УДК 004.056
DOI: 10.53403/9785951505460_559

СОВРЕМЕННОЕ ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ. ТЕХНОЛОГИЯ АВТОМАТИЗИРОВАННОГО ТЕСТИРОВАНИЯ – ФАЗЗИНГ

Станкевичус Антон Андреевич (staff@vniief.ru), Храпунов Павел Александрович, Трищенков Андрей Владимирович

ФГУП «РФЯЦ-ВНИИЭФ», г. Саров Нижегородской обл.

В данной работе приведено описание фаззинга как технологии автоматизированного тестирования. Проведено сравнение фаззинга с другими видами тестирования. Описываются особенности и этапы подготовки к проведению фаззинга, приводятся классификации фаззеров. Представлен пример работы и использования, одного из популярных фаззеров AFL++ на реальном примере.

Ключевые слова: информационная безопасность, уязвимости, фаззинг, методы фаззинга, тестирование.

MODERN SOFTWARE TESTING. TECHNOLOGY OF AUTOMATED TESTING – FUZZING

Stankevichus Anton Andreevich (staff@vniief.ru), Khrapunov Pavel Alexandrovich, Trishchenkov Andrey Vladimirovich

FSUE «RFNC-VNIIEF», Sarov, Nizhny Novgorod region

This paper describes fuzzing as an automated testing technology. Fuzzing is compared with other types of testing. Features and stages of preparation for fuzzing are described, classifications of fuzzers are given. An example of the operation and use of one of the popular AFL ++ fuzzers is presented on a real example.

Keywords: information security, vulnerabilities, fuzzing, fuzzing methods, testing.

Введение

По мере усложнения и активного развития информационных систем, на первые места начинает выходить проблема информационной безопасности. Данный факт предъявляет особые требования к разработке и сопровождению программных продуктов.

Для этих целей используется тестирование программ, при котором программы анализируются на предмет способности противостоять нежелательному внешнему вмешательству в их работу. Для тестирования применяются как статический анализ на этапе создания программного кода, так и динамический анализ в процессе выполнения программы.

Одной из составляющих динамического анализа является фаззинг. Фаззинг – это технология автоматизированного тестирования программ с целью выявления потенциальных уязвимостей, путем подачи в исследуемую программу набора некоренных и неожиданных данных. При таком тестировании практически полностью отсутствуют ложные срабатывания, характерные для статических анализаторов. При этом обеспечивается большое количество граничных значений.

Входными данными выступают обрабатываемые исследуемым приложением файлы и другая информация, в том числе определяемая протоколами обмена, прикладными интерфейсными функциями и т.п.

На сегодняшний день уже разработан довольно обширный инструментарий для фаззинга и есть возможность подобрать фаззер именно под свою задачу.

В данной работе приведено описание фаззинга как технологии автоматизированного тестирования. Проведено сравнение фаззинга с другими видами тестирования. Описываются особенности и этапы подготовки к проведению фаззинга, приводятся классификации фаззеров. Представлены примеры работы и использования, наиболее популярных фаззеров таких как AFL++ и Crusher на реальных примерах.

Теоретическая часть

Определение

Фаззинг – это техника автоматизированного тестирования программного обеспечения, при которой на вход программе подаются неожиданные, недопустимые и случайные наборы данных с целью найти непредусмотренные программистом входные данные, которые приводят к аварийному завершению программы или к ее некорректному поведению. Входные данные могут передаваться через файлы, сетевые сокеты, API, стандартный поток ввода, переменные окружения и т. д. Фаззинг также иногда называют тестированием на устойчивость или негативным тестированием.

История

«Термин «Фаззинг» появился в 1988 году в Университете Висконсина. На семинаре Бартона Миллера была создана программа для тестирования надежности приложений под Unix, эта программа генерировала случайные данные, которые передавались как параметры для тестируемых программ до тех пор, пока они не завершали выполнение с ошибкой.

Однако, можно утверждать, что процесс фаззинга гораздо старше. Так, американский ученый Джеральд Вайнберг рассказывает о том, как еще в 50-х годах он в качестве стандартной практики при тестировании использовал либо взятые из мусорного ведра перфокарты, либо перфокарты случайных чисел, что помогало обнаруживать нежелательное поведение тестируемой программы.

