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:

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.
2.1 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.

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

<table>
<thead>
<tr>
<th>CVE-ID</th>
<th>Tool</th>
<th>Runs</th>
<th>µTTE(s)</th>
<th>Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-4487</td>
<td>FOT</td>
<td>20</td>
<td>177</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>AFLGo</td>
<td>20</td>
<td>390</td>
<td>2.20</td>
</tr>
<tr>
<td></td>
<td>AFL</td>
<td>20</td>
<td>630</td>
<td>3.56</td>
</tr>
<tr>
<td>2016-4492</td>
<td>FOT</td>
<td>20</td>
<td>477</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>AFLGo</td>
<td>20</td>
<td>540</td>
<td>1.21</td>
</tr>
<tr>
<td></td>
<td>AFL</td>
<td>20</td>
<td>960</td>
<td>2.01</td>
</tr>
<tr>
<td>2016-6131</td>
<td>FOT</td>
<td>9</td>
<td>17314</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>AFLGo</td>
<td>6</td>
<td>21180</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>AFL</td>
<td>2</td>
<td>26340</td>
<td>1.32</td>
</tr>
</tbody>
</table>

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 2 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 Espiruino (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.
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) CoIAFL [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.

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