УНИВЕРСИТЕТ ИТМО
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
ноябрь–декабрь 2021 Том 21 № 6 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
November–December 2021 Vol. 21 No 6 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)
НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК УНИВЕРСИТЕТ ИТМО
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

doi: 10.17586/2226-1494-2021-21-6-895-902
УДК 004.052.2

Стохастическое тестирование программного обеспечения для поиска уязвимостей

Андрей Олегович Манеев[1], Антон Игоревич Спивак[2]✉

[1,2] Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация

[1] maneevao@gmail.com, https://orcid.org/0000-0001-9166-0706
[2] Anton.spivak@gmail.com✉, https://orcid.org/0000-0002-6981-8754

Аннотация

Предмет исследования. Стохастическое тестирование при помощи фаззеров является одним из способов поиска уязвимостей в программном обеспечении. Процесс тестирования в общем случае представляет собой генерацию случайных входных данных для исследуемой программы и может занимать значительное время. Снижение времени тестирования — актуальная задача. Одно из направлений улучшения процесса генерации — выделение только тех последовательностей данных, которые значительно влияют на пути выполнения тестируемой программы. Таким образом, новый подход к формированию входных данных, для снижения общего времени тестирования, позволит найти большее количество уязвимостей программ. **Метод.** Предложена модификация генетического алгоритма, используемого популярным фаззером afl (American Fuzzy Lop). Для повышения эффективности генерации входных данных предложена модель перспективных позиций. В модели производится выбор наиболее перспективной с точки зрения обнаружения уязвимости позиции входных данных для дальнейших мутаций с использованием генетического алгоритма фаззера. В отличие от существующих подходов, представленная модель учитывает перспективность конкретного элемента входных данных для увеличения покрытия кода тестом и по данной причине направляет генетический алгоритм на изменение именно этой позиции. **Основные результаты.** Выполнено сравнение эффективности разработанной модели с популярным фаззером afl и его модификациями (aflfast, symfuzz, afl-rb). В ходе экспериментального исследования модель позволяет достичь в среднем на 21 % большее покрытие кода, чем другие представленные решения. Покрытие ветвей исполнения между базовыми блоками кода программы увеличено с 20 897,3 до 17 267,4. **Практическая значимость.** Предложенная модель может быть применена при тестировании программного обеспечения, предполагающего ввод и обработку пользовательских данных. При этом требуемые изменения необходимы только в компоненте генератора псевдослучайных чисел и не затрагивают весь инструмент автоматизированного тестирования.

Ключевые слова

тестирование, динамическое тестирование, стохастическое тестирование, уязвимость, фаззинг, Fuzzing

Ссылка для цитирования: Манеев А.О., Спивак А.И. Стохастическое тестирование программного обеспечения для поиска уязвимостей // Научно-технический вестник информационных технологий, механики и оптики. 2021. Т. 21, № 6. С. 895–902. doi: 10.17586/2226-1494-2021-21-6-895-902

Stochastic software testing for vulnerability analysis

Andrey O. Maneev[1], Anton I. Spivak[2]✉

[1,2] ITMO University, Saint Petersburg, 197101, Russian Federation

[1] maneevao@gmail.com, https://orcid.org/0000-0001-9166-0706
[2] Anton.spivak@gmail.com✉, https://orcid.org/0000-0002-6981-8754

Abstract

Stochastic testing by fuzzing tools is one of the approaches to software vulnerability analysis. A testing process usually generates random input data for a tested program and takes a significant period of time. Reducing testing time

is an important task. One of the areas of research for improving testing is to define only those sets of data sequences, which have an impact on the execution path of the tested program. Thus, a new approach of input data generation that reduces total testing time allows finding more program vulnerabilities. The paper suggests a modification of a genetic algorithm, which is used by fuzzer afl (American Fuzzy Lop). The promising positions model is introduced to improve the efficiency of input data generation. With this model, the most promising position in input data is chosen by the fuzzer genetic algorithm from the viewpoint of vulnerability analysis for next mutation steps. Compared to existing solutions, the suggested model pays attention to the perspective position of a data element to increase code coverage and directs the genetic algorithm to change it. The model was evaluated with the popular fuzzer afl and its modifications (aflfast, symfuzz, afl-rb). During the evaluation study, the suggested model reached 21 % more code coverage than existing solutions. Edge coverage between base program blocks is increased from 20897.3 up to 17267.4. The developed model can be used during software testing, which implies an input and processing of user data. The model can be integrated into stochastic testing tools. The modification should be done only, in the random generator component and does not require redesigning the whole testing tool.

