# More Evaluation Results of UNIFUZZ Paper

Yuwei Li

August 2020

## 1 Introduction

This paper is to present more evaluation results of UNIFUZZ paper [26].

## 2 Flaws of Existing Fuzzers

In this section, we discuss typical flaws of several existing fuzzers found by us, along with the recommendations for repairing them.

**VUzzer64.** VUzzer64 is an updated version of VUzzer [33], which runs in the 64-bit Linux operating system. However, there is an error with the code that determines whether an input triggers a crash. Generally, when a process crashes, the operating system will generate and send a corresponding signal that represents the fault, so that the parent process can determine the state of the child process. Some fuzzers like AFL get the signal directly by calling `waitpid` and `WTERMSIG` to determine a crash. Other fuzzers such as VUzzer leverage the exit code of a process to determine a crash. In particular, VUzzer leverages the Python subprocess library to run the target program directly (`shell=False` when calling `subprocess.Popen`) and when the program crashes, VUzzer gets a minus exit code value which represents the received signal. However, VUzzer64 uses the Linux shell to run a program (`shell=True` when calling `subprocess.Popen`), and the exit code of a crash is 128 plus the signal value. The flaw of VUzzer64 is that it neglects this change, while still using the same code for determining a crash as VUzzer. This flaw is extremely severe as it makes VUzzer64 not being able to find any crash during the fuzz testing. With our report and suggestion, this flaw has been fixed in the current version of VUzzer64.

**T-Fuzz.** The key idea of T-Fuzz is when the fuzzing process gets "stuck" (which means a fuzzer cannot find any new paths for a long time), it will generate a transformed program that negates the path constraint to fuzz deeper paths of a program. However, there are two flaws in the implementation of T-Fuzz published on GitHub [11]. First, T-Fuzz may get into a state that keeps generating many transformed programs for a long time without fuzzing. Therefore, it wastes time that should be used for fuzzing. We call this flaw as *program explosion*. This flaw can be solved by setting a maximum number of transformed programs. If T-Fuzz has generated enough transformed programs, it should continue the fuzzing process. Second, for each transformed program, T-Fuzz gives it a new name which is different from the original binary name. However, there are some programs such as `busybox` whose functionalities are related to their names. If their names are not preserved, their functionalities will be changed. Therefore, the fuzzing process on these name-changed binaries is meaningless. This issue can be solved by putting each transformed program into a new directory and changing the directory name instead of changing the binary name.

**AFLFast.** AFLFast is a coverage-based fuzzer based on AFL. The core idea of AFLFast is paying more attention on generating seeds which can cover few-frequency paths. However, due to the implementation defects, AFLFast may get into a state that generates a large amount of cycles instead of generating new inputs. Zhu et al. [42] also found this flaw and they called this *cycle explosion*.

Figure 1 shows the key code that causes the *cycle explosion* flaw. The return variable of function `calculate_score` is `perf_score`, which represents how many new inputs will be generated based on the seed `q`. The value of `perf_score` is affected by variable `fuzz`, which is the number of inputs that have the same paths as seed `q`. When variable `fuzz` is very large, the value of `perf_score` is close to zero which means no new inputs will be generated based on this seed. If the `perf_score` values of all the seeds in the queue are low, then this cycle will be finished quickly and AFLFast may get into a state where the number of cycles increases very quickly but no new inputs are produced. One solution for this problem is to set a minimum value of variable `perf_score`.

We also find other flaws of the existing fuzzers including inconsistency between documentation and code, the crashes of fuzzers during the fuzzing process, compilation issues, etc. Due to the space limitations, we do not discuss them in detail here. For all these flaws, we have notified the corresponding developers and we will publish the flaws along with our recommendations to fix the flaws on the UniFuzz platform.

```
u32 calculate_score(struct queue_entry* q){
 /* default setting */
 u32 perf_score = 100;
 ...
 switch (schedule){
  case FAST:
   if (q->fuzz_level < 16) {
     factor = ((u32) (1 << q->fuzz_level)) /   (fuzz == 0 ? 1 : fuzz);
   } else
     factor = MAX_FACTOR / (fuzz == 0 ? 1 : next_p2 (fuzz));
   break;
  ...
 }
 if (factor > MAX_FACTOR)
   factor = MAX_FACTOR;
 perf_score *= factor / POWER_BETA;
 return perf_score;
}
```

Figure 1: The key code that causes *cycle explosion* in AFLFast.

