Editors: D. Balzarotti, davide.balzarotti@eurecom.fr | W. Enck, whenk@ncsu.edu | T. Holz, thorsten.holz@rub.de | A. Stavrou, astavrou@gmu.edu

The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes

Mathias Payer | EPFL, Lausanne, Switzerland

Software contains bugs, and some bugs are exploitable. Mitigations protect our systems in the presence of these vulnerabilities, often stopping the program once a security violation has been detected. The alternative is to discover bugs during development and fix them in the code. The task of finding and reproducing bugs is difficult; however, fuzzing is an efficient way to find security-critical bugs by triggering exceptions, such as crashes, memory corruption, or assertion failures automatically (or with a little help). Furthermore, fuzzing comes with a witness (proof of the vulnerability) that enables developers to reproduce the bug and fix it.

Software testing broadly focuses on discovering and patching bugs during development. Unfortunately, a program is only secure if it is free of unwanted exceptions. Security, therefore, requires proof of the absence of security violations. For example, a bug becomes a vulnerability if any attacker-controlled input reaches a program location that allows a security violation, such as memory corruption. Software security testing, therefore, requires reasoning about all possible executions of code at once to produce a witness that violates the security property. As Edsger W. Dijkstra said in 1970: "Program testing can be used to show the presence of bugs, but never to show their absence!"

Digital Object Identifier 10.1109/MSEC.2018.2889892
Date of publication: 20 March 2019

System software, such as a browser, a runtime system, or a kernel, is written in low-level languages (such as C and C++) that are prone to exploitable, low-level defects. Undefined behavior is at the root of low-level vulnerabilities, e.g., invalid pointer dereferences resulting in memory corruption, casting to an incompatible type leading to type confusion, integer overflows, or application programming interface (API) confusion. To cope with the complexity of current programs and find bugs, companies such as Google, Microsoft, and Apple integrate dynamic testing into their software development cycle.

The idea of fuzzing is simple: execute a program in a test environment with random input and see if it crashes.

Fuzzing, the process of providing random input to a program to intentionally trigger crashes, has been around since the early 1980s. A revival of fuzzing techniques is taking place as evidenced by papers presented at top-tier security conferences showing improvements in the techniques' effectiveness. The idea of fuzzing is simple: execute a program in a test environment with random input and see if it crashes. The fuzzing process is inherently sound but incomplete. By producing trial cases and observing whether the tested program crashes, fuzzing produces a witness for each discovered crash. As a dynamic testing technique, fuzzing is incomplete for nontrivial programs as it will neither cover all possible program paths nor all data-flow paths except when run for an infinite amount of time. Fuzzing strategies are inherently optimization problems where the available resources are used to discover as many bugs as possible, covering as much of the program functionality as possible through a probabilistic exploration process. Due to its nature as a dynamic testing technique, fuzzing faces several unique challenges:

- *Input generation*: Fuzzers generate inputs based on a mutation strategy to explore a new state. Because the fuzzer is aware of the program structure, it can tailor input generation to the program. The underlying strategy determines how effectively the fuzzer explores a given state space. A challenge for input generation is finding the balance between exploring new paths (control flow) and executing the same paths with different input (data flow).
- *Execution engine*: The execution engine takes newly generated input and executes the program under test with that input to detect flaws. Fuzzers must distinguish between benign and buggy executions. Not every bug results in an immediate segmentation fault, and detecting a state violation is a challenging task, especially as code generally does

 Copublished by the IEEE Computer and Reliability Societies

not come with a formal model. Additionally, the fuzzer must disambiguate crashes to identify bugs without missing true positives.

- *Coverage wall*: Fuzzing struggles with some aspects of code. It may, for example, have difficulty handling a complex API, checksums in file formats, or hard comparisons, such as a password check. Preparing the fuzzing environment is a crucial step to increase the efficiency of fuzzing.
- *Evaluating fuzzing effectiveness*: Defining the metrics for evaluating the effectiveness of a fuzzing campaign is challenging. For most programs, the state space is (close to) infinite, and fuzzing is a brute-force search in this state space. Deciding, for example, when to move to another target, path, or input is a crucial aspect of fuzzing. Orthogonally, comparing different fuzzing techniques requires an understanding of the strengths of a fuzzer and the underlying statistics to enable a fair comparison.

Input Generation

Input generation is essential to the fuzzing process as every fuzzer must automatically generate test cases to be run on the execution engine. The cost of generating a single input must be low, following the underlying philosophy of fuzzing where iterations are cheap. Through input generation, the fuzzer implicitly selects which parts of the tested program are executed. Input generation must balance data-flow and control-flow exploration (discovering new code areas compared to revisiting previously executed code areas with alternate data) while considering what areas to focus on. There are two fundamental forms of input generation: model- and mutation-based input generation. The first is aware of the input format while the latter is not.

