Санкт–Петербургский государственный университет
Кафедра системного программирования
Группа 20.Б11-мм

Карасев Виктор Алексеевич

Гибридная генерация модульных тестов

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
инженер-исследователь кафедры системного программирования
М.П. Костицын

Санкт-Петербург
2023 г.

Содержание

Введение

Ежегодно сложность программного обеспечения (ПО) растёт и, соответственно задача обеспечения качества становится всё более востребованной. Тестирование — одна из важных составляющих этой задачи и, при этом, достаточно сложная. Написание тестов трудоемкий процесс, на который не всегда удаётся выделить время в процессе разработки ПО. В тоже время, тестирование и отладка не гарантируют отсутствие ошибок или уязвимостей. Для решения данных проблем часто применяются инструменты автоматической генерации модульных тестов.

Техники генерации обычно разделяют на три вида. «Белый ящик» — подразумевает использование исходного кода, «черный ящик» — подразумевает отсутствие знаний об исходном коде и «серый ящик» – гибридный подход, сочетающий в себе два предыдущих.

Одной из техник вида «белый ящик» является символьное исполнение [7, 2, 6] — подход заключающийся в исполнении не на конкретных входных данных, а на так называемых символьных. Это даёт возможность для каждой ветви исполнения получить ограничения на значения входных параметров, при выполнении которых эта ветвь достигается. После этого при помощи SMT-решателей [3, 8, 5] такие значения могут быть найдены и сгенерирован тест, покрывающий данную ветвь. Достоинством символьного исполнения является теоретически полное покрытие кода тестами. Однако необходимость анализировать все пути исполнения, число которых может расти экспоненциально, влечет за собой главный недостаток этого подхода — высокая вычислительная сложность. В то же время имеются и другие проблемы например, невозможность обрабатывать внешние вызовы.

Рассмотрим одну из техник вида «чёрный ящик» – фаззинг [4]. Идея фаззинга в передаче на вход тестируемой программе случайных данных в ожидании аварийного завершения программы, зависания, утечек памяти, нарушений внутренний логики и других потенциальных проблем. Основным достоинством данной техники является низкая, относительно большинства других техник, скорость работы. Однако зачастую фаззинг даёт меньшее покрытие кода, нежели техники вида «белый ящик».

Техники вида «белый ящик» и «чёрный ящик» имеют своих недостатки, поэтому в большинстве реальных инструментов используются техники вида «серый ящик», которые позволяют комбинировать эти подходы и компенсировать их недостатки. Примером инструмента, использующего такой подход, является SAGE [1], который нашел большое количество уязвимостей во время разработки операционной системы Windows 7.

Данная работа была выполнена в рамках проекта V#, который разрабатывается на кафедре системного программирования. V# — инструмент автоматической генерации тестового покрытия для программ на платформе .NET, базирующийся на технике символьного исполнения. В предыдущей работе, был разработан фаззер, который был характеризован некоторыми существенными недостатками, одним из которых является отсутствие изоляции анализируемого кода.

1. Постановка задачи

Целью учебной практики является создание системы взаимодействия фаззинга и символьного исполнения. Для её выполнения были поставлены следующие задачи:

- Выполнить обзор средств изоляции исполняемого кода;
- Разработать архитектуру системы взаимодействия фаззера и символьного исполнеия;
- Реализовать созданную модель взаимодействия в символьной виртуальной машине V#;
- Провести тестирование полученного решения.

2. Обзор

В данной главе представлен обзор используемых техник анализа, средств изоляции исполняемого кода, а также проекта V#.

2.1 Символьное исполнение

Реальные символьные интерпретаторы устроены с учётом различных особенностей языка, таких как поддержка вывода типов, поддержка классов и др., однако они используют общую концепцию абстрактного **символьного исполнителя**.

Рассмотри механизм символьного исполнителя на конкретном примере. Функция Abs (Листинг 1) принимает один аргумент. Требуется ответить на следующие вопросы. Существуют ли конкретные значения аргументов при которых функция Abs вызывает исключение NegativeResultException? Если такие значения существуют то какие?

```
public static int Abs(int x)
{
    int y = 0;
    if (x < 0) {
        y = -x;
    } else {
        y = x;
    }
    if (y < 0) {
        throw NegativeResultException();
    }
    return y;
}
```

Листинг 1: Программа для иллюстрации символьного исполнения

Перед началом исполнения кода, всем входным параметрам сопоставляется символьная переменная. Далее, символьная машина исполняет код по инструкции, накапливая так называемое *условие пути* (изначально $true$). При возникновении ветвлений текущее состояние машины копируется и для каждой новой ветви исполнения условие пути считается как конъюнкция текущего условия пути и условия вхождения в соответствующую ветку. Далее исполнение происходит независимо.

