SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing

Mingzhe Wang[1], Jie Liang[1], Yuanliang Chen[1], Yu Jiang[1*],Xun Jiao[2], Han Liu[1], Xibin Zhao[1], Jiaguang Sun[1]
School of Software, Tsinghua University, KLISS, Beijing, China[1]
Department of Computer Science, University of California, San Diego, USA[2]

ABSTRACT

Mutation-based fuzzing is a widely used software testing technique for bug and vulnerability detection, and the testing performance is greatly affected by the quality of initial seeds and the effectiveness of mutation strategy. In this paper, we present SAFL[1], an efficient fuzzing testing tool augmented with qualified seed generation and efficient coverage-directed mutation. First, symbolic execution is used in a lightweight approach to generate qualified initial seeds. Valuable explore directions are learned from the seeds, thus the later fuzzing process can reach deep paths in program state space earlier and easier. Moreover, we implement a fair and fast coverage-directed mutation algorithm. It helps the fuzzing process to exercise rare and deep paths with higher probability. We implement SAFL based on KLEE and AFL and conduct thoroughly repeated evaluations on real-world program benchmarks against state-of-the-art versions of AFL. After 24 hours, compared to AFL and AFLFast, it discovers 214% and 133% more unique crashes, covers 109% and 63% more paths and achieves 279% and 180% more covered branches.

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

KEYWORDS

Symbolic Execution, Greybox Fuzzing, Software Testing

1 INTRODUCTION

Fuzzing has been identified as one of the most effective technologies for bug and vulnerability detection in real-world software systems. The key idea is to generate random inputs to exercise as many program paths as possible, then execute the program using such inputs to catch crashes [8]. Many fuzzing tools such as libFuzzer [1] and American Fuzzy Lop (AFL) [12] were developed and have caught a large number of dangerous bugs and security vulnerabilities in widespread systems. They are easy to use and have already gained great success in both industrial practice and academic research.

However, due to the characteristics of seed random mutation, it is not easy to ensure the coverage of complex software systems. To improve the fuzzing performance, some researchers try to use coverage information to direct mutation, such as AFLFast [4] and AFLgo [3]. Meanwhile, others try to integrate fuzzers with program analysis techniques like symbolic execution, such as Driller [11] and Mayhem [6]. But with the increasing complexity of software systems, it is still hard to reach deep places in a program [13] and would miss the bugs concealed in deep paths.

Basing on the performance of existing fuzzers, we observe that there are two main obstacles for efficient fuzzing.

(1) **Poor initial seed quality.** Initial seeds are hugely important as they determine the initial direction of fuzzing. Efficient seeds are hard to write manually and random seeds cannot ensure coverage, thus generating directed initial seeds automatically is badly needed.
(2) **Inefficient Mutation algorithm.** Mutation algorithm determines the speed and depth of coverage. How to efficiently mutate the initial seeds to get valuable seeds exploring new paths is an enormous challenge.

In this paper we present SAFL, an efficient fuzzer for C/C++ programs. Augmented with qualified seed generation and efficient coverage-directed mutation, it grapples with the two obstacles. SAFL employs symbolic execution in a lightweight approach to generate initial seeds which can get reasonable fuzzing direction. Also, SAFL uses a fair and fast fuzzing algorithm. It classifies seeds according to path coverage, then mutate them in different ways and different weights. With this algorithm, SAFL is able to explore deep paths as many and fast as possible.

For evaluation, we conduct thoroughly repeated evaluations on real-world program benchmarks against state-of-the-art versions of AFL such as AFLFast. The results demonstrate that SAFL has a better performance in coverage and bugs detection. More specially, after 24 hours, for the 10 cases in total, compared to AFL and AFLFast, it discovers 214% and 133% more unique crashes, covers 109% and 63% more paths and achieves 279% and 180% more covered branches. For each case, the improvement remains.

2 RELATED WORK

Several approaches have been proposed to improve fuzzing [2–4, 12]. Mutation-based fuzzers generate new inputs by mutating the seed inputs with the feedback of coverage information. Several recent works put the information into better use than AFL [12], the originator. For example, AFLFast [4] gives more mutation time to valuable seeds which exercise low-frequency paths. AFLgo [3] proposes a simulated annealing-based power schedule to reach specific program locations. Generation-based fuzzers leverage the knowledge of input format model, which is especially useful for fuzzing complex formats. For example, Glade [2] improves the inference of context-free grammar by consulting the program.

