цели `t` вызываем `stepTo(t)` на боре состояния, которое его достигло. Так как после вызова у состояния обновится история, то нужно предварительно удалить это состояние из поисковых компонент с прошлой историей и добавить в поисковые компоненты с новой.

- Если достигнутое `t` было с ошибкой, то это означает, что было подтверждено одно из предупреждений и его больше не надо пытаться подтвердить: добавляем `t` в множество достигнутых целей с ошибками и для всех последующих обновлений для всех состо-

яний будем вызывать `block(t)`.

- Если `t` стало недостижимым для s, то нужно вызвать `remove(t)` на боре этого состояния.

Каждый выбор следующего состояния делегируется случайной компоненте целевого поиска из тех, что в данный момент находятся в отображении.

4.2. Выводы

Был реализован выборщик путей, который направляет символьное исполнение вдоль трассировок ошибок с помощью уже реализованной в KLEE компоненты целевого поиска и бора целей, реализованного в прошлой главе.

5. Степень уверенности для ложных срабатываний

Мы могли не подтвердить какое-то из предупреждений не потому, что пути, проходящего по трассировке предупреждения не существует, а из-за недостаточного ограничения по времени символьного исполнения или других ограничений символьной виртуальной машины. Поэтому необходимо сообщать пользователю степень уверенности в том, что предупреждение действительно ложно. В данной главе описан алгоритм расчёта степени уверенности и его реализация.

5.1. Описание алгоритма

Пусть хотим подтвердить трассировку целей, тогда в любой момент времени каждое состояние символьного исполнения пытается достичь какого-нибудь элемента трассировки. Пусть s в данный момент пытается достичь множество целей $\{t_i\}$, тогда поддерживается вес для пары $(s, \{t_i\})$. После того, как s достигнет $\{t_i\}$, то будет пытаться достичь следующего элемента трассировки — $\{t_j\}$, соответственно начнём поддерживать вес для пары $(s, \{t_j\})$, который изначально будет равен последнему значению веса для пары $(s, \{t_i\})$. В итоге из уверенности в предупреждении, изначально равной 1 вычитается сумма весов всех пар $(s_i, \{t_k\})$, где s_i — оставшиеся после символьного исполнения состояния и $\{t_k\}$ — соответствующий элемент трассировки целей из этого предупреждения, которого они пытались достичь.

Рассмотрим на примере с рисунка 7, как обновляется вес для пар: изначально он равен 1, затем при первом ветвлении он делится пополам, так как оба состояния достигают хотя бы одной цели из интересующего множества целей, s_1 — t_2, а s_3 — t_1. После второго ветвления деления веса не происходит, так как s_2 не достигает нужных целей. В общем случае при каждом ветвлении вес делится поровну между состояними, достигающими хотя бы одну из целей множества.

Таким образом, уверенность в ложности предупреждения — это

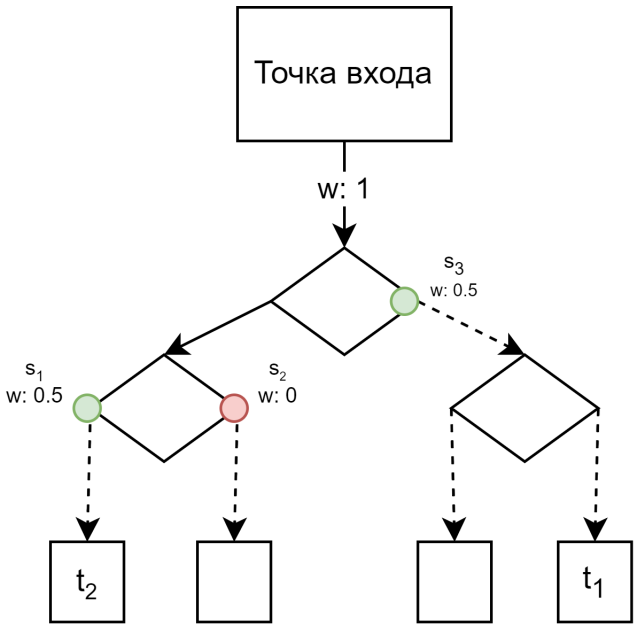

Рис. 7: Пример расчёта веса для $\{t_1, t_2\}$

своеобразный процент путей, приводящих к предупреждению, который мы успели изучить.