В 1991 создается первая программа для фаззинга – crashme, которая проверяла различные системные вызовы в UNIX-системах

В апреле 2012 Google анонсировал ClusterFuzz для фаззинга компонентов браузера Google Chrome, где каждый мог загрузить свой фаззер и попытаться найти ошибку.

В 2014–2015 появляются фаззеры AFL, libFuzzer, go-fuzz и с их помощью было выявлено множество ошибок безопасности.

В 2016 году Microsoft выпускает Project Springfield, предназначенный для поиска критических уязвимостей. А Google выпускает OSS-Fuzz, который позволяет производить непрерывный фаззинг. В 2018 году появляется техника поиска уязвимостей с помощью фаззинга на процессорах с RISC архитектурой.

В сентябре 2020 года Microsoft выпустила OneFuzz, автономную платформу fuzzing-as-a-service, которая автоматизирует обнаружение программных ошибок, с поддержкой Windows и Linux.»

Классификация фаззеров

«В зависимости от метода генерации данных принято разделять подход к фаззингу на несколько типов:

1. На основе грамматики

Таким фаззерам для работы требуется определенный набор правил — грамматика для построения входных данных. Как только фаззеру будут известны эти правила, он сможет генерировать новые комбинации на их основе. При этом можно позволить себе иногда отклоняться от грамматики и не всегда следовать ей, подавая на вход тестируемой программы некорректные данные. Подобные фаззеры генерируют хорошие, достоверные последовательности, доля случайности в них относительно невелика.

2. На основе мутаций

Такой тип фаззеров на каждом этапе случайным образом изменяет входные данные из предыдущих попыток. При этом для старта ему требуется набор репрезентативных входов (корпус), который будет использоваться для дальнейших мутаций. В качестве корпуса можно взять данные пользователей или существующие тесты. Во время своей работы такой фаззер слегка изменяет готовые последовательности (переставляя биты и байты), комбинирует и сочетает их вариации и подает в программу.

3. Построение грамматики по данным

Также есть способ сочетать оба описанных выше подхода. Специалисты по обработке данных и машинному обучению могут попытаться построить грамматику на основе уже имеющейся информации. Однако оценить результат на выходе таких фаззеров зачастую очень сложно.

4. На основе покрытия кода

Данные фаззеры устроены по принципу генетического алгоритма и стремятся максимизировать покрытие тестового кода. С практической точки зрения это один из самых эффективных на сегодня типов фаззеров.» [1].

По типу воздействия фаззеры можно разделить на два основных класса: локальные и удаленные [9]:

Локальные фаззеры делятся на следующие типы:

- фаззеры командной строки. Используются для выявления ошибок, связанных с разбором входных параметров программ;
- фаззеры переменных окружения. Используются для выявления ошибок, связанных с обработкой данных, получаемых через переменные окружения;
- фаззеры файлов. Используются для тестирования программного обеспечения, принимающего файлы в качестве входных данных;

Удаленные фаззеры бывают следующих типов:

- фаззеры сетевых протоколов. В зависимости от сложности протокола применяются фаззеры соответствующей сложности;
- фаззеры web-приложений. Получили особую актуальность с развитием Web 2.0;
- фаззеры web-браузеров. Тестируется правильность разбора, как HTML-тэгов, так и других поддерживаемых расширений. Особо стоит выделить фаззеры com-объектов поддерживаемых браузерами.

К сожалению, до сих пор в индустрии нет четких определений и классификаций фаззеров. На рис. 1 можно увидеть попытку классифицировать существующие виды популярных фаззеров [2].

