Санкт-Петербургский государственный университет

Кузиванов Сергей Юрьевич

Выпускная квалификационная работа

Инструмент генерации юнит-тестов для GoLang

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2018 «Программная инженерия»*

Научный руководитель:
доцент кафедры системного программирования, к. т. н. Ю. В. Литвинов

Рецензент:
руководитель департамента анализа программ ООО Техкомпания “Хуавей” Д. А. Иванов

Санкт-Петербург
2022

Saint Petersburg State University

Kuzivanov Sergei

Bachelor's Thesis

GoLang unit test generation tool

Education level: bachelor

Speciality *09.03.04 "Software Engineering"*

Programme *CB.5080.2018 "Software Engineering"*

Scientific supervisor:
C.Sc., docent Y. V. Litvinov

Reviewer:
Software Analysis Team Leader at "Huawei" D. A. Ivanov

Saint Petersburg
2022

Оглавление

Введение

С каждым годом количество кода, который пишется в мире ежедневно, неуклонно растёт. Это вызвано активным развитием информационных технологий по всему миру, и предпосылок, что за следующие десять лет скорость разработки программных продуктов снизится, нет.

Тестирование – одна из важнейших частей разработки, поскольку даже самые опытные программисты могут допускать ошибки при написании кода или не задокументировать какую-либо особенность реализованного модуля, что может привести к неправильной работе создаваемой системы. Поэтому в крупных проектах тестированию уделяется большое внимание и тратится заметная часть ресурсов компании.

Тестирование кода бывает нескольких видов. Один из видов – модульное тестирование или юнит-тестирование. Проведение данного вида тестирования заключается в написании набора тестов для какого-либо (обычно небольшого) модуля системы, охватывающего все возможные сценарии работы модуля, и обычно возлагается на программиста, реализовавшего данный модуль. Следовательно, программист тратит время на тестирование, что уменьшает время на разработку кода, вследствии чего время разработки проекта в целом увеличивается. Одним из способов сэкономить время и силы разработчика на тестирование и уменьшить время разработки является автоматизация написания модульных тестов с помощью специализированных инструментов, называемых фаззерами.

Существует множество подходов к автоматической генерации модульных тестов. Случайное тестирование и символьное исполнение являются одними из самых распространённых.

Случайное тестирование генерирует тесты случайным образом и достаточно быстро. Символьное же исполнение производит анализ программы и уже потом на основе собранных данных генерирует модульные тесты. Таким образом, основным преимуществом случайного тестирования является высокая скорость генерации модульных тестов, а основное преимущество символьного исполнения – большое покрытие

кода тестами.

Язык программирования Go является относительно новым языком программирования. Он был разработан компанией Google в 2007 году. На данный момент язык достаточно активно развивается, каждые полгода выходит новая версия, расширяющая возможности языка. И, несмотря на то, что этот язык достаточно новый, многие крупные компании, такие как Huawei и Docker Inc., активно используют его в своих проектах.

На данный момент существует достаточно много проектов, реализующих подход случайного тестирования при генерации модульных тестов, однако ощущается нехватка решений, использующих символьное исполнение. Это неудивительно, поскольку реализация подхода случайного тестирования значительно проще, чем символьного исполнения. Однако подход символьного исполнения ценен как подход, позволяющий находить ошибки в коде с большей вероятностью по сравнению со случайным тестированием. И именно возможности применения подхода символьного исполнения к коду на языке Go и будет посвящена данная выпускная квалификационная работа.

Постановка задачи

Целью данной выпускной квалификационной работы является создание инструмента, позволяющего для произвольной функции в коде на языке Go автоматически генерировать тестовый Go-файл с модульными тестами с использованием символьного исполнения при генерации тестов.

Для достижения этой цели в рамках работы были сформулированы следующие задачи.

1. Обзор существующих инструментов для генерации модульных тестов для кода на языке Go.
2. Проектирование нового инструмента для генерации модульных тестов для произвольной функции в виде тестового файла на Go.
3. Реализация спроектированного инструмента.
4. Проведение экспериментов.

1. Обзор

1.1. Виды фаззеров

Существует много различных видов классификации фаззеров, но в контексте данной работы нам будет интересна только классификация фаззеров в зависимости от использования информации о тестируемой функции при генерации набора модульных тестов для неё. Согласно выбранной классификации фаззеры делят на три группы: фаззеры чёрного ящика, фаззеры серого ящика и фаззеры белого ящика. Это разделение весьма условно, поэтому в данной работе каждый фаззер будет рассматриваться либо как фаззер чёрного ящика, либо как фаззер белого ящика.

Фаззер чёрного ящика – это такой фаззер, который почти не использует информацию о тестируемой функции. Часто в таких фаззерах используется подход случайного тестирования.

Случайное тестирование – это подход автоматической генерации модульных тестов, который предполагает, что значение каждого параметра тестируемой функции выбирается случайным образом из множества допустимых значений, определяемого типом каждого конкретного параметра. Данный подход позволяет генерировать большие наборы модульных тестов за короткий промежуток времени. Однако главная проблема данного подхода состоит в том, что для покрытия кода тестами необходимо сгенерировать довольно внушительное количество модульных тестов.

Фаззер белого ящика – это такой фаззер, который максимально полно использует имеющуюся информацию о тестируемой функции. Примером подхода, применяемого в таких фаззерах, может служить подход символьного исполнения.

Символьное исполнение – это подход автоматической генерации модульных тестов, при котором весь доступный код анализируется с целью определения ограничений, налагаемых условными операторами и операторами цикла, на параметры функции. Это позволяет генери-

ровать достаточно малый набор модульных тестов, покрывающий код. Однако сам анализ кода занимает значительное количество времени ввиду теоретической сложности данного подхода.

Фаззер серого ящика – это что-то среднее между фаззером чёрного ящика и фаззером белого ящика. Данный фаззер обычно применяет во время своей работы подходы, родственные символьному исполнению, однако с различными эвристиками с целью сокращения времени анализа. Граница между таким видом фаззеров и между фаззерами белого или чёрного ящика обычно весьма условна, поэтому в данной работе не будет рассматриваться этот вид фаззеров, а все фаззеры, которые могут быть приписаны к фаззерам серого ящика, будут упоминаться как фаззер чёрного ящика или фаззер белого ящика в зависимости от полноты использования информации о тестируемом коде.

