FOT: A Versatile, Configurable, Extensible Fuzzing Framework

Hongxu Chen*
Nanyang Technological University
Singapore

Yuekang Li*
Nanyang Technological University
Singapore

Bihuan Chen†
Fudan University
China

Yinxing Xue
University of Science and Technology of China
China

Yang Liu
Nanyang Technological University
Singapore

ABSTRACT

Greybox fuzzing is one of the most effective approaches for detecting software vulnerabilities. Various new techniques have been continuously emerging to enhance the effectiveness and/or efficiency by incorporating novel ideas into different components of a greybox fuzzer. However, there lacks a modularized fuzzing framework that can easily plugin new techniques and hence facilitate the reuse, integration and comparison of different techniques.

To address this problem, we propose a fuzzing framework, namely Fuzzing Orchestration Toolkit (FOT). FOT is designed to be *versatile*, *configurable* and *extensible*. With FOT and its extensions, we have found 111 new bugs from 11 projects. Among these bugs, 18 CVEs have been assigned.

Video link: https://youtu.be/O6Qu7BJ8RP0.

CCS CONCEPTS

• **Security and privacy → Software security engineering**;

KEYWORDS

Greybox Fuzzing; Software Testing

ACM Reference Format:
Hongxu Chen, Yuekang Li, Bihuan Chen, Yinxing Xue, and Yang Liu. 2018. FOT: A Versatile, Configurable, Extensible Fuzzing Framework. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3236024.3264593

1 INTRODUCTION

Greybox fuzzing has become one of the most effective approaches to detect vulnerabilities in a program under test (PUT). Compared with whitebox and blackbox fuzzing, greybox fuzzing strikes a balance between execution speed and effectiveness. The past years have witnessed a number of greybox fuzzing frameworks, e.g., AFL [16], libFuzzer [11], and honggfuzz [7], followed by various fuzzing extensions [2, 3, 6, 10, 12] to enhance their effectiveness and/or efficiency.

However, there lacks a fuzzing framework to easily reuse, integrate and compare different fuzzing techniques and experiment with new ideas. Take AFL as an example, it is implemented all in one file with around 8K LOC, which contains more than 100 global variables. Hence, the implementation of a single feature often involves modifications in multiple places. In short, AFL is compact but also highly coupled because AFL is designed to *require essentially no configuration* [16]. In fact, most of the existing fuzzers are designed for easy deployment and usage, but not easy extension. Therefore, it is desirable to have a fuzzing framework that allows easy *configuration* and *extension* for new features.

To this end, we propose our fuzzing framework, namely Fuzzing Orchestration Toolkit (FOT). FOT is designed to hold three properties.

(1) **Versatility.** FOT provides a fuzzing ecosystem, including a set of static and dynamic analyses used to aid the fuzzing process.
(2) **Configurability.** FOT provides a set of configurable options. Users can easily tweak the parameters of the fuzzer to improve the fuzzing effectiveness with their experience.
(3) **Extensibility.** FOT is designed to be of high coherence and low coupling. Specially, the implementation mainly consists of two parts: the library containing general fuzzing utilities and miscellaneous tools on top of it. Therefore, apart from the default fuzzer provided by FOT, developers can write their own fuzzers with modest effort based on the library.

2 ARCHITECTURE DESIGN

In this section, we describe the design of FOT framework. Interested readers can refer to FOT's project site for more details: https://sites.google.com/view/fot-the-fuzzer.

Figure 1 depicts the overview of FOT. It consists of three parts, namely the *preprocessor*, the *fuzzer*, and the *complementary toolchain*. Components of the framework are represented with *blue* rectangles. All these components are *configurable* and *extensible*.

2.1 Preprocessor

This part contains various tools for collecting static information and instrumentation with the PUT.

*Hongxu Chen and Yuekang Li have equal contribution in this work.
†Bihuan Chen is the corresponding author, and with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China and the Shanghai Institute of Intelligent Electronics & Systems, China.