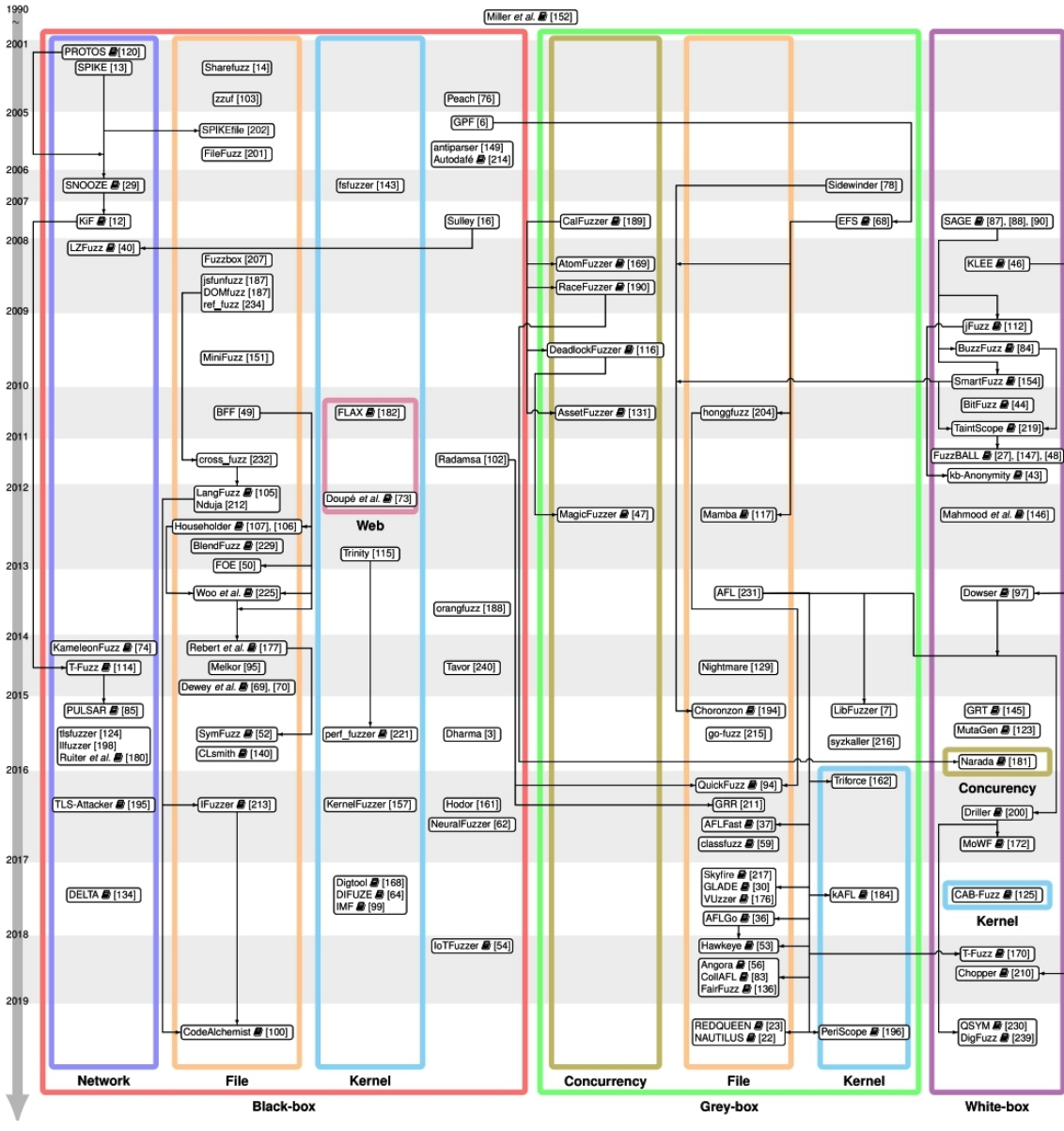

Рис. 1. Классификация фаззеров

Подходы к фаззингу

Выделяется три подхода к выявлению недостатков системы: тестирование методом черного, серого и белого ящиков. Различие между ними определяется теми ресурсами, которые доступны во время тестирования [3].

Фаззинг «черный ящик»

Фаззер во время своей работы не получает никакой информации от тестируемой программы, он может взаимодействовать с программой только через ввод/вывод программы. Большинство традиционных фаззеров относят к этой категории.

Фаззинг «белый ящик»

Здесь программа анализирует внутреннее устройство тестируемой программы. Обычно для того чтобы фаззер мог фиксировать пройденный путь, необходимо предварительно инструментировать исследуемую программу. Инструментация может быть статической (во время компиляции) и динамической (во время работы программы). При данном виде фаззинга необходимо иметь в наличии исходный код программы.