Knowledge of the input structure given through a formal description enables model-based input generation to produce (mostly) valid test cases. The model specifies the input format and implicitly indicates the explorable state space. Based on the model, the fuzzer can produce valid test cases that satisfy many checks in the program, such as valid state checks, dependencies between fields, or checksums such as a CRC32. For example, without an input model, most randomly generated test cases will fail the equality check for a correct checksum and quickly error out without triggering any complex behavior. The model allows input generation to balance the created test inputs according to the underlying input protocol. The disadvantage of model-based input generation is that it needs an actual model. Most input formats are not formally described and will require an analyst to define the intricate dependencies.

Through input generation, the fuzzer implicitly selects which parts of the tested program are executed.

Mutation-based input generation requires a set of seed inputs that trigger valid functionality in the program and then leverages random mutation to modify these seeds. Providing a set of valid inputs is significantly easier than formally specifying an input format. The input-mutation process then constantly modifies these input seeds to trigger behavior that researchers want to study.

Regardless of the input-mutation strategy, fuzzers need a fitness function to assess the quality of the new input and guide the generation of new input. A fuzzer may leverage the program structure and code coverage as fitness functions. There are three approaches to observing the program during fuzzing to provide input to the fitness function. White-box fuzzing infers the program specification through program analysis but often results in untenable cost. For example, the scalable automated guided execution white-box fuzzer leverages symbolic execution to explore different program paths. Black-box fuzzing blindly generates new input without reflection. The lack of a fitness function limits black-box fuzzing to functionality close to the provided test cases. Grey-box fuzzing leverages lightweight program instrumentation instead of heavier program analysis to infer coverage during the fuzzing campaign itself, merging analysis and testing.

Coverage-guided grey-box fuzzing combines mutation-based input generation with program instrumentation to detect whenever a mutated input reaches new coverage. Program instrumentation tracks which areas of the code are executed, and the coverage profile is tied to specific inputs. Whenever an input mutation generates new coverage, it is added to the set of inputs for mutation. This approach is highly efficient due to the low-cost instrumentation but still results in broad program coverage. Coverage-guided fuzzing is the current de facto standard, with American fuzzy lop[1] and honggfuzz[2] as the most prominent implementations. These fuzzers leverage execution feedback to tailor input generation without requiring the analyst to have deep insight into the program structure.

A difficulty for input generation is finding the perfect balance between the need to discover new paths and the need to evaluate existing paths with different data. While the first increases coverage and explores new program areas, the latter explores already covered code through the use of different data. Existing metrics have a heavy control-flow focus as coverage measures how much of the

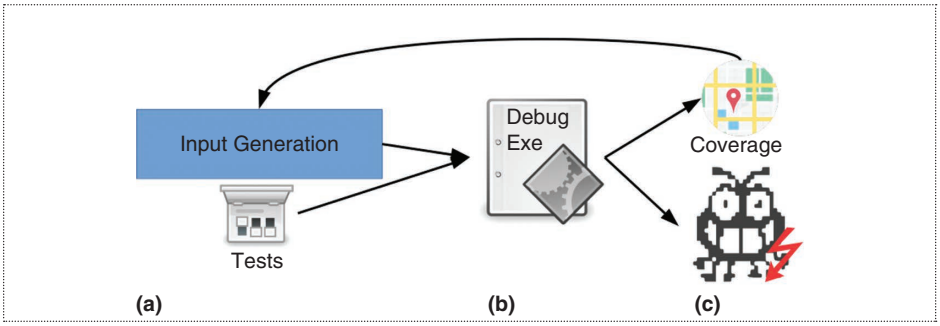

Figure 1. Fuzzing consists of an execution engine and an input-generation process that runs executables, which are often instrumented with explicit memory safety checks. (a) The input-generation mechanism (the blue box marked "Input Generation") may leverage existing test cases ("Tests") and execution coverage to generate new test inputs. For each discovered crash, the fuzzer provides a witness (the input that triggers the crash). (b) The execution engine. (c) A "bug" triggers the crash. The icon marked "Coverage" indicates input that has passed through the execution engine. Some of that input may pass through the input-generation process again. Arrows indicate the direction of process. Exe: executable.

program has already been explored. Data-flow coverage is only measured implicitly with inputs that execute the same paths but with different data values. A good input-generation mechanism balances the explicit goal of extending coverage with the implicit goal of rerunning the same input paths with different data.