Keywords
testing, dynamic testing, stochastic testing, vulnerability, fuzzing

For citation: Maneev A.O., Spivak A.I. Stochastic software testing for vulnerability analysis. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2021, vol. 21, no. 6, pp. 895–902 (in Russian). doi: 10.17586/2226-1494-2021-21-6-895-902

Введение

В современном мире программное обеспечение (ПО) постоянно усложняется. Это вызвано увеличением разнообразия методов проектирования, реализации и особенностей применения программных продуктов в различных областях деятельности человека. Сокращается время, доступное для тестирования ПО, в течение которого усложняется рассмотрение внутренней логики работы каждого программного продукта. Одновременно с этим растет число обнаруживаемых дефектов на поздних стадиях разработки, которые были пропущены в ходе тестирования.

Как правило, полный перебор всех тестов для ПО недостижим. На практике в ходе жизненного цикла разработки ПО время на проведение полного тестирования можно сравнить с бесконечностью, так как оно может и не закончиться, даже когда необходимость в продукте будет полностью исчерпана. В попытках решить данные проблемы исследователями и тестировщиками был предложен метод стохастического тестирования (Fuzzing — фаззинг). Фаззинг — подвид динамического тестирования, заключающийся в передаче приложению случайных данных, которые являются входными аргументами, параметрами работы приложения либо просто наборами обрабатываемых данных [1].

Обработка тестируемой программой случайных данных не всегда приводит к значимому результату. На практике — большая часть тестов (в зависимости от приложения) не представляют интереса для исследователя, который пытается найти ошибки в программе. Тест при фаззинге считается успешным, если передача входных данных программе привела к сбою, утечке памяти, появлению исключения и др. Для того чтобы тестирование происходило в режиме «серого ящика», программа подвергается инструментации, в ходе которой фаззер добавляет в код на этапе компиляции блоки-обертки, фиксирующие исполнение определенных блоков исполняемого кода. Таким образом, фаззер измеряет показатель покрытия кода, процент кода программы, который был выполнен в процессе исполнения цикла тестирования. Для повышения качества тестирования, а именно показателя покрытия кода, стохастическое тестирование или фаззинг дополняются множеством интеллектуальных методов, которые направляют, регламентируют процесс формирования данных для передачи их программе. Тем не менее существующие методы не позволяют полностью покрыть код за короткое время тестирования. Данная работа является актуальной, так как предлагает новое решение для повышения эффективности стохастического тестирования для поиска уязвимостей.

Постановка задачи

Фаззинг появился сравнительно давно в мире информационных технологий — первое упоминание его автоматизированного использования датировано 1983 годом [2]. Тем не менее только сейчас данная технология получила новый толчок в развитии: крупные корпорации внедряют данную технологию в жизненный цикл разработки программ, ученые изучают различные аспекты данного вопроса, а исследователи и участники программ bug-bounty [3] (программы вознаграждений за поиск уязвимостей) давно взяли его на вооружение как основной инструмент. Вполне вероятно, что в будущем данный метод тестирования по применимости сможет вытеснить множество других.

Основные достоинства стохастического тестирования, которые позволяют применять его при тестировании практически любого ПО:

— простота реализации – для запуска теста необходим готовый экземпляр программы, инструментированный на этапе компиляции;
— простота подготовки программного продукта для тестирования – процесс подготовки программы может быть сделан в полностью автоматическом режиме, необходимо лишь описание способов передачи входных данных;
— отсутствие необходимости в изучении исходного кода программного продукта – исходный код программного продукта нужен только для инструментации на этапе компиляции, что выполняется фаззером автоматически;

— повышенная гарантия отсутствия ложного срабатывания – если в ходе теста происходит сбой программы, то это однозначно свидетельствует о том, что в обработке данного набора входных данных существует ошибка.

Также экспертами рассматриваются вопросы использования стохастического тестирования при проведении сертификации ПО [4]. Необходимость внедрения данного вида тестирования диктуется перечисленными достоинствами и малым числом экспертов по выявлению дефектов в программном коде в условиях огромного числа постоянно разрабатываемого ПО. Единственное препятствие для внедрения тестирования при сертификации — стохастическая природа, которая не позволяет достоверно отвечать на вопросы об отсутствии уязвимостей в объекте тестирования, а также невозможность воспроизведения всего хода тестирования.

Современные научные подходы нацелены на исследование совместной работы стохастического и других видов тестирования. Зачастую выбираются методы тестирования «белого ящика», в которых собирается дополнительная информация о тестируемом ПО. При тестировании приложения с использованием такого показателя как покрытие кода, говорят, что это тестирование по типу «серого ящика», где собирается небольшой объем информации о тестируемом приложении, чаще всего в автоматизированном режиме [5].

