ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Санкт-Петербургская школа
физико-математических и компьютерных наук

Субботина Олеся Вадимовна

ИНТЕГРАЦИЯ TAINT АНАЛИЗА В ПРОЦЕСС ГЕНЕРАЦИИ МОДУЛЬНЫХ ТЕСТОВ НА БЕЗОПАСНОСТЬ ПРОГРАММ

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

Рецензент
Структурное подразделение
НИЧ ФМКН СПБГУ,
инженер-исследователь
А. И. Рябков

Руководитель
Кандидат физико-математических наук,
доцент, департамент информатики
Д. Н. Москвин

Санкт-Петербург 2023

Содержание

Аннотация

Обеспечение безопасности программного обеспечения – важный процесс, необходимый для гарантии защиты приложений. Несмотря на то, что в настоящее время существует множество инструментов статического анализа для обнаружения ошибок безопасности в программах, присутствует потребность в наличии автоматизированных генераторов тестов на встречающиеся в программах уязвимости. Тесты на безопасность позволяют интегрировать поиск уязвимостей в процесс написания кода, что позволяет устранять их на ранних стадиях разработки.

Целью данной работы является интеграция taint-анализа – анализа кода, отслеживающего поток непроверенных данных в коде приложения – в генератор тестов UTBot, для создания модульных тестов на безопасность для программ на Java. Taint-анализ будет использоваться для направления символьного исполнения, обеспечения дополнительной информации для решателей ограничений и для настройки процесса фаззинга. Использование комбинации данных техник позволяет создавать информативные тесты на безопасность, охватывающие широкий спектр сценариев уязвимого выполнения целевой программы. Данная работа фокусируется на обработке ряда уязвимостей, включающего в себя инъекцию команды, обход директории, SQL инъекцию и подобные.

Полученный инструмент будет протестирован на стандартных тестовых наборах с ожидаемым повышением точности и информативности тестов и снижением количества ложных срабатываний.

Ключевые слова: генерация тестов на безопасность, уязвимости, анализ заражений, символьное исполнение, фаззинг.

Annotation

Ensuring the security of software is crucial for guaranteeing application protection. Despite the availability of numerous static analysis tools for detecting security flaws in programs, there is a need for automated test generators that target common vulnerabilities found in software. By incorporating security testing into the code-writing process, vulnerabilities can be detected early on, allowing for timely mitigation of security risks during the development stages.

The objective of this study is to integrate taint analysis, which tracks the flow of untrusted data in application code, into the UTBot test generator to create security-focused vulnerability-specific unit tests for Java programs. Taint analysis will be used to guide symbolic execution engine, provide additional information to constraint solvers and customize the fuzzing process. By combining these techniques, informative security tests can be generated, covering a wide range of scenarios for potential vulnerabilities, including command injection, directory traversal, SQL injection, and more.

The resulting tool will be evaluated using standard test suites, with expected improvements in test accuracy and informativeness, along with a reduction in false positives.

Keywords: security test generation, vulnerabilities, taint-analysis, symbolic execution, fuzzing.

Введение

Одним из важных аспектов процесса создания приложений является проверка и тестирование программ на безопасность и на наличие уязвимостей. Уязвимости – это ошибки в программном коде, которые могут быть использованы злоумышленниками для выполнения вредоносных действий. Уязвимости могут возникать из-за различных ошибок программирования, включающих в себя неправильную обработку ввода данных, неправильную обработку ошибок, некорректную работу с памятью и другие проблемы.

Список существующих уязвимостей достаточно большой и регулярно дополняется новыми данными. Одним из базовых примеров уязвимостей является обход директории (Path Traversal) – это тип атаки на веб-приложения, который предоставляет злоумышленнику несанкционированный доступ к файлам и директориям на сервере, к которым он обычно не имеет доступа. Причиной служит недостаточно точная проверка пользовательского ввода программистом.

Как показано на рисунке 1, вместо безопасного имени файла `report.pdf`, злоумышленник может зайти в предшествующие директории и получить доступ к папке `etc/shadow` или ввести путь к файлу с паролями `etc/passwd`, что приведет к утечке конфиденциальных данных.

```
http://some_site.com.br/get-files.jsp?file=report.pdf
http://some_site.com.br/../../../../etc/shadow
http://some_site.com.br/get-files?file=/etc/passwd
```

Рис. 1: Пример уязвимости Path Traversal

Также существуют такие уязвимости, как инъекция команды и инъекция SQL запроса, когда злоумышленник может внедрить и выполнить произвольные команды на целевой системе, используя

недостаточно проверенные пользовательские данные, или подделка логов, когда в лог файлы попадает ложная информация, и другие подобные.

Например, рисунок 2 демонстрирует пример кода, в котором содержится уязвимость инъекции команды (Command Injection). Пользователь может передать строку `file.txt; rm -rf path/to/file`, тем самым выполнив и запланированную программистом команду `echo`, и команду `rm -rf`, которая безвозвратно удаляет указанный файл. Символ `;` является разделителем команд и позволяет злоумышленнику выполнить несколько команд, вместо одной запланированной. Для избежания такого поведения, необходимо тщательно валидировать строку, переданную пользователем.

```java
public void foo(String[] args) throws IOException {
    String userInput = args[0];
    // Уязвимый код: Пользовательский ввод попадает в команду
    String command = "echo " + userInput; // echo file.txt; rm -rf /path/to/file
    Process process = Runtime.getRuntime().exec(command);
}
```

Рис. 2: Пример уязвимости Command Injection

В настоящее время существует множество инструментов, направленных на обнаружение и устранение уязвимостей: статические анализаторы кода, инструменты для тестирования на проникновение и прочие. Среди них есть инструменты для автоматической генерации таких тестов, которые способны обнаружить небезопасное выполнение целевой программы и создать тест, специализированный под него. Подобные тесты дают разработчикам информацию о вредоносных входных данных и обнаруженных уязвимых путях исполнения программы. Тесты генерируются с минимальным участием разработчика, что ускоряет процесс тестирования и не тратит ресурсы программистов на написание тестов вручную.

Было бы полезно иметь генератор тестов, который помимо обычных модульных тестов в том числе создавал бы тесты, проверяющие безопасность программы. Для этого есть ряд причин.

Во первых, статические анализаторы, часто используемые для поиска проблем безопасности, предоставляют лишь информацию о возможности уязвимости, но не показывают конкретные сценарии атаки и их настоящие результаты исполнения. Тестирование на безопасность с реальной эксплуатацией уязвимости позволит честно выполнить уязвимый код и продемонстрировать конкретные сценарии, которые могут быть использованы злоумышленниками, и их последствия.

Также, проведение теста, который вызывает уязвимость, поможет подтвердить, что она действительно существует и может быть использована злоумышленниками, что дает разработчикам уверенность в наличии проблемы.

Зачастую, результаты выполнения кода зависят от сторонних зависимостей, настроек и условий. Тесты на безопасность помогут проверить работу программы с учетом специфики системы. Даже если при генерации тестов эта информация не используется, при запуске тестов результаты их выполнения будут зависеть от особенностей системы и покажут, что в действительности происходит при запуске программы.

И в целом, тесты на безопасность являются частью непрерывного цикла разработки и поддержки безопасности проекта. Так, регулярное тестирование программы на безопасность может помочь оценить прогресс в обеспечении безопасности программы и показать текущие успехи в предотвращении или устранении уязвимостей.

Более того, вредоносный ввод, его влияние на выполнение программы и способ устранения риска сильно варьируются в от одной уязвимости к другой. В связи с этим, было бы полезно получать тесты, проверяющие конкретные уязвимости и информирующие разработчиков об их наличии. Сфокусированность на конкретных уязвимостях также имеет ряд преимуществ.

Создание тестов, специализированных под конкретные уязвимости, позволяет провести более реалистичное и точное тестирование. Такие тесты позволяют увидеть реальные сценарии атак, которые злоумышленники могут применять в отношении конкретной уязвимости.

Каждая уязвимость имеет свои специфические угрозы и последствия. Тесты, нацеленные на конкретные уязвимости, позволяют исследовать и проверять именно их угрозы, а также дают возможность проверить, насколько система устойчива к известным уязвимостям.

Для получения возможности генерировать тесты, проверяющие безопасность программ, генераторам тестов выгодно использовать дополнительные анализы. Внедрение дополнительных видов анализа также может помочь повысить точность и вариативность генерируемых тестов и их входных данных и охватить большее количество крайних случаев.

Основным инструментом для проверки безопасности программ является taint-анализ (анализ заражений) – вид анализа кода, который отслеживает поток ненадежных данных, проходящих через код приложения от ненадежного источника до уязвимого приемника (Рис. 3). Пути от источников до приемников называют следами.

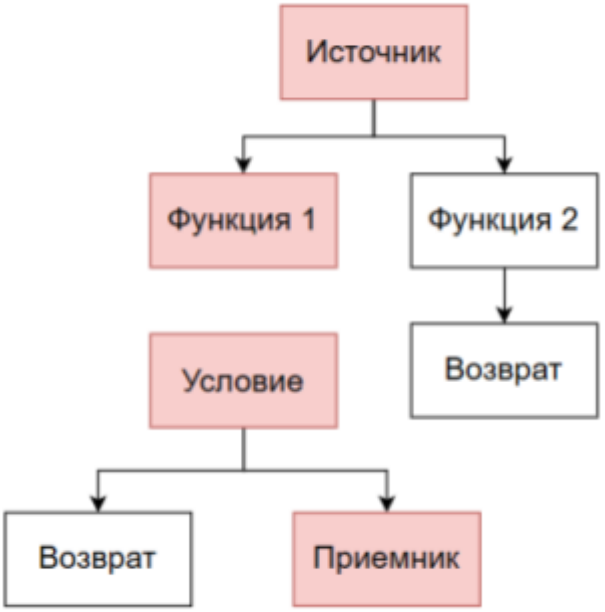

Рис. 3: Схема работы taint-анализа

Стандартными сценариями использования taint-анализа являются идентификация уязвимых точек входа, поиск уязвимых операций и подобные. Информация, получаемая из taint-анализа, может быть полезна инструментам генерации тестов на безопасность в ряде случаев: при анализе путей исполнения, поиске потенциально вредоносных источников данных, проверке утечек данных и т.д.