Фаззинг «серый ящик»

Фаззер серого ящика может получать некоторую информацию о внутренней структуре тестируемой программы и/или ее исполнении. Фаззеры серого ящика полагаются на приблизительную информацию, чтобы увеличить скорость и иметь возможность тестировать программу на большем количестве входных данных.

Почему так важно фаззинг-тестирование?

Цель фаззинга основана на предположении, что в каждой программе есть ошибки, которые ждут сво-

его часа. Значит, систематический подход рано или поздно должен их обнаружить. Фаззинг может добавить другую перспективу к классическим методам тестирования программного обеспечения (ручной просмотр кода, отладка), поскольку это нечеловеческий подход. Он не заменяет их, но является разумным дополнением, благодаря ограниченной работе, необходимой для внедрения процедуры. На рис. 2 представлена схема описывающая принцип работы фаззера, а в таблице сравнение фаззинга и статического тестирования [4].

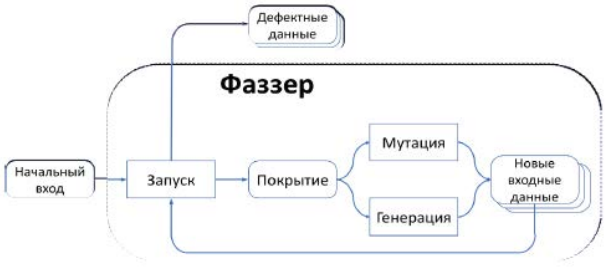

Рис. 2. Принципиальная схема фаззинга

Сравнение фаззинга со статическим типом тестирования

	Статический анализ	Фаззинг
Покрытие кода	~100 %	Зависит от тестовых данных и типа тестируемого ПО
Ложные срабатывания	Много	Практически нет
Пропуск ошибок	Зависит от базы данных анализатора	Зависит от тестовых данных и алгоритмов фаззера
Типы ошибок	Широкий спектр	Определенный спектр ошибок
Ручной анализ	Много	Минимально
Сложность использования	Небольшая (статический анализатор сделает все сам, нужно только указать исходный код программы)	Большая (необходимо определить точки входа в программу, подобрать тестовые данные, настроить при необходимости саму программу, а также сам фаззер)

Преимущества фаззинг-тестирования

Фаззинг-тестирование улучшает тестирование безопасности программного обеспечения. Ошибки, обнаруженные в фаззинге, иногда бывают серьезными и чаще всего используются хакерами, включая сбои, утечку памяти, необработанные исключения и т. д. Если какая-либо из ошибок не может быть замечена тестировщиками из-за ограниченности времени и ресурсов, эти ошибки также обнаруживаются в фаззинг-тестировании.

Недостатки фаззинг-тестирования

Само по себе фаззинг-тестирование не может дать полной картины общей угрозы или ошибок безопасности. Фаззинг-тестирование менее эффективно для борьбы с угрозами безопасности, не вызывающими сбоев программы, такими как некоторые вирусы, черви, трояны и т. д. Фаззинг-тестирование может обнаруживать только простые ошибки или угрозы. Для эффективного выполнения потребуется значительное время.

Установка граничного условия со случайными входными данными очень проблематична, но теперь с помощью детерминированных алгоритмов, основанных на пользовательских входных данных, большинство тестеров решают эту проблему.

Фаззинг не может гарантировать полное обнаружение ошибок в приложении.

Практическая часть

Этапы подготовки к проведению фаззинга

Подготовка делится на несколько этапов:

1. Подготовка входных данных;
2. Определение и анализ поверхности атаки;
3. Изменение исходного кода для реализации возможности принимать данные от фаззера (если необходимо и если есть исходный код);
4. Инструментация исходного кода проекта;
5. Настройка фаззера и его запуск;
6. Сбор покрытия;
7. Анализ полученных результатов.

Тестистирование Xpdf с помощью «AFL++»

Приведу пример работы и использования файлового свободно распространяемого фаззера – AFL++. В качестве операционной системы рекомендуется использовать Linux Ubuntu или другую версию ОС Linux.