1.2. Обзор существующих решений

В данном подразделе будет проведён обзор существующих фаззеров для языка Go.

1.2.1. DUCKEE GO

DUCKEE GO [3] – это кросс-платформенный символьный движок для программ на языке Go. Он принимает на вход код программы на Go, строит по нему AST, делает специальные преобразования AST и с помощью SMT-решателя Z3 [11] ищет входные данные, которые приводят к ошибке.

1.2.2. Пакет gofuzz

Пакет gofuzz [13] предназначен для заполнения произвольного объекта в программе на Go случайным значением. Это достигается за счёт того, что существует конечное множество встроенных Go-типов, генерация случайных значений для которых реализована в данном пакете.

Пакет может быть использован при фаззинге в качестве генератора случайных значений произвольного типа.

1.2.3. Фреймворк go-fuzz и встроенный фаззер

Фреймворк go-fuzz [12] позволяет тестировать Go-код методом фаззинга. На вход фреймворк принимает Go-файл, в котором должна быть функция `Fuzz`. Тестирование происходит посредством запуска функции `Fuzz` для произвольных значений параметров. В результате генерируются файлы с тестами, на которых выполнение функции `Fuzz` завершилось некорректно.

Данное решение подходит для языка Go версии 1.17 и ниже. В версии 1.18 был внедрён встроенный фаззер с подобной функциональностью [1].

1.2.4. Проект gollvm

Проект gollvm [8] — это проект, в который входит компилятор программ на Go, построенный на инфраструктуре LLVM [10]. Данный компилятор позволяет не только компилировать программы, но и транслировать программы в LLVM биткод.

Важным преимуществом LLVM биткода по сравнению с кодом на Go является тот факт, что для LLVM биткода уже существует символьный движок KLEE [9], позволяющий генерировать значения входных параметров с большим покрытием кода.

1.3. Результат обзора

В результате проведённого обзора в подразделе 1.2 был выбран подход с использованием проектов gollvm [8] и klee [9] по следующим причинам.

- Поскольку целью данной выпускной квалификационной работы является исследование возможности применения символьного исполнения для генерации модульных тестов, то фаззеры чёрного ящика не были взяты за основу нового инструмента.
- Проект DUCKEEGO [3], хоть формально и удовлетворяет критерию использования фаззера белого ящика, имеет ряд недостатков.

  - Проект на платформе GitHub выглядит заброшенным, поскольку последний коммит был сделан четыре года назад.
  - При попытке собрать данное решение возникли ошибки, которые не удалось решить. Также отсутствовал Docker-файл, который при наличии мог бы помочь при сборке решения.
  - Поскольку собрать и запустить данное решение не удалось, о работоспособности и функциональности можно судить только исходя из соответствующей статьи [3].

- И проект gollvm [8], и проект klee [9] на данный момент достаточно активно развиваются, вследствии чего можно сделать вывод, что решение, основанное на этих проектах, имеет больший потенциал для дальнейшего развития по сравнению с DUCKEEGO [3].

2. Архитектура инструмента

Основная цель инструмента – это генерация модульных тестов для функций в исходном коде на языке Go. Однако необходимо отметить, что используемый проект klee [9] предназначен только для генерации значений входных параметров, а в рамках данной работы это значения параметров функции. То есть для получения модульного теста в обычном его понимании недостаточно использовать только проект klee [9]. Поэтому в данной работе речь в основном будет идти о генерации значений параметров функций, а не о генерации модульных тестов. Однако разработанный инструмент имеет поддержку генерации модульных тестов по схемам.

В данном разделе будет описана архитектура разработанного на основе gollvm [8] и klee [9] инструмента. В частности, будет описана идея построения данного инструмента, будет обосновано использование в проекте Docker-образа, а также будет описан процесс взаимодействия состовляющих частей инструмента друг с другом.

2.1. Идея построения архитектуры

В рамках данной работы был разработан фаззер белого ящика на основе проектов gollvm [8] и klee [9]. Обоснование выбора данного подхода описано в подразделе 1.3.

В работе данного фаззера активно используются такие утилиты как **klee** и **ktest-tool** из проекта klee [9] и **go** и **llvm-goc** из проекта gollvm [8]. Поскольку основная логика всего инструмента сосредоточена в данных утилитах, то было принято решение разработать архитектуру фаззера в виде набора взаимодействующих между собой утилит.

Каждая утилита в данной архитектуре требует для своей работы одни артефакты и в процессе работы генерирует другие артефакты. Следовательно, всё взаимодействие между утилитами реализовано в виде генерации и использования артефактов, причём ко всем генерируемым артефактам пользователь разработанного инструмента имеет доступ.

Во время работы инструмента предполагается, что имеется определённый Go-пакет, состоящий из файлов с исходным кодом. Эти Go-файлы содержат определения тех функций, которые необходимо протестировать с помощью данного инструмента. Точнее, необходимо для каждой тестируемой функции сгенерировать наборы входных параметров так, чтобы они покрывали как можно больше исходного кода функции.

Таким образом, разработанный фаззер должен принимать Go-пакет, функции в котором надо протестировать, и файл конфигурации, содержащий информацию о тестируемых функциях, а возвращать тестовые Go-файлы и наборы значений параметров для каждой упомянутой в файле конфигурации функции. Следовательно, можно говорить, что входным артефактом всего инструмента в целом является Go-пакет, а выходными артефактами являются тестовые Go-файлы и наборы значений параметров. Необходимо отметить, что файл конфигурации не рассматривается как артефакт системы только из-за концептуальных соображений. Более подробно о содержании файла конфигурации можно прочитать в подразделе 3.3.