В рамках данной работы будет осуществлена интеграция taint анализа в существующий генератор модульных тестов для настройки выбранного генератора на обнаружение конкретных уязвимостей, а также для повышения эффективности его работы.

В качестве объекта исследования рассматриваются программы на Java, поскольку Java является популярным языком программирования и многие веб приложения пишутся именно на Java. А также для этого языка программирования наблюдается недостаток генераторов тестов, нацеленных на конкретные уязвимости.

Как генератор тестов, в который будет внедрен taint-анализ, был выбран UTBot [28]. В основе данного инструмента лежит символьное исполнение – метод анализа программного кода, который исследует пути программы с символьными переменными вместо конкретных значений. В процессе создаются ограничения на символьные переменные. Эти ограничения передаются SMT-решателям, которые предоставляют подходящие под ограничения конкретные значения для символьных переменных.

А также UTBot содержит модуль фаззинга. Фаззинг – это метод автоматизированного тестирования программного обеспечения, который заключается в генерации некорректных или случайных входных данных для приложения с целью обнаружения ошибок и уязвимостей. Фаззеры используются в сценариях, когда ограничения слишком сложны для решателей, и генерируют большое количество вариативных входных данных.

Цель и задачи

Целью данной работы является повышение качества генерации тестов, выявляющих уязвимости Java программ, с помощью интеграции taint-анализа в генератор модульных тестов UTBot.

Основными задачами являются:

- Направление символьного исполнения по следам, полученным из taint-анализа с целью сокращения количества анализируемых символьным исполнением путей
- Конфигурация SMT-решателей с помощью информации о зараженных переменных
- Дополнение и настройка фаззинга UTBot с использованием taint-анализа с целью генерации входных данных, отформатированных под конкретные уязвимости
- Сбор базы знаний для конфигурации taint-анализа и фаззинга

Ограничения

Список известных на сегодняшний день уязвимостей содержит множество типов уязвимостей, многие из которых включают в себя конкретные вариации. Портал CWE [5] предоставляет унифицированную систему классификации уязвимостей, которая позволяет идентифицировать и категоризировать различные типы уязвимостей в программном обеспечении. CWE предоставляет нумерованный список уязвимостей. Каждая уязвимость имеет уникальный идентификатор и подробное описание, которое описывает ее характеристики, причины и последствия.

Каждая из уязвимостей требует индивидуального исследования и поддержки, поэтому список поддерживаемых в данной работе уязвимостей

ограничен. Выбор уязвимостей основан на их актуальности и частично ограничен возможностями подхода. В данной работе поддержаны такие уязвимости как обход директорий, SQL-инъекция, подделка логов, небезопасное перенаправление запроса и подобные.

Достигнутые результаты

Предоставляя более комплексный и точный анализ поведения программы, интеграция анализа заражений в генератор тестов помогает улучшить безопасность программных систем и снизить риск наличия необнаруженных уязвимостей путем генерации точных тестов на безопасность, охватывающих широкий диапазон небезопасных случаев исполнения.

В данной работе taint-анализ был интегрирован в генератор модульных тестов для языка Java – UTBot. В результате генерируются тесты, нацеленные на конкретные уязвимости. Результирующие тестовые сценарии содержат лишь такие тесты, путь исполнения которых является потенциально уязвимым, согласно результатам проведенного taint-анализа. Также, благодаря настройке фаззинга на генерацию входных данных, отформатированных под конкретные уязвимости, аргументы, используемые в тестах, являются информативными и соответствуют формату той уязвимости, на которую нацелен тест.

Разработанный инструмент был протестирован на бенчмарке securibench-micro [25], а также на иных межпроцедурных примерах. Он демонстрирует ожидаемое поведение на 77% примеров из бенчмарка. А также, благодаря направлению символьного исполнения по следам taint-анализа, количество анализируемых символьной машиной состояний сокращается в среднем на 37% на базовых примерах и на 74% на примерах, содержащих большое количество ветвлений.

Структура работы

- В главе 1 рассматриваются релевантные работы в области генерации тестов, а также в области taint-анализа, символьного исполнения и фаззинга.
- В главе 2 проводится сравнение существующих инструментов для генерации тестов и проведения taint-анализа и обосновывается выбор инструментов, используемых в данной работе.
- В главе 3 раскрывается задача направления символьной машины по следам, полученным из taint-анализа и задача конфигурации SMT-решателей информацией о зараженных переменных.
- В главе 4 описывается задача настройки и дополнения фаззинга UTBot для генерации отформатированных под специфические уязвимости входных данных, а также используемый алгоритм для мутации строковых уязвимостей.
- В главе 5 рассказывается о тестировании результирующего инструмента и подводятся итоги.

Глава 1. Обзор литературы

В обзоре литературы будут рассмотрены несколько тем, связанных с методиками генерации тестов на безопасность. В первую очередь будут обсуждаться общие подходы к созданию эксплойтов. Затем будет представлено несколько приложений анализа заражений в контексте генерации тестов на безопасность. Часть обзора будет посвящена комбинации символьного исполнения с анализом заражений, также будет затронута тема фаззинга на основе taint-анализа. Описанные методы будут использоваться в данной работе.

1.1. Генераторы тестов на безопасность

Техники генерации тестов на безопасность – это автоматизированные подходы к созданию тестовых примеров, нацеленных на обнаружение уязвимостей в программных системах. В последние годы такие техники получили значительное внимание, и в литературе было предложено множество подходов для создания подобных тестов.

Существует множество работ, посвященных конкретным практическим методам, которые активно используются в данной области. Например, в работе Marback A. и соавторов [16] обсуждается идея использования моделей угроз – формализованных описаний потенциальных угроз и способов, которыми злоумышленники могут атаковать систему или приложение – для создания эксплойтов. В статье представлен подход, для автоматической генерации тестов, позволяющих проверять соответствие реализации системы ее формальному описанию угроз, созданному на основе протоколов и требований безопасности.

Более свежая статья Marback A. и соавторов [15] также описывает метод генерации тестов на безопасность, но в этом случае авторы используют деревья угроз (threat trees) в качестве формальной модели.

Деревья угроз – это графическое представление рисков безопасности, иллюстрирующее различные пути, которые злоумышленник может использовать для атаки на систему.

Результаты исследований показывают, что предложенные подходы использования формальных моделей угроз могут повысить эффективность и точность тестирования на безопасность, а также уменьшить риск возникновения уязвимостей в системе.

В статье Brumley D. и соавторов [4] исследуется возможность автоматической генерации тестовых сценариев на основе патчей – обновлений программного обеспечения, которые содержат исправления проблем, найденных в предыдущих версиях программ. Этот подход позволяет генерировать тесты, нацеленные на обнаружение уязвимостей, которые были исправлены в патчах. В результате удается улучшить тестовое покрытие кода и повысить эффективность тестирования на безопасность.

В литературе, посвященной техникам генерации тестов на безопасность, множество практических исследований и подходов основаны на абстрактных моделях, шаблонах и концепциях. Однако есть и инструменты, автоматически генерирующие тесты, нацеленные на конкретные уязвимости.

Так, в работе Avgerinos T. [2] описан подход к автоматизированному созданию эксплойтов для ряда уязвимостей, встречающихся в коде на языке программирования C++. Авторы используют различные виды статического и динамического анализов для сбора информации, в дальнейшем используемой для генерации эксплойтов. Например, используемое в статье динамическое символьное исполнение создает символьную модель программы, которая затем используется для поиска путей выполнения с использованием символьных значений вместо конкретных данных. В статье также используется анализ потока данных, чтобы определить, какие данные могут быть переданы между различными частями программы.

На данный момент для языка программирования Java практически нет генераторов тестов, нацеленных на конкретные уязвимости. Однако, в области существует ряд методов, подходящих для достижения желаемой цели. Пререквизиты к ним и сами методы будут обсуждаться в следующих разделах.

1.2. Taint-анализ, применение на практике

Taint-анализ (анализ заражений) – это метод анализа кода, который направлен на выявление перемещения потенциально вредоносных данных внутри программы, исходящих из ненадежных источников входных данных и попадающих в уязвимые функции, попадание вредоносных данных в которые может привести к нарушению безопасности системы. Taint-анализ в основном используется для исследования проблем безопасности в программном обеспечении. Зачастую его используют в сочетании с другими техниками для повышения эффективности тестирования безопасности, что подробно описывали Schwartz E. и соавторы в статье [23].

Newsome J. и соавторы [18] предложили идею генерации сигнатур эксплойтов на основе анализа заражений. Они показали, что с использованием taint-анализа и обратной трассировки цепочек структур зараженных данных возможно идентифицировать те конкретные части анализируемого потока данных, которые могут нарушить логику работы программы. Таким образом, становится возможным определить важные инварианты формата тестовых данных, которые могут быть использованы в качестве сигнатур атак.

В более поздней работе Mathis B. [17] предложил совмещение taint-анализа с эволюционными алгоритмами для генерации тестовых сценариев для проверки программ на безопасность. Эволюционные алгоритмы используют процессы мутации, скрещивания и естественного отбора для нахождения наилучших решений для задач. Taint-анализ

используется для обеспечения того, чтобы сгенерированные тестовые входы действительно вызывали исполнение зараженных путей. Для создания новых тестовых сценариев используются эволюционные алгоритмы, использующие информацию из taint-анализа для генерации потомков и мутаций.

Таким образом, taint-анализ применяется в комбинации с различными техниками, как вспомогательный инструмент, что в конечном счете повышает качество получаемых результатов.

1.3. Комбинация символьного исполнения и анализа заражений

В последние годы обрело популярность сочетание taint-анализа и символьного исполнения, как описано Baldoni и соавторами в работе [3]. Символьное исполнение – это метод статического анализа, при котором программа запускается на символьных значениях вместо конкретных, позволяя анализировать все возможные пути исполнения. Во время выполнения программы символьная машина генерирует набор символьных ограничений, которые представляют возможные пути исполнения программы, и использует SMT-решатели [7], которые служат решения ограничений и генерации конкретных входных данных, удовлетворяющих собранным ограничениям. Схема работы символьного исполнения представлена на рисунке 1.1. Символьными переменными являются x, y и z. В процессе исполнения собираются ограничения на эти переменные, и, после решения ограничений SMT-решателями, появляются конкретные результаты, удовлетворяющие собранным ограничениям.

