Fuzzinator:
An Open-Source Modular
Random Testing Framework

Renát Hodován and Ákos Kiss
Department of Software Engineering
University of Szeged
Dugonics tér 13, 6720 Szeged, Hungary
Email: hodovan@inf.u-szeged.hu, akiss@inf.u-szeged.hu

Abstract—There is no fuzzing without a random test generator, but to be useful in practice, it needs to be accompanied by some similarly important components: solutions to feed test cases to the target system, unique error detectors, automatic test case reducers, or issue reporters. Unfortunately, many projects either still focus only on test case generation and give no support for the rest of the tasks, or they tightly couple several components into a monolithic artifact. In this paper, we introduce the Fuzzinator open-source random testing framework that supports connecting all, potentially independently developed components of a fuzzing setup in a modular way.

Index Terms—Random testing, fuzzing, failure detection, test case reduction, modular framework.

“It’s the best Inator I’ve ever made!”
—Dr. Heinz Doofenshmirtz

I. INTRODUCTION

Although the idea of random or fuzz testing is not a new one [1], the scientific research of the topic is still very active. Multiple papers are published by various authors and research groups every year on how fuzzing can be applied to various targets – e.g., to web engines [2], operating system internals [3], compilers [4], interpreters [5], [6], user interfaces [7], etc. – or how random test case generation can be made more efficient – e.g., using constraint solvers [8], coverage information [9], evolutionary approaches [10], [11], or machine learning [12]. However, fuzzing attracts the attention of not only the members of the scientific community but also of the industry, and even of unaffiliated enthusiasts: a wide variety of fuzzers is available on the Internet at different levels of maturity [13]. The popularity of fuzzing is not surprising, as the idea of replacing or complementing at least some of the expensive human testers with computers that are getting increasingly powerful and also cheaper at the same time is highly appealing – especially because the typical problems fuzzers can reveal are potentially high priority safety critical program abortions, illegal memory accesses, or assertion failures.

However, as soon as the complexity of the tested application grows beyond that of a “Hello World!” program, random test case generators – no matter how cleverly written – are not enough on their own. Undoubtedly, there is no fuzzing without a test generator, but to be useful in practice, it needs to be accompanied by some similarly important components. First of all, whenever a new test case is generated, it must be transported to the system-under-test (SUT): even simple transport infrastructure. Then, as the strength of fuzzing is that it automatically generates large amounts of test cases, it is also important to automatically recognize when a test case drives the SUT into unexpected behaviour. This can be a program crash, a non-zero exit code, an uncaught exception message on the error channel, or a timeout of a remote service, etc. Moreover, not only is it important to detect errors in the SUT but it is also useful to recognize unique bugs, as random test generators can – and tend to – make SUTs fail similarly (e.g., access illegal memory from the same machine instruction, throw an exception at the same location, fail on the same assertion) on different inputs. Keeping multiple test cases that trigger the same error is not only superfluous but it can also lead to the waste of resources when it comes to the analysis of the found bugs. Finally, as fuzzed test cases are randomly generated, even if they trigger a failure in the SUT, they rarely contain the minimum information that is needed for the reproduction of the error but rather they are full of useless content and only a sub-1% is interesting. Thus, automatic test case reducers can come handy [14], [15].

The above highlighted features are almost a must for any practical fuzzing setup, but depending on the use-case, some extras can be considered as ‘nice-to-have’: rapidly evolving SUTs may benefit from an automated update mechanism that fetches, builds, and inserts the latest version of the SUT into the fuzzing infrastructure to ensure that computing power is not wasted on outdated code; or it can improve project management if the found – and hopefully minimized – failure-inducing test cases are automatically reported to an issue
tracking system, thus reducing the time the project team needs to react to it.

Unfortunately, many academic, industrial, or independent projects either still focus only on the test case generator component and give no support for the rest of the tasks, thus making their solution hard to put in practice, or they tightly couple several components into a monolithic artifact which cannot be easily reused by or adapted to the needs of others. (Or they claim to have all the necessary advantages but they are not freely and openly available.)

In this paper, we introduce an open-source random testing framework, called Fuzzinator\(^1\), that aims at connecting all the important components of a fuzzing setup in a highly configurable modular way thus helping the re-use and the easy integration of independently developed projects. Thus, Fuzzinator removes the need for reinventing the wheel of fuzzing over and over again.