Table 1: The fuzzers incorporated in UniFuzz.

| **Fuzzer** | **Mutation/Generation** | **Directed/Coverage** | **Target** |
|---|---|---|---|
| AFL [40] | M | C | S/B [1] |
| AFLFast [14] | M | C | S/B |
| AFLGo [13] | M | D | S |
| AFLPIN [4] | M | C | B |
| AFLSmart [32] | M | C | S/B |
| Angora [15] | M | C | S/B |
| CodeAlchemist [20] | G | n.a. | B |
| Driller [36] | M | C | B |
| Domato [17] | G | n.a. | B |
| Dharma [5] | G | n.a. | B |
| Eclipser [16] | M | C | S |
| FairFuzz [25] | M | C | S |
| Fuzzilli [8] | M | C | S |
| Grammarinator [22] | G | n.a. | B |
| Honggfuzz [18] | M | C | S |
| Jsfuzz [10] | M | C | S |
| jsfunfuzz [9] | G | n.a. | B |
| LearnAFL [38] | M | C | S |
| MoonLight [21] | n.a. | n.a. | n.a. |
| MOPT [28] | M | C | S/B |
| NAUTILUS [12] | G+M | C | S |
| NEUZZ [35] | M | C | S |
| NEZHA [30] | M | C | L [2] |
| Orthrus [34] | n.a. | n.a. | n.a. |
| Peach [6] | G | n.a. | B |
| PTfuzz [41] | M | C | S |
| QSYM [39] | M | C | B |
| QuickFuzz [19] | G+M | n.a. | B |
| radamsa [7] | M | C | B |
| slowfuzz [31] | M | n.a. | L |
| Superion [37] | G+M | C | S |
| T-Fuzz [29] | M | C | S |
| VUzzer [33] | M | C | B |
| VUzzer64 [33] | M | C | B |
| zzuf [24] | M | n.a. | B |

S: source code, B: binary.

L: user needs to write libFuzzer code.

# 3 Usable Fuzzers in UniFuzz

Table 1 presents the detailed information of the usable fuzzers incorporated in UniFuzz.

# 4 The Bug Information of the UniFuzz Benchmark

To add more bug information of the UniFuzz benchmark programs, we combine three static analysis tools (Flawfinder [2], RATS [3], Clang Static Analyzer [1]) with the directed fuzzer, AFLGo [13], to analyze the UniFuzz benchmark programs. Table 2 presents the detection results, where we can observe that the detection results among the three analysis tools vary widely. Indeed, our manual analysis reveals that the static analyzers suffer from many false

Table 2: The flaws detected by the static analysis tools and the number of all known unique bugs of the UniFuzz benchmark.