Чаще всего, для проверки достижимости полученных состояний, используются SMT-решатели, которые проверяют условие пути на выполнимость. Если условие выполнимо, то SMT-решатель также выдает модель — значения переменных на которых формула выполняется.

На рис. 1 представлена схема символьного исполнения для функции Abs. В прямоугольниках на первой строке изображена исполняемая инструкция, на второй — символьная память, на третьей — условие пути. В эллипсах проиллюстрирован результат проверки достижимости состояния.

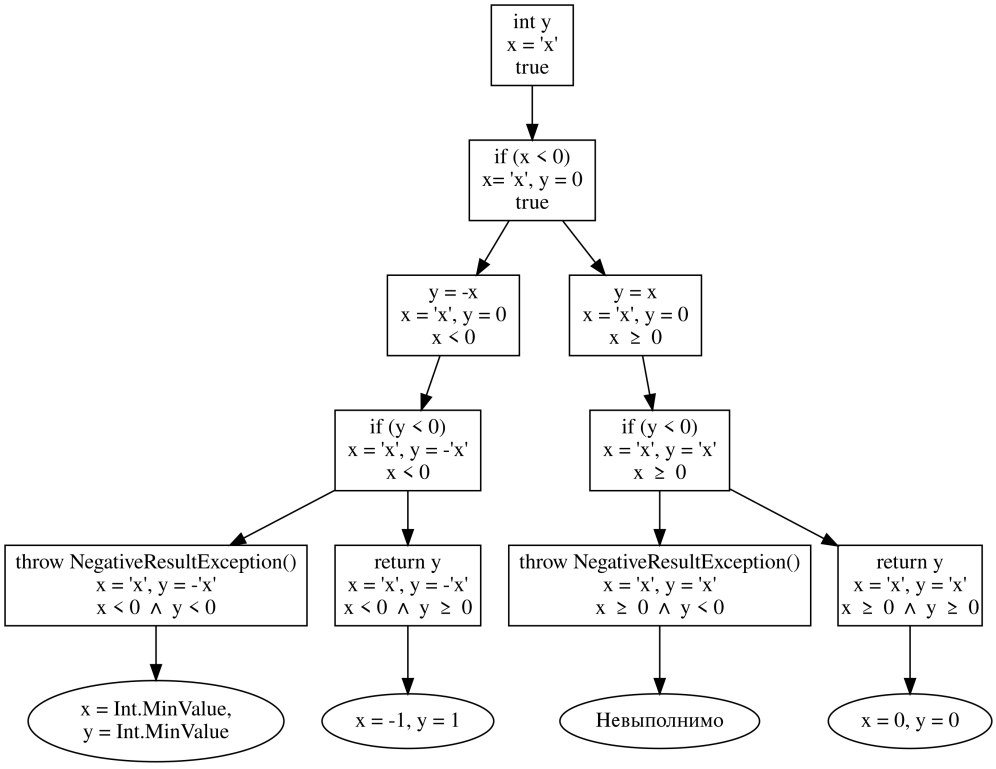

Рис. 1: Схема символьного исполнения функции Abs

2.2 Фаззинг

Фаззинг [4] — техника тестирования программного обеспечения, заключающаяся в передаче на вход приложении случайных значений, в ожидании определенного поведения. Таким поведением может быть утечка памяти, падение приложения, нарушение описанных свойств и другое.

В предыдущей работе, в рамках проекта V#[1], был разработан фаззер, который был характеризован недостатками, связанными с необходимостью изоляции тестируемого кода, а также использованием состояний символьного машины для передачи информации о покрытии. Несмотря на то, что реализованный фаззер позволил проверить применимость алгоритма гибридной генерации тестов, упомянутые выше недостатки препятствуют его использованию в реальных задачах.

