Evaluating the Fork-Awareness of Coverage-Guided Fuzzers

Marcello Maugeri[1][a], Cristian Daniele[2][b], Giampaolo Bella[1][c] and Erik Poll[2][d]
[1]*Department of Maths and Computer Science, University of Catania, Catania, Italy*
[2]*Department of Digital Security, Radboud University, Nijmegen, The Netherlands*
marcello.maugeri@phd.unict.it, cristian.daniele@ru.nl, giampaolo.bella@unict.it, erikpoll@cs.ru.nl

Keywords: Fuzzing, Fork, Security Testing, Software Security.

Abstract: Fuzz testing (or fuzzing) is an effective technique used to find security vulnerabilities. It consists of feeding a software under test with malformed inputs, waiting for a weird system behaviour (often a crash of the system). Over the years, different approaches have been developed, and among the most popular lies the coverage-based one. It relies on the instrumentation of the system to generate inputs able to cover as much code as possible. The success of this approach is also due to its usability as fuzzing techniques research approaches that do not require (or only partial require) human interactions. Despite the efforts, devising a fully-automated fuzzer still seems to be a challenging task. Target systems may be very complex; they may integrate cryptographic primitives, compute and verify check-sums and employ *forks* to enhance the system security, achieve better performances or *manage different connections at the same time*. This paper introduces the *fork-awareness* property to express the fuzzer ability to manage systems using forks. This property is leveraged to evaluate 14 of the most widely coverage-guided fuzzers and highlight how current fuzzers are ineffective against systems using *forks*.

1 INTRODUCTION

In the last years, plenty of fuzzers have been developed to deal with sophisticated software and nowadays it is extremely common that network systems employ forks to deal with different connections at the same time. This leads to 1) the need to devise accurate and ad-hoc fuzzers and 2) the need to evaluate these fuzzer according to their ability to cope with such advanced systems.

Unfortunately, as pointed out in (Hazimeh et al., 2020), it is not easy to benchmark all of them since the fuzzers are very different from each other. Metzman et al. faced this problem by devising *FuzzBench*(Metzman et al., 2021), an open-source service for the evaluations of stateless fuzzers. Later, Natella and Pham presented *ProFuzzBench*(Natella and Pham, 2021), which similarly to *FuzzBench* provides a service to evaluate stateful fuzzers.

Although FuzzBench includes a sample of real word programs and ProFuzzBench includes different network systems (i.e. systems that often employ forks to deal with multiple connections(Tanenbaum, 2009)), they do not evaluate the ability of the fuzzers to cope with programs that use forks. Despite *forks* representing the only way to create a new process (Tanenbaum, 2009), experimental results have shown that current fuzzers cannot deal with forked processes.

The existing approach merely relies on code modifications to remove the forks. Unfortunately, this approach goes against the willingness to reduce manual work and improve automation during a fuzzing campaign.(Boehme et al., 2021).

In this work, we explore and classify the limitations current fuzzers exhibit in front of forking programs.

In summary, this paper:

1. devises a novel property capturing the ability of fuzzers to deal with forks appropriately;
2. evaluates 14 coverage-guided fuzzers based on this property;
3. proposes possible improvements to the current state-of-the-art and future directions.

The paper is organised as follows. Section 2 describes the relevant background, Section 3 presents our contributions to knowledge, Section 4 shows the existing approaches that try to cope with the fork problem and,

[a] https://orcid.org/0000-0002-6585-5494
[b] https://orcid.org/0000-0001-7435-4176
[c] https://orcid.org/0000-0002-7615-8643
[d] https://orcid.org/0000-0003-4635-187X

eventually, Section 5 discuss the results and propose possible future directions.

2 BACKGROUND

2.1 Fuzz Testing

Fuzzing is an automated testing technique pioneered by Miller et al.(Miller et al., 1990) in 1990 to test UNIX utilities. As outlined in Figure 1, coverage-guided fuzzing is composed at least of seed selection, input generation and system execution.