| Program | Num of Flaws | | | Static Analysis + AFLGo | | | Num of all known Unique Bugs |
|---|---|---|---|---|---|---|---|
| | RATS | Flawfinder | Clang Static Analyzer | Num of Targets | Num of Unique Bugs | Num of New Unique Bugs1 | |
| exiv2 | 716 | 1,208 | 57 | 36 | 12 | 0 | 36 |
| gdk | 46 | 147 | 16 | 14 | 0 | 0 | 14 |
| imginfo | 79 | 95 | 63 | 19 | 0 | 0 | 19 |
| jhead | 94 | 190 | 15 | 11 | 13 | 1 | 12 |
| tiffsplit | 319 | 761 | 112 | 24 | 8 | 0 | 24 |
| lame | 192 | 438 | 37 | 22 | 3 | 0 | 22 |
| mp3gain | 33 | 168 | 24 | 1 | 0 | 0 | 1 |
| wav2swf | 494 | 1,749 | 499 | 102 | 1 | 0 | 102 |
| ffmpeg | 1,091 | 4,315 | 448 | 334 | 0 | 0 | 334 |
| flvmeta | 90 | 197 | 14 | 7 | 4 | 0 | 7 |
| mp42aac | 266 | 279 | 49 | 13 | 0 | 0 | 13 |
| cflow | 102 | 486 | 16 | 11 | 3 | 0 | 11 |
| infotocap | 476 | 994 | 39 | 34 | 1 | 0 | 34 |
| jq | 38 | 84 | 4 | 2 | 0 | 0 | 2 |
| mujs | 61 | 121 | 0 | 0 | 0 | 0 | 0 |
| pdftotext | 139 | 571 | 54 | 23 | 2 | 0 | 23 |
| sqlite3 | 431 | 1,335 | 52 | 35 | 4 | 0 | 35 |
| nm | 7,878 | 15,912 | 117 | 3 | 6 | 2 | 5 |
| objdump | 5,757 | 10,408 | 247 | 139 | 1 | 0 | 139 |
| tcpdump | 138 | 372 | 145 | 1 | 0 | 0 | 1 |

The value of this column represents the new unique bugs discovered by combing the static tools with AFLGo, in addition to the bugs discovered by the eight coverage-based fuzzers in Section ??.

positives. We therefore first manually select suspicious flaws, and the corresponding number is shown in the column *Num of Targets* in Table 2. Then, to confirm the existence of the bugs, we use AFLGo to test the programs by setting the suspicious flaws as the targets for 10 days. Finally, we find 58 unique bugs in total, where three of them are new bugs (one bug for `jhead`, and two bugs for `nm`), in addition to the bugs discovered by the eight coverage-based fuzzers evaluated in Section ??. Table 2 also presents the number of all the known unique bugs.

# 5 The Number of Unique Bugs

We present more detailed statistical results of the numebr of unique bugs in Table 3, which includes the minimum, maximum, the arithmetic mean and the median of the number of unique bugs.

Table 3: The minimum, maximum, the arithmetic mean and the median of the number of unique bugs.