AFL++ (American fuzzy lop) – это ориентированный на поиск ошибок инструмент анализа ПО, который использует обширный список типов инструментации кода для получения информации о покрытии и использует множество генетических алгоритмов мутации для автоматического обнаружения различных тестовых примеров, которые вызывают новые внутренние состояния в бинарном коде ПО. Фаззер AFL поддерживает инструментацию, реализующуюся на этапе компиляции из исходного кода с помощью оберток afl-gcc/afl-g++. afl-gcc/afl-g++ подменяет вызываемую команду на обертку, переписывающую ассемблерный код, сгенерированный компилятором [5].

Определение и анализ поверхности атаки

На вход Xpdf, через аргумент командной строки, можно подавать pdf файл, который на выходе будет конвертирован в текстовый файл «.txt». Цель состоит в том, чтобы найти сбой **CVE-2019-13288** в XPDF

3.02. **CVE-2019-13288** — это уязвимость, которая может вызвать бесконечную рекурсию через созданный файл. Данный сбой может привести к исчерпанию памяти стека и сбою программы. В результате удаленный злоумышленник может использовать это для DoS-атаки. Тестирование проводилось на ОС Ubuntu 20.04.4 LTS [6].

Установка соответствующих обновлений и ПО

Сначала получим нашу фаззинговую цель. Создаем новый каталог для проекта, в котором и будем проводить тестирование. Выполняем в терминале:

```
cd $HOME
mkdir fuzzing_xpdf && cd fuzzing_xpdf/
sudo apt install build-essential
wget https://dl.xpdfreader.com/old/xpdf-3.02.tar.gz
tar -xvzf xpdf-3.02.tar.gz
```

Далее создадим две папки: xpdf-3.02-fuzz(для фаззинга) и xpdf-3.02-coverage(для сбора покрытия).

```
mkdir xpdf-3.02-fuzz && mkdir xpdf-3.02-coverage
cp –r xpdf-3.02 xpdf-3.02-fuzz
cp –r xpdf-3.02 xpdf-3.02-coverage
```

Также необходимо скачать несколько примеров в формате PDF:

```
cd $HOME/fuzzing_xpdf
mkdir pdf_examples && cd pdf_examples
wget https://github.com/mozilla/pdf.js-sample-files/raw/master/helloworld.pdf
wget http://www.africau.edu/images/default/sample.pdf
wget https://www.melbpc.org.au/wp-content/uploads/2017/10/small-example-pdf-file.pdf
```

Инструментация исходного кода

Для начала необходимо установить AFL++ [7]:

```
sudo apt-get update
sudo apt-get install -y build-essential python3-dev automake git flex bison libglib2.0-dev libpixman-1-dev python3-setuptools
sudo apt-get install -y lld-11 llvm-11 llvm-11-dev clang-11 || sudo apt-get install -y lld llvm llvm-dev clang
sudo apt-get install -y gcc-$(gcc --version|head -n1|sed 's/.* //'|sed 's/\..*//')-plugin-dev libstdc++-$(gcc --version|head -n1|sed 's/.* //'|sed 's/\..*//')-dev
cd $HOME
git clone https://github.com/AFLplusplus/AFLplusplus && cd AFLplusplus
export LLVM_CONFIG="llvm-config-11"
make distrib
sudo make install
```

Для сборки AFL++ версии необходимо перейти в корень папки «xpdf-3.02-fuzz» с исходными текстами и выполнить в терминале следующие команды:

```
sudo apt update && sudo apt install -y build-essential gcc
export LLVM_CONFIG="llvm-config-11"
CC=$HOME/AFLplusplus/afl-clang-fast
CXX=$HOME/AFLplusplus/afl-clang-fast++
./configure --prefix="$HOME/fuzzing_xpdf/install/"
make && make install
```

Для сборки версии для получения покрытия, необходимо перейти в корень папки «xpdf-3.02-coverage» с исходными текстами и выполнить в терминале следующие команды:

```
CXXFLAGS='-fprofile-arcs -ftest-coverage' CFLAGS='-fprofile-arcs -ftest-coverage' LDFLAGS='-lgcov --coverage' ./configure --prefix="$HOME/fuzzing_xpdf/xpdf-3.02-coverage/install/"
make
make install
```

Настройка и запуск фаззера AFL++

Произведем настройку фаззера AFL++. Для этого необходимо выполнить следующие действия:

1. Открыть терминал и выполнить следующие команды:

```
sudo su
echo core >/proc/sys/kernel/core_pattern
exit
```

2. Переходим в папку «fuzzing_xpdf» и создадим папки, в которых будем хранить входные и выходные данные.