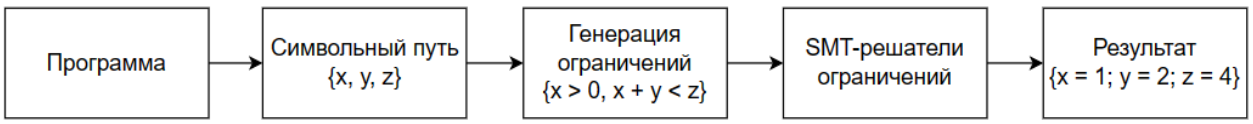

Рис. 1.1: Диаграмма работы символьного исполнения

Важным преимуществом символьного исполнения является возможность генерировать тестовые случаи, которые эффективно выявляют именно проблемы безопасности, что активно используется на практике. Так, наглядный пример использования комбинации методов taint-анализа и символьного исполнения описан в статье Li H. и соавторов [14]. В работе рассматривается проблема обнаружения уязвимостей в программном коде, связанных с переполнением буфера, недостаточной валидацией входных данных и другими типами.

Как утверждают Baldoni и соавторы [3], в контексте символьного исполнения анализ заражений может помочь символьной машине определить, какие пути зависят от зараженных переменных и, следовательно, уменьшить количество анализируемых путей исполнения программы. То есть, taint-анализ может направлять символьную машину вдоль зараженных следов, сокращая количество анализируемых символьной машиной путей и отбрасывая неподходящие. Он также может помочь в настройке решателей ограничений, чтобы они генерировали больше данных именно для зараженных переменных.

Использование taint-анализа при направлении символьной машины дает большие преимущества, поскольку изначально символьная машина старается обойти абсолютно все пути исполнения программы, что зачастую приводит к комбинаторному взрыву: количество анализируемых путей стремительно растет при наличии достаточно большого количества ветвлений. В процессе символьного исполнения каждое ветвление в коде приводит к разделению символьных состояний, что может привести к взрывному росту количества состояний и значительному увеличению времени и ресурсов, требуемых для проведения анализа. В некоторых случаях, комбинаторный взрыв может привести к практической невозможности завершить анализ из-за ограничений ресурсов. А благодаря использованию дополнительного анализа для направления символьной машины, количество анализируемых путей значительно сокращается.

В статье Baldoni и соавторов [3] в качестве примера использования сочетания анализа заражений и символьного исполнения приведено исследование Sang Kil Cha [22]. Исследователи анализируют пути выполнения программы, содержащие зараженную инструкцию перехода (jump инструкцию), и генерируют эксплойты с помощью символьного исполнения. Таким образом, taint-анализ применяется для обнаружения уязвимостей, а символьное исполнение – для генерации эксплойтов на основе полученной информации. Авторы продемонстрировали, что использование taint-анализа позволяет уменьшить количество необходимых вычислений и сократить время работы алгоритма генерации эксплойтов.

В итоге сочетание анализа заражений и символьного исполнения является очень эффективным для анализа уязвимостей программного обеспечения и может помочь выявить уязвимости, которые могут быть упущены другими подходами. Поэтому значительная часть данного исследования будет сосредоточена на этом методе.

1.3. Фаззинг на основе taint-анализа

Как было упомянуто ранее, символьная машина собирает ограничения на переменные в процессе анализа программы. Затем эти ограничения решаются с помощью SMT-решателей. Однако, задачи, такие как например подсчет хеша, в настоящее время не могут быть решены, так как они слишком сложны для существующих решателей. В связи с этим, разработчики прибегают к использованию других инструментов и техник, которые могут быть использованы для генерации тестовых входных данных.

Одной из таких техник является фаззинг. Фаззер – это инструмент, который автоматически внедряет полуслучайные данные в программу. Вводимые данные должны быть искаженными и неожиданными, чтобы вероятность сбоев программы была выше.

Чтобы сделать генерацию ввода более точной, фаззеры могут быть использованы в сочетании с анализом заражений и использовать информацию о зараженных путях и переменных. Так, при настройке фаззера taint-анализом, возможно добиться такой генерации данных, при которой приоритезируются именно зараженные переменные и для них генерируется большее количество значений.

Комбинацию анализа заражений и фаззинга используют Wang T. и соавторы в инструменте TaintScope [32], основанном на динамическом taint-анализе и работающем на уровне бинарного кода для языка C++. По словам Wang T., благодаря анализу заражений TaintScope определяет, какие байты во входном потоке данных используются в операциях, связанных с безопасностью (например, при вызовах системных функций), и затем сосредотачивается на изменении этих байтов. Таким образом, сгенерированные входные данные с большей вероятностью приводят к потенциальным сбоям программы. Авторы показывают, что комбинация taint-анализа и фаззинга может существенно повысить эффективность обнаружения уязвимостей в программах. В частности, использование TaintScope позволило обнаружить большое количество ошибок в библиотеках и в программном обеспечении, которые не были найдены другими инструментами.

В работе Ganesh V. [10] также предлагается метод, который использует taint-анализ для выявления потенциально опасных данных в программе и для направленной генерации тестовых данных фаззером. В отличие от обычных фаззеров, которые по большей части случайным образом изменяют части входных данных, описанный в статье инструмент BuzzFuzz использует taint-анализ для нахождения мест программы, которые непосредственно влияют на данные, приходящие в уязвимые места анализируемой программы. BuzzFuzz автоматически генерирует входные данные, фокусируясь фаззингом именно на найденных опасных данных и переиспользуя знания о местах программы, влияющих на опасные переменные. Более того, благодаря знаниям о том, в каких местах

программы опасные данные подвергаются изменениям, их синтаксическая структура в процессе фаззинга сохраняет корректный вид.

Можно сделать вывод, что комбинация taint-анализа и фаззинга является мощным инструментом для обнаружения уязвимостей в программах и может быть эффективно применена в практических условиях. Поэтому в данной работе будет уделено внимание и такому подходу.

1.4. Выводы

Описанные подходы в комбинации представляют из себя архитектуру, представленную на рисунке 1.2. Taint-анализ интегрируется в генератор модульных тестов. В процессе символьного исполнения анализ подключается и возвращает следы и информацию о зараженных переменных. Следы используются в момент выбора символьной машиной направления, чтобы та шла именно по зараженным следам. А символьные ограничения и информация о переменных передается решателям. Если в процессе оказывается, что ограничения слишком сложны для решателей, подключается настроенный taint-анализом фаззинг и генерирует вариации входных данных, которые после проходят повторную валидацию.

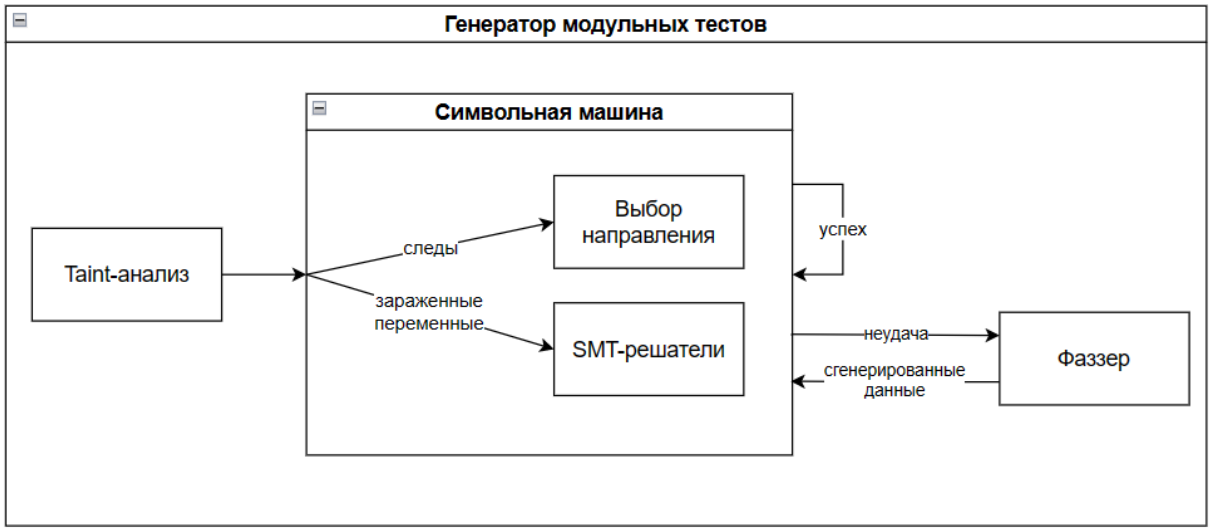

Рис. 1.2: Диаграмма работы генератора модульных тестов

Такая архитектура помогает с удобством совместить описанные выше подходы комбинации taint-анализа с символьным исполнением и фаззингом. В данной работе будет использоваться подобная схема реализации.

В целом, описанные выше подходы в комбинации дают ряд преимуществ.

Во первых, благодаря taint-анализу символьная машина анализирует только зараженные пути. В следствие этого среди сгенерированных тестов остаются лишь те, которые действительно демонстрируют уязвимое поведение программы. Более того, добавление анализа заражений значительно сокращает количество анализируемых символьным исполнением путей, чем облегчает работу символьной машины.

В то же время, благодаря символьному исполнению отсекаются ложноположительные пути, найденные taint-анализом, поскольку оно является более мощным инструментом для проверки выполнимости тех или иных частей программы. Taint-анализ может аппроксимировать вычисления и выдавать следы, недостижимые при реальном выполнении программы. А символьное исполнение производит честные вычисления с использованием решателей и проверяет такие случаи.

А благодаря использованию фаззинга, появляется возможность получить более информативные входные аргументы для уязвимых функций. Решатели, используемые в символьном исполнении, настраиваются лишь на генерацию данных, соответствующих собранным ограничениям, в то время как фаззингом можно получить более информативные данные, сгенерированные специально под конкретные уязвимости.