| | AFL | | | | AFLFast | | | | Angora | | | | Honggfuzz | | | | MOPT | | | | QSYM | | | | T-Fuzz | | | | VUzzer64 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | min | max | avg | mid | min | max | avg | mid | min | max | avg | mid | min | max | avg | mid | min | max | avg | mid | min | max | avg | mid | min | max | avg | mid | min | max | avg | mid |
| exiv2 | 0 | 10 | 1.7 | 1 | 0 | 4 | 0.67 | 0 | 6 | 17 | 9.77 | 10 | 0 | 4 | 1.4 | 1 | 1 | 15 | 4.77 | 3.5 | 0 | 6 | 2.9 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| gdk | 0 | 1 | 0.3 | 0 | 0 | 2 | 0.5 | 0 | 0 | 6 | 2.53 | 3 | 0 | 1 | 0.4 | 0 | 4 | 8 | 6.7 | 7 | 5 | 16 | 9.77 | 10 | 1 | 1 | 1 | 1 | 0 | 3 | 0.6 | 0 |
| imginfo | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 0.7 | 0.5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| jhead | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 5 | 2.07 | 1 | 1 | 2 | 1.13 | 1 | 0 | 3 | 0.83 | 1 | 7 | 22 | 13.13 | 13.5 | 0 | 11 | 1.9 | 0 | 0 | 0 | 0 | 0 |
| tiffsplit | 3 | 9 | 6.6 | 7 | 5 | 8 | 5.57 | 5 | 3 | 6 | 4.73 | 5 | 0 | 2 | 0.7 | 1 | 4 | 9 | 6.9 | 7 | 5 | 9 | 6 | 6 | 3 | 5 | 3.9 | 4 | 0 | 0 | 0 | 0 |
| lame | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 3.5 | 3.5 | 3 | 4 | 3.6 | 4 | 4 | 4 | 4 | 4 | 3 | 4 | 3.97 | 4 | 3 | 3 | 3 | 3 | 0 | 0 | 0 | 0 |
| mp3gain | 3 | 7 | 5.5 | 6 | 4 | 7 | 5.5 | 5 | 4 | 7 | 5.4 | 5 | 7 | 10 | 8.3 | 8 | 4 | 9 | 6.27 | 6 | 6 | 11 | 8.67 | 9 | 3 | 5 | 3.2 | 3 | 1 | 2 | 1.27 | 1 |
| wav2swf | 2 | 3 | 2.2 | 2 | 2 | 3 | 2.07 | 2 | 3 | 5 | 3.47 | 3 | 1 | 3 | 1.87 | 2 | 2 | 3 | 2.63 | 3 | 2 | 3 | 2.27 | 2 | 1 | 1 | 1 | 1 | 1 | 2 | 1.3 | 1 |
| ffmpeg | 0 | 1 | 0.37 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 3 | 1.47 | 1 | 0 | 1 | 0.63 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| flvmeta | 3 | 4 | 3.63 | 4 | 1 | 4 | 3.5 | 4 | 3 | 4 | 3.83 | 4 | 4 | 5 | 4.87 | 5 | 4 | 4 | 4 | 4 | 1 | 4 | 2.77 | 3 | 0 | 4 | 3.5 | 4 | 1 | 2 | 1.2 | 1 |
| mp42aac | 0 | 1 | 0.07 | 0 | 0 | 1 | 0.67 | 1 | 1 | 5 | 2.53 | 2 | 0 | 1 | 0.43 | 0 | 2 | 4 | 2.63 | 2 | 0 | 5 | 2.43 | 2.5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| cflow | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 7 | 5.17 | 5 | 3 | 5 | 3.8 | 4 | 2 | 4 | 2.93 | 3 | 0 | 0 | 0 | 0 | 0 | 1 | 0.33 | 0 |
| infotocap | 0 | 3 | 1.73 | 2 | 1 | 4 | 2.07 | 2 | 0 | 0 | 0 | 0 | 0 | 1 | 0.13 | 0 | 3 | 8 | 4.7 | 5 | 0 | 7 | 4.37 | 4.5 | 0 | 0 | 0 | 0 | 0 | 1 | 0.07 | 0 |
| jq | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 1.13 | 1 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mujs | 0 | 0 | 0 | 0 | 0 | 1 | 0.1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0.1 | 0 | 0 | 1 | 0.17 | 0 | 1 | 2 | 1.1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pdftotext | 0 | 2 | 0.3 | 0 | 0 | 2 | 0.73 | 1 | 1 | 1 | 1 | 1 | 0 | 3 | 0.93 | 1 | 3 | 14 | 7.3 | 5.5 | 1 | 4 | 2.17 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sqlite3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0.43 | 0 | 1 | 5 | 3.47 | 3.5 | 0 | 5 | 2.47 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| nm | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 2.23 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| objdump | 1 | 3 | 1.33 | 1 | 1 | 3 | 1.33 | 1 | 1 | 13 | 4.9 | 3 | 0 | 2 | 0.73 | 1 | 3 | 11 | 6.1 | 6 | 0 | 4 | 1.63 | 1.5 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| tcpdump | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 1.6 | 1 | 0 | 0 | 0 | 0 | 1 | 4 | 1.87 | 2 | 13 | 48 | 31.07 | 33 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# 6 The CVEs Found By the Fuzzers.

Here we present the detailed information of the CVEs found by the fuzzers in Section 4 of the UNIFUZZ paper [26], including the concrete CVE ID, CVSS score and the vulnerability type in Table 4.