Основной недостаток стохастического тестирования — значительное время, необходимое для передачи всех возможных вариантов входных данных исследуемой на уязвимости программе. Существующие решения используют различного рода оптимизации для ускорения процесса тестирования, но, тем не менее, получаемые результаты далеки от совершенства и требуют улучшений.

В настоящей работе рассмотрен один из вариантов повышения эффективности фаззера путем внедрения модели перспективных позиций для направленной генерации входных данных. Предложенная модель, в отличие от существующих подходов, учитывает перспективность конкретного элемента входных данных для увеличения покрытия кода тестом и поэтому направляет генетический алгоритм на изменение именно этой позиции.

Обзор существующих решений по теме исследования

American Fuzzy Lop. American Fuzzy Lop (afl) давно и широко используется исследователям ПО для стохастического тестирования вследствие открытости, а также качества исполнения. С его помощью было найдено множество уязвимостей в различных крупных, распространенных приложениях [6]. Большая часть научных работ [7, 8] не только упоминает afl при исследовании, но и использует в качестве базового инструмента.

Afl — фаззер для тестирования по типу «серого ящика», в котором присутствуют механизмы для тестирования по типу «черного ящика», только на данный момент они находятся в стадии бета-тестирования.

Для сбора информации о покрытии afl использует следующую инструментацию программ (язык C++)[1]:

Листинг 1. Инструментация кода в afl

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem [cur_location^prev_
location]++;
prev_location = cur_location>>1 ;
```

В ходе генерации входных данных фаззер afl может применять не только случайные числа, но и выполнять последовательное изменение входных данных в соответствии с некоторой логикой. Одной из реализаций направленного формирования входных данных может являться генетический алгоритм [9–11]. В его основе лежат точечные изменения фрагментов данных — мутации, которые потом оцениваются функцией приспособленности, и, в случае использования фаззера, вычисляются покрытия кода тестом. В случае роста показателя покрытия такая мутация сохраняется и используется далее в цикле тестирования.

Основные мутации, которые способен выполнять фаззер afl над данными:
— инверсия битов;
— алгебраические вычисления;
— вставка «магических» значений (значения, которые встречаются чаще остальных).

Принцип работы фаззера afl можно представить в виде UML (Unified Modeling Language) диаграммы активности, которая демонстрирует основные этапы формирования входных данных, оценку результата теста и сохранение промежуточных результатов (рис. 1).

Основной компонент, определяющий эффективность фаззера, построенного на базе afl, — блок выбора фрагментов входных данных для мутации. На основе afl разработано несколько модификаций с различными реализациями данного блока, а также изменениями процесса тестирования: symfuzz[2], aflfast[3], afl-rb[4]. Для сравнения данные продукты и оригинальный afl фаззер версии 2.52b рассмотрены в настоящей работе.

Symfuzz проводит формирование входных данных, основываясь на одном из типов мутаций — инвертирование бита, при этом исключена детерминированная стадия поиска. Основная идея данного фаззера — поиск оптимального значения интенсивности мутации, методика поиска которого формируется при изучении приложения как «белого ящика». Данный подход к стохастическому тестированию может быть интересен за

[1] Lcamtuf Technical whitepaper for afl-fuzz [Электронный ресурс]. Режим доступа: http://lcamtuf.coredump.cx/afl/technical_details.txt (дата обращения: 21.10.2021).

[2] SymFuzz: Program-Adaptive Mutational Fuzzing [Электронный ресурс]. Режим доступа: http://security.ece.cmu.edu/symfuzz/ (дата обращения: 21.10.2021).

[3] Github repository [Электронный ресурс]. Режим доступа: https://github.com/mboehme/aflfast (дата обращения: 21.10.2021).

[4] Github repository [Электронный ресурс]. Режим доступа: https://github.com/carolemieux/afl-rb (дата обращения: 21.10.2021).

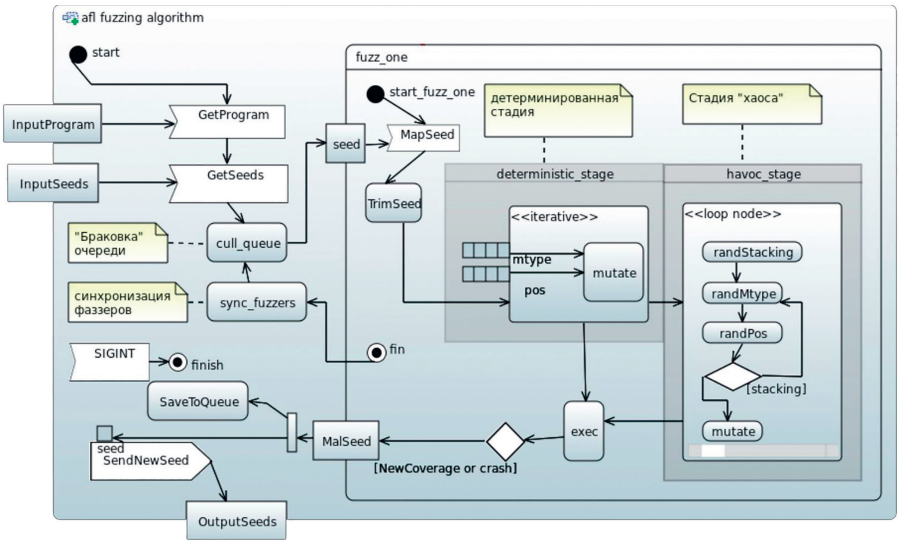

Рис. 1. Диаграмма активности фаззера afl
Fig. 1. Afl fuzzer activity diagram

счет высокой интенсивности проведения тестирования, так как операция инвертирования бита легковесна и требует низких затрат процессорного времени.

Aflfast — один из самых современных фаззеров, он основан на цепях Маркова, которую внедрили в процесс фаззинга ПО. Данная модель отражает внутреннее состояние исследуемого приложения за счет его моделирования с помощью цепей Маркова. Такое направление исследований перспективно, так как позволяет упростить ручное исследование приложения после стохастического за счет предоставления тестировщику информации о структуре исполнения программы.

Afl-rb проводит генерацию входных данных, основываясь на редкости посещения некоторых ветвей исполнения программы, а также минимальном отборе позиций для мутаций, отбрасывая некоторые из них на детерминированной стадии. В ходе детерминированной стадии происходит жесткое отсеивание позиций, которые не позволили достичь нового покрытия. Авторы настоящей работы считают, что полное исключение некоторых позиций для мутаций не является целесообразным для подобного типа тестирования, а возможно и вовсе ему вредит. Тем не менее результаты стохастического тестирования данного фаззера на некоторых приложениях достаточно высоки [12].