5.2. Реализация алгоритма

В каждом узле бора хранится вес. Вес для пары $(s, \{t_i\})$ хранится в узле, связанном с корнем бора s по ребру $\{t_i\}$. При вызове `update(states)` у выборщика путей, реализованного в предыдущей главе, аккумулируется отображение из цели в множество состояний, которые могут этой цели достичь. Далее на основании этого отображения обновляются веса: для каждого множества $\{t_i\}$ подсчитывается количество состояний, достигающих хотя бы одной цели из множества, после этого вес для пары $(s, \{t_i\})$ делится на это количество и обновляется в соответствующем узле. В конце символьного исполнения для всех листов в поддереве, связанном с корнем множеством $\{t_i\}$ вес будет равен весу для пары $(s, \{t_i\})$, на который и оштрафуется уверенность для всех предупреждений, соответствующих листьям в данном поддереве.

5.3. Выводы

Символьной виртуальной машине может не хватить ресурсов (времени/памяти) для подтверждения трассы срабатывания. Поэтому был реализован алгоритм расчёта степени уверенности для ложных срабатываний, которая показывает сколько процентов путей, потенциально приводящих к предупреждению, было изучено в ходе исполнения. Основываясь на этой информации, можно изменить ограничения символьной машины и запуститься на предупреждениях с низкой уверенностью ещё раз.

6. Оценка инструмента

В данной главе описана постановка эксперимента и его инфраструктура, а также результаты запуска эксперимента на Juliet Test Suite и проектах с открытым исходным кодом.

6.1. Постановка эксперимента и инфраструктура

Входными данными для эксперимента является SARIF файл с поддержаными предупреждениями. Это предупреждения с поддержаными типами ошибок, которые находятся в соответствии с CWE. В данный момент поддерживается:

- Разыменование нулевого указателя (NullDeref)[5]
- Разыменование освобождённого указателя (UseAfterFree)[6]
- Повторное освобождение указателя (DoubleFree)[7]

Все предупреждения размечены на истинные и ложные. Целью эксперимента является нахождение числа подтверждённых истинных предупреждений — чем больше, тем лучше. SARIF отчёты генерировались с помощью трёх инструментов статического анализа для C/C++:

- Clang Static Analyzer [4]
- CppCheck [5]
- Infer [10]

Эксперименты проводились на платформе с операционной системой Ubuntu 20.04.6 LTS 64-bit и процессором Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz × 8, с ограничением по памяти в 8 ГБ и следующим ограничением по времени в секундах: $\frac{bcSize}{250000} + 20 \times warningsNum$, где $bcSize$ — размер файла с байткодом программы в байтах, а $warningsNum$ — количество предупреждений в отчёте.

[5]Определение CWE-476
[6]Определение CWE-416
[7]Определение CWE-415

6.2. Эксперименты на Juliet Test Suite

Для первого этапа экспериментов был выбран Juliet Test Suite [20] — тестовый набор для оценки статических анализаторов, где тестовые случаи разделены по типам ошибок, а также на истинные и ложные, так что здесь разметка отчётов была произведена автоматически. Результаты представлены в таблице 1. Можно заметить, что в данном эксперименте мы подтверждаем почти 100% истинных предупреждений, что и является нашей целью, однако есть особые случаи. Ни одного подтверждённого предупреждения для предупреждений Clang SA с типом ошибки $UseAfterFree$ связано с тем, что в моей работе и статическом анализаторе различаются семантики этих ошибок: KLEE реагирует только на разыменование, в то время как статический анализатор указывает на любое упоминание указателя в коде, в том числе безопасное, из-за чего трассировка не доходит до места ошибки. Также было обнаружено, что мы не можем покрыть 12 предупреждений Infer с типом ошибки $NullDeref$, так как трассировка составлена так, что из предыдущего элемента нельзя достичь следующего.

