DOI: 10.15514/ISPRAS-2020-32(2)-13

Платформа автоматического фаззинга программного интерфейса приложений

[1]*С.С. Саргсян, ORCID: 0000-0002-8831-4965 <sevaksargsyan@ispras.ru>*
[1]*В.Г. Варданян, ORCID: 0000-0002-4899-2999 <vaag@ispras.ru>*
[1]*Дж.А. Акопян, ORCID: 0000-0002-4094-2727 <jivan@ispras.ru>*
[1]*А.М. Агабалян, ORCID: 0000-0003-2331-6692 <anna.aghabalyan@ispras.ru>*
[1]*М.С. Меграбян, ORCID: 0000-0001-9846-3414 <matos@ispras.ru>*
[2]*Ш.Ф. Курмангалеев, ORCID: 0000-0002-0558-2850 <kursh@ispras.ru>*
[2]*А.Ю. Герасимов, ORCID: 0000-0001-9964-5850 <agerasimov@ispras.ru>*
[2]*М.К. Ермаков, ORCID: 0000-0002-0483-6097 <mermakov@ispras.ru>*
[2]*С.П. Вартанов, ORCID: 0000-0003-3786-2248 <svartanov@ispras.ru>*

[1]*Российско-Армянский университет,*
0051, Армения, г. Ереван, ул. Овсепа Эмина 123
[2]*Институт системного программирования им. В.П. Иванникова РАН,*
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Рандомизированное тестирование приложений (фаззинг, фаззинг-тестирование) является одним из широко используемых методов поиска ошибок. Цель фаззинг-тестирования – определить стабильность приложений при обработке псевдослучайно сгенерированных входных данных. В ходе тестирования приложение запускается на множестве произвольных входных данных, которые могут быть недействительными/неожиданными. Современное программное обеспечение часто предоставляет программный интерфейс (Application programming interface) для расширения возможностей программы. Это еще больше усложняет тестирование программного обеспечения, поскольку становится необходимым учитывать все возможные сценарии использования предоставленных интерфейсных функций. Применение фаззинга для генерации разных сценариев использования программного интерфейса приложения и соответствующих входных данных позволяет эффективным образом выявить ошибки в реализации функций программного интерфейса. В данной статье описывается новый метод фаззинг-тестирования для Android приложений и библиотек, написанных на языке Java. Разработанный инструмент фаззинг-тестирования выявил 15 уникальных дефектов, приводящих к аварийному завершению приложения SmartThings, разработанного компанией Samsung.

Ключевые слова: интернет вещей; фаззинг; генерация вызовов API

Для цитирования: Саргсян С.С., Варданян В.Г., Акопян Дж.А., Агабалян А.М., Меграбян М.С., Курмангалеев Ш.Ф., Герасимов А.Ю., Ермаков М.К., Вартанов С.П. Платформа автоматического фаззинга программного интерфейса приложений. Труды ИСП РАН, том 32, вып. 2, 2020 г., стр. 161-174. DOI: 10.15514/ISPRAS–2020–32(2)–13

Благодарности: Работа поддержана компанией Samsung-Electronics.

Automatic API fuzzing framework

[1]*S.S. Sargsyan, ORCID: 0000-0002-8831-4965 <sevaksargsyan@ispras.ru>*
[1]*V.G. Vardanyan, ORCID: 0000-0002-4899-2999 <vaag@ispras.ru>*
[1]*J.A. Hakobyan, ORCID: 0000-0002-4094-2727 <jivan@ispras.ru>*
[1]*A.M. Aghabalyan, ORCID: 0000-0003-2331-6692 <anna.aghabalyan@ispras.ru>*
[1] *M.S. Mehrabyan, ORCID: 0000-0001-9846-3414 <matos@ispras.ru>*
[2]*Sh.F. Kurmangaleev, ORCID: 0000-0002-0558-2850 <kursh@ispras.ru>*
[2]*A.Yu. Gerasimov, ORCID: 0000-0001-9964-5850 <agerasimov@ispras.ru>*
[2]*M.K. Ermakov, ORCID: 0000-0002-0483-6097 <mermakov@ispras.ru>*
[2]*S.P. Vartanov, ORCID: 0000-0003-3786-2248 <svartanov@ispras.ru>*

[1] *Russian-Armenian University,*
123 Hovsep Emin str., Yerevan, 0051, Armenia
[2]*Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Randomized testing (fuzzing) is a well-known approach for finding bugs in programs. Fuzzing is typically performed during the finishing stage of quality assurance in order to check the stability of the target program in the face of malformed or unexpected input data. Modern software more than often provides an API for extending its functionality by third-party developers; since an API is an entry point to software internals, its functionality and usage scenarios must be tested as well. Thorough API testing must involve checking a large number of possible scenarios and it is fairly obvious that fuzzing can be applied to this task by generating usage scenarios in an automatic randomized way—which brings us to the concept of API fuzzing. In this paper we describe an automatic approach to randomized testing of API libraries for Android/desktop Java. Proposed method is able to change the sequence of called API functions in order to discover new execution paths. It consists of two basic stages. In the first stage the arguments of currently called API functions are mutated. When mutation of called API functions arguments can't find new execution path the tool switches to the second stage. In the second stage current sequence of API functions calls is mutated. Mutation can add new API functions calls or remove some of them. After API calls sequence mutation, the tool switches back to the first stage. Switches between the first and the second stages are continued during whole process of fuzzing. During the experimental setup developed method of randomized testing were able to find 15 crashes in SmartThings application developed by Samsung.