2.3 Средства изоляции

В данном разделе будет представлен обзор средств изоляции как для платформы .NET, так и не зависящих от неё.

Harmony[2] – библиотека для платформы .NET, предназначенная для изменения кода во время исполнения. Удаление потенциально взаимодействующих с операционной системой внешних вызовов, позволило бы изолировать анализируемый код. Однако, для этого требуется идентифицировать такие вызовы во время исполнения, также для изменения кода Harmony использует метод системной библиотеки *System.Reflection.Emit*, который позволяет вносить изменения заметно медленнее чем инструментирование кода через *Core Profiler API*[3].

Code Access Security[4] – это система разрешений, которая позволяет управлять правами доступа для .NET приложений. Она работает на основе политик безопасности, которые определяются на уровне домена приложения. Ключевым минусом данного средства изоляции, является отсутствие поддержки .NET Core.

[1] https://github.com/VSharp-team/VSharp/

[2] https://github.com/pardeike/Harmony

[3] https://github.com/dotnet/runtime/tree/main/docs/design/coreclr/profiling

[4] https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/code-access-security

Unbreakable[5] – библиотека для платформы .NET, предназначенная для изоляции не доверенного кода. В качестве изоляции Unbreakable, предлагает ограничивать набор библиотек доступных для использования приложению, что является принципиальным ограничением для анализа произвольного кода, более того, в данный момент, эта библиотека не находится в активной поддержке.

Docker[6] – инструмент для контейнеризации приложений. Docker обеспечивает полную изоляцию запускаемых контейнеров на уровне файловой системы, процессов и сети. Важным ограничением в использовании, является требования к наличию Docker-демона на системе, где запускается контейнер.

2.4 Проект V#

Инструмент V# анализирует промежуточное представление программы — язык CIL (Common Intermediate Language), используя техники символьного исполнения и фаззинга. На рис. 2 показана модульная архитектура проекта V#. Желтым цветом отмечены компоненты, которые были модифицированы в процессе работы.