[1]The tool can be downloaded at https://github.com/hghwng/tools-date
[*] Yu Jiang is the correspondence author, and the work is supported by Tsinghua-Huawei Collabration Project YBN2017010031

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183494

Symbolic execution is also applied to optimize fuzzing [5, 7]. Symbolic execution tools collect the constraints of a program while simulating the execution, therefore they can generate a concrete input for any given feasible paths as long as the solver works. For example, S2E [7], namely a symbolic QEMU, improves environment modeling by simulating the whole operating system altogether. KLEE [5], which runs LLVM bitcode, is another lightweight and robust tool. BitBlaze [10] reasons about a single execution path at a time and picks new paths to explore iteratively. Recently, researchers tried to combine mutation based fuzzing with symbolic execution [6, 9, 11]. For example, Driller [11] switches to symbolic executor when AFL gets stuck, and switches back to the efficient fuzzer as soon as the complex constraint is bypassed.

Main Difference. SAFL also takes advantage of mutation-based fuzzing and symbolic execution. Compared to traditional mutation-based fuzzers, SAFL exercises more rare paths quickly because of its high-quality initial seeds and novel guided mutation algorithm. Compared to symbolic tools, SAFL is more efficient as SAFL won't get stuck at complex constraints which solvers are unable to solve. Although the components of Driller and SAFL look similar, there's an essential difference: with symbolic execution, SAFL helps the fuzzer with high-quality seeds, while Driller and Mayhem help by overcoming complex checks. Within Driller, even if the check is bypassed, the fuzzer is likely to get stuck again as the mutation algorithm invalidates the delicate byte sequence generated by the smart symbolic executor. The oscillating is costly. The lightweight design of SAFL is free of such oscillation. SAFL only runs symbolic execution in one pass. The evolved mutator is able to detect delicate parts of a byte sequence by using the result of symbolic execution effectively.

3 SAFL DESIGN

As presented in Figure 1, SAFL consists of three components. The `Toolchain` component builds two versions of tailored binary for symbolic execution and fuzzing. The symbolic version is passed to the lightweight `Symbolic Executor` component to generate qualified seeds. With high-quality seeds and hardened binary available, the `Fuzzer` component implements the guided fuzzing algorithm, runs the testing and produces the bug report efficiently.

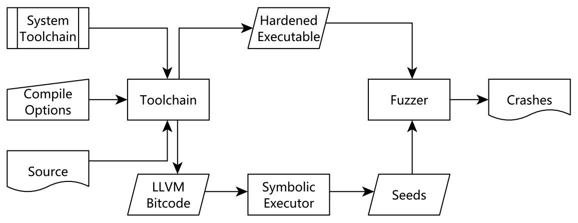

Figure 1: The interaction of SAFL components

3.1 Toolchain

Different execution mechanisms of symbolic execution for seed generation and concrete execution for fuzzing require different sets of compiler options. The `Toolchain` component is built to accomplish the following four tasks:

(1) Build LLVM bitcode for symbolic execution. Symbolic execution favors a higher-level program representation. High-level representation carries more information per instruction, thus the program representation contains fewer instructions. In other words, symbolic execution runs faster. More importantly, higher-level representation carries more semantics than machine code does. Symbolic execution benefits from the enriched semantics and more bugs can be identified.
(2) Leverage instrumentation and binary hardening techniques to build hardened binaries for dynamic execution. Dynamic execution works best with hardened binaries. Many types of vulnerabilities such as heap overflow usually won't bite on the spot. They are hard to detect when running a vanilla binary because the target program runs too short for the latent bug to take effect.
(3) Maintain subtle equilibrium of compiler optimization to prevent over-optimization and under-optimization. Generally, optimization speeds up the execution, which is beneficial to compute-intensive fuzzing. Loop unrolling is especially useful, which helps to track the execution time of simple loops. However, optimization does backfire sometimes. Many optimizations exploit undefined behaviors to aggressively optimize the program. For example, an overwrite to a local array can be optimized out.
(4) Simplify the build procedure. A large codebase is usually accompanied by a complex build procedure. The complexity hinders the adoption of fuzzing testing. The `Toolchain` component simplifies the procedure by intercepting the build command invocation and automatically replace it with the optimal options above.