The rest of this paper is structured as follows: Section II describes the Fuzzinator framework in detail, Section III gives an overview of related work and similar solutions, and Section IV concludes the paper with a summary and with directions for future work.

II. Fuzzinator

During the design of Fuzzinator, we had two sets of requirements: one is for which tasks we wanted it to automate, and the other is how we wanted to achieve it. The requirements for ‘which’ have already been discussed in the previous section, while those for ‘how’ were broken down to three sub-requirements: we wanted it to help the integration of existing tools by requiring no programming from the user in basic use-cases, to enable arbitrary customization and extension if the existing solutions were not satisfactory, and to advocate and facilitate the re-use of its building blocks (be them built-ins or extensions).

Subsection II-A gives details on the high-level architecture of Fuzzinator (the ‘which’), Subsection II-B discusses the way its finer-grained components work together (the ‘how’), and finally, Subsection II-C gives a working example to show how all pieces fit together.

A. Architecture

To put it simply, Fuzzinator is nothing but a job scheduler. However, it is a scheduler that deals with jobs that are tailored for the tasks of fuzz testing. First and foremost, it has the concept of fuzz jobs, which take a fuzzer to generate a batch of test cases and call a SUT with the test cases one-by-one. Whenever the SUT runs into an issue that has not been seen before, it records it in a database and schedules a reduce job for running. The task of these jobs is to connect reducers to SUTs, i.e., to make a test case reducer generate simpler versions of a previously found failure-inducing input and feed these versions into the same SUT that was called by the fuzzer to find whether they still reproduce the issue. Once a reduce job finishes, its result is used to update the database with the compact version of the test case. Moreover, when a reduce job drives its SUT into a previously unseen issue (which is not that frequent but does happen, as the reduction process is usually also stress-testing the SUT), it is also recorded and a separate reduce job is scheduled again. The scheduler also works with the concept of update jobs, which have the duty to detect when a SUT becomes outdated (e.g., based on timestamps or VCS information) and then to perform a SUT-specific update. Finally, as SUTs can change over time, the scheduler can run validation jobs to check whether a previously found and recorded issue can still be reproduced on the current SUT version; and when an issue becomes obsolete, it is marked so in the database. Similarly to reduce jobs, if the corresponding test case triggers another new issue in the SUT during validation, it will go through the record and reduce process again.

Figure 1 (especially its bottom part) gives a high-level view of the above mentioned concepts.

B. Components

The previous subsection only enumerated the job concepts of the core scheduler of Fuzzinator but it was intentionally vague about what ‘generating a batch of test cases’, ‘calling a SUT’, or ‘detecting an issue’ meant. These concepts belong to a lower level of abstraction and require the mentioning of some implementation details. Fuzzinator is a Python 3-based framework using the NoSQL MongoDB system as its database. Within the framework, every fuzzer is simply represented by a function (or a callable, to use a more generic Python term), and whatever data such a fuzzer function returns is considered as a new test case. Similarly, a SUT is also only a function that shall take the result of the fuzzer function and return something that it considers as an adequate description of an issue (if it failed, nothing otherwise), the only requirement being that it should be a dictionary (a key-to-value mapping) so that the framework can add annotations later when necessary (e.g., to put the test case that triggered the issue in the mapping).

\(^1\)Fuzzinator is open-sourced at https://github.com/renatahodovan/fuzzinator, and is also available via pip install fuzzinator.
Reducers follow the same logic, i.e., they are functions that take a SUT (the function) and a test case (the data previously returned by the fuzzer function), and return a version of the input that is more compact but still makes the SUT return a similar issue description. And finally, the identification of unique issues is also working on the above abstractions by enabling the SUTs to put a special key–value pair in the returned dictionary, which is then checked for equivalence in comparisons (to prevent comparing the whole contents of issue dictionaries).