2.2. Использование Docker-образа

Для обеспечения правильной работы фаззера необходима установка следующих проектов (приведён список основных проектов, более полный список рассматривается ниже):

- LLVM [10] – инфраструктура для создания компиляторов и вспомогательных утилит;
- Gollvm [8] – компилятор языка Go на основе LLVM;
- Gofrontend [7] – фронтенд для компиляторов языка Go;
- Z3 [11] – SMT-решатель;
- KLEE [9] – символьный движок LLVM биткода.

Более того, для сборки проекта gollvm [8] требуется определённая версия LLVM [10], не являющаяся официальным релизом. Также существует зависимость между проектами gollvm [8] и gofrontend [7]. Следовательно, установка правильных версий данных проектов может быть сложной задачей для пользователя разработанного фаззера.

На сегодняшний день всё большую популярность набирает идея использования Docker-контейнеров для решения разного рода проблем, в частности проблем со сложной сборкой проектов с зависимостями. Основные преимущества использования Docker-контейнеров описаны ниже.

- Нет необходимости тратить время на сборку решения, достаточно скачать уже настроенное и готовое к работе окружение по сети Интернет.
- Окружение Docker-контейнера изолировано от операционной системы пользователя, что гарантирует, что программы, запущенные внутри Docker-контейнера, не смогут изменить настройки операционной системы или влиять на другие процессы.
- Также вследствие изолированности Docker-контейнера от операционной системы приложение, работающее в Docker-контейнере, работает одинаково вне зависимости от настроек операционной системы.
- Docker-контейнер может работать на любом компьютере, на котором установлен Docker [5].

В рамках данной работы был разработан Docker-образ, включающий в себя все необходимые для работы реализованного инструмента проекты. Данный Docker-образ используется для запуска Docker-контейнера, в котором происходит работа всех утилит разработанного инструмента.

Необходимо отметить, что Docker-образ содержит несколько патчей для проектов gollvm [8], gofrontend [7] и klee [9]. Все используемые патчи расположены в папке `utbotgo/patches` проекта.

- Патч для klee расширяет возможности утилиты **ktest-tool**. Более подробно о данном расширении можно прочитать в подразделе 2.3.4.

- Патчи для gollvm и gofrontend добавляют возможность компиляции стандартной библиотеки Go в LLVM биткод во время сборки проекта gollvm системой сборки CMake [2] путём установки значения переменной `GOLLVM_BUILD_LLVM_BC_LIBRARY` в `On`.

Ниже перечислено основное программное обеспечение, установленное в рассматриваемом Docker-образе.

- Операционная система – Ubuntu 20.04.

- Программное обеспечение из git-репозиториев (таблица 1).

Название	Ссылка на репозиторий	Коммит	Патч
Z3 [11]	github.com/Z3Prover/z3	df8f9d7dcb	–
LLVM [10]	github.com/llvm/llvm-project	2c5590adfe	–
Gollvm [8]	go.googlesource.com/gollvm	f17ba8c770	+
Go frontend	go.googlesource.com/gofrontend	3e9f4ee166	+
libffi	github.com/libffi/libffi	b60d4fc7bb	–
libbacktrace	github.com/ianlancetaylor/libbacktrace	d0f5e95a87	–
KLEE [9]	github.com/klee/klee	dfec2ced4e	+

Таблица 1: Программное обеспечение из git-репозиториев

- Программное обеспечение, установленное через утилиту **apt** (таблица 2).

- Пакеты для языка Python3 (таблица 3).

2.3. Взаимодействие утилит

Разработанный инструмент представляет собой набор утилит, взаимодействующих друг с другом в специальном Docker-контейнере. Каждая

Название	Версия	Пакет
git	2.25.1	`git`
Make	4.2.1	`make`
Ninja	1.10.0	`ninja-build`
CMake	3.16.3	`cmake`
GCC	9.3.0	`gcc`
Python2	2.7.18	`python`
Python3	3.8.10	`python3`

Таблица 2: Программное обеспечение из **apt**-репозиториев

Название	Версия
PyYAML	6.0

Таблица 3: Пакеты Python, установленные через **pip3**

утилита может запускаться отдельно от остальных и имеет свою спецификацию аргументов командной строки, входных и выходных данных.

Упрощённая архитектура разработанного инструмента представлена на рис. 1.

На диаграмме компонентов на рис. 1 компонентами являются утилиты, работающие в Docker-контейнере, а интерфейсы являются генерируемыми артефактами. Также с помощью цвета компоненты закодирована информация о типе утилиты:

- белая компонента – утилита одного из проектов, перечисленных в подразделе 2.2;
- жёлтая компонента – утилита одного из проектов, изменённая с помощью одного из патчей;
- зелёная компонента – утилита, реализованная в рамках данной выпускной квалификационной работы.

Краткое описание утилит, изображённых на рис. 1:

- **llvm-goc** – компилятор языка Go на базе LLVM из проекта gollvm [8];