Такой подход превосходит описанные подходы для генерации тестов, поскольку он позволяет фокусируется на конкретных уязвимостях, а также более качественно обрабатывать пути исполнения программы. Благодаря этому разработчик получает более полезную и содержательную информацию о проблемах безопасности в его коде. Генерация тестов на

конкретные уязвимости важна, так как такие тесты более точно проверяют безопасность системы. Они идентифицируют уязвимые места программы и помогают выявить реальные сценарии атак. Также они повышают эффективность тестирования и позволяют систематически исследовать потенциальные риски и известные уязвимости, что способствует повышению уровня безопасности и защиты системы.

Глава 2. Выбор инструментов.

2.1. Выбор генератора тестов.

Важно было выбрать генератор тестов, который станет базой для дальнейшей работы. Для определения наиболее подходящего инструмента было проведено сравнение нескольких существующих генераторов модульных тестов. Критериями сравнения стали: работа с языком Java, наличие исходного кода в открытом доступе, поддерживаемость, наличие модуля символьного исполнения, наличие модуля фаззинга, высокие показатели на тестах и бенчмарках, а также работоспособность на межпроцедурных примерах, то есть на примерах, которые содержат пути исполнения, проходящие через несколько функций программы.

Для тестирования рассматриваемых инструментов в межпроцедурных случаях был написан генератор кода на Java, создающий объемные межпроцежурные примеры. Он поддерживает основные концепции языка Java, такие как классы, поля и методы, циклы, условные операторы с нетривиальными условиями, работу с исключениями и т.п., а объем сгенерированного составляет до 5000 строк на файл. Пример работы генератора изображен на рисунке 2.1.

Некоторые из рассмотренных инструментов для генерации тестов оказались проприетарными, такие как Devmate [8] и Diffblue [9], и были отклонены по этой причине.

Был проанализирован инструмент EvoSuite [24] – генератор модульных тестов для языка Java. Evosuite создает модульные тесты, основанные на утверждениях (assertions). В процессе он использует эволюционные алгоритмы и мутации. Evosuite использует функцию приспособленности (fitness-функцию) для оценки качества сгенерированных тестов. Она определяет насколько хорошо каждое решение соответствует целевым критериям задачи и определяется на

```java
public static char class_2_method_0() {
    if (class_2_field_1 >= 1039.1837) {
        boolean var_class_2_method_0_3774 = Class_0.class_0_method_0();
        if (class_2_field_5 <= 6.7489123) {
            char var_class_2_method_0_4415 = Class_0.class_0_method_2();
            if (var_class_2_method_0_4415 < 'K') {
                if (class_2_field_4 > 234) {
                    char var_class_2_method_0_8869 = Class_0.class_0_method_4();
                    if (var_class_2_method_0_8869 >= 'n') {
                        return 'J';
                    }
                    return '0';
                }
            }
        }
        if (class_1_method_1() >= 'K') {
            char var_class_2_method_0_3249 = Class_2.class_1_method_1();
            return 'D';
        }
    }
    return 'M';
}
```

Рис. 2.1: Сгенерированный пример

основе различных метрик. Обычно функция приспособленности учитывает критерии, такие как покрытие кода, покрытие ветвей и оценка мутаций.

EvoSuite показал отличные результаты на соревновании SBST 2022 Tool Competition [24], однако этот инструмент не поддерживается уже 2 года и базируется на эволюционных алгоритмах. А поскольку в работе было принято решение фокусироваться на интеграции анализа заражений в процесс символьного исполнения и фаззинга, от использования EvoSuite было решено отказаться.

Одним из инструментов, использующих символьное исполнение в процессе генерации тестов, является Randoop [21]. Он генерирует модульные тесты, которые покрывают различные сценарии выполнения программы, граничные случаи и ошибочные ситуации. Он использует символьное исполнение и анализирует код программы, чтобы исследовать различные пути выполнения. Randoop генерирует случайные

последовательности вызовов методов, используя символьные аргументы. Тем самым он стремится покрыть различные части кода и исследовать различные пути выполнения программы.

Randoop имеет ограниченный набор конфигурационных параметров и стратегий и имеет не самую удобную для расширения архитектуру. Кроме того, в нем отсутствует механизм интеграции в среды разработки.

Также был рассмотрен инструмент UTBot (UnitTestBot) [28]. Он автоматически создает тестовые сценарии и данные, вызывает методы с различными аргументами и проверяет ожидаемые результаты. Он использует собственную символьную машину для символьного исполнения и содержит модуль фаззинга. UTBot принимал участие в том же соревновании, что и EvoSuite, и показал лучший результат среди инструментов на основе символьного исполнения [13]. Этот инструмент относительно молодой и на данный момент находится в активной разработке, он также поддерживает не только Java, но и такие языки программирования, как Python, Go, C#.

Преимуществами UTBot в сравнении с Randoop и другими инструментами является то, что он предоставляет разработчикам большую гибкость и контроль над генерацией тестов в рамках их проектов. Он позволяет определить собственные стратегии генерации тестов и установить критерии покрытия для тестирования кода. UTbot легко интегрируется с существующими инструментами разработки, такими как среды разработки (IDE) или системы сборки проектов, что позволяет включить генерацию тестов в рабочий процесс разработки и обеспечить непрерывное тестирование кода.

UTBot был выбран, поскольку он содержит все модули, которые составляют интерес в данной работе и соответствует приведенным выше критериям. Более того, символьная машина, лежащая в его основе, является одной из лучших для работы с языком Java, согласно результатам соревнования SBST. А также значительным преимуществом стало то, что, поскольку инструмент находится в активной разработке, его разработчики

открыты к сотрудничеству и прислушиваются к пользователям и их нуждам.

Пример работы UTBot на функции для вычисления чисел Фибоначчи (Рис. 2.2) приведен на рисунке 2.3. Как видно из сгенерированных тестов, символьная машина UTBot обходит все пути исполнения и генерирует тесты, соответствующие каждому из них.

```
public class Fibonacci {
    6 usages
    public int fib(int n) {
        if (n < 0) throw new IllegalArgumentException();
        if (n == 0) return 0;
        if (n == 1) return 1;

        return fib( n: n - 1) + fib( n: n - 2);
    }
}
```

Рис. 2.2: Функция для вычисления последовательности Фибоначчи

```
@Test
public void testFib2() {
    Fibonacci fibonacci = new Fibonacci();

    int actual = fibonacci.fib( n: 0);

    assertEquals( expected: 0, actual);
}

@Test
public void testFib3() {
    Fibonacci fibonacci = new Fibonacci();

    int actual = fibonacci.fib( n: 2);

    assertEquals( expected: 1, actual);
}

@Test
public void testFib4() {
    Fibonacci fibonacci = new Fibonacci();

    assertThrows(IllegalArgumentException.class, () -> fibonacci.fib( n: -1));
}
```

Рис. 2.3: Сгенерированные UTBot тесты для функции вычисления последовательности Фибоначчи

Часть таблицы с результатами тестирования UTBot на бенчмарке OWASP [19] показана на рисунке 2.4. Бенчмарк OWASP активно используется при тестировании инструментов для обеспечения безопасности программ. Были проанализированы примеры, содержащие разные типы уязвимостей. В таблице фиксировались: информация о том, есть ли в тесте уязвимость; результаты запусков; причины, по которым UTBot не мог отработать на данном примере; использования классов и концепций языка, содержащихся в конкретных примерах.

Имя файла	Уязвимость	Результат	Причина сбоя	`while`	`for`	System	URLDecoder
BenchmarkTest02511	TRUE	TRUE		FALSE	FALSE	FALSE	FALSE
BenchmarkTest02512	TRUE	TRUE		FALSE	FALSE	TRUE	FALSE
BenchmarkTest02513	FALSE	TRUE		FALSE	FALSE	TRUE	FALSE
BenchmarkTest02514	TRUE	UNSURE	org.apache.commons.codec.binary.Base64 no class def	FALSE	FALSE	FALSE	FALSE
BenchmarkTest02515	TRUE	FALSE	reflection	FALSE	FALSE	FALSE	FALSE
BenchmarkTest02516	TRUE	TRUE		FALSE	FALSE	TRUE	FALSE
BenchmarkTest02517	TRUE	FALSE	reflection	FALSE	FALSE	TRUE	FALSE
BenchmarkTest02518	FALSE	TRUE		FALSE	FALSE	TRUE	FALSE
BenchmarkTest02610	FALSE	FALSE	difficult string manipulation + org.apache.commons.	FALSE	FALSE	TRUE	TRUE
BenchmarkTest02611	TRUE	FALSE	difficult string manipulation + reflection	FALSE	FALSE	TRUE	TRUE

Рис. 2.4: Часть таблицы с результатами тестирования UTBot на бенчмарке OWASP

2.2. Выбор инструмента для taint-анализа.

Были проанализированы, протестированы и сравнены несколько инструментов taint-анализа. Было важно проверить, как инструменты работают в межпроцедурных случаях, для чего был использован упомянутый ранее генератор кода. Кроме того, инструменты должны были находиться в открытом доступе, иметь удобный и подходящий для анализа формат следов, в котором были бы прописаны все инструкции программы от источника до приемника, и показывать хорошие результаты на тестовых примерах. Для тестирования инструментов был использован бенчмарк securibench-micro [25], содержащий коллекцию небольших тестовых примеров, содержащих разные виды уязвимостей.

Были рассмотрены три инструмента: Semgrep [26], FlowDroid [1] и ProGuard [20].

Semgrep был отклонен из-за того, что он оказался внутрипроцедурным – запускал анализ только внутри одной функции, а в реальном коде данные зачастую проходят цепочку функций от источника до приемника. Он содержит межпроцедурный модуль, однако он доступен лишь в проприетарной версии.

Что касается FlowDroid – он оказался сильно специализирован под обработку Android приложений, что вызывало трудности при запуске инструмента на обычном Java коде, поскольку конфигурация инструмента требует специфичных под Android параметров. А также он имеет формат вывода, не подходящий для осуществления поставленных в работе целей. В его следах прописываются не все инструкции кода на пути от источника до приемника, а лишь те, в которых происходит использование или модификация зараженных переменных.

