Grey-Box Fuzzing: Variants and Design

Brandon Lara

School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge 70803, USA

April 2023

Abstract. Fuzzing has become a substantial component of software security testing. Grey-Box fuzzing has become the predominant fuzzer type for it's substantial performance improvements over White-Box fuzzing. Many variations of Grey-Box fuzzers exist, and each variation comes with different potential design decisions. These design decisions then imply implementation constraints. In this work, we investigate the limitations of typical design decisions made by fuzzers. We systematize and discuss the limitations imposed by fuzzing domain, design, and implementation.

Key words. Fuzzing, Symbolic Execution, Computer Security, Software Testing

1. Introduction

The expansive and complex nature of software architecture requires automated testing, as human auditing is insufficient to cover millions of lines of code. For this reason, fuzzing has become a staple tool for security professionals. Fuzzing is a form of automated testing that consists of two components: a Fuzz Generator and an Input Manager. The Fuzz Generator generates inputs for a given program (or hardware, network protocol, etc). The Input Manager takes this generated input and provides it to the entity being tested. Together these components make up what we call a Fuzzing Engine. This formulation is broad enough to allow for many approaches and variations to fuzzing implementation. This work focuses on how Grey-Box Fuzzers approach these problems. Even with this limited scope, Grey-Box fuzzers still vary wildly in design. It is worthwhile to parse the variations in fuzzers and investigate the performance, design, and implementation of fuzzing tools. Google reported that their Open Source Project Fuzzer OSSFuzz found 28,000 bugs [1]. Given the effectiveness of fuzzers as opposed to human auditing, improving fuzzer design and implementation directly reduces bugs and security vulnerabilities in software products. This makes fuzzing research a critical component of modern computer security. In the next section 3, we discuss the design of Grey-Box fuzzers and their variants. In section 4, we discuss implementation details of Grey-Box fuzzers.

2. Related Works

Fuzzing has recently spawned a large amount of research. As such, many literature reviews have also been published attempting to assess the state of fuzzing research and effectiveness. Zhu et al reviewed fuzzing literature and found significant drawbacks with the method [6]. The computational overhead of fuzzing has remained a large roadblock for the method [6]. Wang et al explored Gre-Box fuzzing [5]. They explored and systematized knowledge about various domain and intent specific fuzzers [5]. Our work adds to this on-

going review of fuzzing literature by discussing the implementation limitations of Grey-Box fuzzers.

3. Types of Fuzzers

Fuzzers can vary in design and implementation, but we will cover common categories that fuzzers typically fall under. In practice, advanced fuzzers will cross boundaries from one category to the other to generate better results [3]. Regardless of design, every fuzzer must solve two core problems:

1. How does the engine generate new inputs?
2. How does the engine supply inputs to the medium?

3.1. GreyBox vs WhiteBox Fuzzing

In the White-Box fuzzing variant, the fuzzer uses symbolic execution to find inputs that will generate desirable paths and then use a fuzzer to test code along that path [3]. In the Grey-Box Fuzzing variant, the fuzzing engine starts with a seed and mutates it through iterative testing [5]. This is different; because White-Box fuzzing generates a definitive answer about how to trigger certain execution paths and Grey-Box fuzzing performs iterative tests to search for new paths.

White-Box Fuzzing suffers, severely, from the computational overhead involved with symbolic execution. Symbolic execution engines work by scanning the conditional branches in a program and populating a SAT problem based on how the inputs affect the conditionals. This process is believed to be NP-Complete as it is reducible to SAT solving, also believed to be NP-Complete.

Additionally, determining properties of arbitrary code is an undecidable problem. Symbolic execution, which attempts to leverage SAT solving as a means to make deterministic statements about code, is not guaranteed to even produce results (no matter how much time is given) for a given program.

Grey-Box Fuzzing attempts to side step this problem with a mutating seed rather than a SAT solver. The fuzzing engine starts with a set of test inputs (the seed). Based on the outcome of supplying a given seed, the fuzzing engine will alter the input in some way. The fuzzer needs some method of determining what mutations are producing desirable outcomes. This is covered in Section 2.2.

The move and focus on Grey-Box fuzzing has produced fuzzers that find vulnerabilities much faster than their symbolic execution counterparts. The trade off is that Grey-Box fuzzers are not exhaustive while symbolic execution is[3, 5].

3.2. Coverage Guided vs Directed Fuzzing

Coverage Guided Fuzzing is a term for a fuzzing system that attempts to maximize code coverage. This concept is separated from how a fuzzer might try to maximize code coverage. There are both White-Box and Grey-Box Coverage Guided Fuzzers [5]. The aim of a coverage guided fuzzer is to execute as much of the code as possible. This is computationally intensive, and fuzzers can make improvements by only being concerned with certain types of code (Memory Access, System Calls, Shared Multi Threaded Resources, etc). This produces considerable improvements in the compute time at the cost of reducing total code coverage [2, 5]. This improvement is called Directed Fuzzing.

4. Implementation

While many fuzzers are simple in concept, implementation can be a significant barrier. Most Grey-Box fuzzers, and most fuzzers in general, are concerned with user-land binaries run on a standard OS (Linux, MacOS, Windows) [1, 2, 3, 5, 6]. To adequately assess the results of any particular fuzz test,

the fuzzer must have access to the internal execution of the program being tested. Determining code coverage required that the fuzzer know what code was covered by any given execution. Further, for directed fuzzers, determining if a certain condition was met, such as if a certain block of memory was accessed, requires knowing what memory the program accessed during runtime. Acquiring this information is nontrivial.

The standard implementation is to outfit a compiler to produce breakpoints at each branch of execution [2]. A debugger would then catch these breakpoints and record the status of the programs execution. What the fuzzer is interested in testing is unique to the fuzzer design [5]. As the program becomes more complicated, so does the implementation. Asynchronous programs could experience long idle time, waiting for inputs to trigger async callbacks. Fuzzers must either wait this time on each run, or simulate this asynchronous functionality. This further complicates the design of the fuzzer as it now has to deal with program specific async triggers.

Hardware presents a complicated problem for fuzzers, as well. Fuzzers can generate a large amount of IO, reducing the lifespan of Solid State Drives [4]. Further, Modern advanced fuzzers demand an enormous amount of compute power. Mayhem solves this problem by creating a distributed system of fuzzing networks that split the task among many machines [3]. This further complicates the development of fuzzers as scale and efficiency become paramount to fuzzing meaningfully sized programs.

5. Conclusions

Fuzzers are fast becoming cyber securities best tool to audit, uncover, demonstrate, and correct software bugs. The amount of bugs discovered via fuzzing dwarfs what could be done by hand audits of software. Despite how prolific the tactic is at making software safer, the research is still in it's infancy. Many methods are naive in their approach. Fuzzing development and research is making strides to improve the methods beyond the naive approach, but the field still suffers from making many domain and task specific fuzzers [5]. General fuzzers, capable of adapting to many targets and domains, are still quite far away. We believe that an increased research effort in fuzzing technology will yield more general techniques and lead towards a generalized fuzzing framework.

References

Taking the next step: Oss-fuzz in 2023, Feb 2023.

Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. *2012 IEEE Symposium on Security and Privacy*, 2012.

Google. Google/afl: American fuzzy lop - a security-oriented fuzzer.

Pengfei Wang, Xu Zhou, Kai Lu, Tai Yue, and Yingying Liu. The progress, challenges, and perspectives of directed greybox fuzzing. *arXiv preprint arXiv:2005.11907*, 2020.

Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Computing Surveys*, 54(11s):1–36, 2022.