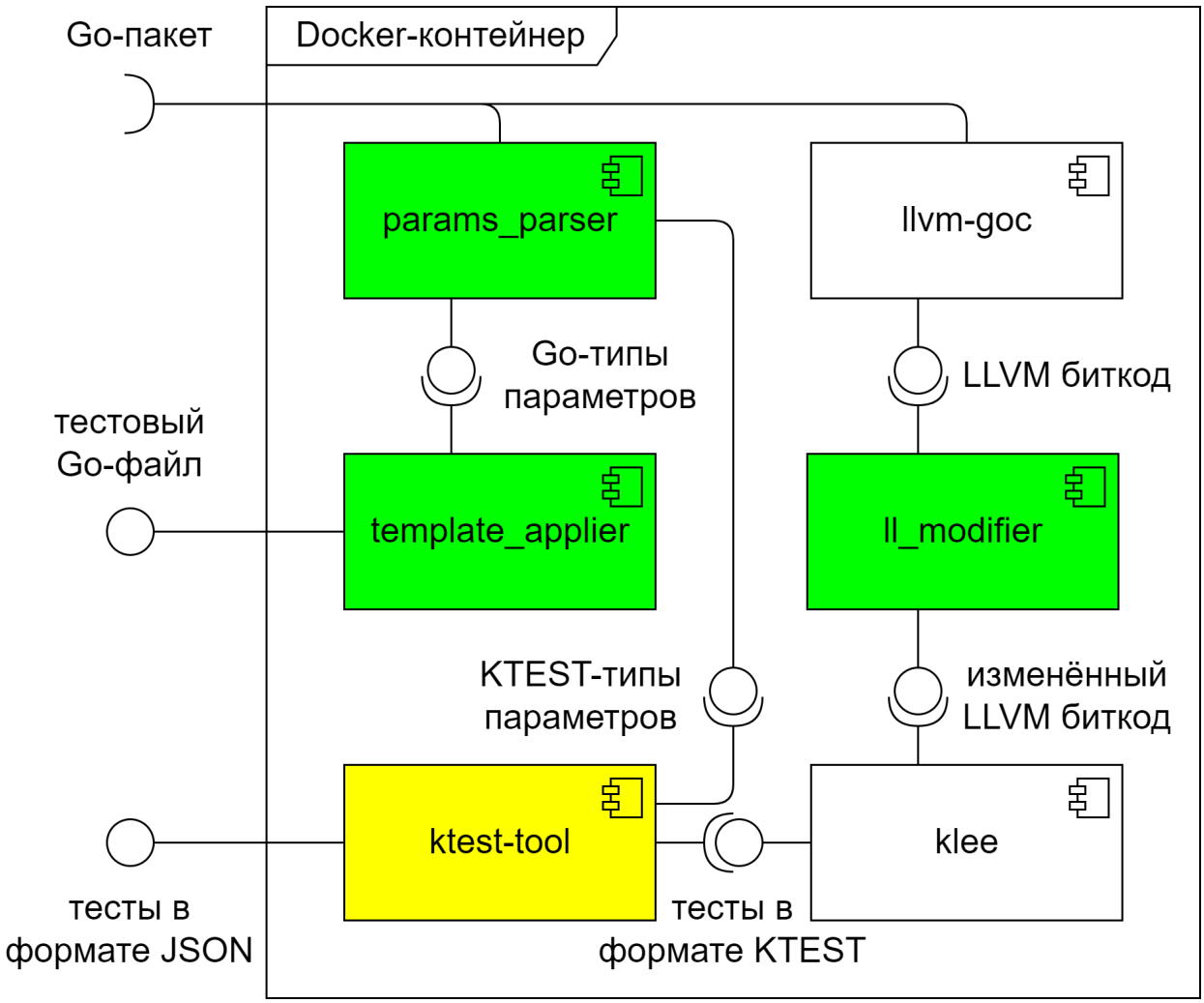

Рис. 1: Архитектура взаимодействия основных утилит

- **ll_modifier** – утилита для преобразования LLVM биткода, полученного от компилятора **llvm-goc**, в LLVM биткод, понятный **klee**;
- **klee** – символьный движок для LLVM биткода, главная утилита проекта klee [9];
- **ktest-tool** – модифицированная версия утилиты для парсинга тестов в формате KTEST, также входит в состав проекта klee [9];
- **params_parser** – утилита для представления типов параметров функций в различных форматах;
- **template_applier** – генератор тестовых Go-файлов по заданному шаблону.

Более подробно про данные утилиты можно прочитать в следующих подразделах.

2.3.1. Утилита llvm-goc

Утилита **llvm-goc** является компилятором языка Go на базе инфраструктуры LLVM [10] и входит в состав проекта gollvm [8]. То есть компилятор **llvm-goc** сначала транслирует исходный код на языке Go в специальное промежуточное представление, называемое LLVM биткодом, а потом компилирует полученный LLVM биткод в исполняемый файл. Таким образом, данный компилятор, в отличие от стандартного компилятора языка Go, может компилировать исходный код как в исполняемый код, так и в LLVM биткод, что и используется в данной работе.

2.3.2. Утилита klee

Утилита **klee** является символьным движком на базе инфраструктуры LLVM и основной утилитой в рамках проекта klee [9]. Суть работы данной утилиты заключается в генерации входных данных программы, представленной в виде LLVM биткода. Утилита **klee** специализирована на работе с LLVM биткодом, полученном из кода на C-подобном языке с помощью компилятора clang [4]. Поэтому работа **klee** с LLVM биткодом, генерируемым **llvm-goc**, имела ряд сложностей.

Для того, чтобы сгенерировать входные данные для какой-либо части программы с помощью утилиты **klee**, необходимо отметить определённым образом в исходном коде переменные, от которых зависит работа тестируемой части. Данные переменные в klee [9] называются символьными. Например, в случае, рассматриваемом в данной работе, необходимо генерировать входные данные для функции. Следовательно, все параметры рассматриваемой функции и только они будут символьными. В результате после запуска утилиты **klee** будут возвращены наборы значений параметров тестируемой функции. Формат, в котором **klee** возвращает сгенерированные входные данные, более подробно рас-

сматривается в подразделе 2.3.4.

2.3.3. Утилита ll_modifier

Утилита **ll_modifier** необходима для преобразования LLVM биткода, генерируемого компилятором **llvm-goc**, в LLVM биткод, понятный утилите **klee**.

Существует несколько проблем, не позволяющих использовать LLVM биткод от **llvm-goc** напрямую в **klee**.

Во-первых, LLVM биткод, генерируемый **llvm-goc**, не включает в себя стандартную библиотеку Go, вследствии чего некоторые вызываемые функции и некоторые глобальные переменные неопределены. При этом **klee** не работает с функциями и переменными без определения, если только это не специальные функции языка C, такие как `printf` и `cos`. При подключении же LLVM биткода стандартной библиотеки Go **klee** не может интерпретировать вставки кода на ассемблере.

Эта проблема была решена не в полном объёме в рамках данной выпускной квалификационной работы. Решение, реализованное в утилите **ll_modifier**, определяет внешние глобальные переменные нулевыми значениями, но никак не борется с вызовом внешних функций.