# 7 The Correlation Between Bug Number and Line Coverage

To explore the relationship between the number of unique bugs and line coverage, we calculate the *Spearman correlation coefficient* $r_s$ between them, which is a non-parametric measure of correlation between two variables and

Table 4: The CVEs found by the fuzzers.

| Program | CVE ID | CVSS | Vulnerability Type | Fuzzer |
|---|---|---|---|---|
| exiv2 | CVE-2017-12955 | 8.8 | Buffer Overflow | Angora |
| | CVE-2018-9305 | 8.1 | Information Leak | Angora, Honggfuzz, MOPT, QSYM |
| | CVE-2018-9144 | 8.1 | Buffer Overflow | Angora, MOPT, QSYM |
| | CVE-2017-11337 | 7.5 | Free Error | AFLFast, AFL, Angora, Honggfuzz, MOPT |
| | CVE-2017-11339 | 6.5 | Buffer Overflow | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM |
| | CVE-2018-17229, CVE-2018-17230 | 6.5 | Buffer Overflow | AFL, MOPT |
| | CVE-2017-17724 | 6.5 | Buffer Overflow | Angora, Honggfuzz, MOPT, QSYM |
| | CVE-2017-12956 | 6.5 | SEGV | AFLFast, Angora |
| | CVE-2019-13112 | 6.5 | Excessive Memory Allocation | Angora |
| | CVE-2018-10780 | 6.5 | Buffer Overflow | AFL |
| | CVE-2018-19535 | 6.5 | Buffer Overflow | Angora |
| | CVE-2017-11683 | 6.5 | Assertion Failure | Angora, MOPT |
| | CVE-2019-13113 | 6.5 | Assertion Failure | Angora, QSYM |
| | CVE-2018-10999 | 6.5 | Buffer Overflow | Angora, QSYM |
| | CVE-2017-11336 | 6.5 | Buffer Overflow | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM |
| | CVE-2017-18005 | 5.5 | Null Pointer Dereference | Angora |
| | CVE-2017-14863 | 5.5 | Null Pointer Dereference | Angora, Honggfuzz, MOPT |
| | CVE-2017-14866 | 5.5 | Buffer Overflow | AFL, MOPT |
| | CVE-2017-1000128, CVE-2017-1000126 | 5.5 | Buffer Overflow | Angora |
| | CVE-2017-14865, CVE-2017-14858 | 5.5 | Buffer Overflow | AFL, MOPT |
| | CVE-2017-14861 | 5.5 | Stack Overflow | AFLFast, AFL, Angora, Honggfuzz, MOPT |
| | CVE-2017-1000127 | 5.5 | Buffer Overflow | AFL, Angora, MOPT |
| | CVE-2017-17669 | 5.5 | Buffer Overflow | Angora, QSYM |
| gdk | CVE-2015-7674 | 6.8 | Integer Overflow | Angora, MOPT |
| | CVE-2015-7673 | 6.8 | Buffer Overflow | QSYM |
| | CVE-2015-4491 | 6.8 | Other | MOPT, QSYM |
| imginfo | CVE-2016-9396, CVE-2017-13747, CVE-2017-13751, CVE-2017-13745 | 7.5 | Assertion Failure | MOPT |
| | CVE-2017-13746, CVE-2016-9397 | 7.5 | Assertion Failure | MOPT |
| | CVE-2016-9393 | 5.5 | Assertion Failure | MOPT |
| jhead | CVE-2018-17088 | 7.8 | Integer Overflow | Angora, QSYM, T-Fuzz |
| | CVE-2018-6612, CVE-2019-1010301 | 5.5 | Buffer Overflow | Angora, QSYM, T-Fuzz |
| | CVE-2019-1010302 | 5.5 | Other | Honggfuzz, MOPT, QSYM, T-Fuzz |
| tiffsplit | CVE-2016-5318 | 9.8 | Buffer Overflow | AFLFast, AFL, Angora, MOPT, T-Fuzz |
| | CVE-2016-6223 | 9.1 | Other | AFLFast, AFL, Angora, MOPT, QSYM, T-Fuzz |
| | CVE-2017-7602 | 7.8 | Integer Overflow | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz |
| | CVE-2017-9147 | 6.5 | Buffer Overflow | AFL, MOPT |
| | CVE-2014-8127 | 6.5 | Buffer Overflow | AFLFast, AFL, Angora, MOPT |
| | CVE-2016-9273 | 5.5 | Buffer Overflow | MOPT, QSYM |
| | CVE-2016-10095 | 5.5 | Buffer Overflow | AFLFast, AFL, Angora, MOPT, QSYM, T-Fuzz |