This – seemingly simple – generic representation of all actors (that they are all Python functions) and artifacts (that they are dictionaries of arbitrary structures) opens up the possibility of great modular flexibility and extensibility. E.g., if the SUT is a project written in Python and exposes an API, its functions can be directly called, and if assertion errors are raised, they can be wrapped into a dictionary to signal the found failure. Alternatively, if the SUT is a command line tool that reads from the console, we can write a simple wrapper that uses Python’s standard libraries to start a child process, pipe the test case to the standard input, and create a new dictionary with the exit code, if it is not zero. Similarly, a fuzzer can be a simple Python function constructing arrays of arbitrary bytes, or a command line tool that generates files in some directory wrapped into a small Python callable to execute the tool and return either the contents or the file system paths of the generated files (depending on the needs of the SUT). Moreover, as functions can wrap or decorate other functions in Python, it becomes easy to separate orthogonal aspects into separate building blocks. E.g., various fuzzer implementations may generate random web content but all those fuzzers should not implement their own web server to deliver the test cases – the fuzzer functions should only focus on the content generation and let a decorator be applied to handle the transport over HTTP, if it is so expected by the SUT.

To make Fuzzinator useful out-of-the box, we analyzed the fuzzing scenarios we had faced in the past and identified their common aspects and their differences. We turned the smallest units of common aspects into parametric building blocks (i.e., SUT call, fuzzer, reducer, updater functions, etc.) and made them parts of the framework. Table I gives a list of the more important built-in building blocks thus created. Hopefully, the most common fuzzing setups shall not require additional programming but can re-use these built-in components.

The glue between these components is a standard INI configuration file format with predefined section and property names. The configuration files can name Python entities (functions) and their arguments, as well as their order of composition – and the dynamic inspection capabilities of Python enable and ensure their proper execution. The next subsection gives a concrete example of how this works in practice.

C. Example

In this subsection, we show how a working configuration file can be created for the fuzz testing of a real target. We use the JerryScript project\(^2\) as an example, a JavaScript execution engine created for resource-constrained IoT devices which provides a Linux command line wrapper around the engine for testing purposes.

We start with a minimum configuration that defines one SUT and one test generator using built-in functions, as shown in Listing 1. The sections of the configuration starting with a sut. prefix define how SUTs will be handled and they also give a name to the SUT (the rest of the section name after the prefix). In the example, the call property of the sut.jerry section defines that StdinSubprocessCall (which writes the test case to the stdin of its subprocess target and returns an issue dictionary if the target exits with a non-zero code) will be used for executing the jerry SUT. The parameters of the call (those specific to StdinSubprocessCall) are defined in the sut.jerry.call section: how (command) and where (cwd) the target must be executed. The sections starting with fuzz. connect SUTs with test case generators. In the example, the fuzz section references the appropriate SUT section and the RandomContent fuzzer that simply produces random strings.

As registering all the non-zero exits of the SUT as issues is not really useful in practice, we should filter out all issues when the SUT did not exit with specifically interesting exit codes and did not print messages to stderr matching a given pattern. We can make use of the fact that StdinSubprocessCall puts exit code information and stdout, stderr dumps into its issue dictionaries, and of the built-in decorator solutions that can use this data, as shown in

2 http://jerryscript.net

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>BUILT-IN BUILDING BLOCKS OF FUZZINATOR.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SUT calls</strong></td>
<td>Subprocess executor with input on command line</td>
</tr>
<tr>
<td></td>
<td>Exit code filter</td>
</tr>
<tr>
<td></td>
<td>GDB and LLDB backtrace creators</td>
</tr>
<tr>
<td></td>
<td>Sensitive data anonymizer</td>
</tr>
<tr>
<td></td>
<td>AFL integration</td>
</tr>
<tr>
<td></td>
<td>Random byte array generator</td>
</tr>
<tr>
<td></td>
<td>HTTP server of fuzzer output</td>
</tr>
<tr>
<td></td>
<td><strong>Updaters</strong></td>
</tr>
<tr>
<td></td>
<td>File age-based update condition</td>
</tr>
<tr>
<td></td>
<td>Update script executor</td>
</tr>
</tbody>
</table>

418
Listing 1

MINIMUM CONFIGURATION FOR FUZZING JERRYSCRIPT.

```python
[sut.jerry]
call=fuzzinator.call.StdinSubprocessCall

[sut.jerry.call]
command=./build/bin/jerry
    cwd=</path/to/jerryscript/root/directory>

[fuzz.jerry-with--random]
sut=sut.jerry
fuzzer=fuzzinator.fuzzer.RandomContent
```