Достоинства afl признаны среди множества различных крупных компаний, в том числе Microsoft, Facebook, исследовательские центры которых постоянно используют этот инструмент для тестирования в своих проектах [13].

Пользователю доступны исходные тексты afl для исследования работы данного фаззера, а также для расширения его возможностей путем модификации исходного кода. Большую часть информации о тестируемом проекте и прогрессе тестирования фаззер записывает в файлы журналов, создающихся для каждого цикла тестирования, кроме того, краткая сводка о процессе тестирования выводится на экран.

Модель перспективных позиций

Основной недостаток существующих фаззеров заключается в том, что большая часть времени тестирования затрачивается не на самые оптимальные позиции для мутации. Например, если в качестве входных данных используется bmp- или wav-файлы, то более 90 % размера файла занимают байты, отвечающие за цвет отрисовки изображения на экране или за характеристики звуковой волны, которые не влияют на исполнение программы. Таким образом, фаззер в 90 % случаев мутирует байты, не приводящие к новому покрытию кода, а следовательно, выполняет работу, которой следует отдать меньший приоритет.

Для повышения эффективности процесса стохастического тестирования предложено использовать модель перспективных позиций для мутации (рис. 2). Для каждого байта из входных данных длиной n ставится в соответствие его вес в диапазоне значении от 10 до 255 (1 Б), который показывает с какой частотной характеристикой следует выбирать данную позицию для мутации. Чем выше значение, тем чаще необходимо производить мутации. Таким образом, задается распределение для генератора псевдослучайных чисел, которым он руководствуется при изменении входных данных в ходе тестирования.

Начальная инициализация весов позиций входных данных производится значением 0b01000000 = 64.

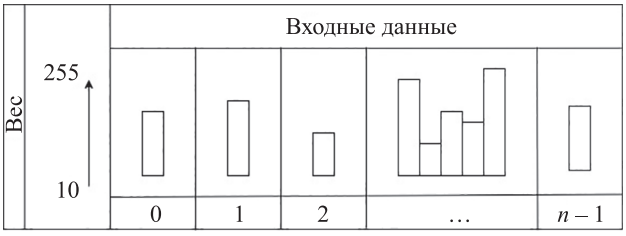

Рис. 2. Модель перспективных позиций
Fig. 2. Promising positions model

В ходе процесса тестирования веса корректируются за счет сбора информации об исполнении и достижения нового покрытия кода. Таким образом, модель не является статичной — веса меняются в процессе тестирования на основе получаемой информации о покрытии кода. При использовании модели перспективных позиций веса в модели меняются вероятностно на основе типа события, полученного в результате выполнения теста, для сохранения стохастической природы изменения входных данных.

Правила корректировки весов можно определить в соответствии с тремя видами исходов выполнения теста:

1) достигнуто новое покрытие (ошибка исполнения):
   — выполнить or 0b10000000 над весами измененных позиций;
   — запомнить позицию как сигнатурную – позиция, способная изменить ход выполнения программы, запретить повышение веса позиции;
2) достигнуто известное покрытие:
   — если позиция сигнатурная, то отметить, что она может принимать больше одного значения, разрешить повышение веса позиции;
   — выполнить случайное действие над весом позиции:
     — 25 % — увеличить вес на 1;
     — 5 % — уменьшить вес на 5;
     — 70 % — ничего не делать;
3) покрытие осталось прежним:
   — выполнить случайное действие над позициями:
     — 10 % — and 0b00111111;
     — 40 % — уменьшить вес на 1;
     — 50 % — ничего не делать.

Таким образом, модель перспективных позиций, представляющая собой веса позиций входных данных, используется генератором псевдослучайных чисел для направленной генерации новых тестирующих последовательностей данных. Схема использования модели перспективных позиций фаззером представлена на рис. 3.