Во-вторых, для того, чтобы **klee** сгенерировал наборы значений параметров функции, необходимо указать все параметры тестируемой функции в виде символьных переменных. Это делается с помощью функции `klee_make_symbolic`. В коде на языке C использование данной функции для предоставления утилите **klee** информации о символьных переменных происходит с помощью вызова этой функции для каждой символьной переменной. Однако для кода на языке Go использование этой функции становится трудной задачей. Были рассмотрены следующие варианты использования функции `klee_make_symbolic` в коде на языке Go.

- Язык Go поддерживает возможность вызова функций на языке C, однако данный подход использует файлы стандартной библиотеки Go, подключить которые в рамках данной работы не удалось.

- Функция `klee_make_symbolic` на самом деле не имеет определения, то есть это своего рода маркер, указывающий **klee** на символьную переменную. Поэтому была проведена попытка определения подобной функции в коде на Go. Данную идею не удалось реализовать в силу особенностей компилятора **llvm-goc**.

- В силу вышесказанного появилась идея изменить генерируемый LLVM биткод с помощью LLVM API на языке C++ с целью описания функции `klee_make_symbolic` и определения новой функции, создающей символьные переменные для каждого параметра тестируемой функции. Данный подход был реализован в утилите **ll_modifier**.

Таким образом, утилита **ll_modifier** содержит в себе решения двух описанных выше проблем, позволяя делать преобразование LLVM биткода из генерируемого компилятором **llvm-goc** в используемый **klee**.

2.3.4. Утилита ktest-tool

Утилита **klee** после завершения своей работы генерирует различные артефакты. В частности, она генерирует тесты в файлах в формате KTEST.

Каждый файл KTEST имеет расширение `.ktest`. В случае с генерацией наборов значений параметров для функции на языке Go объекты в файле KTEST являются параметрами определённой функции, то есть каждый сгенерированный утилитой **klee** файл с тестами в формате KTEST описывает все входные параметры тестируемой функции.

Из рис. 2 видно, что объекты, а в данном случае параметры Go-функции, не имеют типа, то есть каждый объект может содержать в качестве своего значения строчку или число, но непонятно, что именно.

Утилита **ktest-tool** принимает в качестве параметра командной строки файлы с входными параметрами в формате KTEST, которые были сгенерированны утилитой **klee**, и выводит их содержимое в человекочитаемом виде. Поскольку, как уже было сказано ранее, в формате

формат KTEST		
HDR	Заголовок (обычно KTEST)	5 бит
VERSION	Номер версии	4 бита
NUM_ARGS	Количество аргументов	4 бита
аргументы (NUM_ARGS штук)		
SIZE	Размер аргумента в битах	4 бита
ARG	Значение аргумента	SIZE бит
SYM_ARGVS	Зарезервированное поле	4 бита
SYM_ARGVLEN	Зарезервированное поле	4 бита
NUM_OBJECTS	Количество объектов	4 бита
объекты (NUM_OBJECTS штук)		
NAME_SIZE	Размер имени объекта в битах	4 бита
NAME	Имя объекта	NAME_SIZE бит
BYTES_SIZE	Размер значения в битах	4 бита
BYTES	Значение объекта	BYTES_SIZE бит

Рис. 2: Структура файла с форматом KTEST

KTEST не содержится информация о типах представленных объектов, **ktest-tool** не может выводить данные в каком-то одном типе данных для каждого объекта. Поэтому вывод значений объектов производится сразу в нескольких различных типах. Пример этого можно видеть на листинге 1. Данный пример был взят из официального описания работы утилиты **ktest-tool**.

Другая проблема заключается в том, что утилита **ktest-tool** может

```
ktest file : 'klee-last/test000003.ktest'
args       : ['get_sign.bc']
num objects: 1
object 0: name: 'a'
object 0: size: 4
object 0: data: b'\x00\x00\x00\x80'
object 0: hex : 0x00000080
object 0: int : -2147483648
object 0: uint: 2147483648
object 0: text: ....
```

Листинг 1: Пример вывода утилиты ktest-tool

вывести либо всё содержимое всех файлов KTEST в человекочитаемом виде, непригодном для использования другими утилитами, либо значения указанных в параметрах командной строки объектов в виде набора битов, что тоже неудобно для дальнейшей работы со значениями объектов.

Для решения этих проблем в рамках данной работы был разработан патч `utbotgo/patches/ktest-tool.patch` для утилиты **ktest-tool**. Данный патч добавляет возможность указать тип каждого объекта из KTEST-файлов в отдельном файле в формате JSON, а также вывести информацию об объектах тоже в формате JSON. Дополнительно была добавлена поддержка типа `base64`. Таким образом, новая версия утилиты **ktest-tool** поддерживает следующие типы данных.

- **hex** – hex-строка;
- **int** – целое число со знаком (применимо только если данные имеют длину 1, 2, 4 или 8 байт);
- **uint** – беззнаковое целое число (применимо только если данные имеют длину 1, 2, 4 или 8 байт);
- **text** – ASCII-строка;
- **b64** – данные в формате base64.

Как итог, вышеупомянутый патч для утилиты **ktest-tool** позволяет записывать значения параметров в виде, пригодном для последующей обработки другими утилитами инструмента.

2.3.5. Утилита params_parser

Утилита **params_parser** предназначена для извлечения имён и типов аргументов и возвращаемых значений функций, которые необходимо протестировать. Возвращаемым форматом файлов с полученной информацией является формат JSON. Поддерживается вывод типов параметров как в виде KTEST-типов, так и в виде типов языка Go.

2.3.6. Утилита template_applier

Утилита **template_applier** применяется как генератор тестовых Go-файлов по шаблону `utbotgo/utils/other_files/test.go_template`. Каждый сгенерированный разработанным инструментом тестовый Go-файл является подстановкой имён и типов параметров тестируемой функции и другой важной информации в используемый шаблон. Данное решение позволяет при необходимости достаточно просто улучшать генерируемые тестовые Go-файлы посредством изменения файла-шаблона.

Файл-шаблон `utbotgo/utils/other_files/test.go_template` представляет собой обычный Go-код, в котором каждая подстрока, находящаяся в двух парах фигурных скобок `{{...}}`, заменяется на соответствующее значение из входного JSON-файла с описанием имён и типов параметров функции и другой информацией.