	NullDeref			UseAfterFree			DoubleFree		
	T	F	C	T	F	C	T	F	C
Clang SA	156	35	156	18	18	0	105	-	105
CppCheck	102	-	102	-	-	-	238	-	238
Infer	210	-	198	84	-	84	276	-	276

Таблица 1: Оценка на Juliet Test Suite

T — количество истинных предупреждений, F — ложных, C — подтверждённых

6.3. Эксперименты на открытых проектах

Также были проведены эксперименты на пяти проектах с открытым исходным кодом. В их число вошли a2ps[8], discount[9], libtiff[10], coreutils[11] и flite[12]. Здесь разметка предупреждений в SARIF репортах была проведена вручную. Подробные результаты по каждому проекту и статическому анализатору представлены в таблице 2. Большинство истинных предупреждений было не подтверждено из-за проблемы взрыва путей и того, что символьное исполнение не успевает исследовать все пути исполнения.

Также по результатам этого эксперимента был подсчитан общий процент подтверждения по следующей формуле: $\frac{allConfirmed}{allTrue}$, где $allConfirmed$ — количество подтверждённых по всем исследуемым проектам и $allTrue$ — количество истинных по всем исследуемым проектам. Результаты сравнения с [1], реализацией подхода направленного символьного исполнения для C/C++, по этой метрике приведены в таблице 3.

[8]Проект a2ps: www.gnu.org/software/a2ps/
[9]Проект discount: github.com/Orc/discount
[10]Проект libtiff: gitlab.com/libtiff/libtiff
[11]Проект coreutils: github.com/coreutils/coreutils
[12]Проект flite: github.com/festvox/flite

	a2ps			discount			libtiff			coreutils			flite		
	T	F	C	T	F	C	T	F	C	T	F	C	T	F	C
Clang SA	12	5	7	0	1	0	2	6	2	1	20	1	5	4	3
CppCheck	0	10	0	0	0	0	0	3	0	0	1	0	0	0	0
Infer	2	9	2	20	0	12	4	3	2	0	3	0	-	-	-

Таблица 2: Оценка на открытых проектах

T — количество истинных предупреждений, F — ложных, C — подтверждённых

	Gerasimov et al [1]	KLEE с Clang SA	KLEE с Infer
Общий процент подтверждения	12%	65%	61%

Таблица 3: Сравнение с аналогом

6.4. Выводы

Для тестирования предлагаемого подхода были проведены эксперименты с размеченными SARIF репортами. Были созданы две подгруппы проектов для генерации предупреждений. Первой был Juliet Test Suite, на котором общий процент подтверждения составил 100% для всех статических анализаторов почти для всех типов ошибок, что доказывает корректность данного подхода с различными статическими анализаторами и типами ошибок. Второй группой были проекты с открытым исходным кодом, на которых удалось достичь общего процента подтверждения предупреждений в 65% для Clang SA и 61% для Infer, что превосходит результаты аналога с 12% и показывает применимость данного подхода на реальных проектах.

Заключение

Главным результатом данной работы стала реализация подхода направленного символьного исполнения для фильтрации ложных срабатываний в результатах статического анализа в KLEE[13]. Также в рамках данной работы были достигнуты следующие результаты:

- Реализована поддержка SARIF формата, как входного для результатов статического анализа и трансляция отчётов в данном формате в массив трассировок целей, что позволяет данному подходу работать с любым анализатором, поддерживающим SARIF формат и предоставившим отображение между его и нашими типами ошибок.

- Реализован бор целей — структура данных для хранения и работы с трассировками целей, которая позволяет обрабатывать все предупреждения одновременно и экономить на обработке трассировок с одинаковыми префиксами.

- Реализована система оценки уверенности для ложных срабатываний, основанная на количестве изученных путей, потенциально проходящих по трассировке предупреждения, которая показывает пользователям, насколько система уверена в том, что срабатывание действительно ложное.