3.2 Seed Generation

SAFL uses KLEE as the kernel of the `Symbolic Executor` component, to generate high-quality seeds. LLVM bitcode is passed to the symbolic execution virtual machine to explore the typical program states. SAFL further tweaks the symbolic execution to match the purpose of seed generation:

(1) Environment simulation. Libc is required by almost all programs and the invocation of utility functions inside libc such as `strcmp` is ubiquitous. SAFL ships KLEE built with uclibc, a tiny libc implementation originally developed for embedded use. This enables the symbolic execution of utility functions and basic POSIX API simulation.
(2) Time limitation. A program executes several magnitudes slower inside a symbolic virtual machine [7], so it's necessary to limit the default exhaustive exploration of KLEE. Because we do not need to explore all paths and only execute one pass to favor the efficient fuzzing phase, the execution time of symbolic execution is limited to 20 minutes.
(3) Seed selection. The symbolic execution phase of SAFL is targeted at accelerating the fuzz phase by reaching deep program states first. But KLEE tends to create too many initial seeds that slow down the fuzzing. To avoid redundant seeds, only the concrete inputs that result in new coverage are kept. A conversion module is developed to connect KLEE and `Fuzzer` component together.

3.3 Guided Fuzz

SAFL avoids unfruitful runs as much as possible with the novel guided fuzzing algorithm implemented in `Fuzzer`. It avoids three pitfalls to increase the execution speed and path depth.

(1) Unfair seed selection. It's crucial to choose seed right because once a low-quality seed is chosen, efficient mutation algorithm provides a little remedy. The seed selection algorithm of the original AFL favors seeds that have unique coverage and run fast. It doesn't take the branch rarity into account, which is unfair for the rare branches. SAFL fixes the problem by only picking seeds exercising branches which are rare enough.

(2) Invalid seed mutation. Complex checks are difficult to bypass, thus once the exact byte sequence is generated, the mutator had better keep the magic byte sequence intact. AFL is unaware of the magic sequence, putting lots of energy on mutating the magic sequence in vain. SAFL tries to discover the way where a byte can be mutated: deletable, overwritable, and insertable. This strategy prevents mutation that outputs seeds drifting from the rare branch.

(3) Slow seed mutation. While the crude heuristic works efficiently, it puts too much constraint on the mutation. When the algorithm above gets stuck, SAFL falls back to the original AFL mutation algorithm, but the execution time is shortened. The execution time scheduler of AFL has the similar weakness of seed selection algorithm. When assigning mutation time of each seed, AFL misses the metric of branch rarity. It's slow when computing resource is wasted on a less-valued seed. Drawing upon the work of AFLFast, SAFL uses power schedule to decide the execution time.

4 SAFL IMPLEMENTATION

As Figure 2 illustrates, SAFL is divided into three layers, to provide developer-friendly interface and cross-distribution compatibility.