3. Реализация

В рамках данной выпускной квалификационной работы был разработан инструмент для генерации модульных тестов для функций языка Go, а именно были разработаны различные утилиты, Docker-контейнер для их работы и организовано взаимодействие утилит друг с другом посредством генерации артефактов. Также была написана утилита **utbotgo**, предоставляющая простой и понятный интерфейс пользователя для взаимодействия с разработанным решением. Разработка проекта велась с применением языков Go, Makefile, C++, Dockerfile, CMake, Shell и Python.

В этом разделе будет описано, как установить и использовать утилиту **utbotgo** и как происходит процесс генерации и запуска тестов.

3.1. Установка утилиты utbotgo

Для сборки и работы утилиты **utbotgo** потребуется компьютер с операционной системой семейства GNU/Linux. Также необходимым условием для установки и работы данной утилиты является наличие таких утилит как **bash**, **make** и **docker**.

Для установки утилиты **utbotgo** на компьютер необходимо вызвать команду **make install**, находясь в папке `utbotgo` проекта. Следует отметить, что эта команда может потребовать root права для правильной работы. Рассмотренная команда скачает по сети Интернет разработанный Docker-образ и установит утилиту **utbotgo**. Путь установки, а также версия и название Docker-образа описаны в файле `utbotgo/.env`.

При перемещении директории с данным проектом необходимо пересобрать утилиту **utbotgo** поскольку при установке она “запоминает” абсолютный путь к директории проекта.

3.2. Допустимые команды

Утилита **utbotgo** при запуске принимает одну команду и в зависимости от указанной команды выполняет те или иные действия. Флаги к

командам не передаются. Вся настройка работы инструмента хранится в файле конфигурации, о чём пойдёт речь в подразделе 3.3.

Вышеупомянутая утилита принимает одну из следующих команд:

- **init** – создаёт начальную конфигурацию для последующей настройки работы утилиты;
- **generate** – генерирует наборы значений параметров и тестовые Go-файлы для указанных функций, также может генерировать указанный вторым аргументом артефакт;
- **test** – делает то же, что и **generate**, но с последующим запуском сгенерированных тестов;
- **update_answers** – принимает текущие возвращаемые значения функций как правильные ответы;
- **clean** – уничтожает все сгенерированные тесты.

3.3. Файл конфигурации

Настройка работы инструмента производится посредством изменения файла конфигурации `utbotgo/config.yml`. Данный файл генерируется после вызова команды **utbotgo init**.

Структура файла `utbotgo/config.yml` состоит из единственного раздела **tested-functions**, содержащего список имён функций, для которых необходимо сгенерировать тестовые Go-файлы и наборы значений параметров.

Например, если необходимо настроить фаззер на генерацию тестов для функций `f` и `someOtherFunction`, файл конфигурации может выглядеть следующим образом:

```
tested-functions:
  - f
  - someOtherFunction
```

При редактировании файла конфигурации наборы значений параметров и тестовые Go-файлы тех функций, которые были удалены из списка **tested-functions**, также будут удалены при первом вызове команды **utbotgo generate** или **utbotgo test**.

3.4. Использование утилиты utbotgo

На рис. 3 показана диаграмма деятельности, отражающая предполагаемый вариант использования утилиты **utbotgo** при разработке Go-пакета.

Для того, чтобы начать использование разработанного фаззера, необходимо запустить команду **utbotgo init** для создания начальной конфигурации инструмента, после чего требуется изменить файл конфигурации с целью задания тех функций, работу которых нужно протестировать. Все файлы, генерируемые во время работы утилиты **utbotgo**, сохраняются в текущем каталоге, так что при необходимости можно скопировать сгенерированные артефакты на другой компьютер и запустить на нём модульные тесты без установки утилиты **utbotgo**.

После изменения файла конфигурации следует запустить команду **utbotgo generate** для генерации значений параметров для указанных функций. Также можно вместо данной команды выполнить команду **utbotgo test**, которая после генерации тестов сразу их запускает. Однако для запуска тестов можно использовать и стандартную команду **go test**.

Поскольку, как было отмечено в разделе 2, инструмент в основном направлен на то, чтобы генерировать значения параметров тестируемых функций, то возникает вопрос, как с помощью данного инструмента проверять правильность работы тестируемых функций. Для этого поддерживаются два подхода:

1. **ручная проверка** – это проверка соответствия заявленной логики работы с получаемыми результатами вручную, то есть программист, реализовавший данную функцию, просматривает все