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3264593

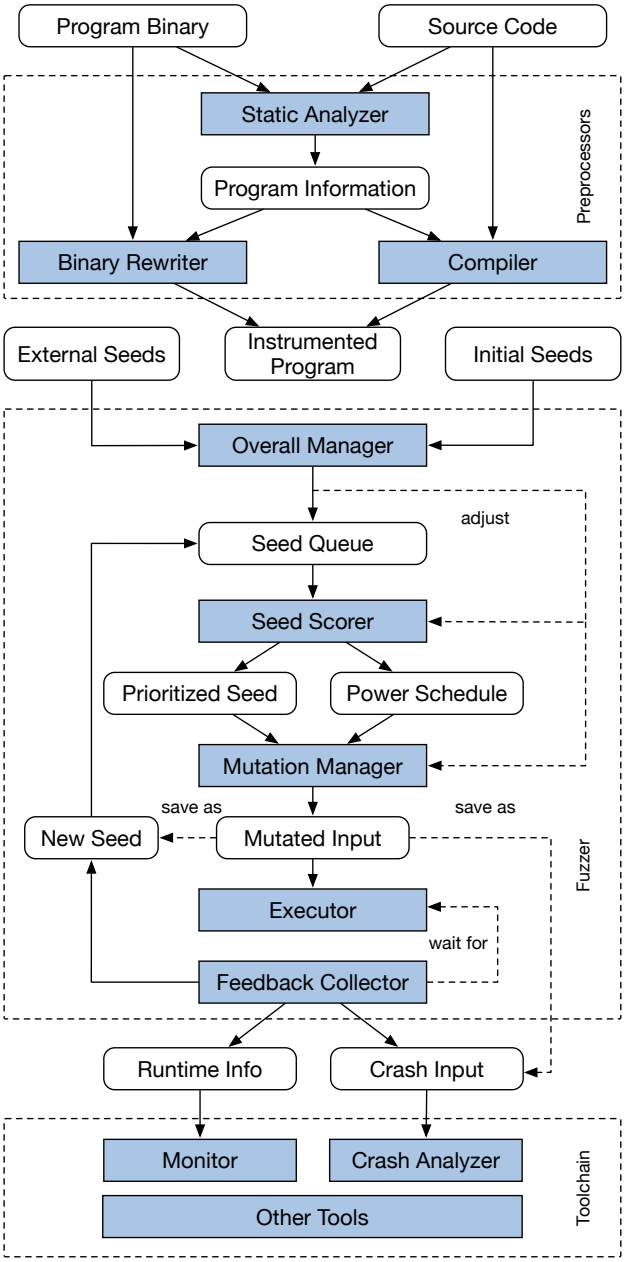

Figure 1: Overview of the FOT Fuzzing Framework

2.1.1 Static Analyzer. This includes various tools to extract semantic understandings from the PUT. For example, we have tools to generate the control flow graph, call graph or statically collected vulnerability information and convert them into suitable representations that can later be instrumented into the PUT and utilized during the fuzzing process. This part is *configurable* to generate different levels of static information. It is *extensible* as developers are allowed to add new types of static analysis as long as the generated result follows the specified format.

2.1.2 Instrumentor. The *binary rewriter* and the *compiler* instrument additional static information generated by the static analyzer into the PUT so that the fuzzer can collect feedback from the latter during execution. FOT supports Dyninst [1] based instrumentation when only binary is provided, and LLVM based instrumentation when the source code is available. This part is *configurable* as the users can choose to either instrument at source code level or at binary level. It is *extensible* since developers can use other tools such as Intel Pin [9] for instrumentation as long as the instrumented code can embed the static information and follow the regulations to provide feedback for the fuzzer.

2.2 Fuzzer

This part explains FOT's the main fuzzing process. It is essentially a loop that continuously selects seeds from the queue, applies mutations to the selected seeds, executes the PUT against mutated inputs, and collects feedback for the next iteration.