[5]https://github.com/ashmind/Unbreakable
[6]https://www.docker.com/

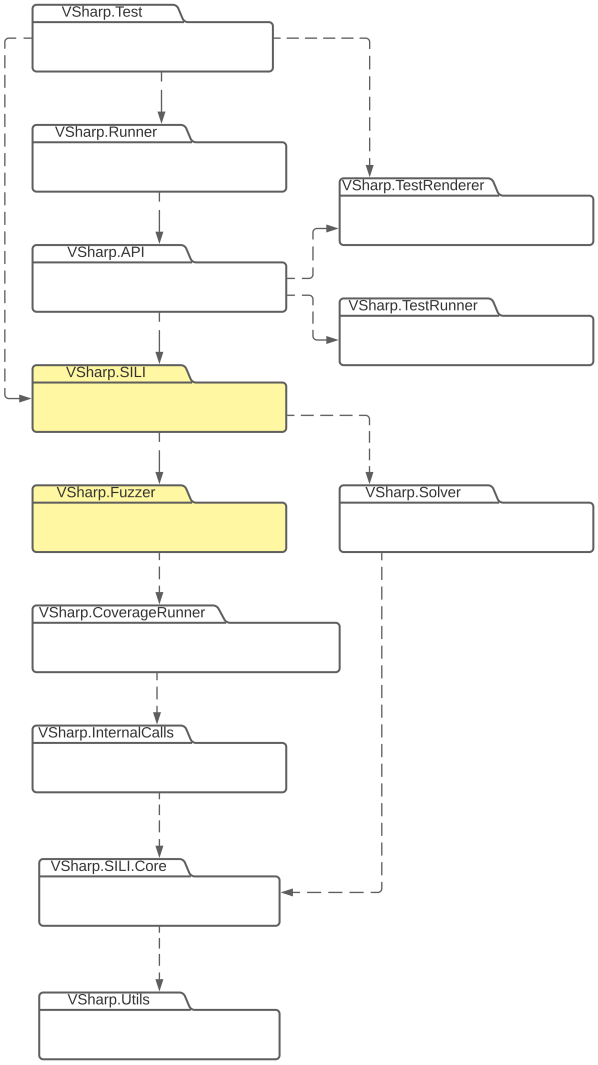

Рис. 2: Архитектура проекта V#

- VSharp.Test используется для тестирования проекта. В качестве тестов используются написанные на C# функции. Для каждой из них запускается анализ и сравнивается полученное покрытие кода с ожидаемым.
- VSharp.Runner используется для конфигурации и запуска проекта из консоли.
- VSharp.API представляет интерфейс для использования V# из других проектов.
- VSharp.TestRenderer используется для создания кода NUnit тестов на языке C#
- VSharp.TestRunner используется для проверки и воспроизведения сгенерированных тестов.
- VSharp.SILI реализует символьной интерпретатор CIL.
- VSharp.Solver реализует взаимодействия с решателем: кодирование запроса и декодирование результата.
- VSharp.InternalCalls реализует символьные модели для некоторых функций из CLR (Common Language Runtime). Вместо полноценного исследования этих функций на их место будет подставлена соответствующая модель.
- VSharp.SILI.Core реализует «ядро» проекта — функции, выполняющие основную содержательную работу, например, функции для работы с состояниями символьной машины.
- VSharp.Utils реализует вспомогательные функции для работы системы, например, дополнительные методы рефлексии.
- VSharp.CoverageRunner реализует инструмент сбора информации о покрытии кода.
- VSharp.Fuzzer реализует алгоритм фаззинга.

3. Архитектура

В этой главе представлена архитектура системы взаимодействия фаззинга и символьного в проекте V#, разработанная в рамках данной работы. На рис. 2 была показана компонентная архитектура всего проекта V#, а на рис. 3 отображены компоненты обозреваемой системы.

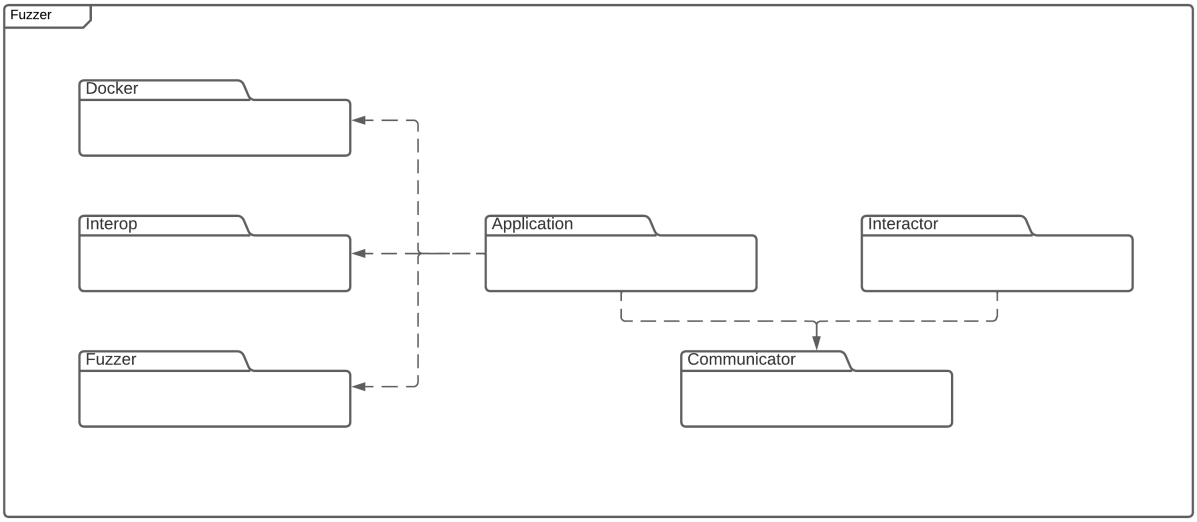

Рис. 3: Архитектура модуля Fuzzer

Разработанная система состоит из следующих компонент:

- **Fuzzer** – отвечает за непосредственно алгоритм фаззинга и генерацию данных.
- **Docker** – реализует взаимодействие с docker-контейнером содержащим фаззер.
- **Interop** – реализует взаимодействие с инструментом сбора статистики.
- **Communicator** – инкапсулирует интерфейс взаимодействие фаззера и символьного исполнения.
- **Application** – реализует фаззер как самостоятельное приложение.
- **Interaction** – предоставляет интерфейс взаимодействия с фаззером как с приложением.