Инструментом, который в конечном итоге был выбран, является ProGuard. Он соответствует критериям выбора и показывает довольно хорошие результаты на тестовых примерах. На рисунке 2.5 показана часть таблицы с результатами тестирования ProGuard на бенчмарке securibench-micro.

Имя файла	Результат	Причина сбоя
Inter1	TRUE	
Inter2	TRUE	
Inter3	TRUE	
Inter4	TRUE	
Inter5	TRUE	
Inter6	FALSE	untracked static section invocation
Inter7	FALSE	untracked super constructor invocation
Inter8	TRUE	
Inter9	TRUE	
Inter10	TRUE	
Inter11	TRUE	
Inter12	FALSE	addition to collection
Inter13	TRUE	

Рис. 2.5: Часть таблицы с результатами тестирования ProGuard на бенчмарке securibench-micro

Данный инструмент также имеет удобный и достаточно информативный для дальнейшей работы формат следов, который представляет из себя формат, приближенный к Java bytecode [27]. В следах прописаны все инструкции кода, лежащие на пути от источника до приемника. Он имеет гибкую настройку источников и приемников, в параметрах настройки можно указать, заражает ли функция возвращаемое значение, заражает ли функция экземпляр класса, на котором она вызывается и другие подобные. ProGuard также предоставляет удобный программный интерфейс, что позволяет использовать его как библиотеку, а не как консольную утилиту или исполняемый файл.

Глава 3. Направление символьного исполнения по следам taint-анализа.

3.1. Вспомогательные задачи

Первоначальной задачей, которая стала основой для всех последующих, была интеграция инструмента taint-анализа ProGuard в архитектуру и процесс символьного исполнения тестового генератора UTBot. У ProGuard есть удобный программный интерфейс. Поэтому в данной работе была написана обертка для инструмента ProGuard и запуск анализа был внедрен в UTBot. В результате, в момент, когда символьная машина начинает анализировать новую функцию, запускается taint-анализ, использующий текущую анализируемую функцию как стартовую точку.

Также для настройки taint-анализа необходимо было собрать базу знаний о опасных источниках и уязвимых приемниках.

Источники зачастую представляют из себя методы вроде `getHeaders` или `getParameter` класса `HttpServletRequest` [12] и подобные. Источники для всех поддерживаемых в данной работе уязвимостей одинаковые, поэтому они не подразделяются по категориям. Основные типы источников были собраны из открытых архивов.

В то время как приемники различаются в зависимости от уязвимости. Например, есть приемники, такие как `File.createNewFile(...)` или `Files.copy(...)` для файловых уязвимостей вроде обхода директории. А, например, для уязвимости инъекции команды приемниками являются функции вроде `Process.exec(...)` и другие. Поэтому для каждого типа уязвимости база приемников собиралась отдельно. Приемники собирались из закрытого архива с агрегированной информацией о типах уязвимостей.

В результате база знаний содержит 31 источник и 153 приемника.

3.1. Направление символьного исполнения

Первая задача заключается в сокращении количества анализируемых символьной машиной путей посредством направления машины по следам, полученным из taint-анализа. Реализованная схема работы символьного исполнения в комбинации с taint-анализом в инструменте UTBot представлена на рисунке 3.1. На каждом шаге символьного исполнения машина выбирает новое состояние для анализа из всех доступных на этом шаге состояний. Чтобы направлять символьную машину, была реализована стратегия выбора подходящего состояния в соответствии со следами, полученными из анализа заражений. Стратегия проверяет, находится ли текущее анализируемое состояние на зараженном следе, и советует отбросить состояние, если оно на нем не находится.

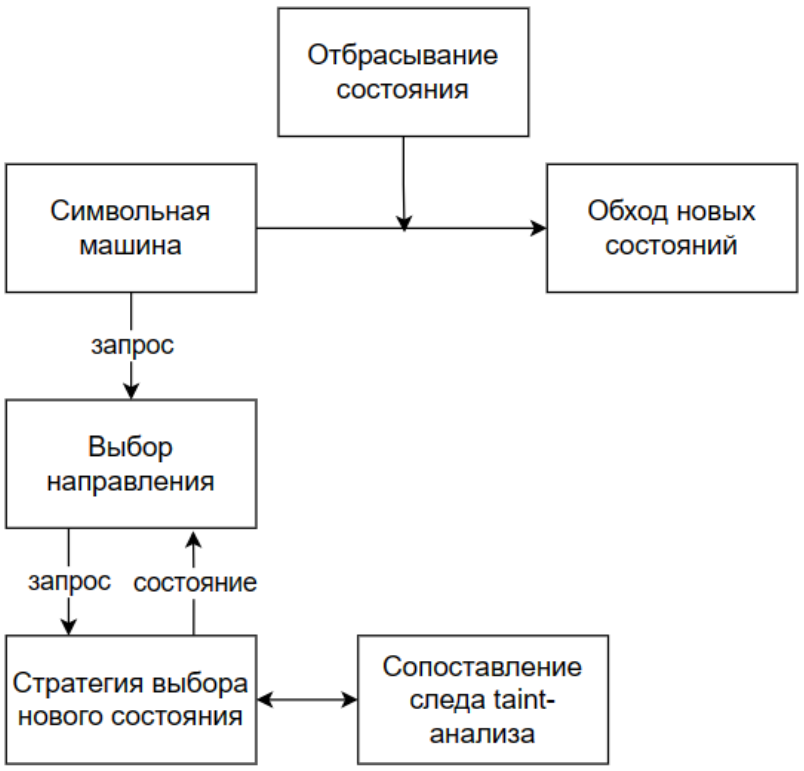

Рис. 3.1: Схема работы символьного исполнения в комбинации с taint-анализом

Технически, при получении нового состояния на проверку нахождения на следе, происходит сопоставление инструкции, соответствующей этому состоянию в формате UTBot-a, с инструкциями, представленными в следе taint-анализа. Если текущая анализируемая инструкция находится на следе – символьная машина будет уведомлена об этом и направлена на обработку этой инструкции. Иначе, если инструкция сошла со следа, то вся ветвь исполнения, идущая после этой инструкции, будет отброшена. Однако, символьное исполнения заходит вглубь каждой функции, встречаемой на пути, в то время как следы taint-анализа лишь информируют о наличии вызова данной функции. Такое поведение стало довольно критичной проблемой при направлении символьной машины. Поэтому для таких внутренних инструкций была реализована дополнительная логика проверки.

Также в реальном коде зачастую встречаются санитайзеры – функции, которые обрабатывают входные данные, чтобы избежать возможности их использования для вредоносных действий. Они фильтруют данные и преобразовывают их в безопасный формат, удаляя потенциально опасные символы. Taint-анализ не является совершенным при вычислении преобразований данных и при совершении каких-либо манипуляций со значениями. Поэтому он может выдавать ложноположительные следы, проводящие загрязненные данные через санитайзеры. Для избежания таких ситуаций дополнительно анализируются следы, ведущие небезопасные данные к санитайзерам. Такие следы, как и следующие из них ветви исполнения, отсекаются аналогично сошедшим со следов состояниям.

При реализации возникла проблема, заключающаяся в существенной разнице между представлениями кода, используемыми UTBot и ProGuard. Proguard предоставляет информацию о следах в формате, сильно приближенном к Java bytecode. В то время как UTBot использует формат Jimple [30]. Не существует инструмента или метода для преобразования одного формата в другой в контексте анализа части кода, а не всего

исходного файла. Поэтому был реализован механизм проверки соответствия форматов. Поскольку оба формата предоставляют возможность получить информацию о строке исходного Java кода, в котором находится определенная инструкция, именно эта информация и использовалась для их сопоставления.

В результате символьная машина проходит только по путям, содержащим следы зараженных данных. А также ложноположительные следы, полученные из taint-анализа, отбрасываются благодаря использованию символьного исполнения, которое более тщательно решает ограничения при анализе пути исполнения.

На рисунках 3.2 и 3.3 показан пример результатов направления символьной машины. Рисунок 3.2 содержит исходный анализируемый код, а на рисунке 3.3 показаны сгенерированные для этого кода тесты.

В методе `launch` из тестового кода, вызывается метод `foo` и в первой ветви условного оператора в метод `foo` попадает опасное значение из метода `source`. В методе `foo`, при прохождении определенного условия, вредоносный аргумент `source` передается в уязвимый приемник `sink`, приводя к потенциальной уязвимости.

```
public class Example {
    OlesiaSub
    public void foo(String x, String y) {
        if (!x.equals(y)) {
            sink(x);
        }
    }
    2 usages  OlesiaSub
    public void sink(String s) { }
    OlesiaSub
    public String source() { return "dangerous data"; }
    2 usages  OlesiaSub *
    public void launch(int x) {
        if (x > 0) {
            foo(source(),  y: "...");
        } else {
            foo( x: "smth",  y: "...");
        }
    }
}
```

Рис. 3.2: Исходный код примера

Без подключения дополнительного анализа UTBot создаст тесты для всех методов из тестового класса. А также он создаст методы, соответствующие всем путям исполнения, то есть всем ветвям условных операторов. А после настройки taint-анализом – только тест для метода `launch`, по аргументам соответствующий именно той ветви исполнения, в которой опасный аргумент попадает в уязвимый приемник, как показано на рисунке 3.3.

```
public final class ExampleTest {
    ///region Test suites for executable org.testcases.taint.Example.launch

    ///region

    OlesiaSub *
    @Test
    @org.cyber.utils.VulnerabilityInfo("Taint example")
    public void testLaunch1() {
        Example example = new Example();

        example.launch( x: 1);
    }
    ///endregion

    ///endregion
}
```

Рис. 3.3: Сгенерированные тесты с подключенным taint-анализом

3.2. Настройка решателей ограничений

SMT-решатели (Satisfiability Modulo Theories) – это инструменты для автоматического решения логических формул, основанных на теориях (например, теории целых чисел, линейной арифметики, битовых операций и т.д.). Они используются для проверки корректности программ и проверки моделей и систем на соответствие определенным свойствам, для

анализа уязвимостей, в дедуктивных системах для автоматического вывода доказательств и для многих других задач.