2.2.1 Overall Manager. As FOT is designed to support multi-threaded parallel fuzzing, it contains an overall manager for fuzzing, managing the workload of each worker thread. Particularly, it can listen to a special directory to actively import seed inputs from external sources such as symbolic executors like KLEE [4] or mutation generators like Radamsa [8]. This part is *configurable* as the users can choose different strategies for the overall management. It is *extensible* as it can interoperate with other seed generation tools.

2.2.2 Seed Scorer. The seed scorer is in charge of selecting a seed from the queue for mutation (seed prioritization) and determining how many new inputs should be generated based on the selected seed (power scheduling). This part is *configurable* as the users can select from several built-in scoring strategies to evaluate seeds. It is *extensible* as the users can implement their own strategies with the interfaces provided in FOT.

2.2.3 Mutation Manager. The mutation manager is in charge of incorporating different mutators. It can mutate the seeds in a pure random manner or according to predefined specifications. This part is *configurable* as FOT provides various mutators for the users to choose from. It is *extensible* as the developers can implement their own mutators with the provided library.

2.2.4 Executor. The executor drives the execution of the PUT. This part is *configurable* as the default executor in FOT allows users to choose whether or not to use forkserver [16] during fuzzing. It is *extensible* as the developers can extend the executor for different scenarios. For example, they may add a secondary executor to execute another PUT to perform differential testing.

2.2.5 Feedback Collector. The feedback collector collects the feedback emitted by the PUT. The exact feedback often corresponds to the instrumented information. This part is *configurable* as the users are allowed to select from the default feedback options provided by FOT. For now, the feedback can be at basic-block level (like AFL) or function level. It is *extensible* as the users can specify their customized types of feedback for collection.

2.3 Complementary Toolchain

FOT additionally contains various tools helping to make the framework *versatile*. For instance, we implemented a web-based frontend user interface to monitor the fuzzing results. It provides fruitful

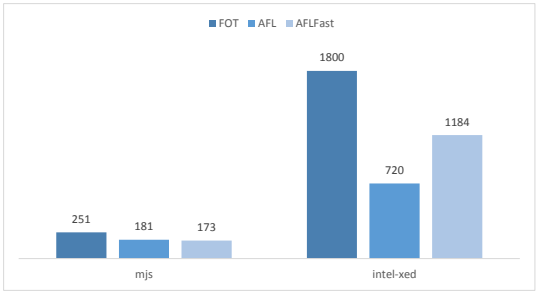

Figure 2: Average Number of Unique Crashes Found in 24 Hours on *mjs* and *intel-xed*.

information to make the fuzzing process more transparent. We also implemented a crash analyzer to analyze the detected crashes and generate reports automatically. This reduces the manual efforts of crash triaging. Last but not the least, several other tools have been developed with different purposes to complement the fuzzer.

3 IMPLEMENTATION AND EXTENSIONS

We have implemented the FOT framework and developed two extensions to FOT.

3.1 Implementation

The FOT project started from June, 2017 and has been actively developed by two researchers. It is implemented with 15000 lines of Rust for core fuzzing modules, together with 2600 lines of C/C++ for the preprocessor, 4800 lines of Java for structure-aware mutation, and 2400 lines of Python for complementary toolchain.

3.2 Static Vulnerability Analysis Integration

Greybox fuzzers are typically aware of quantitative changes of code coverage and use such feedback for keeping the *interesting* seeds. However, the performance of collecting code coverage feedback quantitatively is often not ideal, and greybox fuzzers also need to evaluate the code coverage qualitatively [3]. One of the approaches to bring qualitative awareness about the covered code is to combine fuzzing with static vulnerability analysis, as mentioned in § 2.1.1.

Existing fuzzing works seldom use static analysis information to facilitate seed prioritization and power scheduling since existing fuzzing frameworks have little support for them. In contrast, integration with vulnerability static analysis is trivial in FOT: we used *static analyzer* to calculate vulnerability metrics (e.g., calls to unsafe functions and cyclomatic complexity) and customized *seed scorer* to take them into account during fuzzing. This extension added about 330 lines of C++ and 190 lines of Rust code.