1) Seeds Selection. The user must provide some input messages (seeds) representative of some usual inputs for the system.

2) Input Generation. The core of every fuzzer is the generation of slightly malformed input messages to forward to the software under test. A fuzzer is as efficient as the generated inputs are able to break the system. According to the approach used to generate the messages, the fuzzers may be classified into:

- *dumb:* generate random strings (as the first fuzzer (Miller et al., 1995) did);
- *dumb mutational:* blindly mutate seed messages provided by the user;
- *grammar-based:* leverage the grammar of the system to craft the input messages;
- *smart mutational:* (often called *evolutionary*) require a sample of inputs and leverage *feedback mechanisms* to craft system-tailored messages. An example of feedback mechanisms is the code coverage feedback, explored in Section 2.2.

3) System Execution. Each execution of the fuzzer involves three components:

- *Bugs detector:* it reports eventual bugs. The majority of the bugs detectors only report crashes, however for many systems, also a weird deviation from the happy flow of the protocol may represent significant security issues;
- *Hangs detector:* it detects program execution hangs;
- *Code coverage detector:* as further explained in Section 2.2, the code coverage represents one of the feedbacks the fuzzer leverages to improve the quality of the input messages.

2.2 Coverage-Guided Fuzzing

Smart mutational fuzzers use feedback mechanisms to steer the generation of the messages. Different types of feedback mechanisms exist (Shahid et al., 2011), and often different terms are used to express the same idea. To avoid further noise, in this work we use the term *code coverage* to express the lines of code that are reached by a specific message.

Code coverage fuzzers need to recompile the code with ad-hoc compilers (e.g. the AFL compiler) to instrument the code and obtain run-time information.

AFL (Zalewski, 2017), for example, instruments the code to fill a bitmap that represents the lines of the code covered by the inputs.

Later, it uses this bitmap to assign a higher score to messages able to explore previously unseen lines of code.

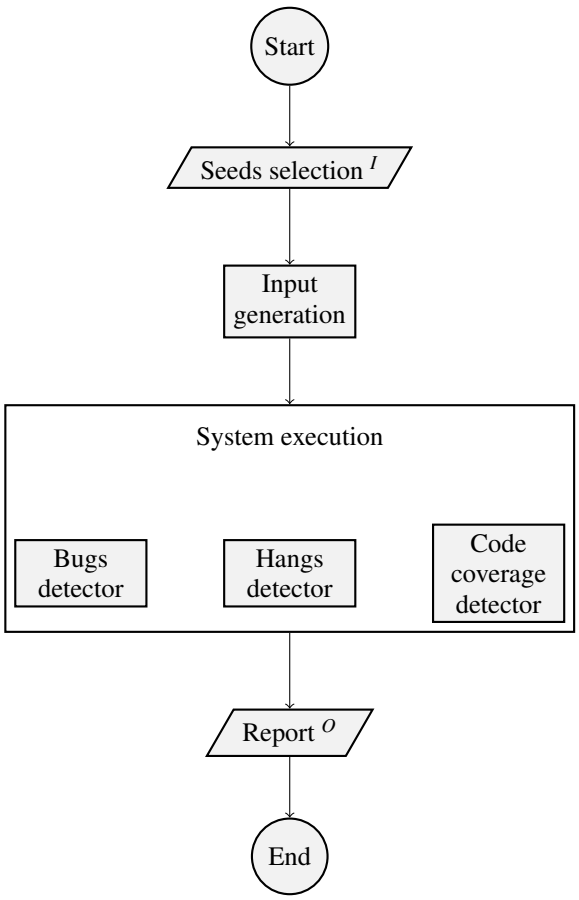

Figure 1: Coverage-guided fuzzing process.