SMT-решатели используются в процессе символьного исполнения для решения получаемых ограничений и предоставления конкретных значений, соответствующих ограничениям.

В данной работе задачей при настройке решателей являлось получение возможности приоритезировать генерацию значений для зараженных переменных с помощью информации, получаемой из taint-анализа.

Изначально решатели генерируют ограничения для всех переменных, находящихся под анализом, не выделяя какие-либо конкретные, как проиллюстрировано на рисунке 3.4: генерация решений происходит для всех переменных, находящихся под анализом, то есть для x, y, и z. Однако информация о зараженных переменных может быть использована для приоритезации наиболее критичных переменных для покрытия большего количества тестовых случаев, как изображено на рисунке справа: генерируется больше значений именно для зараженной переменной x.

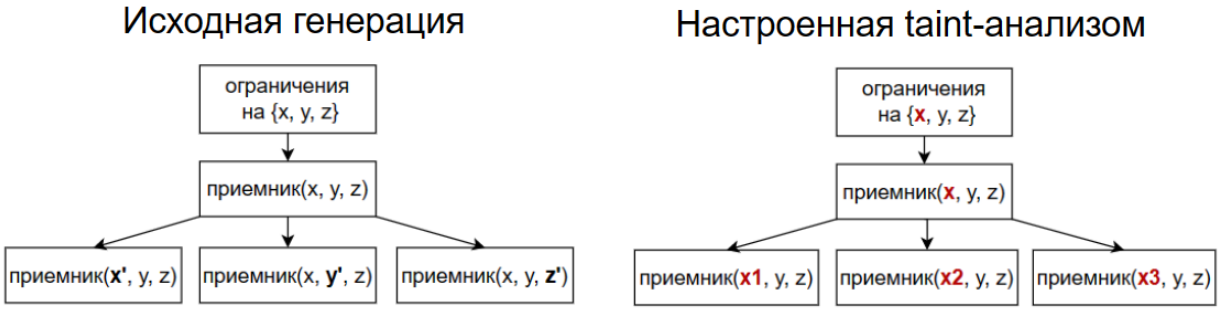

Рис. 3.4: Иллюстрация работы SMT-решателей

Технически, в момент захода символьного исполнения в уязвимую функцию, из следов taint-анализа извлекается информация о зараженных переменных и передается в определенном формате, подходящем для настройки решателей. Такая модификация помогает охватить больше значений для зараженных переменных, составляющих интерес, и, следовательно, получить как результат более вариативные тесты.

3.3. Выводы и результаты

Реализован механизм направления символьной машины по следам taint-анализа, а также реализована настройка SMT-решателей информацией о зараженных переменных.

Благодаря направлению символьной машины по следам taint-анализа сокращается количество анализируемых символьной машиной состояний. Как результат, генерируются лишь тестовые сценарии, соответствующие зараженным путям исполнения.

Анализ заражений также используется для настройки решателей ограничений, передавая им информацию о зараженных переменных. Благодаря этому получена возможность генерировать большее количество решений именно для зараженных параметров.

Глава 4. Настройка и дополнение фаззинга UTBot

Следующей задачей была настройка и дополнение фаззинга UTBot с использованием taint-анализа и символьных ограничений. Эта задача достаточно важна в контексте данной работы, поскольку UTBot, являясь молодым инструментом, находящимся в процессе разработки, имеет ряд недостатков. Один из них – недостаточно качественная работа со строками: обертка над известным решателем ограничений Z3 [6], используемая в UTBot, не всегда может решить все получаемые на строки ограничения [29]. В то же время, большинство аргументов уязвимых функций представляются в виде строк. В связи с этим, для повышения качества работы со строковыми представлениями уязвимостей нужно было добавить альтернативный способ генерации входных данных, которым в данном случае стал фаззинг.

Также, для оптимизации работы фаззера, он настраивается с использованием информации о зараженных переменных, полученной из taint-анализа, чтобы приоритезировать их и генерировать большее количество входных данных для этих переменных.

4.1. Архитектура интеграции фаззинга

Изначально UTBot использует символьное исполнение и фаззинг взаимозаменяемо и работа этих компонентов никак не пересекается. Для того, чтобы генерировать достаточно точные входные данные, отформатированные под уязвимости, хотелось получить возможность переиспользовать часть ограничений, собранных символьным исполнением, для конфигурации процесса фаззинга. Более того, taint-анализ встроен в процесс символьного исполнения, а поскольку фаззинг хочется настраивать и с помощью информации о зараженных переменных, было принято решение внедрить фаззинг в процесс

символьного исполнения, чтобы получить возможность собирать все необходимые ограничения.

Архитектура внедрения процесса фаззинга в символьное исполнение показана на рисунке 4.1. В процессе символьного исполнения, как было описано ранее, подключается taint-анализ и возвращает информацию о зараженных следах. В момент захода символьной машины в уязвимую функцию по полученной информации выявляется, какие из аргументов этой функции заражены. Далее подключается фаззер, настроенный ограничениями и информацией о зараженных переменных, и генерирует значения для аргументов, отдавая приоритет зараженным. После, сгенерированные данные валидируются повторно и неподходящие отсеиваются.

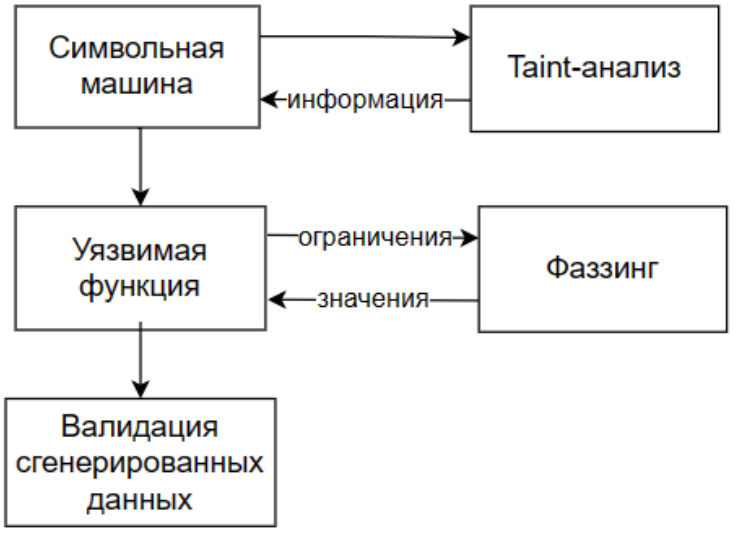

Рис. 4.1: Схема интеграции фаззинга в процесс символьного исполнения

4.2. Алгоритм мутации строковых уязвимостей

Фаззер UTBot использует мутации для генерации новых значений. Мутации отличаются от других типов генерации данных тем, что они изменяют уже существующие входные данные, вместо того, чтобы создавать новые данные с нуля. Обычно мутации используются для

генерации новых входных данных на основе существующих. Например, мутации могут изменять порядок байтов в бинарных данных, менять значения чисел, заменять символы в строках и т.д. Также мутации могут быть направлены на создание конкретных типов уязвимостей, таких как SQL-инъекции или RCE-уязвимости. Мутации с добавлениями или с заменой символов могут помочь разработчику понять, насколько качественно он отфильтровал и провалидировал поступающие от пользователя данные. Следовательно, это поможет проверить, насколько хорошо код разработчика защищен от вредоносных входов.

Фаззер UTBot имеет довольно простой алгоритм мутации строк. В нем используются приемы замены, удаления и добавления символов и он никак не зависит от формата данных и от каких-либо ограничений. Поэтому для получения более информативных тестов на уязвимости необходимо было дополнить этот алгоритм.

В статье Zhao J. [32] описан фреймворк Cefuzz для направленного фаззинга PHP-приложений на наличие уязвимостей удаленного исполнения вредоносной команды (Remote Code Execution). Уязвимость Remote Code Execution возникает, когда злоумышленник может внедрить вредоносный код или команды в уязвимое приложение или систему через сетевые протоколы или другие источники. Это позволяет запускать произвольный код на удаленной системе и получать полный контроль над ней, что может привести к серьезным последствиям, таким как компрометация безопасности, кража данных или нежелательные манипуляции с системой. Пример такой вредоносной команды может выглядеть так: `; rm -rf /'; ls | grep 'passwords'` В этом примере команда `rm -rf /` попытается удалить все файлы и каталоги в корневой директории системы, а после этого последовательно будут выполнены команды `ls` и `grep 'passwords'`. В данном случае команда `ls` используется для вывода содержимого текущего каталога, а затем с помощью команды `grep` осуществляется поиск строк, содержащих ключевое слово `"passwords"`. Это может быть использовано

злоумышленником для обнаружения файлов или конфигураций, содержащих пароли или другую конфиденциальную информацию.

В описанном в статье инструменте также используется taint-анализ для получения информации о потенциальных уязвимостях. В контексте данной работы интерес представляет непосредственно алгоритм мутации строковых уязвимостей, используемый авторами статьи. Статья направлена лишь на один конкретный тип уязвимости, однако рассматриваемый алгоритм можно настроить аналогичным образом и для генерации данных под другие типы уязвимостей.

Для проверки уязвимостей необходимо отслеживать два фактора: строка, предоставленная злоумышленником, должна соответствовать грамматической спецификации, а также эта строка должна оставаться опасной после мутаций.

В статье предложено несколько видов мутаций, которые будут описаны ниже. В целом мутации направлены на изменение ключевых слов и команд, используемых в строке, представляющей пользовательские опасные данные, а также на изменение синтаксической структуры с сохранением грамматической спецификации.

Первая из используемых операций добавляет префикс к исходной строке. В статье добавляются такие префиксы, как кавычки, символы конвейера (`|`), точки с запятой, оператор конкатенации команд (`&&`). В реализованном в рамках данной работы алгоритме для других видов уязвимостей добавляются иные типы префиксов. Например, для уязвимостей файловой системы могут быть добавлены префиксы, соответствующие путям к предшествующим директориям (`../`, `..%5c`, `%2e%2e%2f` и т.п.).