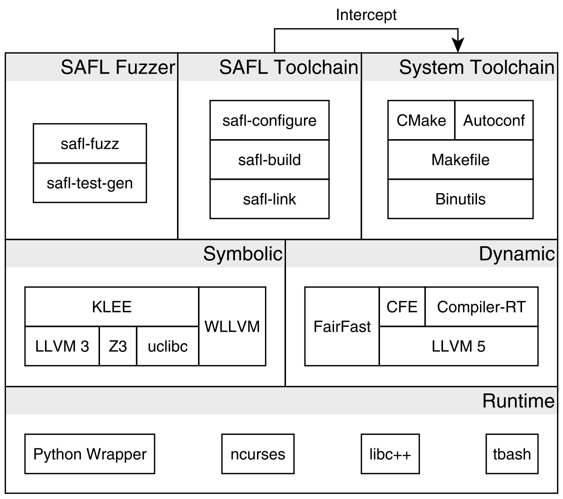

Figure 2: Layers of SAFL

- Interface Layer. The interface layer encapsulates the complexity of internal tools. The toolchain part intercepts and replaces the system toolchain invocation. The fuzzer part prepares the environment and optimal options for execution. It also connects the symbolic and dynamic executor, by exporting the results of symbolic execution to seeds for fuzzing.
- Tool Layer. The tool layer consists of the core tools for the build and execution stage. The symbolic part bundles KLEE, LLVM, Z3 and uclibc for execution, and WLLVM for LLVM bitcode collection. The dynamic counterpart bundles LLVM to build the hardened binary and our improved fuzz engine named FairFast for random testing.
- Runtime support Layer. The runtime support layer contains shared libraries used by upper layers. This enables compatibility across Linux distributions, as long as the kernel version is recent enough.

To use SAFL, little modification to normal build procedure is required. The developer replaces each build command with the corresponding SAFL toolchain command as presented in Table 1. A simple analysis is performed on the command line options by SAFL. Depending on the build stage, the command line is replaced to build LLVM bitcode or sanitizer-hardened executable. For example, when `safl-build g++` is invoked in dynamic mode, SAFL injects `-fsanitize=address` for address sanitizer and `-Xclang -load` for an additional LLVM pass to enable coverage reporting behind the scenes. When invoked in symbolic mode, SAFL redirects the invocation to WLLVM to collect bitcode.

Table 1: SAFL and normal build procedure

	SAFL	Normal build
Prepare	safl-prepare	
Configure	safl-configure ./configure	./configure
Build lib	safl-build make -j	make -j
Build driver	safl-build g++ target.cc -c	g++ target.cc -c
Link	safl-link target.o lib.a	g++ -o app target.o lib.a
Generate seed	safl-test-gen ...	
Fuzz	safl-fuzz app.bc	./app

5 EVALUATION

We evaluate SAFL on 10 different real-world program benchmarks selected from Google's fuzzer-test-suite for libFuzzer, including boringssl, re2, guetzli, libxml2, lcms, pcre2, proj4, libssh, libarchive and c-ares. We compare SAFL with AFL and AFLFast. libFuzzer is not included because the fuzzing engine will shut down when it finds a crash, while other tools continue fuzzing unless manually terminated. AFLgo is not included either because it targets fuzzing of revised code. In order to demonstrate the contribution of seed generation and guided fuzzing respectively, a weaker version called SAFL$^-$ is developed. It has the symbolic executor disabled, while our fuzzing engine, FairFast, is kept.

We run each tool on ASan (address sanitizer) hardened binaries using a single core for 24 hours. The machine has 36 cores (E5-2630 v3 @ 2.40GHz), 128 GB of main memory, and Ubuntu 16.04 (AMD64) as the host OS.

Table 2 shows the number of paths covered by each tool and Table 3 shows the number of covered branches. From the third column and the fourth column of the two tables, we can observe that the optimized fuzzing engine in SAFL$^-$ covers 26.8% more paths and 3.7% more branches compared to AFLFast. From the fourth column and the fifth column of the two tables, we can observe that the qualified seed generation further contribute to the coverage. SAFL increases covered paths by 28.6% and covered branches by 169% compared to SAFL$^-$. Combining lightweight symbolic execution and enhanced guided fuzzing, the increasing coverage for paths and branches is 63% and 180% respectively, compared to the AFLFast basing on the most related AFL version.

From these comparisons and statistics, we can conclude that the guided fuzzing strategy and symbolic-execution-based seed generation are effective, and they help SAFL to exercise more and deeper paths in the program with a higher probability.