Keywords: internet of things; fuzzing; API call generation

For citation: Sargsyan S.S., Vardanyan V.G., Hakobyan J.A., Aghabalyan A.M., Mehrabyan M.S., Kurmangaleev Sh.F., Gerasimov A.Yu., Ermakov M.K., Vartanov S.P. Automatic API fuzzing framework. Trudy ISP RAN/Proc. ISP RAS, vol. 32, issue 2, 2020. pp. 161-174 (in Russian). DOI: 10.15514/ISPRAS–2020–32(2)–13

Acknowledgements: The work has been done with support of Samsung Electronics.

1. Введение

Контроль качества программного обеспечения является одной из наиболее важных задач при его разработке. Для решения этой проблемы компании внедряют в жизненный цикл разработки программного обеспечения различные инструменты анализа программ [1]. Спектр инструментов достаточно широк и может включать в себя статический и динамический анализ кода программ. Фаззинг [2] является одним из наиболее распространенных и эффективных методов динамического анализа. Метод применялся ещё в середине 1950-х годов, но первые был описан в работе посвященной оценке надёжности Unix утилит [3]. Эффективность предложенного метода подтверждается множеством дефектов, найденных на разных утилитах из инструментария UNIX [4]. Методы и подходы фаззинга постепенно совершенствуются, и в данный момент существует множество инструментов, предназначенных для фаззинг-тестирования. AFL [5] использует

генетический алгоритм выборки входных данных, что позволяет эффективно увеличивать покрытие кода. В работе [6] описывается генерация структурированных данных, заданных в форме BNF, для фаззинга компиляторов, интерпретаторов и трансляторов. Инструмент [7] производит направленный фаззинг на основе динамической инструментации целевой программы. В работе [8] фаззинг совмещается с динамическим символьным исполнением программы с целю увеличения покрытия кода. Проект libFuzzer [9] используется для фаззинга библиотечных функций написанных на языке C/C++. Разработчик должен дописать код функции-обёртки для фаззинга конкретной функции из библиотеки. Программные инструменты Trinity [10] и syzkaller [11] предназначены для фаззинга системных вызовов операционной системы Linux [12]. Эти инструменты генерируют множество маленьких программ, содержащих системные вызовы, и запускают их в окружении целевой операционной системы. При запуске производится мониторинг состояний системы в целях обнаружения сбоев.

Для фаззинга библиотечных/интерфейсных функций существующие инструменты используют либо аннотации этих функций, либо адаптеры, которые обеспечивают их вызов. Это приводит к невозможности полностью автоматического их применения для новых библиотечных или интерфейсных функций. Современное программное обеспечение часто предоставляет большое число интерфейсных функций, которые используются для реализации подключаемых модулей и позволяют расширять функциональные возможности программы. Предоставляемые интерфейсные функции должны иметь конкретную спецификацию и сценарии использования. Система предоставляющая интерфейсные функций должна уметь обрабатывать все недействительные сценарии их использования. В общем случае задача может быть неразрешимой, поскольку количество сценариев возрастает экспоненциально от количества предоставляемых функций. Применение фаззинга к таким системам может эффективным образом выявлять недействительные сценарии использования предоставляемой функциональности.

В данной работе предлагается новый подход для фаззинга интерфейсных функций из библиотек языка Java (JAR файлы [13]). Предлагаемый метод имеет несколько преимуществ:

1. Полностью автоматически восстанавливаются аннотации функций и классов из JAR файла, что позволяет при изменении версии библиотеки (добавление/удаление интерфейсных функций) заново генерировать аннотации.
2. Создается генератор последовательности вызовов интерфейсных функций и их аргументов, который может отправлять сгенерированные данные на выбранные устройства (например, мобильные телефоны). В том числе он может производить параллельный запуск функций на множестве устройств с синхронизацией результатов.
3. Обеспечивается поддержка шаблонов начальных сценариев использования интерфейсных функций (например, инициализация и очищение ресурсов до или после основных вызовов интерфейсных функций). Это позволяет эффективно увеличивать покрытие кода и снижать количество неправильных сценариев, увеличивая эффективность применения фаззера.