Далее рассмотрим алгоритм взаимодействия фаззера и символьного исполнения. Его можно разбить на три смысловые части: конфигурация, фаззинг, завершение работы. На рис. 4 продемонстрирована диаграмма описывающая работу системы.

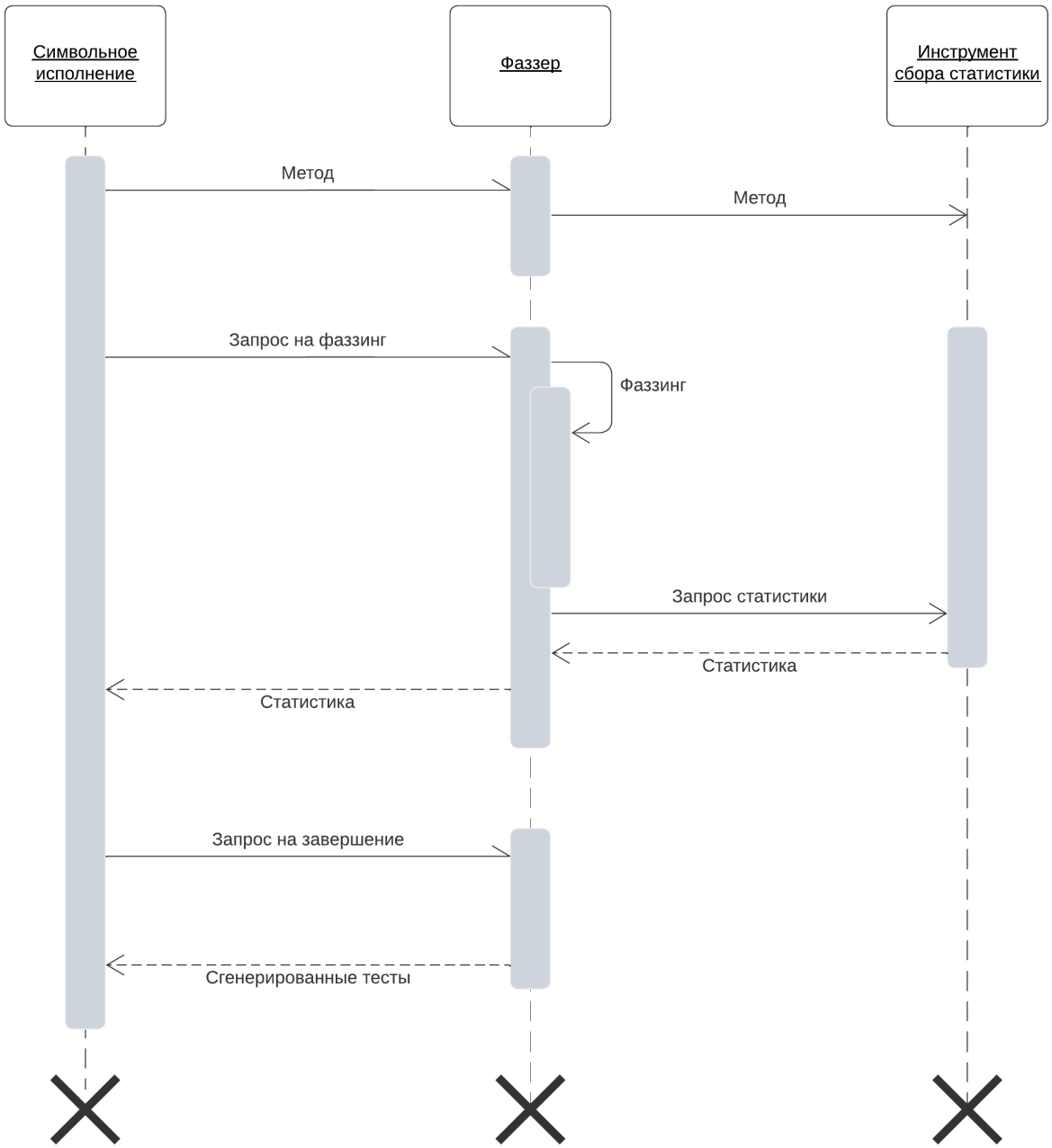

Рис. 4: Работа системы

Конфигурация системы представляет собой передачу информации об анализируемом методе в фаззер и инструмент для сбора статистики. Далее

рассмотрим наиболее интересную часть взаимодействия – фаззинг.