Execution Engine

After the fuzzer generates test cases, it must execute them in a controlled environment and detect when a bug is triggered. The execution engine takes the fuzz input, executes the program under test, extracts runtime information, such as coverage, and detects crashes (Figure 1). Ideally, a program would terminate whenever a flaw is triggered. For example, an illegal pointer dereference on an unmapped memory page results in a segmentation fault, which terminates the program, allowing the executing engine to detect the flaw. Unfortunately, only a small subset of security violations will result in program crashes. Buffer overflows into adjacent memory locations, for instance, may never be detected at all or may only be detected later if the overwritten data are used. The challenge for this component of the fuzzing process is to efficiently enable the detection of security violations. For example, without instrumentation, only illegal pointer dereferences to unmapped memory, control-flow transfers to nonexecutable memory, division by zero, or similar violations will trigger an exception.

To detect security violations early, the tested program may be instrumented with additional software guards. It is especially tricky to find security violations through undefined behavior for code written in system languages. Sanitization analyzes and instruments the program during the compilation process to detect security violations. Address Sanitizer,[3] the most commonly used sanitizer, employs probability to detect spatial and temporal memory safety violations by placing red zones around allocated memory objects, keeping track of allocated memory, and checking memory accesses. Other LLVM-based sanitizers cover undefined behavior, uninitialized memory, or type safety violations through illegal casts.[4] Each sanitizer requires a certain type of instrumentation, which increases the performance cost. The use of sanitizers for fuzzing, therefore, has to be carefully evaluated as, on one hand, it makes error detection more likely but, on the other hand, it reduces fuzzing throughput.

The main goal of the execution engine is to conduct inputs as fast as possible.

The main goal of the execution engine is to conduct inputs as fast as possible. Several fuzzing optimizations, such as fork servers, persistent fuzzing, or special operating system (OS) primitives, reduce the time for each execution by adjusting system parameters. Fuzzing with a fork server executes the program up to a certain point and then forks new processes at that location for each new input. This allows the execution engine to skip over initialization code that would be the same for each execution. Persistent fuzzing allows the execution engine to reuse processes in a pool with new fuzzing input, resetting the state between executions. Different OS primitives for fuzzing reduce the cost of process creation by, for example, simplifying the creation of page tables and optimizing scheduling for short-lived processes.

Modern fuzzing is heavily optimized and focuses on efficiency, measured by the number of bugs found per unit of time. Sometimes fuzzing efficiency is implicitly measured by the number of crashes found per unit of time. However, crashes are not necessarily unique, and many crashes could point to the same bug. Disambiguating crashes to locate unique bugs is an important but challenging task. Multiple bugs may cause a program crash at the same location, whereas one input may trigger multiple bugs. A fuzzer must triage crashes conservatively

so that no true bugs are removed. Yet the triaging must not overload the analyst with redundant crashes.

Coverage Wall

In addition to massive parallelism, a key advantage of fuzzing compared to more heavyweight analysis techniques is its simplicity. However, due to this simplicity, fuzzing can get stuck in local minima in front of a coverage wall. When this happens, continuous input generation will not result in either additional crashes or new coverage. A common approach to circumvent the coverage wall is to extract seed values used for comparisons. These seed values are then used during the input-generation process. Orthogonally, a developer can comment out hard checks, such as CRC32 comparisons, or checks for magic values. Removing these noncritical checks from the program requires a knowledgeable developer to tailor fuzzing for each program.

Several recent extensions[5–8] try to bypass the coverage wall by automatically detecting when the fuzzer gets stuck and, then, if the problem is detected, leveraging an auxiliary analysis to either produce new inputs or modify the program. It is essential that this (sometimes heavyweight) analysis is executed only rarely, as alternating between analysis and fuzzing is costly and reduces fuzzing throughput.

Fuzzing libraries also face the challenge of experiencing low coverage during unguided fuzzing campaigns. Programs often call exported library functions in sequence, building up a complex state in the process. The library functions execute sanity checks and quickly detect an illegal or missing state. These checks make library fuzzing challenging, as the fuzzer is not aware of the dependencies between library functions. Existing approaches, such as LibFuzzer, require an analyst to prepare a test program that calls the library functions in a valid sequence to build up the necessary state to fuzz complex functions.

Evaluating Fuzzing

In theory, evaluating fuzzing is straightforward: in a given domain, if technique A finds more unique bugs than technique B, then technique A is superior to technique B. In practice, evaluating fuzzing is very difficult due to the randomness of the process and domain specialization (e.g., a fuzzer may only work for a certain type of bug or in a certain environment). Rerunning the same experiment with a different random seed may result in vastly different numbers of crashes, discovered bugs, and iterations. A recent overview of the state of the art[9] evaluated the common practices of recently published fuzzing techniques. The study's authors, after identifying common benchmarking mistakes when comparing different fuzzers, drew four observations from their findings:

- *Multiple executions*: A single execution is not enough due to the randomness in the fuzzing process. Input mutation relies on randomness to decide, according to the mutation strategy, where to mutate input and what to mutate. In a single run, one mechanism could discover more bugs simply by chance. To evaluate different mechanisms and measure noise, we require multiple trials and statistical tests.
- *Crash triaging*: Heuristics cannot be the only way to measure performance. For example, collecting crashing inputs or even stack bucketing is insufficient to identify unique bugs. Ground truth is needed to disambiguate crashing inputs and correctly count the number of discovered bugs. A benchmark suite with ground truth will help.
- *Seed justification*: The choice of seed must be documented, as different starting seeds provide vastly different starting configurations, and not all techniques cope equally well with different seed characteristics. Some mechanisms require a head start with seeds to execute reasonable functionality, while others are perfectly fine to start with empty inputs.
- *Reasonable execution time*: Fuzzing campaigns are generally executed over days or weeks. Comparing different mechanisms based on a few hours of execution time is not enough. A realistic evaluation, therefore, must run fuzzing campaigns for at least 24 h.

Rerunning the same experiment with a different random seed may result in vastly different numbers of crashes, discovered bugs, and iterations.

These recommendations make fuzzing evaluation more complex. Evaluating each mechanism now takes considerable time with experiments running multiple days to get enough statistical data for a fair and valid comparison. Unfortunately, such a thorough evaluation is required for a true comparison and analysis of factors leading to better fuzzing results.

A Call for Future Work

With the advent of coverage-guided grey-box fuzzing,[1,2] dynamic testing has seen a renaissance. Many new techniques that improve security testing have appeared. An important advantage of fuzzing is that each reported bug comes with

a witness that enables the deterministic reproduction of the bug. Sanitization, the process of instrumenting code with additional software guards, helps in discovering bugs closer to their source. Overall, security testing remains challenging, especially for libraries or complex code, such as kernels or large software systems. As fuzzers become more domain specific, an interesting challenge will be to make comparisons across different domains (e.g., comparing a grey-box kernel fuzzer for use-after-free vulnerabilities with a black-box protocol fuzzer). Given the significant recent improvements in fuzzing, exciting new results can be expected. Fuzzing will help make our systems more secure by finding bugs during the development of code before they can cause harm during deployment.

Fuzzing is a hot research area with researchers striving to improve input generation, reduce the impact of each execution on performance, better detect security violations, and push fuzzing to new domains, such as kernel fuzzing or hardware fuzzing. These efforts bring excitement to the field. ■

References

1. M. Zalewski, "American fuzzy lop (AFL)," 2013. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical_details.txt
2. R. Swiecki, "Honggfuzz," 2010. [Online]. Available: https://github.com/google/honggfuzz
3. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," presented at the 2012 USENIX Annual Technical Conference, Boston, MA. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany
4. Y. Jeon, P. Biswas, S. A. Carr, B. Lee, and M. Payer, "HexType: Efficient detection of type confusion errors for C++," in *Proc. 2017 ACM SIGSAC Conf. Computer and Communications Security*, pp. 2373–2387. doi: 10.1145/3133956.3134062.
5. N. Stephens et al., "Driller: Augmenting fuzzing through selective symbolic execution," in *Proc. ISOC Network and Security System Symp.*, 2016. doi: 10.14722/ndss.2016.23368.
6. S. Raway, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware evolutionary fuzzing," in *Proc. ISOC Network and Security System Symp.*, 2017. doi: 10.14722/ndss.2017.23404.
7. H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: Fuzzing by program transformation," in *Proc. 2018 IEEE Symp. Security and Privacy*. doi: 10.1109/SP.2018.00056.
8. I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," presented at the 27th USENIX Security Symp., Baltimore, MD, 2018.
9. G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proc. ACM Conf. Computer and Communications Security (CCS)*, 2018. doi: 10.1145/3243734.3243804.

Mathias Payer is a security researcher and an assistant professor at the EPFL School of Computer and Communication Sciences, leading the HexHive group. His research focuses on protecting applications in the presence of vulnerabilities, with a focus on memory corruption and type violations. Contact him at mathias.payer@nebelwelt.net.

Last Word *continued from p. 84*

Public-interest technology isn't new. Many organizations are working in this area, from older organizations, such as EFF and EPIC, to newer ones, such as Verified Voting and Access Now. Many academic classes and programs combine technology and public policy. My cybersecurity policy class at the Harvard Kennedy School is just one example. Media startups like The Markup are doing technology-driven journalism. There are even programs and initiatives related to public-interest technology inside for-profit corporations.

This might all seem like a lot, but it's really not. There aren't enough people doing it, there aren't enough people who know it needs to be done, and there aren't enough places to do it. We need to build a world where there is a viable career path for public-interest technologists.

There are many barriers. A report titled "A Pivotal Moment" (https://www.netgainpartnership.org/s/pivotalmoment.pdf) includes this quote:

> While we cite individual instances of visionary leadership and successful deployment of technology skill for the public interest, there was