Предлагаемый подход был реализован в рамках инструмента ISP-Fuzzer [14], разработанного в Институте системного программирования им. В. П. Иванникова РАН и протестированного на библиотеке PluginBase, которая входит в состав платформы SmartThings [15], созданной компанией Samsung [16]. В результате экспериментального запуска разработанный инструмент смог выявить 15 уникальных дефектов, приводящих к отказу системы, что демонстрирует эффективность предлагаемого решения. Полученные результаты были подтверждены компанией Samsung.

2. Высокоуровневое описание платформы

На рис. 1 приводится общая схема предлагаемой платформы. Серверная часть инструмента производит генерацию данных (последовательность вызовов интерфейсных функций с соответствующими аргументами), которые будут выполнены на клиентской стороне. Серверная часть может одновременно работать с множеством клиентских приложений и синхронизовать полученные результаты. Это позволяет многократно увеличить эффективность фаззинга.

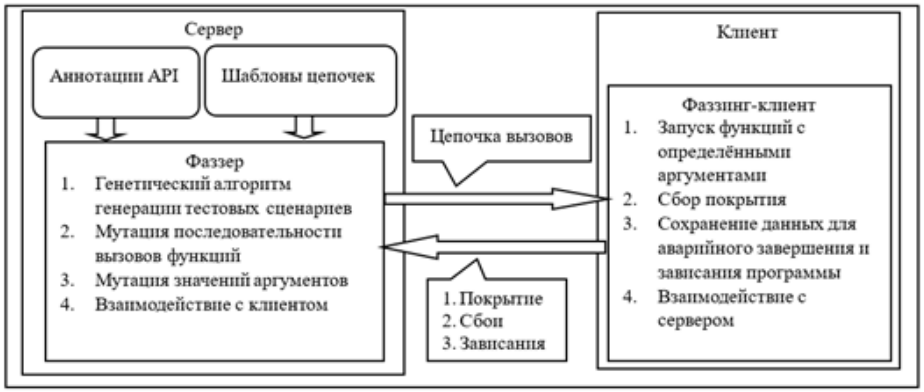

Рис. 1. Общая схема предложенной платформы
Fig. 1. The general scheme of the proposed platform

На основе аннотаций функций, которые передаются инструменту, строятся последовательности (цепочки) вызовов интерфейсных функций. Разработан специальный протокол передачи данных между сервером и клиентами, который позволяет отправлять сгенерированную последовательность вызовов интерфейсных функций и получать необходимое покрытие кода. В зависимости от того, какое покрытие обеспечивает сгенерированная последовательность вызовов, генетический алгоритм подбирает следующую последовательность вызовов и набор аргументов. Последовательности вызовов, которые привели к аварийному завершению или зависанию программы, сохраняются для последующего анализа экспертами.

С помощью конфигурационного файла можно указать шаблоны генерации новых последовательностей вызовов. Например, можно указать интерфейсные функций, которые будут вызваны перед и после каждой сгенерированной последовательности вызовов для реализации специфического протокола использования программного интерфейса.

2.1 Фаззинг интерфейсных функций на примере библиотеки PluginBase

На рис. 2 приводится схема фаззинга интерфейсных функций из библиотеки PluginBase (JAR файл), входящей в состав SmartThings. SmartThings платформа связывает разные устройства интернета вещей и осуществляет управление ими. Управление устройствами производится через интерфейсные функции, входящие в библиотеку PluginBase. Для выполнения фаззинга библиотеки PluginBase были разработаны два инструмента. Первый инструмент генерирует аннотации для классов и методов, входящих в JAR библиотеку. На основе этих аннотаций генерируются последовательности вызовов интерфейсных функций для исполнения на клиентской машине. Если функция получает как аргумент производный объект, система сгенерирует код для создания этого объекта. Построение сложных объектов производится следующим образом:

1. Если есть интерфейсная функция, которая возвращает объект данного типа, производится вызов этой функции.
2. В противном случае вызывается конструктор данного объекта. Если конструктор принимает как аргумент сложный объект, процесс повторяется рекурсивно пока все аргументы создаваемого объекта не будут иметь примитивный тип или тип, сконструированный из примитивных типов.