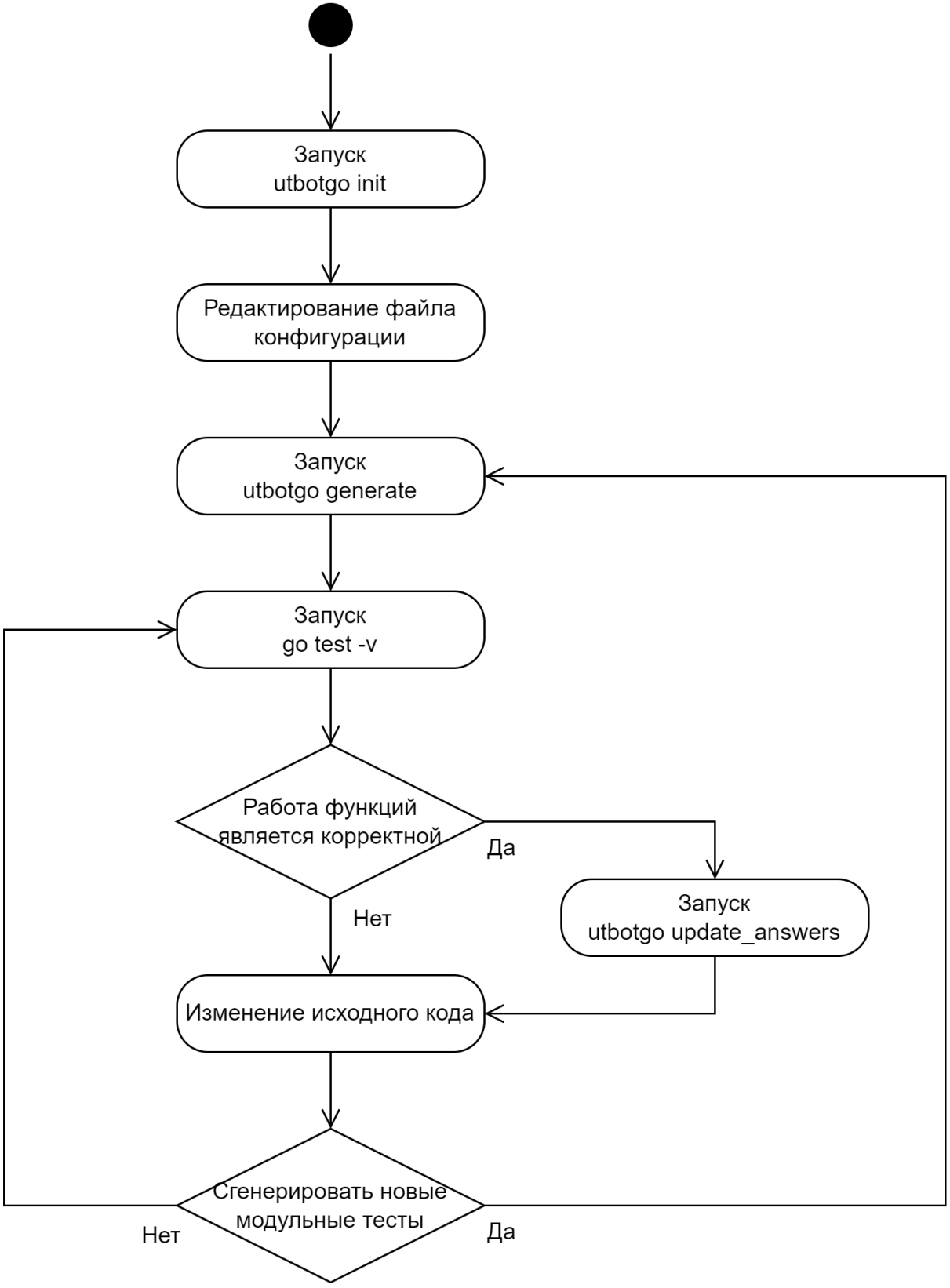

Рис. 3: Диаграмма деятельности использования утилиты **utbotgo** при разработке

возвращаемые значения и значения параметров и принимает решение, работает ли реализованная им функция так, как ожидалось, или нет;

2. **автоматическое обнаружение изменений** – это автоматизированное сравнение текущих результатов, полученных при вызове функции, с эталонными результатами, полученными при последнем запуске команды **utbotgo update_answers**.

Если текущая реализация проверяемых функций считается правильной, необходимо запустить команду **utbotgo update_answers**. Она сохраняет результаты работы функций как правильные и при следующих тестированиях сравнивает их с возвращаемыми значениями изменённых функций. Таким образом создаются модульные тесты для тестируемой функции.

После изменения исходного кода возможно возникновение необходимости сгенерировать новые наборы значений параметров для проверяемых функций, например, при изменении количества или типа параметров или возвращаемых значений. Для этого потребуется снова запустить команду **utbotgo generate** или **utbotgo test**.

Если необходимо удалить все артефакты, сгенерированные утилитой **utbotgo**, рекомендуется воспользоваться командой **utbotgo clean**.

3.5. Генерируемые артефакты

Основные артефакты, генерируемые утилитой **utbotgo**, представлены на рис. 4 ниже.

На рис. 4 `${F}` означает имя функции, указанной в файле конфигурации. Если функций больше одной, вместо `${F}` подставляется имя каждой указанной функции, таким образом генерируется, например, несколько тестовых Go-файлов, один на каждую указанную функцию.

Основные генерируемые артефакты следующие:

- `utbotgo_${F}_test.go` – тестовый Go-файл, содержащий в себе

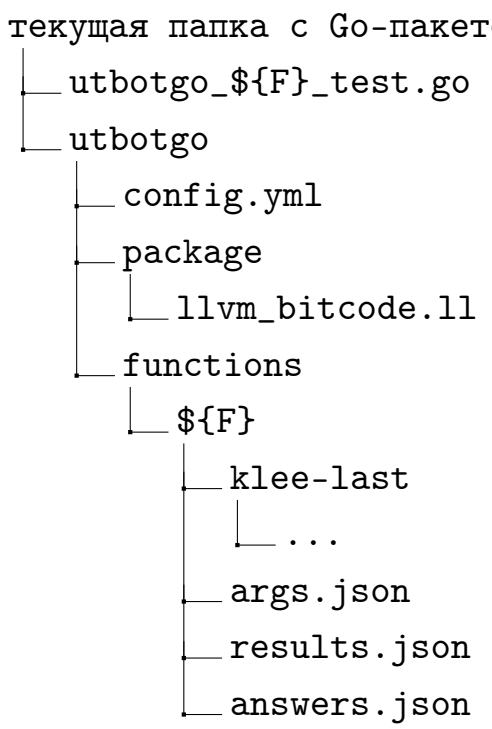

Рис. 4: Основные генерируемые утилитой **utbotgo** артефакты

все типы и функции, необходимые для тестирования функции с именем `${F}`;

- `utbotgo/package/ll_bitcode.ll` – сгенерированный с помощью компилятора **llvm-goc** [8] LLVM биткод целого Go-пакета;
- `utbotgo/functions/${F}/klee-last/` – ссылка на папку с последней версией набора значений параметров функции `${F}`, сгенерированная утилитой **klee** [9];
- `utbotgo/functions/${F}/args.json` – набор значений параметров функции `${F}`;
- `utbotgo/functions/${F}/results.json` – результаты работы функции `${F}` на наборе значений параметров, полученные при последнем запуске тестов;
- `utbotgo/functions/${F}/answers.json` – эталонные результаты работы функции `${F}` на наборе значений параметров.