После того как система сконфигурирована начинается фаззинг, после каждого запуска анализируемого метода фаззер запрашивает собранную статистику у инструментера и передает ее в движок символьного исполнения, который в это время также занимается анализом. Далее рассмотрим процесс завершения работы системы.

После того как система считает анализ завершенным (достигнуто 100% покрытие кода или ограничение по времени), фаззеру отправляется сигнал о завершении работы, после чего фаззер пересылает сгенерированные тесты и завершает свою работу.

4. Детали реализации

Данная глава описывает особенности реализации подсистемы генерации модульных тестов в проекте V#. Реализация была написана на языке F#. Код реализации доступен по ссылке[7]. Наиболее интересной частью реализации является изоляция анализируемого кода и коммуникация фаззера и символьного исполнения, а именно:

- Инициализация системы.
- Сбор и пересылка статистики;
- Обработка собранной статистики.
- Пересылка сгенерированных тестов;

4.1 Изоляция исполняемого кода

В качестве инструмента изоляции был выбран Docker, так как он:

- Обеспечивает необходимый уровень изоляции;
- Не имеет ограничений для анализа произвольного .NET кода
- Поддерживает необходимые операционные системы (Windows, MacOS, Linux) и архитектуры (X86-64, ARM).

Исполняемый код вместе с фаззером запускается в Docker-контейнере. Так как фаззеру необходимо коммуницировать с символьным исполнением, контейнер получает доступ к одному сетевому порту для коммуникации по средству TCP, к одной папке с разрешением на запись для сериализации сгенерированных тестов, и к определенному множеству папок на чтение для загрузки необходимых для запуска анализируемого кода динамических библиотек.

[7] https://github.com/KarasssDev/VSharp/tree/Fuzzer-v5

4.2 Коммуникация фаззера и символьного исполнения

Работы системы начинается с запуска контейнером с фаззером. В процессе работы системы, фаззер может быть перекофигурирован любое количество раз для анализа другой сборки или метода. Под кофигурацией подразумевается передача следующей информации в фаззер:

- Путь к анализируемой сборке;
- Путь к локации для сериализации тестов;
- Токен анализируемого метода.

Далее происходит процесс загрузки анализируемого метод с помощью собственной реализации AssemblyManager. Одновременно с фаззером автоматически запускается инструмент для сбора информации об исполненных инструкциях в анализируемом методе. Это происходит благодаря возможностям CorProfilerAPI и специальному окружению. Все взаимодействие с инструментом для сбора информации строится на использовании механизма P/Invoke. Перед началом фаззинга, в инструмент для сбора статистики передается токен анализируемого метода, для того чтобы он собирал информацию только об интересующей области кода.

После описанных выше действий систему можно считать проинициализированной. Далее начинается сам процесс фаззинга. После каждого запуска анализируемого метода, фаззер запрашивает статистику у инструментера и пересылает её в движок символьное исполнение по сокету. Движок символьного исполнения переводит полученную статистику во внутреннее представление и помечает соответствующую ветвь исполнения посещенной. В данный момент фаззер не использует информацию об исполненных инструкциях, однако факт наличия у фаззера данной информации, позволит в будущем использовать более продвинутые техники фаззинга.

После того как анализ считается завершенным (достигнуто 100% покрытие кода или ограничение по времени), фаззеру отправляется сигнал о завершении работы по сокету, после чего он сериализует сгенерированные тесты во внутренне представление (.vst файлы) по соответствующему пути. В

случае если фаззер, по непредвиденным обстоятельствам, не завершает свою рботу, соответствующий ему процесс принудительно завершается.

5. Тестирование

В данной главе описан способ тестирования реализованной системы взаимодействия символьного исполнения и фаззинга для платформы .NET.

5.1 Способ тестирования

В проекте V# в качестве тестов используется набор заранее написанных на языке C# функций, для которых генерируются тесты. Далее происходит проверка корректности: каждый сгенерированный тест запускается на виртуальной машине .NET, а полученный результат запоминается. Если он совпадает с ожидаемым, то тест считается корректным. После проверки каждого теста вычисляется процент полученного покрытия: если он соответствует заранее определенному числу, то тестовая функция считается успешно выполненной.