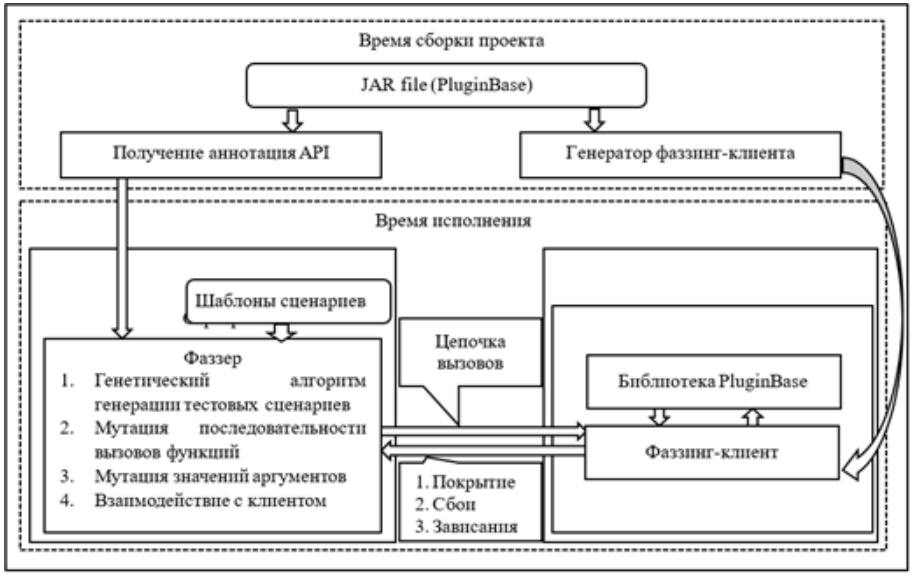

Рис. 2. Схема фаззинга интерфейсных функций PluginBase
Fig. 2. Fuzzing scheme of PluginBase interface functions

Второй инструмент генерирует встраиваемый модуль (плагин) для приложений SmartThings, который получает последовательность вызовов интерфейсных функций от сервера и выполняет их на клиенте (мобильном телефоне). Реализация встраиваемого модуля представляет собой конечный автомат, поддерживающий протокол общения с сервером (получает последовательность команд, отправляет покрытие кода и информацию о падениях/зависаниях приложения). Дополнительно инструментируется код самой библиотеки PluginBase для получения покрытия кода в процессе исполнения цепочек вызовов.

2.2 Генерация вызовов интерфейсных функций

Генерация последовательности вызовов API функций состоит из двух основных этапов. На первом этапе производится вызов всех API функций (разминка – warm-up). Цель данного этапа – сбор начального покрытия кода, который обеспечивает эффективную работу генетического алгоритма. Если этот этап пропущен, добавление вызова новой функции в цепочку вызовов всегда будет увеличивать покрытие кода и генетический алгоритм не сможет отличить эффективные мутации последовательности вызовов и аргументов функций от неэффективных. Кроме этого, на данном этапе всем функциям передаются нулевые указатели на объекты, в целях определения необработанных случаев использования нулевых

указателей (*NullPointerException*). На этапе разминки, как правило, достигается покрытие кода по базовым блокам в 30-40%.

На втором этапе (рис. 3) начинается основной цикл фаззинга (мутация цепочки вызовов функций и их аргументов). Произвольным образом генерируется набор начальных цепочек (мутацию цепочки вызовов можно ограничить через конфигурационный файл шаблонов цепочек). Одна из начальных цепочек вызовов функций загружается для обработки. Производятся мутации всех аргументов функций из цепочки, которые продолжаются до тех пор, пока покрытие кода перестает увеличивается. В этот момент инструмент переходит к мутациям последовательности вызовов функций в цепочке. В процессе этой мутации могут быть добавлены или удалены некоторые вызовы. Если сгенерированная цепочка вызовов увеличивает покрытие кода, она сохраняется для дальнейшей обработки. Чем больше увеличивается покрытие кода, тем дольше будут обрабатываться соответствующая цепочка вызовов и аргументы функций. Наблюдаемый прирост покрытия на второй стадии фаззинга в течение недели составляет ещё 30-40%. Пользователь может управлять процессом мутации, задав в шаблонном файле следующие параметры:

1. последовательности вызовов функций, которые должны быть обработаны первыми;
2. набор мутаций для аргументов функций;
3. набор функций, которые должны быть вызваны до и после каждой цепочки вызовов;
4. зависимости между функциями. В частности, возвращаемое значение одной функций должно быть использовано, как аргумент вызова другой.

Рис. 3. Схема основного цикла фаззинга на серверной машине
Fig. 3. The scheme of the main fuzzing cycle on the server machine

После того, как цепочка вызовов функций и соответствующие аргументы сгенерированы, они передаются следующему компоненту (генератор последовательности команд выполнения – ГПКВ, Execution command sequence generator). ГПКВ получает последовательность команд,

выполнение которых на клиентской стороне обеспечит вызовы интерфейсных функций с соответствующими аргументами.

2.2.1. Мутация данных

Модуль мутации данных получает на вход начальное значение (может быть пустым) и длину требуемой последовательности данных. Возвращает мутированные/сгенерированные данные заданной длинны. В инструменте поддерживается следующий набор мутаций данных:

1. расширение данных: с одинаковым произвольным значением, произвольными значениями, с нулевыми байтами;
2. сокращение данных путем удаления произвольного количества байтов;
3. модификация данных: присвоение последовательности байтов произвольных значения, присвоение последовательности байтов одного и того же произвольного значения, присвоение произвольному интервалу байтов нулевого/произвольного значения;
4. инверсия произвольного бита;
5. генерация строковых данных: повторение строки, создание не валидного списка UTF-8 строк, создание BOM (byte order mark U+FEFF);
6. модификация строковых данных: перевод символов в нижний/верхний регистр;
7. модификация числовых значений: присвоение максимального/минимального значения, применение арифметических операций.