Listing 2

FILTERING ISSUES.

```python
[sut.jerry]
call.decorate(0)=fuzzinator.call.ExitCodeFilter
call.decorate(1)=fuzzinator.call.RegexFilter

[sut.jerry.call.decorate(0)]
exit_codes=[132, 129]

[sut.jerry.call.decorate(1)]
stderr=["(?P<msg>Assertion \'.\' failed )at
  (\?P<file>[\(\?P<path>\[\(\?P<line>\[0-9]\]+)]")
```

Listing 3

EXTENDING ISSUES.

```python
[sut.jerry]
call.decorate(2)=fuzzinator.call.PlatformInfoDecorator
call.decorate(3)=fuzzinator.call.SubprocessPropertyDecorator
call.decorate(4)=fuzzinator.call.UniqueIdDecorator

[sut.jerry.call.decorate(3)]
property=version
    command=git rev-parse --short HEAD
cwd=${{sut.jerry.call:cwd}}

[sut.jerry.call.decorate(4)]
properties=["msg", "path"]
```

Listing 4

UPDATING THE SUT AND REDUCING TESTS.

```python
[sut.jerry]
update_condition=fuzzinator.update.TimestampUpdateCondition
update=fuzzinator.update.SubprocessUpdate
reduce=fuzzinator.reduce.Picire

[sut.jerry.update_condition]
    age=12:00:00
        path=${{sut.jerry.call:cwd}}/build/bin/jerry

[sut.jerry.update]
    command=git pull origin master &&
        /tools/build.py --debug --clean
cwd=${{sut.jerry.call:cwd}}
```

Listing 5

Listing 6

III. RELATED WORK

As it was mentioned earlier, having a good framework around a test generator is just as important as having the test generator itself. The minimum expected functionality of such a framework is to recognize the arising issues and save all the required information for reproduction. Otherwise, testing is completely in vain. However, the framework is open to replace or mix any and all parts of the setup with custom-coded external building blocks, if necessary.

Figure 2 shows a screenshot of Fuzzinator configured for multiple SUTs – including the above introduced JerryScript – in action. (The shown text-based user interface is one of the two options currently available, the other is a classic line-based logger.)

The issues cannot only be filtered but also extended with arbitrary information that helps describe the circumstances of the failure, which can also be implemented with decorators. Listing 3 shows how platform, git version, and ID information can be added using built-in solutions: PlatformInfoDecorator adds an extra platform field to the issue dictionary, filled with OS information, SubprocessPropertyDecorator adds a user-defined field (version) with the output of a user-defined script, and UniqueIdDecorator combines the existing fields (msg and path fields previously found by RegexFilter) into an ID to help detect whether an issue is unique or it is a duplicate of an already known one.

Finally, we can also control how our SUT is updated and how the test cases of the found issues should be reduced. Listing 4 shows the use of built-in building blocks again: TimestampUpdateCondition triggers the update based on the last modification time of the target binary, SubprocessUpdate updates JerryScript using a script, and Picire reduces the test cases with the help of the Picire\(^3\) project.

The four short listings presented here show how multiple aspects of the fuzzing of a real target can be customized with a simple configuration file using built-in components only. However, the framework is open to replace or mix any and all parts of the setup with custom-coded external building blocks, if necessary.

3. \(\text{https://github.com/renatahodovan/picire}\)
Perhaps the most often mentioned fuzzer openly available ‘out there’ is American Fuzzy Lop\(^4\), a coverage-guided mutational test-generator and fuzzing framework in a single tool. It provides an easy-to-use command line interface with built-in test generation, crash detection, unification and reduction. However, it only works with such SUTs that were compiled with its specially crafted GCC, and the test generators cannot be customized without digging into its source code.

Another project where the framework and the generator is incorporated into a single tool is PeachFuzzer\(^5\). Peach applies a model-based generational and mutational approach for input test generation. Both the input model definitions and the framework configurations are given with XML-based descriptors, so called PIT files. Although Peach started as an open-source project, it was forked into a private repository several years ago and now it is only commercially available. Mozilla has forked its last public release into a public repository but it is not actively developed according to the logs. Instead, they focus on FuzzManager\(^6\), which is the closest approach to our work both conceptually and realization-wise. It is also an open-source, pure fuzzer framework that intends to work with arbitrary SUTs and test generators. Most of the setup can be set via configuration files. Its two main components are a server and a client module where the server is responsible for managing the issues and the client helps to process the SUT’s output and transform that information into a form that is suitable for the server. Although the basics are similar to ours, there are some differences too. E.g., it expects the users to adapt their fuzzer to communicate with the provided interface, and it does not give support for scheduling multiple fuzz, update, or reduce sessions. On the other hand, it has a nice web interface to browse and edit reported issues.