Как видно из рис. 3 генератор псевдослучайных чисел генерирует новые последовательности на основе предыдущих путем точечного изменения некоторых позиций. За это отвечает генетический алгоритм, встроенный в фаззер. В стандартной реализации выбор позиций для мутации выполняется случайно. Модель перспективных позиций позволяет корректировать поведение алгоритма путем вероятностного повышения частоты изменения определенных позиций, используя информацию о весах позиций. Далее, после прохождения теста с новыми входными данными и получения информации о новом покрытии кода, благодаря обратной связи выполняется корректировка весов в соответствии с правилами, указанными выше. На следующем шаге происходит новая генерация входных данных уже с учетом обновленных весов в модели перспективных позиций.

Разработанная модель позволяет процессу тестирования уделять больше внимания тем позициям во входных данных, которые отвечают за наиболее важные функциональные элементы тестируемого ПО. Тем самым повышается эффективность работы фаззера, которая в общем случае характеризуется временем выполнения тестирования и достигнутым покрытием кода тестируемого ПО.

Экспериментальное исследование

Для экспериментального исследования выбран фаззер с открытыми исходными кодами — afl, модификация которого осуществлялась на языке программирования C++. В схеме работы afl (рис. 1) произведено изменение функции генерации псевдослучайного числа без модификаций основной функциональности фаззера. В работе выполнено исследование по сравнению предлагаемого авторами фаззера с реализованной моделью перспективных позиций (myfuzzer) на следующих проектах: bmp2png, bzip2, gif2png, lz4, objdump, readelf. Пример одного из сравнений показан на рис. 4. В качестве аппаратного обеспечения для проведения экспериментального исследования использован персональный компьютер со следующими характеристиками: процессор Intel Core2 Quad Q8200 с частотой 2,33 ГГц,

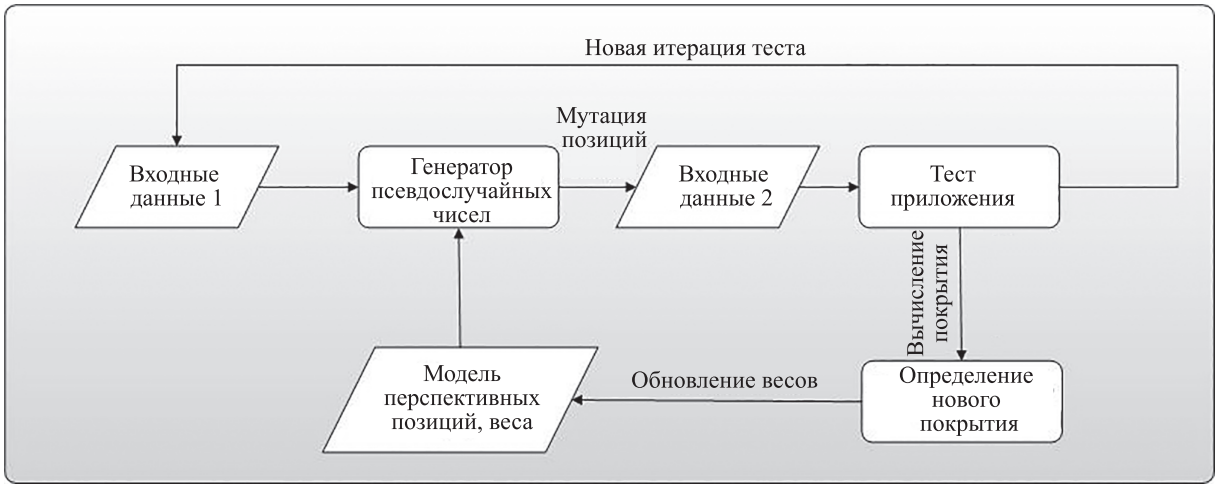

Рис. 3. Схема использования модели перспективных позиций

Fig. 3. Schema of using the promising positions model

4 ядрами, размер оперативной памяти — 2 ГБ, операционная система — Ubuntu 16.04.

Входные данные в ходе исследования представлены файлами соответствующего формата. Например, для bmp2png входные данные представляли собой изображение в формате bmp, которое затем модифицировалось фаззером.

bzip2 — программа архивации, в которой входными данными могут являться файлы любых форматов, поэтому в ходе исследования использовались бинарные файлы со случайным содержимым.

Веса модели перспективных позиций инициализировались начальными значениями 64, а затем на основе правил и получаемых покрытий модифицировались.

В ходе исследования выполнялся запуск приложения в среде каждого из сравниваемых фаззеров — afl, aflfast, symfuzz, afl-rb, а также разработанного myfuzzer. Каждое из приложений подвергалось стохастическому тестированию в течение 3600 с итеративно в 50 запусков. Результат усреднялся для того, чтобы исключить влияние выбросов, которые в случае стохастического тестирования неизбежны. Главным показателем в ходе исследования было принято значение, характеризующее покрытие кода тестируемой программы, показывающее число уникальных путей исполнения программы, которое было найдено фаззером. Соответственно, чем выше данный показатель, тем эффективнее работает фаззер — более полно проверяет код программы на наличие уязвимости (рис. 4, *a*).

Для универсальности сравнений фаззеров между собой, без зависимости от времени исполнения и вида тестируемого продукта, выполним нормировку показателей покрытия кода в каждый момент времени (рис. 4, *b*):

$$cov_f^*(t) = \frac{cov_f(t)}{\max(\{cov_{fuz}(t), fuz \in fuzzers\})},$$

где *fuzzers* — множество всех рассматриваемых фаззеров.

Дальнейшие вычисления выполнены на основе интегрирования значений покрытия кода по времени.

$$D_{f1-f2} = \frac{\int_0^{3600} cov_{f_1}^*(t)dt}{\int_0^{3600} cov_{f_2}^*(t)dt},$$

где D_{f1-f2} — показатель сравнения фаззеров f_1 и f_2.

Используя выражение $\int_0^{3600} cov_f^*(t)dt$, вычислим значения покрытия для каждого сравниваемого фаззера в сумме для всех приложений. Результаты представлены в таблице.

Результаты показали, что фаззер myfuzzer с моделью перспективных позиций для мутации имеет самое высокое значение среднего покрытия кода среди сравниваемых средств стохастического тестирования.