- Реализованный подход был протестирован на двух группах проектов. Оценка на Juliet Test Suite показала корректность данного подхода при работе с различными анализаторами и обработке различных типов ошибок. Оценка на открытых проектах показала прирост производительности в сравнении с аналогом и применимость подхода на реальных проектах.

- Реализованный в рамках работы инструмент используется инженерами компании Huawei.

[13] github.com/UnitTestBot/klee

Дальнейшая работа возможна в следующих направлениях:

- Поддержка новых типов ошибок для предупреждений статического анализа
- Разделение логики расчёта достижимости цели и выбора следующего состояния: это позволит применить другие стратегии выбора путей.

Список литературы

[1] Gerasimov A. Yu., Kruglov L. V., Ermakov M. K., and Vartanov S. P. An Approach to Reachability Determination for Static Analysis Defects with the Help of Dynamic Symbolic Execution // Program. Comput. Softw. — 2018. — nov. — Vol. 44, no. 6. — P. 467–475. — Access mode: https://doi.org/10.1134/S0361768818060051.

[2] Cadar Cristian, Dunbar Daniel, and Engler Dawson. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. — USA : USENIX Association. — 2008. — P. 209–224.

[3] Cheirdari Foteini and Karabatis George. Analyzing False Positive Source Code Vulnerabilities Using Static Analysis Tools // 2018 IEEE International Conference on Big Data (Big Data). — 2018. — P. 4782–4788.

[4] Clang Static Analyzer: source code analysis tool. — Access mode: https://clang-analyzer.llvm.org/.

[5] CppCheck: a tool for static C/C++ analysis. — Access mode: https://cppcheck.sourceforge.io/.

[6] Ma Kin-Keung, Khoo Yit, Foster Jeffrey, and Hicks Michael. Directed Symbolic Execution. — 2011. — 09. — Vol. 6887. — P. 95–111.

[7] Ge Xi, Taneja Kunal, Xie Tao, and Tillmann Nikolai. DyTa: Dynamic Symbolic Execution Guided with Static Verification Results // Proceedings of the 33rd International Conference on Software Engineering. — New York, NY, USA : Association for Computing Machinery. — 2011. — P. 992–994. — Access mode: https://doi.org/10.1145/1985793.1985971.

[8] Yi Kwangkeun, Choi Hosik, Kim Jaehwang, and Kim Yongdai. An Empirical Study on Classification Methods for Alarms from a Bug-

Finding Static C Analyzer // Inf. Process. Lett. — 2007. — Vol. 102, no. 2–3. — P. 118–123. — Access mode: https://doi.org/10.1016/j.ipl.2006.11.004.

[9] Bessey Al, Block Ken, Chelf Benjamin, Chou Andy, Fulton Bryan, Hallem Seth, Henri-Gros Charles, Kamsky Asya, McPeak Scott, and Engler Dawson. A Few Billion Lines of Code Later Using Static Analysis to Find Bugs in the Real World // Commun. ACM. — 2010. — 02. — Vol. 53. — P. 66–75.

[10] Infer: Static Analyzer. — Access mode: https://fbinfer.com/.

[11] King James C. Symbolic Execution and Program Testing // Commun. ACM. — 1976. — jul. — Vol. 19, no. 7. — P. 385–394. — Access mode: https://doi.org/10.1145/360248.360252.

[12] Lattner C. and Adve V. LLVM: a compilation framework for lifelong program analysis & transformation // International Symposium on Code Generation and Optimization, 2004. CGO 2004. — 2004. — P. 75–86.

[13] Poeplau Sebastian and Francillon Aurélien. Symbolic Execution with SYMCC: Don't Interpret, Compile! — USA : USENIX Association. — 2020. — SEC'20.

[14] Yun Insu, Lee Sangho, Xu Meng, Jang Yeongjin, and Kim Taesoo. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing // Proceedings of the 27th USENIX Conference on Security Symposium. — USA : USENIX Association. — 2018. — P. 745–761.