В качестве входных файлов буде использовать папку «pdf_examples» с примерами.

Для выходных файлов создаем папку «out».

3. Выполняем команду в окне терминала на запуск фаззинга:

```
afl-fuzz -i $HOME/fuzzing_xpdf/pdf_examples/ -o $HOME/fuzzing_xpdf/out/ -s 123 -- $HOME/fuzzing_xpdf/install/bin/pdftotext @@ $HOME/fuzzing_xpdf/output
```

Описание параметров и опций команды:

- ***-i*** указывает каталог, в который мы должны поместить входные данные
- ***-o*** указывает каталог, в котором AFL++ будет хранить измененные файлы.
- ***-s*** указывает статическое случайное начальное число для использования
- *--* – после данного параметра указываем путь, до файла, который будет фаззить.
- ***@@*** – это командная строка целизаполнителя, которую AFL будет заменять каждым именем входного файла.

Далее ждем пока фаззер обнаружит хотя бы одну ошибку (см. рис. 3). Уникальные ошибки указаны красным цветом в поле: ***saved craches.*** После этого можно отключать фаззинг и приступать к сбору покрытия и анализу результатов.

4. Чтобы завершить работу фаззера нажмите сочетание клавиш: ***Ctrl+C***.

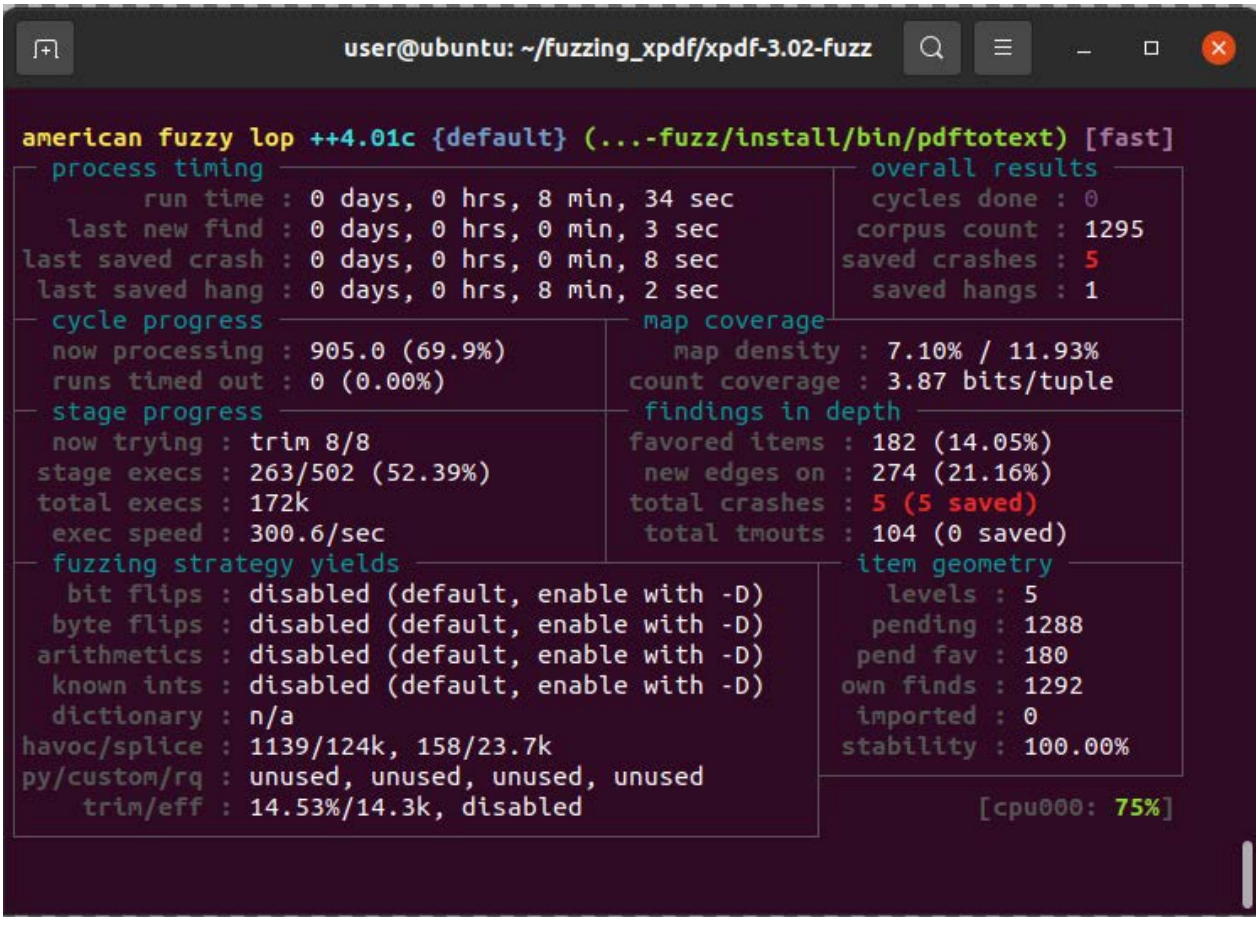

Рис. 3. Пользовательский интерфейс AFL++

Сбор покрытия