The workflow is as follows. First, the static analyzer calculates vulnerability score for each function. Then we instrument the PUT to provide function level coverage information. After detecting a seed that brings new coverage, the feedback collector will collect its function level coverage and map it with the static analysis result to get the function level vulnerability scores. The seed scorer will then accumulate the function-level scores to form the execution trace level vulnerability scores for the exercised seeds. Finally, the power schedule determined by the seed scorer prioritizes and allocates more powers to the seeds with higher vulnerability scores.

Table 1: Crash Reproduction in FOT, AFLGo and AFL Against Binutils (Taken from [5]).

CVE-ID	Tool	Runs	μTTE(s)	Factor
2016-4487	FOT	20	177	–
	AFLGo	20	390	2.20
	AFL	20	630	3.56
2016-4492	FOT	20	477	–
	AFLGo	20	540	1.21
	AFL	20	960	2.01
2016-6131	FOT	9	17314	–
	AFLGo	6	21180	1.22
	AFL	2	26340	1.52

Figure 2 shows the average number of unique crashes detected on mjs and intel-xed of different fuzzers over 10 runs. We can see that with the help of static vulnerability analysis, FOT can detect more unique crashes in a limited time budget.

3.3 Directed Greybox Fuzzing (DGF)

Guiding the greybox fuzzer towards certain predefined locations in the PUT can fit multiple scenarios such as patching testing, crash reproduction, and static analysis report verification [2].

DGF requires the fuzzer to evaluate seeds according to their distances towards target locations. AFLGo [2] is a DGF based on AFL. It applies a simulated-annealing-based power schedule for the seeds according to their distances from target locations. However, building an effective directed fuzzer requires not only adjustments of *power schedules* but also *seed prioritization* and *mutation strategies*.

DGF in FOT is done by generating the static distances to the target locations with the help of *static analyzer* and customized *program instrumentation*, *feedback collector*, *seed scorer* as well as the *mutation manager*. The implementation added about 240 lines of C++ and 510 lines of Rust code.

The workflow is as follows. The preprocessor calculates the distances to target locations for each basic-block and function, and then instruments the basic-block level distance information during compilation. During fuzzing, the fuzzer collects the distance information along the executed traces for the seeds. The seed scoring module will prioritize the seeds closer to the targets and assign more powers to them. Moreover, the mutation manager will favor fine-grained mutations once the target function is reached.

Table 1 compares the results of FOT with AFL and AFLGo on the c++filt tool in GNU Binutils (each experiment was conducted 20 times with 8 hours as the budget). **μTTE** is the average time-to-exposure in seconds to trigger a vulnerability. We can see that FOT is able to decrease the exposure time greatly.

3.4 New Vulnerabilities

Till now, FOT has been used to fuzz more than 100 projects. Table 2 lists some of the 0-day vulnerabilities we found with FOT. Among them, 6 CVEs have been assigned to Oniguruma (a widely used regular expression library used by PHP, Ruby, etc) and 9 CVEs have been assigned to Espruino (a Javascript engine for IoT devices). GNU diffutils, GNU bc and apcalc have been used for many years. Other projects such as radare2 (an open source reverse engineering framework) and libsass (the SASS library) have been fuzzed for multiple times by others.

Table 2: Selected Trophies and the Projects

Project Name	0-day Bugs	Time Since Release	GitHub Stars	KLOC
mjs	21	1y7m	787	16.5
liblnk	20	8y9m	42	56.9
GNU bc	18	26y8m	–	31.4
radare2	10	9y4m	7645	857.1
Espruino	10	4y9m	1395	1392.6
libsass	10	6y5m	3813	43.8
libpff	7	3y8m	85	137.7
Oniguruma	6	5y8m	556	119.1
apcalc	4	19y	–	98.7
FLIF	3	2y9m	2989	43.6
diffutils	2	29y7m	–	147.4

4 RELATED WORK

In this section, we first compare FOT with other fuzzing frameworks, and then discuss its relationship to current fuzzing extensions.