2.2.2. Мутация цепочек вызовов интерфейсных функций

В ходе мутаций цепочек вызовов используется информация о зависимостях интерфейсных функций (автоматически восстановленная или указанная пользователем). Это делается в целях получения семантически «связанных» программ, когда возвращаемое значение одной из функций используется другой. При создании новой цепочки вызовов произвольным образом выбирается некоторая функция *f* из списка доступных интерфейсных функций. Если существует функция, возвращаемое значение которой может быть использовано как аргумент функции *f*, то перед вызовом *f* добавляется вызов этой функции, а возвращаемое значение передаётся как аргумент вызову *f*. Процесс повторяется рекурсивно для добавленных функций до тех пор, пока длина цепочки не превысит заданную границу.

В случае мутаций цепочки производится добавление/удаление некоторой последовательности вызовов, которые не зависят друг от друга. Максимальная длина таких последовательностей задается пользователем. На мутации цепочек вызовов можно повлиять с помощью конфигурационного файла шаблонов. Предоставляется возможность:

1. определить набор функций, которые будут вызваны перед/после каждой цепочки;
2. определить конкретные значения или интервалы значений для аргументов функций;
3. составить список функций, которые могут или не могут присутствовать в цепочках вызовов;
4. задавать зависимости между функциями.

2.2.3. Генерация последовательности команд выполнения

Инструмент имеет два основных назначения:

1. для данной цепочки вызовов вычислять, сколько данных потребуется, чтобы получить конкретные аргументы всех функций;
2. генерировать последовательность байтов, которая будет выполнена/интерпретирована на клиентской машине.

Для выполнения первой своей задачи инструменту необходима цепочка вызовов и аннотаций интерфейсных функций и классов. Для выполнения второй задачи инструменту также необходимо получить конкретные аргументы функций (данные от модуля мутации данных).

Вычисление размера необходимых данных (задача 1) производится рекурсивным обходом составных типов сложного объекта. Размер данных вычисляется суммированием размера составных типов. На этом этапе собирается дополнительная информация о возможных значениях конкретных аргументов. Есть три основных вида таких значений: *MustBeNull* (аргумент всегда должен иметь значение null), *NotNull* (аргумент никогда не должен иметь значение null), *Any* (аргумент может иметь любое значение). Эта информация передается модулю мутаций для генерации корректных данных. Это обеспечивает семантическую корректность вызовов функций. Например, некоторые примитивные типы в Java не могут иметь значение null. Если есть функция с аргументом такого типа, то вызов этой функции (вызов производит клиент фаззинга – интерпретатор последовательности команд) с null аргументом приведет к некорректному поведению программы. На практике такая ситуация возникнуть не может, поскольку компилятор Java не позволит провести компиляцию такого кода.

Табл. 1. Коды операций регистровой машины
Table 1. Codes of operations of the register machine

Описание команды	Код операций	Операнды
Положить значение в переменную	1-8 (в зависимости от типа)	«переменная», «значение»
Положить UTF значение в переменную	9	«переменная», «UTF строка»
Положить массив в переменную	10	«переменная», «идентификатор типа», «длина», {«значении»}
Положить null в переменную	11	«переменная»
Положить строку в переменную	12	«переменная», «длина», {«значении»}
Положить массив переменных в переменную	13	«переменная», «длина», {«переменные»}
Вызов метода	«method ID»	[«переменная»], «ссылка объекта», {«параметры метода»}
Вызов статического метода	«method ID»	[«переменная»], {«параметры метода»}
Получить значения поля объекта	«get field ID»	«переменная», «переменная, в которой ссылка на объект»
Получить значения статического поля	«get field ID»	«переменная»
Положить значение в поле объекта	«put field ID»	«переменная, в которой ссылка на объект», «переменная, в которой значение»
Положить значение в статическое поле	«put field ID»	«переменная, в которой значение»

Генерация последовательности команд выполнения производится на основе простейшей регистровой машины. Каждая операция (положить значение в переменную, вызвать функцию и так далее) кодируется в протоколе (в табл. 1 приведены коды). Алгоритм проходит по цепочке вызовов функций и получает последовательность команд для регистровой машины.