[15] SARIF: Static Analysis Results Interchange Format. — Access mode: https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

[16] Shoshitaishvili Yan, Wang Ruoyu, Salls Christopher, Stephens Nick, Polino Mario, Dutcher Andrew, Grosen John, Feng Siji, Hauser Christophe, Kruegel Christopher, and Vigna Giovanni.

SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis // 2016 IEEE Symposium on Security and Privacy (SP). — 2016. — P. 138–157.

[17] Stefanović Darko, Nikolić Danilo, Dakic Dusanka, Spasojević Ivana, and Ristic Sonja. Static Code Analysis Tools: A Systematic Literature Review. — 2020. — 01. — P. 0565–0573. — ISBN: 9783902734297.

[18] Ayewah Nathaniel, Pugh William, Hovemeyer David, Morgenthaler J. David, and Penix John. Using Static Analysis to Find Bugs // IEEE Software. — 2008. — Vol. 25, no. 5. — P. 22–29.

[19] Arzt Steven, Rasthofer Siegfried, Hahn Robert, and Bodden Eric. Using Targeted Symbolic Execution for Reducing False-Positives in Dataflow Analysis // Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis. — New York, NY, USA : Association for Computing Machinery. — 2015. — SOAP 2015. — P. 1–6. — Access mode: https://doi.org/10.1145/2771284.2771285.

[20] Wagner Andreas and Sametinger Johannes. Using the Juliet Test Suite to compare static security scanners // 2014 11th International Conference on Security and Cryptography (SECRYPT). — 2014. — P. 1–9.

[21] Johnson Brittany, Song Yoonki, Murphy-Hill Emerson, and Bowdidge Robert. Why don't software developers use static analysis tools to find bugs? // 2013 35th International Conference on Software Engineering (ICSE). — 2013. — P. 672–681.

[22] The way static analyzers fight against false positives, and why they do it. — Access mode: https://pvs-studio.com/en/blog/posts/cpp/0488/.

Приложения

1. Пример отчёта статического анализатора в формате SARIF


```
{
  "runs": [
    {
      "results": [
        {
          "codeFlows": [
            {
              "threadFlows": [
                {
                  "locations": [
                    {
                      "importance": "unimportant",
                      "location": {
                        "message": {
                          "text": "Control jumps to 'case 100:'  at line 1520"
                        },
                        "physicalLocation": {
                          "artifactLocation": {
                            "index": 0,
                            "uri": "file:///home/marvel1337/2022/klee/coreutils/src/numfmt.c"
                          },
                          "region": {
                            "endColumn": 7,
                            "startColumn": 7,
                            "startLine": 1472
                          }
                        }
                      }
                    },
                    {
                      "importance": "essential",
                      "location": {
                        "message": {
                          "text": "Array access (from variable 'optarg') results in a null pointer dereference"
                        },
                        "physicalLocation": {
                          "artifactLocation": {
                            "index": 0,
```

```
                            "uri": "file:///home/marvel1337/2022/klee/coreutils/src/numfmt.c"
                          },
                          "region": {
                            "endColumn": 24,
                            "endLine": 1522,
                            "startColumn": 15,
                            "startLine": 1522
                          }
                        }
                      }
                    }
                  ]
                }
              ]
            }
          ],
          "locations": [
            {
              "physicalLocation": {
                "artifactLocation": {
                  "index": 0,
                  "uri": "file:///home/marvel1337/2022/klee/coreutils/src/numfmt.c"
                },
                "region": {
                  "endColumn": 24,
                  "endLine": 1522,
                  "startColumn": 15,
                  "startLine": 1522
                }
              }
            }
          ],
          "message": {
            "text": "Array access (from variable 'optarg') results in a null pointer dereference"
          },
          "ruleId": "core.NullDereference"
        }
      ],
      "tool": {
        "driver": {
          "fullName": "clang static analyzer",
          "language": "en-US",
          "name": "clang",
```

```
            "version": "Ubuntu clang version 12.0.0-3ubuntu1~20.04.5"
          }
        }
      }
    ],
    "version": "2.1.0"
}
```