4.1 Comparisons to Other Fuzzing Frameworks

Table 3 compares FOT with existing fuzzing frameworks with respect to 10 major features. As we can see, the existing fuzzing frameworks AFL, libFuzzer and honggfuzz lack features in different aspects, while FOT integrates all of them. FOT stands out in that it provides various configurations for advanced users; it is also highly modularized to be easily extended with other fuzzing techniques. Further more, FOT also partially supports structure-aware mutations (by specifying semantic grammars) and interoperability with other seed generation tools such as symbolic executors (by monitoring and scheduling newly incoming seed input directory).

4.2 Relationship to Current Fuzzing Extensions

Most current fuzzing techniques can be easily integrated into FOT thanks to its highly-modularized design. In fact, these techniques can be applied with some extensions to the different components in Figure 1 and can be used together with the configuration interface.

1) AFLFast [3] can be implemented by applying a Markov Chain model based seed *power scheduling* in the fuzzer.
2) AFLGo [2] can be implemented by a combination of *static analyzer*, *instrumentation* and *power scheduling*.
3) CollAFL [6] can be implemented by using a collision-resistant algorithm to increase the uniqueness of the path trace labeling during *instrumentation*.
4) Skyfire [14], Radamsa [8], Csmith [15] can be used in the preprocessor to generate seeds for the *external seeds* and a structure-aware mutator assigned by *mutation manager*.
5) Symbolic executors such as KLEE [4] can be integrated in the Driller's [13] style with the help of *overall manager*.

5 CONCLUSIONS

We have proposed FOT, a versatile, configurable, and extensible fuzzing framework to facilitate the reuse, integration, comparison and development of different fuzzing techniques. We briefly explained the workflow of FOT and showed its applications in the aspects of static vulnerability analysis enhanced fuzzing and directed grey-box fuzzing. The experimental results have indicated that FOT can be quite effective for different fuzzing scenarios.

Table 3: Comparisons between Different Fuzzers (○: not supported, ◐: partially supported, ●: fully supported)

Features \ Framework	AFL	libFuzzer	honggfuzz	FOT
Binary-Fuzzing Support	●	○	●	●
Multi-threading Mode	○	●	●	●
In-memory Fuzzing	●	●	●	●
Advanced Configuration	○	◐	○	●
Modularized Functionality	○	◐	○	●
Structure-aware Mutation	○	○	○	◐
Interoperability	○	○	○	◐
Toolchain Support	●	○	○	●
Precise Crash Analysis	○	○	●	●
Runtime Visualization	◐	○	○	●

ACKNOWLEDGMENT

This work is supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2016NCR-NCR002-026) and administered by the National Cybersecurity R&D Directorate; the research of Dr Xue is also supported by Chinese Academy of Sciences (CAS) Pioneer Hundred Talents Program of China.

REFERENCES

[1] Andrew R. Bernat and Barton P. Miller. 2011. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools (PASTE '11)*. ACM, New York, NY, USA, 9–16. https://doi.org/10.1145/2024569.2024572
[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *CCS*. 2329–2344.
[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing As Markov Chain. In *CCS*. 1032–1043.
[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI*. 209–224.
[5] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a Desired Directed Grey-box Fuzzing. In *CCS*. https://doi.org/10.1145/3243734.3243849
[6] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *S&P*. 660–677.
[7] Google. 2018. honggfuzz. https://github.com/google/honggfuzz
[8] Aki Helin. 2018. radamsa - a general-purpose fuzzer. https://gitlab.com/akihe/radamsa
[9] Osnat Levi (Intel). 2018. Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool
[10] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: Program-state Based Binary Fuzzing. In *ESEC/FSE*. 627–637.
[11] LLVM. 2018. libFuzzer. https://llvm.org/docs/LibFuzzer.html
[12] Theofilos Petsios, Adrian Tang, Salvatore J. Stolfo, Angelos D. Keromytis, and Suman Jana. 2017. NEZHA: Efficient Domain-Independent Differential Testing. In *S&P*. 615–632.
[13] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.
[14] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-Driven Seed Generation for Fuzzing. In *S&P*. 579–594.
[15] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*. 283–294.
[16] Michal Zalewski. 2014. American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.