CERT has developed the Basic Fuzzing Framework\(^7\) (BFF for short). It is an open-source cross-platform framework for file-based fuzzing with built-in mutational test generation support. It is also configurable via files and it can recognize, unify, and reduce the found issues. Unfortunately, there has been no public activity on the project since the end of 2016.

Grinder\(^8\) is also a well-known framework specialized to manage browser fuzzer sessions. It has a client-server architecture where the server collects and manages the issues reported from various clients. According to its GitHub page, it is only available for Windows and there has been no recent activity.

Another target-specific framework are Sulley\(^9\) and its successor boofuzz\(^10\). Both are network protocol-specific fuzzing frameworks with built-in test generators, issue detectors and minimization tools. Since Sulley has fallen out of maintenance, its development is continued as a part of the boofuzz project.

Beside these self-hosted approaches, various online fuzzing services exist to choose from.

Google runs the OSS-Fuzz\(^11\) project, a fuzzing service for open-source projects with large user bases. OSS-Fuzz uses libFuzzer\(^12\) — a coverage guided in-process fuzzer – for test generation, ClusterFuzz\(^13\) as the execution environment, and the Monorail\(^14\) reporting engine. To use this service, the project must be successfully nominated for testing. As a

\(^4\)[http://lcamtuf.coredump.cx/afl/]
\(^5\)[https://www.peach.tech]
\(^6\)[https://github.com/MozillaSecurity/FuzzManager]
\(^7\)[https://www.cert.org/vulnerability-analysis/tools/bff.cfm]
\(^8\)[https://github.com/stephenfewer/grinder]
\(^9\)[https://github.com/OpenRCE/sulley]
\(^10\)[https://github.com/jtpereyda/boofuzz]
\(^11\)[https://github.com/google/oss-fuzz]
\(^12\)[http://llvm.org/docs/LibFuzzer.html]
\(^13\)[https://github.com/google/oss-fuzz/blob/master/docs/clusterfuzz.md]
\(^14\)[https://bugs.chromium.org/p/monorail]
further requirement, the project must be equipped with one or more fuzz targets – such functions that can cleverly mutate an input byte array –, and a build script that creates an instrumented binary.

Microsoft also has a similar program – originally called Project Springfield, then rebranded as Microsoft Security Risk Detection[15] –, to test Windows applications. This service runs in the Microsoft Azure environment – where the tested application has to be uploaded to –, and it uses the well-known SAGE [8] fuzzer for test generation.

FuzzStati0n16, a recently started commercial fuzzing service for testing Node JS applications with AFL-based test generation should also be mentioned. Similarly to the previous services, FuzzStati0n reports the found issues to the user on a web interface.

IV. SUMMARY

In this paper, we have introduced the Fuzzinor random testing framework and how it modularized the various tasks and components of a fuzz testing setup. We have also shown how it can be used to configure the fuzzing of an industrially used project without writing any code. The source of the framework has been published and ready-to-run packages of the framework are also available from the standard PyPI software repository.

The framework has been actively used to test multiple projects (Duktape17, JerryScript, IoT.js18, WebKit19, Chromium20) with various random test generators (AFL, Grammarinator21, Generinator:RATS22) over the past few years. It has helped finding, managing, and reporting close to 1000 issues (e.g., 170+ for JerryScript, 450+ for WebKit, and 280+ for Chromium).

The framework is still under development, we have plans to add a graphical or web-based user interface, to support multi-machine scheduling, and to support feedback-based fuzzers that can use the information from SUT executions (e.g., coverage data) in the generation of the next test cases. A non-functional plan, already in progress, is to publish known-to-work configuration files in a repository23 to help the adoption of the framework.

ACKNOWLEDGMENT

Renáta Hodován was supported in part by the EU-supported Hungarian national grant GINOP-2.3.2-15-2016-00037. Ákos Kiss was supported by EFOP-3.6.3-VEKOP-16-2017-00002, a project supported and co-financed by the European Union and the European Social Fund. The authors are thankful for the work of Dániel Vince, also supported by EFOP-3.6.3-VEKOP-16-2017-00002.

REFERENCES