Для тестирования разработанной системы, был создан набор тестов с проблемными для символьного исполнения участками кода: внешними вызовами (взаимодействие с окружением, вызовы функций из нативных библиотек), большими циклами и рекурсией. Все тесты выполнялись на фиксированном генераторе случайных чисел и ограничением по времени в 20 секунд. Также было проведено тестирование на библиотеке Cosmos.

5.2 Сравнение результатов

В табл. 1 показано сравнение результатов тестирования для инструмента V# с системой гибридной генерации модульных тестов и без нее. Первый столбец таблицы определяют тестовый метод, для которого выполнялся анализ с помощью инструмента V#. Второй столбец показывает уровень тестового покрытия в процентах, полученного символьным движком V# без гибридной генерации модульных тестов («Symbolic»). Последний столбец показывает уровень тестового покрытия в процентах, сгенерированного инструментом V# с системой гибридной генерации модульных тестов («Hybrid»).

Название теста	Покрытие «Symbolic» (%)	Покрытие «Hybrid» (%)
OnlyExternalCall	0	100
ExternalCallOneBranch	67	100
ExternalCallManyBranches	60	100
ExternalCallReadWrite	0	100
NativeRegexp	0	100
ForLoop	100	100
NestedForLoop	100	100
MutualRec	100	100
Fibonacci	0	100
Cosmos	23	26
Lifetimes.Utils	12	19

Таблица 1: Результаты тестирования разработанной системы

Проведенное тестирование показывает, что для функций, в которых присутствуют вызовы внешних методов (взаимодействие с окружением, вызовы функций из нативных библиотек), V# с системой гибридной генерации модульных тестов генерирует лучшее тестовое покрытие, чем символьный движок V# без этой системы. Ячейки таблицы, которые помечены зеленым цветом, показывают случаи, в которых уровень покрытия увеличился. При этом для остальных уровень покрытия не уменьшился.

6. Заключение

В ходе работы была разработна система взаимодействия фаззинга и символьного исполнения, устраняющая недостатки предыдущей реализации и предоставляющая фаззеру информацию о покрытии анализируемого метода, что позволит в будущем использовать более продвинутые техники фаззинга. Также в ходе работы были выполнены следующие задачи:

- Выполнен обзор следующих средств изоляции кода: Harmony, Code Access Security, Unbreakable, Docker;
- Разработана модель системы взаимодействия фаззера и символьного исполнения;
- Разработана архитектура системы взаимодействия фаззера и символьного исполнения фаззера для платформы .NET, содержащая две основные компоненты: Fuzzer.Interactor и Fuzzer.Application;
- Созданная модель реализована в рамках проекта V#;
- Проведено тестирование полученного решения.

Список литературы

[1] Bounimova Ella, Godefroid Patrice, and Molnar David. Billions and billions of constraints: Whitebox fuzz testing in production // 2013 35th International Conference on Software Engineering (ICSE) / IEEE. — 2013. — P. 122–131.

[2] Cadar Cristian and Sen Koushik. Symbolic execution for software testing: three decades later // Communications of the ACM. — 2013. — Vol. 56, no. 2. — P. 82–90.

[3] De Moura Leonardo and Bjørner Nikolaj. *Z3: An efficient SMT solver* // International conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2008. — P. 337–340.

[4] Li Jun, Zhao Bodong, and Zhang Chao. Fuzzing: a survey // Cybersecurity. — 2018. — Vol. 1, no. 1. — P. 1–13.

[5] Niemetz Aina and Preiner Mathias. Bitwuzla at the SMT-COMP 2020 // arXiv preprint arXiv:2006.01621. — 2020.

[6] Păsăreanu Corina S and Visser Willem. A survey of new trends in symbolic execution for software testing and analysis // International journal on software tools for technology transfer. — 2009. — Vol. 11, no. 4. — P. 339–353.

[7] Baldoni Roberto, Coppa Emilio, D'elia Daniele Cono, Demetrescu Camil, and Finocchi Irene. A survey of symbolic execution techniques // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.

[8] Barrett Clark, Conway Christopher L, Deters Morgan, Hadarean Liana, Jovanović Dejan, King Tim, Reynolds Andrew, and Tinelli Cesare. *Cvc4* // International Conference on Computer Aided Verification / Springer. — 2011. — P. 171–177.