Произведем сравнение результатов myfuzzer с каждым фаззером, а также со средним значением работы всех фаззеров, для получения комплексного показателя улучшения покрытия кода. Данный способ сравнения в меньшей степени зависит от количества найденных путей и выполняет сравнение в каждый момент времени тестирования.

Вычислим среднее значение прироста покрытия кода фаззера myfuzzer относительно сравниваемых фаззеров.

$$\frac{(|fuzzers| - 1)\int_0^{3600} cov_{myfuzzer}^*(t)dt}{\sum_{i \in fuzzers \setminus myfuzzer} \int_0^{3600} cov_i^*(t)dt} = 1{,}21.$$

Средний процент повышения достижения покрытия кода равен 21 %.

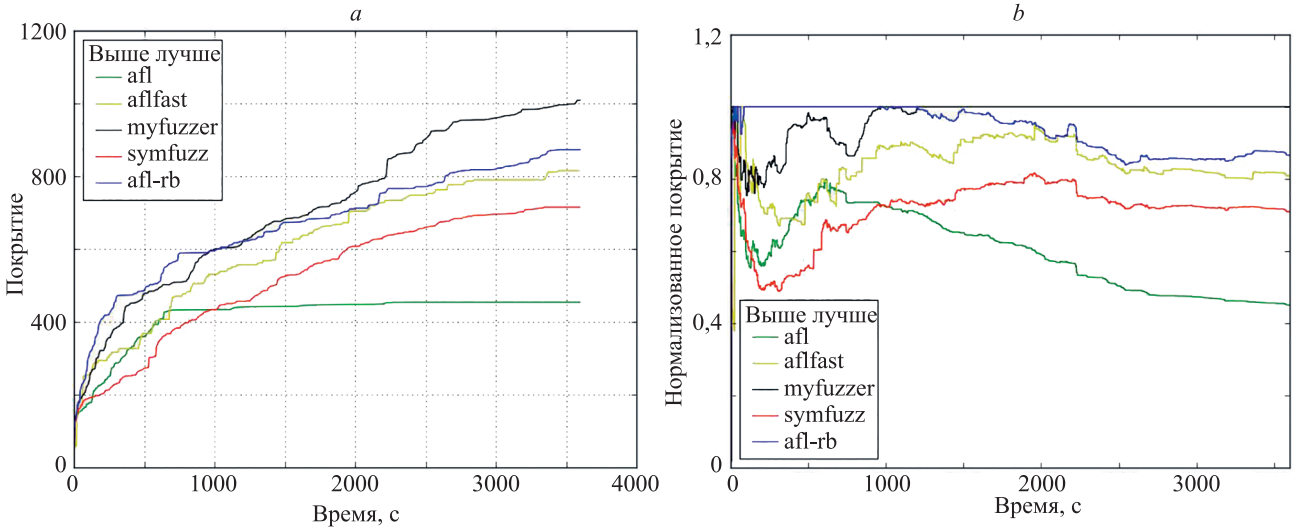

Рис. 4. Зависимость покрытия кода от времени (*a*) и нормированное значение покрытия в каждый момент времени (*b*) при фаззинге приложения gif2png

Fig. 4. Time dependence of code coverage (*a*) and normalized coverage in each period of time (*b*) during gif2png application fuzzing

Таблица. Результаты экспериментального исследования фаззеров
Table. Fuzzers evaluation study results

Наименование фаззера	Прирост myfuzzer относительно сравниваемого, %	Покрытие кода, среднее количество ветвей
myfuzzer	0	20 897,3
afl-rb	9,7	19 048,9
afl	24,7	16 764,5
symfuzz	24,9	16 736,5
aflfast	26,5	16 519,7

Заключение

В ходе выполнения исследования разработана модель перспективных позиций для средств стохастического тестирования для поиска уязвимостей. Модель позволяет повысить эффективность работы генератора псевдослучайных чисел для направленного выбора позиции входных данных в целях выполнения мутаций в ходе подготовки данных для тестирования программного обеспечения. Научная значимость результата заключается в интеграции разработанной модели перспективных позиций в работу генетического алгоритма фаззера, который демонстрирует прирост показателя покрытия кода по сравнению с существующими подходами. Выполнено экспериментальное исследование пяти фаззеров на наборе приложений с целью сравнения их показателей покрытия кода. Благодаря предложенной модели покрытие кода при тестировании с помощью фаззера myfuzzer в среднем увеличилось на 21 %. Коэффициент эффективности вычислялся для различных типов тестируемых приложений.

Литература

1. Саттон М., Грин А., Амини П. Fuzzing: исследование уязвимостей методом грубой силы. СПб.: Символ-Полюс, 2009. 555 с.
2. Hertzfeld A. Monkey Lives [Электронный ресурс]. URL: http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt (дата обращения: 21.10.2021).
3. Bug bounty // Manalyzer [Электронный ресурс]. URL: https://manalyzer.org/bounty (дата обращения: 21.10.2021).
4. Мельникова В.В., Котов С.Л., Палюх Б.В., Проскуряков М.А. Тестирование программ с использованием генетических алгоритмов // Программные продукты и системы. 2011. № 4. С. 107–110.
5. Godefroid P. Random testing for security: blackbox vs. whitebox fuzzing // RT '07: Proc. of the 2nd International Workshop on Random Testing: Co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007). 2007. P. 1. https://doi.org/10.1145/1292414.1292416
6. American Fuzzy Loop (2.52b) [Электронный ресурс]. URL: http://lcamtuf.coredump.cx/afl/ (дата обращения: 21.10.2021).
7. Yue T., Tang Y., Yu B., Wang P., Wang E. LearnAFL: Greybox fuzzing with knowledge enhancement // IEEE Access. 2019. V. 7. P. 117029–117043. https://doi.org/10.1109/ACCESS.2019.2936235
8. Kersten R., Luckow K., Păsăreanu C.S. Poster: AFL-based Fuzzing for Java with Kelinci // Proc. of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS 2017). 2017. P. 2511–2513. https://doi.org/10.1145/3133956.3138820
9. Smetsers R., Moerman J., Janssen M., Verwer S. Complementing Model Learning with Mutation-Based Fuzzing // arXiv: 1611.02429. 2016 [Электронный ресурс]. URL: http://arxiv.org/abs/1611.02429 (дата обращения: 21.10.2021).