Для генерации файлов покрытия необходимо запустить программу на сгенерированных входных данных, например от предыдущего запуска AFL++ (при этом будут генерироваться файлы вида – *.gcda). Для этого напишем специальный bash скрипт «run_for_coverage.sh», и поместим его в папку «scripts» В скрипте вторым аргументом указываем директорию с входными данными [8]:

```
bash
/home/user/fuzzing_xpdf/scripts/run_for_coverage.sh
/home/user/fuzzing_xpdf/out/default/queue/
```

Далее необходимо установить программу «lcov», которая позволит собрать покрытие кода и представить его в удобочитаемом формате.

Для сбора информации о покрытии запускается команда:

```
lcov -d . -c --gcov-tool /usr/bin/gcov-9  -o test.info
```

Далее запускаем команду на формирования html отчета:

```
genhtml -o html/ test.info
```

Информации о покрытии будет находится в директории «html». В удобном формате можно открыть файл «html/index.html» с помощью браузера. Стоит отметить, что собирать информацию о покрытии стоит в директории с исходными текстами. Результат сбора покрытия показан на рис. 4 и 5.

Из результатов видно, что AFL++ успешно увеличивает покрытие по выполненным строкам кода на 1715 и по функциям на 154.

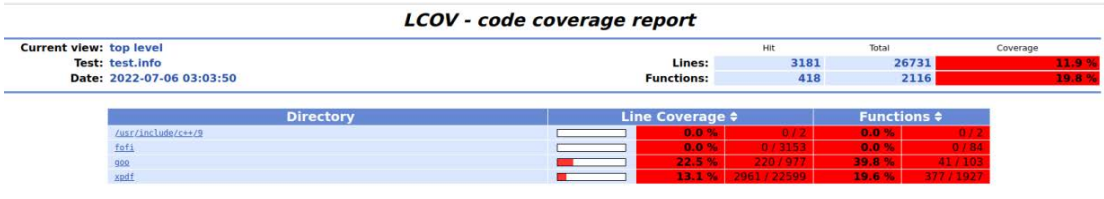

Рис. 4. Результаты покрытия при запуске на начальных входных данных

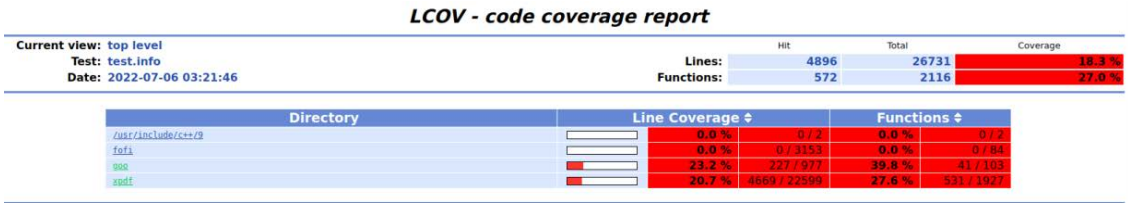

Рис. 5. Результаты покрытия при запуске на сгенерированных входных данных, полученных после анализа инструментом AFL++