2.3 Inter-Process Communication

Operating systems provide *system calls* to perform different tasks (e.g. writing and reading files, accessing hardware services, creating and executing new processes). On UNIX systems, new processes are created by using the *fork system call*(Tanenbaum, 2009). In short, the first process, called *parent process*, gen-

erates a clone, called *child process*, that is an exact copy of the parent process. After the fork, file descriptors and registers are duplicated, thus a change in one of the processes does not affect the other one. Also, the parent and child process will follow *separate execution paths*.

3 OUR CONTRIBUTION

This paper aims to understand how the state-of-the-art coverage-guided fuzzers deal with software under tests containing forks.

It was not obvious to come up with a way to compare and contrast the various tools. We devised a novel property, the *fork awareness*, that must be satisfied when a fuzzer deals with forks effectively and efficiently. As we shall see below, fork awareness rests upon three aspects representing the ability to deal with child processes.

Also, we evaluate the novel property over the most widely used fuzzers from two benchmark frameworks, reaching a total of 14 evaluated tools, 11 drawn from FuzzBench and 3 from ProFuzzBench.

3.1 Fork-Awareness

Abstractly, fork awareness insists that every fuzzer should address the child process as the parent one. During the system execution, the system monitor should detect bugs or hangs regardless of their location and the coverage should be measured also in the child process. This is formalised through Definition 1.

Definition 1. A coverage-guided fuzzer is *fork-aware* if it can detect bugs and hangs and measure coverage in the same way for both the child and the parent's branch.

The three aspects in this definition are called:

[C.1] Child Bugs Detection: any anomaly is reported also if it occurs in child processes;

[C.2] Child Hangs Detection: any infinite hang is reported also if it occurs in child processes;

[C.3] Child Code Coverage: code coverage is measured also for child processes.

3.2 Example Challenges

We wrote three simple C programs to use as challenges for the fuzzers, namely to test whether the fuzzers satisfy the aspects given above.

a) *Bugs Detection Challenge:*

```
if(fork()==0){ //Child process
  raise(SIGSEGV); //Simulated crash
} else { //Parent process
  wait(NULL); //Waiting child
              //termination
}
```

The snippet sends a *SIGSEGV* signal to simulate a bug in the child process. This signal is used to report a segmentation fault, i.e. a memory access violation, which is common in programs written in low-level languages. The fuzzer must detect this bug also after the parent's termination.

b) *Hangs Detection Challenge:*

```
if(fork()==0){ //Child process
  while(1){ ; } //Simulation of
                //blocking code
}
```

The snippet simulates an infinite loop in the child process. The fuzzers must report processes still in execution after the loop and must kill child processes at the end of the fuzzing campaign, avoiding pending process executions.

c) *Code Coverage Challenge:*

```
pid_t pid = fork();
if(pid==0){ //Child process
  if(data %2 == 0){ do_something(); }
     else { do_something(); }
  if(data %3 == 0){ do_something(); }
     else { do_something(); }
  if(data %5 == 0){ do_something(); }
     else { do_something(); }
  if(data %7 == 0){ do_something(); }
     else { do_something(); }
   }
else { //Parent process
     wait(NULL); //Waiting child
                 //termination
     }
```

This snippet simulates a child with several branches. A fuzzer must cover and consider every child's branches.

We run the 14 fuzzers over these challenges and organised the results in Table 1. We noticed that none of the fuzzers succeeded through all three challenges.

3.3 Testbed

We decided to analyse only the coverage-guided fuzzers present in FuzzBench (Metzman et al., 2021) and ProFuzzBench (Natella and Pham, 2021) even though the property applies to every coverage-guided fuzzer. All fuzzers were executed on an *Ubuntu 20.04* server machine and all our source codes are freely

Table 1: Coverage guided fuzzers evaluation.