2.3 Интерпретатор команд

Интерпретатор представляет собой оператор-переключатель, который в зависимости от кода операций выполняет соответствующую операцию. Также имеется буфер переменных, в

котором хранятся необходимые переменные (это, по сути, регистры регистровой машины). В табл. 1 приводятся доступные коды операций. Каждый метод и класс (также поля класса) имеют уникальный идентификатор ID, который позволяет определить нужный метод, класс и поля класса. В таблице этот идентификатор является кодом некоторой команды. На рис. 4 приводится фрагмент кода и соответствующая последовательность команд регистровой машины.

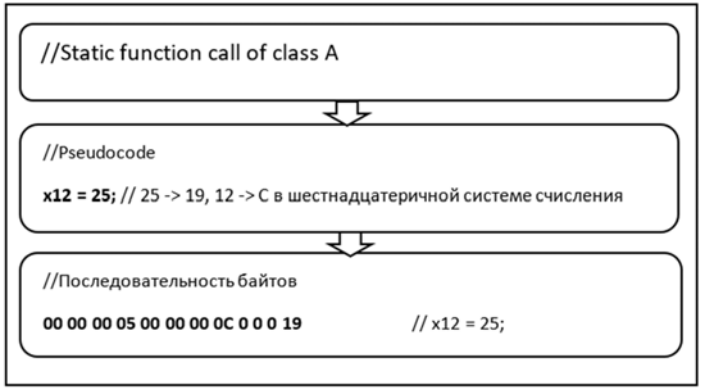

Рис. 4. Пример последовательности команд
Fig. 4. Example sequence of commands

Авторами разработан отдельный инструмент, который из аннотаций JAR-файла генерирует интерпретатор. Это позволяет при обновлении библиотеки автоматически генерировать новый интерпретатор, поддерживающий вызовы добавленных/модифицированных функций.

2.4 Получение покрытия кода

Для получения покрытия кода используется техника, реализованная в инструменте AFL [5]:

1. Каждому базовому блоку целевой библиотеки сопоставляется целое число из интервала [0, 65536). Для сбора покрытия используются две глобальные переменные: карта памяти (map) в размере 65536 элементов и переменная *prevLocation*. Обе переменные инициализированы нулями.
2. При выполнении базового блока X, которому было присвоено целое число bbIndexX (шаг 1), производится следующее изменение в карте памяти:

```
id := bbIndexX ^ prevLocation;
map[id] := map[id] + 1;
prevLocation := bbIndexX >> 1;
```

   Таким образом в карте памяти хранится информация о выполненных базовых блоках.

Такой подход позволяет выявить уникальные пути выполнения (хотя возможны коллизии). Реализация инструментации была выполнена на основе платформы ASM [17].

3. Результаты

В ходе экспериментального запуска на библиотеке PluginBase входящий в состав платформы SmartThings (версия 1.7.9) разработанный инструмент выявил 15 уникальных аварийных завершений программы. Результаты были подтверждены командой тестирования Samsung Electronics R&D. На рис. 5 приводятся некоторые методы, вызовы которых с указанными параметрами приводят к возникновению исключительных ситуаций. Как видно из результатов инструмент может найти не только «простые» ошибки, когда передаются null

объекты. Успешно были найдены исключения выхода индекса за пределы строки и другие типы исключений, определенных в самой платформе SmartThings. Данные результаты были получены в результате параллельного фаззинга на двух устройствах в течении семи дней. На рис. 6 приводится фрагмент кода, демонстрирующий выше указанные ошибки

```
// com.samsung.android.plugins.PluginDataStorageException
JSONConverter.jsonToRcsRep ("{}");

// java.lang.StringIndexOutOfBoundsException
ShpConverter.vidToShpSSID ("nohyphen");

// java.lang.NullPointerException
QcPluginDevice.getQcPluginDevice ("foo");

// os.android.RemoteException
AutomationManager.getInstance ().getRegisteredAutomation ("1");
```

Рис. 5. Найденные необработанные исключительные ситуации
Fig. 5. Unhandled exceptions found

```
package com.samsung.android.plugin.tv;
import android.os.Bundle;
import android.view.Window;
import android.view.WindowManager;
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.InetSocketAddress;

import com.samsung.android.oneconnect.utils.ShpConverter;
import com.samsung.android.plugins.QcPluginDevice;
import com.samsung.android.plugins.automation.AutomationManager;
import com.samsung.android.sccIient.JSONConverter;

public class MainActivity extends PluginBaseActivity {

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    final Window window = getWindow();
    window.setSoftInputMode(
        WindowManager.LayoutParams.SOFT_INPUT_ADJUST_RESIZE |
        WindowManager.LayoutParams.SOFT_INPUT_STATE_HIDDEN);
    getTheme().applyStyle(R.style.AppTheme, true);

    setContentView(R.layout.activity_main);

    System.out.println(" JSONConverter.jsonToRcsRep");
    try {
      JSONConverter.jsonToRcsRep("{}");
    } catch (Throwable e) {
      e.printStackTrace();
    }

    System.out.println(" ShpConverter.vidToShpSSID");
    try {
      ShpConverter.vidToShpSSID("nohyphen");
    } catch (Throwable e) {
      e.printStackTrace();
    }

    System.out.println(" QcPluginDevice.getQcPluginDevice");

    try {
      QcPluginDevice.getQcPluginDevice("foo");
    } catch (Throwable e) {
      e.printStackTrace();
    }

    System.out.println(" AutomationManager.getRegisteredAutomation");
    try {
      AutomationManager mgr = AutomationManager.getInstance();
      mgr.getRegisteredAutomation("1");
    } catch (Throwable e) {
      e.printStackTrace();
    }
  }
}
```