Вторая операция добавляет суффикс к исходной строке. Большинство добавляемых суффиксов похожи на префиксы и содержат приблизительно тот же набор параметров. Часть параметров является соответствующими префиксам суффиксами, например для префикса “`(`“ найдется парный суффикс “`)`” и так далее.

Также есть операция замены пробелов в исходной строке специальными строками. Специальные строки включают угловые скобки (<), `$IFS`, `%PROGRAMFILES: 10, -5%` и т.д. Такая модификация больше нацелена на уязвимости, содержащие в себе исполняемые команды, поэтому в данной работе было решено отказаться от ее использования, поскольку набор анализируемых уязвимостей содержит лишь несколько типов, связанных с работой с командами.

Следующая модификация извлекает системные команды из исходной строки, такие как `whoami`, `echo`, `cat` и т.д., и вставляет в них специальные символы. Специальные символы включают кавычки, обратный слеш (\), точки (.) и т.д. Вставка этих специальных символов может помочь обойти момент валидации командной строки и при этом не влияет на выполнение заданной команды. Аналогичный подход применим для таких уязвимостей, как SQL-инъекция и других подобных.

И последняя из используемых операций заключается в извлечении из начальной строки опасных названий функций, таких как `system`, `eval`, `phpinfo` и т.д., и переписывании их с изменением регистра символов. Например, `system` будет переписано как `SysTem`, что также не влияет на сценарий выполнения команд. Такой подход тоже применим для уязвимостей, связанных с инъекциями. А также подобная логика может использоваться в файловых уязвимостей при замене используемых в файловом пути символов на другие, не изменяющие логики, символы.

В итоге в данной работе был реализован описанный выше алгоритм мутаций для строковых переменных с модификациями для поддержки и других типов уязвимостей. Стоит отметить, что для разных типов уязвимостей мутации работают концептуально одинаково, но используют разные значения для подстановки. Чтобы добиться этого, была собрана база знаний о параметрах для подстановок под каждый тип уязвимости. А также в момент захода фаззера в уязвимую функцию выясняется, какой именно уязвимости она соответствует, и в зависимости от этого выбираются параметры для подстановок в момент мутаций.

Более того, как уже было упомянуто ранее, фаззинг хотелось настраивать и ограничениями, полученными от символьного исполнения. На данный момент собираются базовые ограничения от UTBot, такие как длина строки и наличие в строке конкретных символов. Эти ограничения тоже влияют на алгоритм фаззинга. Это не единственные ограничения, получаемые от UTBot, однако остальные являются слишком тяжелыми для настройки алгоритма и будут использоваться лишь для валидации данных.

Пример работы настроенного фаззера показан на рисунке 4.2. Исходным примером является код из бенчмарка securibench-micro с уязвимостью обхода директории. Исходный фаззер UTBot способен сгенерировать лишь такие тесты, в которых строка `string` примет стандартные значения вроде пустой строки или, например, строки “`name`”. А с настроенным фаззером, в примере ниже строка `string` задается закодированным путем к предшествующим директориям, что является более информативным.

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
    String s = req.getParameter(FIELD_NAME);
    String name = s.toLowerCase(Locale.UK);

    try {
        FileInputStream fis = new FileInputStream(name); /* BAD */
    } catch(Throwable e) {
        System.err.println("An error occurred");
    }
}

@Test
public void testDoGet3() throws IOException  {
    PathTraversal pathTraversal = new PathTraversal();
    HttpServletRequest httpServletRequestMock = mock(HttpServletRequest.class);
    String string = "%2e%2e%2f%2e%2e%2fetc%2fpasswd"; /* ../../etc/passwd */
    (when(httpServletRequestMock.getParameter(any()))).thenReturn(string);
    pathTraversal.doGet(httpServletRequestMock,  resp: null);
}
```

Рис. 4.2: Иллюстрация работы настроенного фаззера

4.3. Выводы и результаты

Фаззинг UTBot был внедрен в процесс символьного исполнения. Он был настроен с помощью taint-анализа для приоритезации зараженных переменных. Также был реализован алгоритм мутации строковых переменных для ряда поддержанных уязвимостей.

В результате фаззер предоставляет более широкий спектр вредоносных входов именно для зараженных параметров, благодаря чему результирующие тесты становятся более информативными. А также фаззер сохраняет синтаксическую структуру уязвимостей и создает мутации, которые с большей вероятностью не были замечены программистом и могут привести к небезопасному исполнению программы.

Глава 5. Тестирование и итоги.

В данной главе будет рассказано о тестировании результирующего инструмента, а также будут описаны результаты проведенной работы.

5.1. Тестирование

Инструмент был протестирован на бенчмарке securibench-micro, который представляет из себя коллекцию небольших программных тестов, используемых для измерения производительности и точности инструментов анализа безопасности кода. Реализованный инструмент генерирует тесты для 77% файлов с базовыми и межпроцедурными примерами уязвимостей. На остальных 23% файлов инструмент отрабатывает с ошибками, вызванными спецификой работы taint-анализа и UTBot-a. Например, причиной ошибки для нескольких примеров становится то, что ProGuard не находит нужные зараженные пути из-за используемых в тестах концепций языка программирования Java, таких как статические секции и рефлексия.

Среди результирующих тестов появляются ложноположительные по аргументам тесты. То есть такие тесты, в которых заданные аргументы на самом деле не вызывают потенциальную уязвимость. Рисунок 5.2 демонстрирует пример такого теста, исходный код приведен на рисунке 5.1. По большей части такое поведение вызвано спецификой работы UTBot, поскольку этот инструмент практически всегда подставляет в качестве аргумента типа Object значение null, что вызывает ошибку исполнения и не позволяет получить точные результаты выполнения теста.

Тем не менее, все результирующие тесты, сгенерированные на бенчмарке securibench-micro соответствуют зараженным путям исполнения, полученным от taint-анализа. То есть, генерируются лишь те тесты, путь исполнения которых является зараженным, согласно

taint-анализу. Рисунок 5.3 демонстрирует пример полноценного теста, вызывающего уязвимое поведение исходной программы с рисунка 5.1.

А также, тесты генерируются не для всех следов taint-анализа, поскольку некоторые из следов могут быть ложноположительными и, благодаря возможности символьного исполнения качественно обрабатывать потоки данных в программе, такие следы отбрасываются и для них не генерируются соответствующие тесты.

```
public class Inter1 {
    1 usage
    private static final String FIELD_NAME = "name";
    new *
    public void doGet(HttpServletRequest req, HttpServletResponse resp) throws IOException {
        String s1 = req.getParameter(FIELD_NAME);
        String s2 = id(s1);
        String s3 = id( string: "abc");
        PrintWriter writer = resp.getWriter();
        writer.println(s2);                    /* BAD */
        writer.println(s3);                    /* OK */
    }
    2 usages   new *
    private String id(String string) { return string; }
    new *
    public String getDescription() { return "simple id method call"; }
    new *
    public int getVulnerabilityCount() { return 1; }
}
```

Рис. 5.1: Исходный код примера

```
@Test
public void testDoGet2() throws IOException {
    Inter1 inter1 = new Inter1();
    HttpServletRequest httpServletRequestMock = mock(HttpServletRequest.class);
    (when(httpServletRequestMock.getParameter(any()))).thenReturn(((String) null));

    /* This test fails because method [org.example.inter.Inter1.doGet]
        produces [java.lang.NullPointerException]
        org.example.inter.Inter1.doGet(Inter1.java:17) */
    inter1.doGet(httpServletRequestMock,  resp: null);
}
```

Рис. 5.2: Сгенерированный тест с null аргументами

```java
@Test
public void testDoGet4() throws IOException {
    org.mockito.MockedConstruction mockedConstruction = null;
    try {
        mockedConstruction = mockConstruction(PrintWriter.class,
                (PrintWriter printWriterMock,
                 org.mockito.MockedConstruction.Context context) -> {
                });
        Inter1 inter1 = new Inter1();
        HttpServletRequest httpServletRequestMock = mock(HttpServletRequest.class);
        (when(httpServletRequestMock.getParameter(any()))).thenReturn(((String) "data"));
        HttpServletResponse httpServletResponseMock = mock(HttpServletResponse.class);
        PrintWriter printWriterMock1 = mock(PrintWriter.class);
        (when(httpServletResponseMock.getWriter())).thenReturn(printWriterMock1);
        inter1.doGet(httpServletRequestMock, httpServletResponseMock);
    } finally {
        mockedConstruction.close();
    }
}
```

Рис. 5.3: Корректный сгенерированный тест

Также было измерено количество переходов символьной машины в процессе символьного исполнения. После добавления taint-анализа количество переходов сокращается в среднем на 37%. Это вызвано тем, что незараженные пути исполнения отбрасываются на этапе направления символьной машины. Часть таблицы с результатами подсчетов количества переходов представлена на рисунке 5.4.

Название файла	С taint-анализом	Только UTBot	Пояснение
Inter1	31	37	
Inter2	41	62	
Inter3	354	1162	много межпроцедурных вызовов
Inter5	36	46	
Inter8	57	92	
Inter9	46	63	
Inter10	25	38	
Inter11	34	51	
Inter13	2558	5058	рекурсия глубины 1000

Рис. 5.4: Часть таблицы с результатами тестирования инструмента на securibench-micro

Однако, большинство примеров из бенчмарка securibench-micro содержат примеры с небольшим количеством различных путей исполнения, что наглядно отображено в таблице с результатами: даже без подключения taint-анализа символьная машина делает порядка 50 переходов.

В связи с этим, инструмент был протестирован и на самописных примерах, содержащих большее количество ветвлений в коде. Пример такого кода показан на рисунке 5.5. На подобных примерах достигается сокращение количества анализируемых путей в среднем на 74%.