10. Cha S.K., Woo M., Brumley D. Program-adaptive mutational fuzzing // Proc. of the 36th IEEE Symposium on Security and Privacy (SP 2015). 2015. P. 725–741. https://doi.org/10.1109/SP.2015.50
11. Householder A.D. Well There's Your Problem: Isolating the Crash Inducing Bits in a Fuzzed File: technical note CMU/SEI-2012-TN-012 / Software Engineering Institute. Carnegie Mellon University, 2012. 19 p.
12. Rajpal M., Blum W., Singh R. Not all bytes are equal: Neural byte sieve for fuzzing // arXiv: 1711.04596. 2017 [Электронный ресурс]. URL: http://arxiv.org/abs/1711.04596 (дата обращения: 21.10.2021).
13. Böhme M., Pham V.-T., Roychoudhury A. Coverage-based Greybox Fuzzing As Markov Chain // Proc. of the 23rd ACM Conference on Computer and Communications Security (CCS 2016). 2016. P. 1032–1043. https://doi.org/10.1145/2976749.2978428

References

1. Sutton M., Greene A., Armini P. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007, 576 p.
2. Hertzfeld A. *Monkey Lives*. Available at: http://www.folklore.org/StoryView.py?story=Monkey_Lives.txt (accessed: 21.10.2021).
3. Bug bounty. *Manalyzer*. Available at: https://manalyzer.org/bounty (accessed: 21.10.2021).
4. Melnikova V.V., Kotov S.L., Palyukh B.V., Proskuryakov M.A. Testing program using genetic algorithms. *Software & Systems*, 2011, no. 4, pp. 107–110. (in Russian)
5. Godefroid P. Random testing for security: blackbox vs. whitebox fuzzing. *RT '07: Proc. of the 2nd International Workshop on Random Testing: Co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, 2007, pp. 1. https://doi.org/10.1145/1292414.1292416
6. *American Fuzzy Loop (2.52b)*. Available at: http://lcamtuf.coredump.cx/afl/ (accessed: 21.10.2021).
7. Yue T., Tang Y., Yu B., Wang P., Wang E. LearnAFL: Greybox fuzzing with knowledge enhancement. *IEEE Access*, 2019, vol. 7, pp. 117029–117043. https://doi.org/10.1109/ACCESS.2019.2936235
8. Kersten R., Luckow K., Păsăreanu C.S. Poster: AFL-based Fuzzing for Java with Kelinci. *Proc. of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS 2017)*, 2017, pp. 2511–2513. https://doi.org/10.1145/3133956.3138820
9. Smetsers R., Moerman J., Janssen M., Verwer S. Complementing Model Learning with Mutation-Based Fuzzing. *arXiv: 1611.02429*. 2016. Available at: http://arxiv.org/abs/1611.02429 (accessed: 21.10.2021).
10. Cha S.K., Woo M., Brumley D. Program-adaptive mutational fuzzing. *Proc. of the 36th IEEE Symposium on Security and Privacy (SP 2015)*, 2015, pp. 725–741. https://doi.org/10.1109/SP.2015.50
11. Householder A.D. *Well There's Your Problem: Isolating the Crash Inducing Bits in a Fuzzed File.* Technical Note CMU/SEI-2012-TN-012. Software Engineering Institute. Carnegie Mellon University, 2012, 19 p.
12. Rajpal M., Blum W., Singh R. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv: 1711.045962017*. 2017. Available at: http://arxiv.org/abs/1711.04596 (accessed: 21.10.2021).
13. Böhme M., Pham V.-T., Roychoudhury A. Coverage-based Greybox Fuzzing As Markov Chain. *Proc. of the 23rd ACM Conference on Computer and Communications Security (CCS 2016)*, 2016, pp. 1032–1043. https://doi.org/10.1145/2976749.2978428

Авторы

Манеев Андрей Олегович — студент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, https://orcid.org/0000-0001-9166-0706, maneevao@gmail.com
Спивак Антон Игоревич — кандидат технических наук, доцент, Университет ИТМО, Санкт-Петербург, 197101, Российская Федерация, SC 56779715800, https://orcid.org/0000-0002-6981-8754, Anton.spivak@gmail.com

Authors

Andrey O. Maneev — Student, ITMO University, Saint Petersburg, 197101, Russian Federation, https://orcid.org/0000-0001-9166-0706, maneevao@gmail.com
Anton I. Spivak — PhD, Associate Professor, ITMO University, Saint Petersburg, 197101, Russian Federation, SC 56779715800, https://orcid.org/0000-0002-6981-8754, Anton.spivak@gmail.com

Статья поступила в редакцию 03.08.2021
Одобрена после рецензирования 25.09.2021
Принята к печати 30.10.2021

Received 03.08.2021
Approved after reviewing 25.09.2021
Accepted 30.10.2021