Рис. 6. Фрагмент кода, демонстрирующий найденные исключения
Fig. 6. Code fragment showing exceptions found

4. Сравнение с аналогичными решениями

EvoSuite [18] является одной из известных платформ, которая автоматически генерирует тесты для Java-классов. Инструмент учитывает покрытие кода, и путем генерации разных тестов может достичь большого/полного покрытия кода. EvoSuite также может генерировать JUnit4 [19] тесты. Данный инструмент имеет некоторые ограничения.

1. Нет возможности генерировать последовательности вызовов функций. Таким образом ошибки, которые воспроизводятся при некоторой последовательности вызовов функций не могут быть найдены.
2. Нельзя указать шаблоны для генерации последовательности вызовов функций. Например, указать корректное создание/освобождение ресурсов перед/после вызовов функций.
3. Если запуск программы производится в специальной среде, EvoSuite тоже не может быть применен. Например, в случае библиотеки PluginBase вызов её методов можно производить только в том случае, когда библиотека загружена через платформу SmartThings (в среде ОС Android [20]).

DSD-Crasher [21] (его предшественник – JCrasher [22]) генерирует тесты для методов Java классов в целях выявления необъявленных исключений. Он также умеет генерировать JUnit тесты. Для достижений этой цели инструмент комбинированно использует динамический и статический анализ. Как и в случае EvoSuite, этот инструмент не может решать поставленную задачу, поскольку имеет те же самые ограничения.

Randoop [23] в отличие EvoSuite и DSD-Crasher может вызывать несколько методов одного класса в одном тесте. Но вызов методов из разных классов не производится. Таким образом генерация последовательности вызовов функций решается частично. Кроме этого, Randoop не учитывает покрытие кода для генераций новых тестов, что является еще одним недостатком.

Таким образом, предложенный авторами метод имеет ряд преимуществ при рандомизированном тестировании интерфейсных функций по сравнению с вышеуказанными инструментами.

Список литературы / References

[1]. SDL. Available at: https://www.microsoft.com/en-us/securityengineering/sdl, accessed 26.11.2019.
[2]. Fuzzingю Available at: https://en.wikipedia.org/wiki/Fuzzing, accessed 26.11.2019.
[3]. Borton P. Miller, Lars Fedriksen. Bryan So. An Empirical Study of the Reliability of UNIX utilities. Communications of the ACM, vol. 33, № 12, 1990, pp. 32-44.
[4]. Unix. Available at: https://en.wikipedia.org/wiki/Unix, accessed 26.11.2019.
[5]. Michael Zelewski. American Fuzzy Lop. 2014. Available at: http://lcamtuf.coredump.cx/afl. Accessed 26.11.2019.
[6]. Sevak Sargsyan, Shamil Kurmangaleev, Matevos Mehrabyan, Maksim Mishechkin, Tsolak Ghukasyan, Sergey Asryan. Grammar-Based Fuzzing. In Proc. of the *2018 Ivannikov Memorial Workshop (IVMEM)*, Yerevan, Armenia, 2018, pp. 32-35, doi: 10.1109/IVMEM.2018.00013.
[7]. Sevak Sargsyan, Shamil Kurmangaleev, Jivan Hakobyan, Matevos Mehrabyan, Sergey Asryan, Hovhannes Movsisyan. Directed Fuzzing Based on Program Dynamic Instrumentation. In Proc. of the 2019 International Conference on Engineering Technologies and Computer Science (EnT), Moscow, Russia, 2019, pp. 30-33, doi: 10.1109/EnT.2019.00011.
[8]. Gerasimov A.Yu., Sargsyan S.S., Kurmangaleev S.F., Hakobyan J.A., Asryan S.A., Ermakov M.K. Combining dynamic symbolic execution and fuzzing. Trudy ISP RAN/Proc. ISP RAS, vol. 30, issue 6, 2018, pp. 25-38. DOI: 10.15514/ISPRAS-2018-30(6)-2.
[9]. libFuzzer. Available at: https://llvm.org/docs/LibFuzzer.html, accessed 26.11.2019.
[10]. Trinity: Linux system call fuzzer. Available at: https://github.com/kernelslacker/trinity, accessed 26.11.2019.