Fuzzer	Based on	Monitor technique	Bugs Detection (C1)	Hangs Detection (C2)	Code coverage (C3)
AFL(Zalewski, 2017)	-	POSIX signals	×	×	✓
AFL++(Fioraldi et al., 2020)	AFL	POSIX signals	×	×	✓
AFLFast(Bohme et al., 2017)	AFL	POSIX signals	×	×	✓
AFLSmart(Pham et al., 2021)	AFL	POSIX signals	×	×	✓
Eclipser(Choi et al., 2019)	AFL	POSIX signals	×	×	✓
FairFuzz(Lemieux and Sen, 2018)	AFL	POSIX signals	×	×	✓
lafintel(Besler and Frederic, 2016)	AFL	POSIX signals	×	×	✓
AFLnwe[1]	AFL	POSIX signals	×	×	✓
AFLNet(Pham et al., 2020)	AFL	POSIX signals	×	×	✓
MOpt-AFL(Lyu et al., 2019)	AFL	POSIX signals	×	×	✓
StateAFL(Natella, 2022)	- AFL - AFLNet	POSIX signals	×	×	✓
LibFuzzer[2]	-	- UBSAN - ASAN - MSAN	✓	×	✓
Entropic(Bohme et al., 2020)	LibFuzzer	- UBSAN - ASAN - MSAN	✓	×	✓
Honggfuzz[3]	-	ptrace (Linux)	✓	×	✓

available online[4] so that our experiments are fully reproducible.

3.4 Fuzzers Evaluation

We run all selected fuzzers against our three example challenges. Table 1 summarises our findings.

All the fuzzers based on AFL use *POSIX signals* and a bitmap respectively to report bugs and keep track of the code coverage.

As shown in the Table 1, while the bitmaps are able to keep track of the child's code coverage, bugs triggered in the child's processes are not detected since AFL catches signals from the main process only, as pointed out in the documentation[5]. The only fuzzers able to detect bugs in the child process are LibFuzzer[6], Entropic(Bohme et al., 2020) and Honggfuzz[7], as discussed in more detail below:

- *LibFuzzer* [8] and *Entropic*(Bohme et al., 2020) employ a set of sanitizers[9] to report bugs. These mechanisms make the fuzzers able to find the bug in Challenge 1 and measure the different code paths in Challenge 3, thereby satisfying challenges C.1 and C.3, as seen above. Unfortunately, challenge C.2 is not satisfied since the fuzzer cannot detect hangs in the child process.
- *Honggfuzz* supports different software/hardware feedback mechanisms and a low-level interface to monitor targets. When executed on Linux machines, Honggfuzz uses the *ptrace* system call to manage processes. This mechanism allows the fuzzer to capture a wide range of signals. As shown in Table 1, the use of ptrace (along with the *SanitizerCoverage*) allows the fuzzer to detect bugs and to consider coverage also in the child process. Unfortunately, neither this mechanism is able to detect hangs in the child process.

In summary, while all selected fuzzers detect the code coverage (C3), none detect hangs (C2) and only a few detect bugs (C1) in the child process. The evaluation underlines that:

- *Loops detection challenge* is the most difficult because fuzzers do not wait for all the child processes but only for the main one;
- *Code coverage challenge* is the easiest because the instrumentation allows measuring coverage from the execution, regardless of the process involved;
- *Bug detection challenge* depends on the technique

[4] https://github.com/marcellomaugeri/forks-break-afl

[5] https://github.com/google/AFL/blob/master/README.md

[6] https://llvm.org/docs/LibFuzzer.html

[7] https://honggfuzz.dev/

[8] https://llvm.org/docs/LibFuzzer.html

[9] AddressSanitizer, UndefinedBehaviorSanitizer and MemorySanitizer

used to observe bugs, as well as the use of sanitisers.

We interpret this general outcome as a clear call for future research and developments.

4 EXISTING SOLUTIONS

Nowadays the only solutions to fuzz programs that use forks are manually modifying the code or breaking the multi-process nature of the system (by employing tools like defork[10]) in order to get rid of the forks.

Unfortunately, making modifications to the code, as pointed out in the AFLNet documentation [11], to remove all the forks is a challenging and error-prone task and break the multi-process nature of the system often leads to weird system behaviours. The only solution, therefore, remains to modify the fuzzers.