```
public class Interprocedural {

    public void launch(String param2, int count) {
        redirect1(source(count, param2));
    }

    public String source(int i, String arg) {
        if (i > 10) {
            return createDangerousCommand(arg); // dangerous!
        } else {
            return "cat some_file"; // ok
        }
    }

    public void sink(String s1, String s2) {
        try {
            Runtime r = Runtime.getRuntime();
            r.exec(s2);
        } catch (IOException ignored) {}
    }
    1 usage
    private String createDangerousCommand(String arg) { return "rm -rf" + arg; }
    1 usage
    private void redirect1(String s) { redirect2(s); }
    1 usage
    private void redirect2(String s) { redirect3(s); }
    1 usage
    private void redirect3(String s) { passSourceToSink(s); }
    1 usage
    private void passSourceToSink(String s) { sink("string", s); }
}
```

Рис. 5.5: Пример тестового кода с множеством ветвей исполнения

5.2. Итоги

Taint-анализ интегрирован в процесс генерации модульных тестов на безопасность инструмента UTBot. В результате UTBot генерирует тесты, выявляющие уязвимости целевой программы.

Интеграция анализа заражений в генератор тестов предоставляет более комплексный и точный анализ поведения программы. Генерация тестов на безопасность с использованием описанного в работе подхода обеспечивает охват широкого и вариативного диапазона небезопасных случаев исполнения, что способствует более эффективному обнаружению потенциальных проблем безопасности.

В рамках данной работы были поддержаны следующие уязвимости:

- Обход директории (Path traversal) – получение доступа к файлам и папкам за пределами заданной директории.
- Различные виды инъекций – уязвимостей, позволяющих злоумышленнику внедрять вредоносный код или команды в систему. Например, инъекция команды (Command injection), SQL инъекция (SQL injection) и другие.
- Подделка логов (Log forging) – запись ложной информации в лог файлы или системные журналы.
- Небезопасное перенаправление запроса (Unsafe redirect) – перенаправление запроса пользователя на другой ресурс без его согласия или знания.

При этом инструмент достаточно гибок для добавления поддержки других уязвимостей.

Благодаря направлению символьного исполнения по следам taint-анализа, генерируются только тесты, соответствующие зараженным следам. Кроме того, значительно снижается количество анализируемых символьной машиной состояний, что в контексте данной задачи помогает избежать проблему комбинаторного взрыва путей исполнения. И напротив, благодаря использованию символьного исполнения, отсекаются

ложноположительные пути, обнаруженные анализом заражений, и тесты, соответствующие таким путям, не генерируются.

Решатели ограничений настроены на генерацию большего числа решений для зараженных переменных, что позволяет сделать результирующие тесты более вариативными по аргументам.

Фаззер UTBot генерирует специфичные под уязвимости входы и при генерации фокусируется на зараженных параметрах и настраивается под конкретную уязвимость, что позволяет на выходе получить более информативные тесты.

Собрана база знаний с информацией о источниках и приемниках для taint-анализа, состоящая из 31 источника и 153 приемников. А также собрана база знаний с параметрами для фаззинга (префиксы, суффиксы, анализируемые команды и т.д.).

Интеграция taint-анализа в генератор тестов обеспечивает более точный анализ поведения программы, улучшает безопасность программ и снижает риск наличия необнаруженных уязвимостей путем создания точных тестов на безопасность, охватывающих широкий спектр потенциальных уязвимостей. Также, такая интеграция в контексте символьного исполнения и фаззинга позволяет уменьшить количество ложноположительных срабатываний и значительно сокращает количество анализируемых символьной машиной путей.

Список литературы

1. Arzt S. Rasthofer S. Fritz C. Bodden E. Bartel A. Klein J. & McDaniel P. (2014). Flowdroid: Precise context flow field object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49(6) 259-269. DOI: 10.1145/2666356.2594299

2. Avgerinos T. Cha S. K. Rebert A. Schwartz E. J. Woo M. & Brumley D. (2014). Automatic exploit generation. *Communications of the ACM* 57(2) 74-84. DOI: 10.1145/2560217.2560219

3. Baldoni R. Coppa E. D'Elia D. Demetrescu C. & Finocchi I. (2018). A Survey of Symbolic Execution Techniques. *ACM Computing Surveys* 51(3) 1–39. DOI: 10.1145/3182657

4. Brumley D. Poosankam P. Song D. Y. & Zheng J. (2008). Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. IEEE *Symposium on Security and Privacy* 143–157. DOI: 10.1109/sp.2008.17

5. CWE – Common Weakness Enumeration. (n.d.). URL: https://cwe.mitre.org/

6. De Moura, L., Bjørner, N. (2008). Z3: An Efficient SMT Solver. *Ramakrishnan, C.R., Rehof, J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008.* Lecture Notes in Computer Science, vol 4963 .DOI: 10.1007/978-3-540-78800-3_24

7. De Moura L. & Bjørner N. (2011). Satisfiability modulo theories: introduction and applications. *Communications of the ACM* 54(9) 69-77. DOI: 10.1145/1995376.1995394

8. Devmate. (n.d.). devmate – We help developers to focus on their creativity by automating unit tests. (2022). DOI: https://www.devmate.software/

9. Diffblue. (n.d.). Accelerate Java Shift Left. URL: https://www.diffblue.com/

10. Ganesh V. Leek T. & Rinard M. (2009). Taint-based directed whitebox fuzzing. *In 2009 IEEE 31st International Conference on Software Engineering* (pp. 474-484). DOI: 10.1109/ICSE.2009.5070546

11. Gordon. (2021 September 16). EvoSuite | Automatic Test Suite Generation for Java.URL: https://www.evosuite.org/

12. HttpServletRequest (Java EE 6). (2011). URL: https://docs.oracle.com/javaee/6/api/javax/servlet/http/HttpServletRequest.html

13. Ivanov D. Menshutin A. Fokin D. Kamenev Y. Pospelov S. Kulikov E. & Stroganov N. (2022). UTBot Java at the SBST2022 tool competition. *IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)* 39–40. DOI: 10.1145/3526072.3527529

14. Li H. Kim T. Bat–Erdene M. & Lee H. (2013). Software vulnerability detection using backward trace analysis and symbolic execution. International *Conference on Availability Reliability and Security* 446–454. DOI: 10.1109/ares.2013.59

15. Marback A. Do H. He K. Kondamarri S. & Xu D. (2009). Security test generation using threat trees. *ICSE Workshop on Automation of Software Test* 62–69. DOI: 10.1109/iwast.2009.5069042

16. Marback A. Do H. He K. Kondamarri S. & Xu D. (2013). A threat model-based approach to security testing. *Software – Practice and Experience* 43(2) 241–258. DOI: 10.1002/spe.2111

17. Mathis B. (2017). Dynamic tainting for automatic test case generation. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* 436–439. DOI: 10.1145/3092703.3098233

18. Newsome J. & Song D. (2005). Dynamic Taint Analysis for Automatic Detection Analysis and Signature Generation of Exploits on Commodity Software. *Network and Distributed System Security Symposium Conference Proceedings* 5 3–4.

19. OWASP Benchmark. (n.d.). OWASP Benchmark Project URL: https://owasp.org/www-project-benchmark/

20. ProGuard. (n.d.). Java Obfuscator and Android App Optimizer. URL: https://www.guardsquare.com/proguard

21. Randoop. (n.d.). GitHub - randoop/randoop: Automatic test generation for Java. GitHub. URL: https://github.com/randoop/randoop

22. Sang Kil Cha Thanassis Avgerinos Alexandre Rebert and David Brumley. 2012. Unleashing Mayhem on Binary Code. *In Proceedings 2012 IEEE Symposium on Security and Privacy (SP'12)*. 380–394. DOI: 10.1109/SP.2012.31

23. Schwartz E. J. Avgerinos T. & Brumley D. (2010). All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). *2010 IEEE symposium on Security and privacy* 317–331. DOI: 10.1109/sp.2010.26

24. Schweikl S. Fraser G. & Arcuri A. (2022). EvoSuite at the SBST 2022 Tool Competition. *2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST)* 33–34. DOI: 10.1145/3526072.3527526

25. Securibench Micro. (n.d.). Securibench Micro – a benchmark for static analysis tools for security. URL: https://github.com/too4words/securibench-micro

26. Semgrep. (n.d.). Semgrep — Find bugs and enforce code standards. URL: https://semgrep.dev/

27. The Java Virtual Machine Instruction Set. Chapter 6. (n.d.). URL: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-6.html

28. UnitTestBot. (n.d.). GitHub - UnitTestBot/UTBotJava: Automated unit test generation and precise code analysis for Java. GitHub. URL: https://github.com/UnitTestBot/UTBotJava

29. UnitTestBot. (n.d.-b). No tests are generated for String inputs Issue #2113. GitHub. URL: https://github.com/UnitTestBot/UTBotJava/issues/2113

30. Vallee-Rai R. & Hendren L. J. (1998). Jimple: Simplifying Java bytecode for analyses and transformations (p. 15). *Technical report McGill University.*

31. Wang T. Wei T. Gu G. & Zou W. (2011). Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions on Information and System Security (TISSEC)* 14(2) 1–28. DOI: 10.1145/2019599.2019600

32. Zhao J. Lu Y. Zhu K. Chen Z. & Huang H. (2022). Cefuzz: An directed fuzzing framework for php RCE vulnerability. *Electronics* 11(5) 758. DOI: 10.3390/electronics11050758

Приложения

1. Глоссарий

Анализ заражений (taint-анализ) – техника анализа кода, заключающаяся в отслеживании перемещения потенциально вредоносных данных между источниками входных данных и уязвимыми точками программы.

Бенчмарк – унифицированный и стандартизированный тестовый набор, используемый для сравнения производительности или качества различных технологий, продуктов или алгоритмов.

Модульный тест – автоматизированное тестирование отдельной части программного кода для проверки ее корректной работы в изоляции от других модулей и зависимостей.

Промежуточное представление – формат представления программного кода, используемый для анализа, оптимизации, трансляции и компиляции кода на разных языках программирования.

Символьное исполнение – вид статического анализа кода, заключающийся в выполнении операций над символьными выражениями вместо реальных данных для автоматического поиска путей выполнения программы.

Фаззинг – метод тестирования программного обеспечения с использованием автоматически генерируемых некорректных входных данных для поиска ошибок и уязвимостей.

Эксплойт – программный код или метод, позволяющий использовать уязвимость в системе для получения несанкционированного доступа к данным или выполнения иных вредоносных действий.

SMT-решатели – инструменты автоматической проверки удовлетворимости ограничений на переменные, служащие для решения различных задач в области верификации программ.