Table 2: Number of paths

Project	AFL	AFLFast	SAFL$^-$	SAFL
boringssl-2016-02-12	693	579	789	988
re2-2014-12-09	1788	1931	2623	3169
c-ares-CVE-2016-5180	37	35	37	42
libssh-2017-1272	19	20	20	28
guetzli-2017-3-30	368	689	690	1329
libxml2-v2.9.2	1335	2235	2494	4276
lcms-2017-03-21	333	202	249	335
pcre2-10.00	9949	12387	16229	18883
libarchive-2017-01-04	648	1448	1660	2093
proj4-2017-08-14	84	84	86	850
Total	15254	19610	24877	31993

Table 3: Number of branches

Project	AFL	AFLFast	SAFL$^-$	SAFL
boringssl-2016-02-12	3215	2651	3504	10231
re2-2014-12-09	13813	16245	16257	30593
c-ares-CVE-2016-5180	103	105	105	277
libssh-2017-1272	595	602	603	1592
guetzli-2017-3-30	1112	2123	2174	8982
libxml2-v2.9.2	6726	9546	10037	40933
lcms-2017-03-21	1787	2060	2529	6587
pcre2-10.00	21798	31821	31939	75800
libarchive-2017-01-04	2706	5123	5786	16296
proj4-2017-08-14	258	263	263	6281
Total	52113	70539	73197	197572

Table 4: Number of crashes

Project	AFL	AFLFast	SAFL$^-$	SAFL
boringssl-2016-02-12	0	0	0	0
re2-2014-12-09	0	0	1	1
c-ares-CVE-2016-5180	4	4	4	4
libssh-2017-1272	5	6	7	7
guetzli-2017-3-30	0	0	0	0
libxml2-v2.9.2	0	0	0	5
lcms-2017-03-21	3	8	10	10
pcre2-10.00	68	90	107	103
libarchive-2017-01-04	1	1	1	4
proj4-2017-08-14	0	0	0	121
Total	81	109	130	255

The number of the unique crash found by each tool in 24 hours is presented in Table 4. This table shows that AFL and AFLFast find crashes in 5 programs, SAFL$^-$ finds crashes in 6 programs, and SAFL finds crashes in 8 programs.

We check the unique crashes in Table 4 for each program, and identify that the unique crashes are caused by one bug — AFL defines unique crashes by unique paths, yet different paths can trigger the same bug. Therefore, the number of unique crashes can be considered as the probability to trigger the bug. We can observe that the optimized fuzzing engine FairFast Fuzzer in SAFL$^-$ increases 19.3% bug detection probability compared to AFLFast. SAFL with qualified seed generation increases 96.2% bug detection probability compared to SAFL$^-$. Combining the techniques above, SAFL improves the bug detection probability of AFLFast by 133.2%.

From these comparisons and statistics, we can conclude that the guided fuzzing strategy and symbolic-execution-based seed generation help SAFL hunt more bugs and trigger each bug with a higher probability.

6 CONCLUSION

In this paper, we present SAFL, an efficient fuzzer for C/C++ programs. The efficiency is mainly embodied in two aspects, namely efficient initial seeds and guided fuzzing algorithm. Initial qualified seeds are generated by lightweight symbolic execution; enhanced guided fuzzing algorithm can rapidly cover the rare path while avoiding quick local convergence. We evaluate the performance of SAFL by fuzzing 10 different real-word programs from Google's libFuzzer test suite. Compared with other tools, SAFL can expose a larger number of unique crashes, exercise more paths and explore deeper states. In the future, we plan to adapt SAFL for systems with a complex architecture, such as database or operating system kernel.

REFERENCES

[1] 2017. libFuzzer in Chrome. https://chromium.googlesource.com/chromium/src/+/master/testing/libfuzzer/README.md. (2017). [Online; accessed 12-November-2017].

[2] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 95–110.

[3] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1032–1043.

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[6] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing mayhem on binary code. In *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 380–394.

[7] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* 46, 3 (2011), 265–278.

[8] Joe W Duran and Simeon Ntafos. 1981. A report on random testing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 179–183.

[9] Julian Fietkau and Bhargava Shastry. 2017. KleeFL - Seeding Fuzzers With Symbolic Execution. Poster presented at USENIX Security'17, Vancouver, BC, Canada, TU Berlin.

[10] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. *Information systems security* (2008), 1–25.

[11] 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.. In *NDSS*, Vol. 16. 1–16.

[12] Michal Zalewski. 2015. American fuzzy lop. (2015).

[13] Michal Zalewski. 2015. Symbolic execution in vuln research. https://lcamtuf.blogspot.com/2015/02/symbolic-execution-in-vuln-research.html. (2015). [Online; accessed 11-November-2017].