5 CONCLUSIONS

This paper analyses the fork awareness of the coverage-guided fuzzers using three different aspects. The analysis conducted on 14 well-known fuzzers highlights that while is it clear how important is to handle multi-process programs, the majority of the fuzzers overlook the problem. 11 of 14 fuzzers are not able to detect bugs in the child process. The intuition behind these outcomes is related to the way these fuzzers detect bugs. All the AFL-derived fuzzers use signals (SIGSEGV, SIGABRT, etc) to detect bugs and this mechanism misses bugs in child processes. We noticed that dealing with forks is not the only problem and other issues may be related to the IPC scheduling. For example, the IPC may influence the success of the fuzzing process since some bugs may be triggered only after a specific process schedule and only after access to a particular cell of memory. We believe this paper represents a first step towards the devising of fuzzers aware of the eventual multiprocess nature of the software. The first step to achieve this goal might be the implementation of a *loop detector* at an early stage, e.g. by leveraging a dynamic library to keep track of all process identifiers of forked processes. To summarise, this work not only provides the first concrete way to evaluate the fuzzers according to their fork awareness but sheds light for the first time on a class of problems that have been ignored until now, showing interesting future directions.

[10] https://github.com/zardus/preeny/blob/master/src/defork.c

[11] https://github.com/aflnet/aflnet

REFERENCES

Besler and Frederic (2016). Circumventing fuzzing roadblocks with compiler transformations. https://lafintel.wordpress.com.

Boehme, M., Cadar, C., and Roychoudhury, A. (2021). Fuzzing: Challenges and reflections. *IEEE Softw.*, 38(3):79–86.

Bohme, M., Manes, V. J., and Cha, S. K. (2020). Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 678–689.

Bohme, M., Pham, V.-T., and Roychoudhury, A. (2017). Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506.

Choi, J., Jang, J., Han, C., and Cha, S. K. (2019). Grey-box concolic testing on binary code. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 736–747. IEEE.

Fioraldi, A., Maier, D., Eißfeldt, H., and Heuse, M. (2020). Afl++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

Hazimeh, A., Herrera, A., and Payer, M. (2020). Magma: A ground-truth fuzzing benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29.

Lemieux, C. and Sen, K. (2018). Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 475–485.

Lyu, C., Ji, S., Zhang, C., Li, Y., Lee, W.-H., Song, Y., and Beyah, R. (2019). {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966.

Metzman, J., Szekeres, L., Maurice Romain Simon, L., Trevelin Sprabery, R., and Arya, A. (2021). FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 1393–1403, New York, NY, USA. Association for Computing Machinery.

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44.

Miller, B. P., Koski, D., Lee, C. P., Maganty, V., Murthy, R., Natarajan, A., and Steidl, J. (1995). Fuzz revisited: A re-examination of the reliability of unix utilities and services. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

Natella, R. (2022). Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering*, 27(7):191.

Natella, R. and Pham, V. T. (2021). Profuzzbench: A benchmark for stateful protocol fuzzing. In *ISSTA 2021 - Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 662–665. Association for Computing Machinery, Inc.

Pham, V.-T., Böhme, M., Santosa, A. E., Căciulescu, A. R., and Roychoudhury, A. (2021). Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997.

Pham, V.-T., Bohme, M., and Roychoudhury, A. (2020). Aflnet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 460–465. IEEE.

Shahid, M., Ibrahim, S., and Mahrin, M. N. (2011). A study on test coverage in software testing. *Advanced Informatics School (AIS), Universiti Teknologi Malaysia, International Campus, Jalan Semarak, Kuala Lumpur, Malaysia*.

Tanenbaum, A. (2009). *Modern operating systems*. Pearson Education, Inc.,.

Zalewski, M. (2017). American fuzzy lop. https://lcamtuf.coredump.cx/afl/.