[11]. syzkaller: Linux syscall fuzzer. Available at: https://github.com/google/syzkaller syzkaller, accessed 26.11.2019.
[12]. Linux. Available at: https://www.linux.org/, accessed 26.11.2019.
[13]. JAR. Available at: https://en.wikipedia.org/wiki/JAR_(file_format), accessed 26.11.2019.
[14]. Sevak Sargsyan, Jivan Hakobyan, Matevos Mehrabyan, Maxim Mishechkin, Vitaliy Akozin, Shamil Kurmangaleev. ISP-Fuzzer: Extendable Fuzzing Framework. In Proc. of the 2019 Ivannikov Memorial Workshop (IVMEM), Velikiy Novgorod, Russia, 2019, pp. 68-71, doi: 10.1109/IVMEM.2019.00017.
[15]. SmartThings. Available at: https://www.samsung.com/global/galaxy/apps/smartthings/, accessed 26.11.2019.
[16]. Samsung. Available at: https://www.samsung.com, accessed 26.11.2019.
[17]. Bruneton E., Lenglet R., Coupaye T. ASM: A code manipulation tool to implement adaptable systems. In Proc. of the ASF (ACM SIGOPS France) Journées Composants 2002: Syst`emes `a composants adaptables et extensibles (Adaptable and Extensible Component Systems), Grenoble, France, 2002.
[18]. Gordon Fraser, Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, Szeged, Hungary, 2011, pp. 416-419.*
[19]. JUnit4. Available at: https://junit.org/junit4/, accessed 26.11.2019.
[20]. Android. Available at: https://www.android.com/, accessed 26.11.2019.
[21]. Christoph Csallner, Yannis Smaragdakis, Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. ACM Transactions on Software Engineering and Methodology, vol.17, issue 2, 2008, Article No. 8.
[22]. Christoph Csallner, Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. Software – Practice & Experience, vol. 34, issue 11, 2004, pp. 1025 - 1050.
[23]. Carlos Pacheco, Michael D. Ernst. Randoop: feedback-directed random testing for Java. In Proc.of the OOPSLA '07 Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion, Montreal, Quebec, Canada, 2007, pp. 815-816.

Информация об авторах / Information about authors

Севак Сеникович САРГСЯН – научный сотрудник, преподаватель, заведующий кафедрой, кандидат физико-математических наук. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Sevak Senikovich SARGSYAN – researcher, lecturer, head of department, Ph.D in physical and mathematical sciences. Research interests: program analysis, dynamic analysis of code, fuzzing.

Ваагн Геворгович ВАРДАНЯН – научный сотрудник, преподаватель, кандидат технических наук. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Vahagn Gevorgovich VARDANYAN – researcher, lecturer, Ph.D in technical sciences. Research interests: program analysis, dynamic analysis of code, fuzzing.

Дживан Андраникович АКОПЯН – научный сотрудник, преподаватель, аспирант. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Jivan Andranikovich HAKOBYAN – researcher, lecturer, PhD student. Research interests: program analysis, dynamic analysis of code, fuzzing.

Анна Мартиросовна АГАБАЛЯН – научный сотрудник, магистр. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Anna Martirosovna AGHABALYAN – researcher, master. Research interests: program analysis, dynamic analysis of code, fuzzing.

Матевос Саргисович Меграбян – научный сотрудник, магистр. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Matevos Sargisovich MEHRABYAN – researcher, master. Research interests: program analysis, dynamic analysis of code, fuzzing.

Шамиль Фаимович КУРМАНГАЛЕЕВ – кандидат физико-математических наук. Сфера научных интересов: анализ программ, динамический анализ кода, фаззинг.

Shamil Faimovich KURMANGALEEV – Senior researcher, Ph.D in physical and mathematical sciences. Research interests: program analysis, dynamic analysis of code, fuzzing.

Александр Юрьевич Герасимов – кандидат физико-математических наук. Сфера научных интересов: автоматический анализ программ, статический анализ программ, динамический анализ программ, комбинированные методы анализа программ, управление научными исследованиями и разработками, жизненный цикл разработки безопасного ПО.

Alexander Yurievich GERASIMOV – PhD in Computer Sciences. Research interests: automatic program analysis, static program analysis, dynamic program analysis, combined methods for program analysis, R&D management, SSDLC.

Михаил Кириллович ЕРМАКОВ – кандидат технических наук. Сфера научных интересов: динамический анализ кода, фаззинг, символьное исполнение, статический анализ кода.

Mikhail Kirilovich ERMAKOV – Candidate of Technical Sciences. Research interests: dynamic program analysis, fuzzing. Symbolic execution, static program analysis.

Сергей Павлович ВАРТАНОВ – сфера научных интересов: символьное исполнение, динамический и статический анализ программ и SMT-решатели.

Sergey Pavlovitch VARTANOV – research interests: symbolic execution, dynamic and static analysis, and SMT solvers.