Анализ полученных результатов

После окончания тестирования можно приступать к анализу результатов. Для этого необходимо перейти в папку с выходными данными. В каталоге «$HOME/fuzzing_xpdf/out/crashes» хранятся все файлы которые вызывают сбой в программе. Для подтверждения найденного сбоя необходимо запустить «Xpdf» и передать на вход найденный файл. Если данный файл приведет к аварийному завершению работы программы, значит сбой (уязвимость) подтвержден.

Заключение

Приведено описание фаззинга как технологии автоматизированного тестирования. Сделано сравнение фаззинга как одного из видов динамического тестирования со статическим тестированием программного кода. Были подробно описаны особенности и этапы подготовки к проведению фаззинга, а также была приведена широкая классификация различных видов фаззеров. Представлен пример работы и использования фаззера AFL++ на примере тестирования программы «Xpdf».

На вход программе были поданы подготовленные pdf файлы с целью их дальнейшей конвертации в текстовый формат. AFL++ в процессе работы в течении 8 минут успел выявить 5 уникальных сбоев в программе «Xpdf». Также был произведен сбор покрытия программного кода, до запуска фаззера и после. Полученные результаты показали, что AFL++ успешно увеличивает покрытие по выполненным строкам кода на 1715 (с 11,9 до 18,3 %) и по функциям на 154 (с 19,8 до 27 %).

Фаззинг – только один из видов тестирования, и он не обеспечит 100 % результата в тестировании безопасности. Самый эффективный подход это системный, необходимо использовать фаззинг совместно с другими видами тестирования. Ведь тестирование безопасности в общем, и фаззинг в частности, необходимы для выпуска стабильного программного продукта, а также для экономии денег и времени на этапе поддержки.

Список литературы

1. Саттон М., Грин А., Амини П. Fuzzing: исследование уязвимостей методом грубой силы. СПб.: Символ-Плюс, 2009.

2. Fuzzing: State of the Art / H. Liang, X. Pei, X. Jia et al. // IEEE Transactions on Reliability. 2018. Vol. 67, N. 3. P. 1199–1218.

3. King James C. Symbolic Execution and Program Testing // Commun. ACM. 1976.Vol. 19, N. 7. P. 385–394. URL: https://doi.org/10.1145/360248.360252 (Дата обращения: 18.07.2022).

4. Томилов И. О., Карманов И. Н., Звягинцева П. А., Грицкевич Е. В. Разработка методики применения фаззинга для анализа уязвимостей программного обеспечения // Системы управления, связи и безопасности. 2018. № 4. С. 48–63.

5. Томилов И. О. Трифанов А. В. Фаззинг. Поиск уязвимостей в программном обеспечении без наличия исходного кода / XIII Международные научный конгресс и выставка. Интерэкспо Гео-Сибирь-2017. Магистерская научная сессия «Первые шаги в науке» // Сборник материалов. 2017. Т. 2. С. 75–80.

6. Официальный сайт фаззера AFL++ [Электронный ресурс]: [веб-сайт]. Aflplus.plus – 2022. URL: https://aflplus.plus/ (дата обращения: 18.07.2022).

7. Уроки по AFL++ [Электронный ресурс]: [веб-сайт]. GitHub.com – 2022. URL: https://github.com/ antonio-morales/Fuzzing101 (дата обращения: 18.07.2022).

8. Организация фаззинга исходного кода [Электронный ресурс]: Блог компании Digital Security.: [веб-сайт]. habr.com – 2022. URL: https://habr.com/ru/ company/dsec/blog/517596/ (дата обращения: 18.07.2022).

9. Классификация видов тестирования [Электронный ресурс]: [веб-сайт]. qa-academy.by – 2022. URL: https://qa-academy.by/qaacademy/ news/ klassifikaciya-vidov-testirovaniya/ (дата обращения: 18.07.2022).