3.6. Алгоритм тестирования

Файлы `args.json`, `results.json` и `answers.json` содержат JSON-массивы равной длины. Для любого допустимого индекса `i` `i`-ый тест заключается в вызове функции с набором параметров из массива в `args.json` с индексом `i`, сравнении полученного результата со значением из массива в `answers.json` с индексом `i` и записи результата в массив в `results.json` по индексу `i`.

Псевдокод данного алгоритма:

Algorithm 1 Test #i for function F

```
1: procedure TEST_${F}[I]
2:     params ← args(i)
3:     result ← ${F}(params)
4:     if answers exists then
5:         answer ← answers(i)
6:         assert if result = answer
7:     results(i) ← result
```

3.7. Ограничения на тестируемые функции

Существует несколько ограничений, накладываемых на те функции, для которых происходит генерация тестов. Если функция не удовлетворяет одному из этих ограничений, она либо не будет обрабатываться, либо обработается неправильно.

1. Результат функции не должен зависеть от глобальных переменных и не должен вызывать функцию, результат которой зависит от глобальных переменных.

2. Функция не должна вызывать функцию из другого пакета; при необходимости рекомендуется перенести весь используемый код из других пакетов в текущий, при этом код не должен содержать вставок кода на языке C/C++ или на ассемблере.

3. Все параметры функции должны быть целочисленными; этому ограничению соответствуют следующие типы языка Go: `rune`, `int`,

`int8`, `int16`, `int32`, `int64`, `byte`, `uint`, `uint8`, `uint16`, `uint32` и `uint64`.

4. Функция должна возвращать ровно одно значение.

5. Тип возвращаемого значения должен быть простым базовым Go-типом, сложные типы, такие как массив, карта и структура, не допускаются.

4. Эксперименты

Поскольку разработанный инструмент является генератором значений параметров тестируемых функций, имеет смысл провести эксперименты с целью выявления соответствия сгенерированных значений параметров с теми, которые точно покрывают весь код тестируемой функции.

В данном разделе будут описаны набор тестовых Go-проектов и полученные результаты.

4.1. Набор тестов

Набор тестовых Go-пакетов находится в папке `examples` проекта на GitHub. С помощью этих тестовых Go-пакетов проверяется следующая функциональность разработанного инструмента (в скобках перечисляются тестовые Go-пакеты, проверяющие указанную функциональность):

- работа с многофайловыми Go-пакетами (`test1`);
- работа с исполняемым Go-пакетом (`test1`);
- работа с несколькими тестовыми функциями (`test2`, `test3`);
- работа с арифметическими выражениями (`test2`);
- работа с оператором `if` (`test1`, `test3`);
- работа с циклом `for` (`test3`).

Также важно отметить, что в Go-пакете `test3` решение работает с вызовом функции `runtime.concatstrings` из стандартной библиотеки Go, что выходит за рамки описанных ограничений, однако других подобных случаев работы разработанного инструмента с внешними функциями пока не обнаружено.

4.2. Результаты

Эксперименты проводились с помощью инструмента GitHub Actions [6]. Для проведения экспериментов для каждого тестового Go-пакета была разработана утилита для проверки сгенерированных значений параметров на соответствие минимальным требованиям и для генерации правильных результатов для каждого вызова функции.

Эксперименты прошли успешно. Они показали, что текущее решение, разработанное в рамках данной выпускной квалификационной работы, работоспособно, однако ограничения, накладываемые на тестируемые функции, описанные в подразделе 3.7, являются критическими и не позволяют использовать текущее решение в реальных проектах. Следовательно, разработанный фаззер является доказательством возможности использования проектов gollvm [8] и klee [9] для генерации модульных тестов для проектов на языке Go, однако инструмент для возможности применения в реальных проектах требует доработок в будущем.

Заключение

В ходе выполнения данной выпускной квалификационной работы были достигнуты следующие результаты.

1. Проведён анализ существующих решений для генерации модульных тестов для кода на языке Go, за основу разрабатываемого инструмента был взят проект gollvm.

2. Разработана архитектура инструмента с использованием проектов gollvm и klee.

3. Реализован инструмент, состоящий из утилит, работающих в Docker-контейнере и взаимодействующих друг с другом.

4. Проведено тестирование работы реализованного инструмента на различных тестовых Go-проектах, находящихся в папке `examples`.

Репозиторий с исходным кодом инструмента находится здесь:
https://github.com/Software-Analysis-Team/UTBotGo

Список литературы

[1] Build-in Fuzzing. — Last access: 02 May 2022. URL: https://go.dev/doc/fuzz/.

[2] CMake. — Last access: 02 May 2022. URL: https://cmake.org/.

[3] Christopher Shao (cshao) Grace Yin (graceyin) Justin Restivo (jrestivo). DUCKEE GO: Dynamic and User-friendly ConcoliK Execution Engine in GO. — 2018. — Last access: 16 December 2021. URL: https://css.csail.mit.edu/6.858/2018/projects/cshao-graceyin-jrestivo.pdf.

[4] Clang. — Last access: 02 May 2022. URL: https://clang.llvm.org/.

[5] Docker. — Last access: 02 May 2022. URL: https://www.docker.com/.

[6] GitHub Actions. — Last access: 03 May 2022. URL: https://docs.github.com/en/actions.

[7] Go frontend. — Last access: 02 May 2022. URL: https://go.googlesource.com/gofrontend/.

[8] Gollvm project. — Last access: 02 May 2022. URL: https://go.googlesource.com/gollvm/.

[9] KLEE project. — Last access: 02 May 2022. URL: https://klee.github.io/.

[10] LLVM project. — Last access: 02 May 2022. URL: https://llvm.org/.

[11] Z3 SMT-solver. — Last access: 16 December 2021. URL: https://github.com/Z3Prover/z3.

[12] go-fuzz. — Last access: 16 December 2021. URL: https://github.com/dvyukov/go-fuzz.

[13] gofuzz. — Last access: 16 December 2021. URL: https://github.com/google/gofuzz.