| | CVE-2010-2631 | 4.3 | Other | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz |
| lame | CVE-2017-11720 | 9.8 | Floating Point Exception | AFL, AFLFast, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz |
| | CVE-2017-8419 | 7.8 | Buffer Overflow | Angora, Honggfuzz, MOPT, QSYM |
| | CVE-2017-15045, CVE-2015-9101 | 5.5 | Buffer Overflow | AFL, AFLFast, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz |
| mp3gain | CVE-2018-10777 | 7.8 | Buffer Overflow | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz |
| | CVE-2017-12912 | 7.8 | SEGV | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM |
| | CVE-2017-14412 | 7.8 | SEGV | Honggfuzz, MOPT, QSYM |
| | CVE-2017-14411 | 7.8 | Buffer Overflow | Honggfuzz, MOPT, QSYM |
| | CVE-2017-14409 | 7.8 | Buffer Overflow | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz, VUzzer64 |
| | CVE-2017-14408 | 5.5 | Buffer Overflow | Honggfuzz |
| | CVE-2017-14410, CVE-2017-14407 | 5.5 | Buffer Overflow | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz, VUzzer64 |
| | CVE-2017-14406 | 5.5 | Null Pointer Dereference | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz |
| wav2swf | CVE-2017-16793 | 7.8 | Buffer Overflow | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM, VUzzer64 |
| | CVE-2017-16868 | 5.5 | Null Pointer Dereference | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM |
| | CVE-2017-1000182 | 5.5 | Memory Leak | Angora |
| | CVE-2017-16890 | 5.5 | Floating Point Exception | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz, VUzzer64 |
| ffmpeg | CVE-2018-13302 | 8.8 | Buffer Overflow | Honggfuzz |
| mp42aac | CVE-2018-14587, CVE-2018-14584 | 8.8 | Buffer Overflow | QSYM |
| | CVE-2018-14588 | 7.5 | Null Pointer Dereference | MOPT, QSYM |
| | CVE-2018-20407 | 6.5 | Memory Leak | AFLFast, Angora, MOPT |
| | CVE-2018-20095 | 6.5 | Excessive Memory Allocation | AFL, Angora, Honggfuzz, MOPT, QSYM |
| cflow | CVE-2019-16165 | 6.5 | Use After Free | Honggfuzz, MOPT, QSYM, VUzzer64 |
| infotocap | CVE-2017-10685 | 9.8 | Format String Vulnerability | MOPT, QSYM |
| | CVE-2017-11113 | 7.5 | Null Pointer Dereference | AFLFast, AFL, Honggfuzz, MOPT, QSYM |
| jq | CVE-2015-8863 | 9.8 | Buffer Overflow | Honggfuzz, QSYM |
| mujs | CVE-2017-5627 | 7.8 | Integer Overflow | QSYM |
| | CVE-2018-6191 | 5.5 | Integer Overflow | AFLFast, MOPT |
| | CVE-2018-5759 | 5.5 | Stack Overflow | QSYM |
| pdftotext | CVE-2019-9587 | 7.8 | Stack Overflow | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM |
| | CVE-2019-13281 | 7.8 | Information Leak | MOPT |
| | CVE-2019-13283 | 7.8 | Information Leak | MOPT, T-Fuzz |
| | CVE-2019-9877 | 7.8 | SEGV | AFLFast, AFL, Honggfuzz, MOPT, QSYM |
| | CVE-2019-13291 | 5.5 | Buffer Overflow | MOPT |
| | CVE-2018-18650 | 5.5 | Integer Overflow | AFLFast, AFL, Honggfuzz, MOPT, QSYM |
| | CVE-2019-13288 | 5.5 | Stack Overflow | MOPT, QSYM |
| nm | CVE-2019-17450 | 6.5 | Stack Overflow | Angora |
| | CVE-2019-17451 | 6.5 | Integer Overflow | Angora |
| objdump | CVE-2017-14130 | 5.5 | Buffer Overflow | QSYM |
| | CVE-2017-15024 | 5.5 | Stack Overflow | AFLFast, AFL, Angora |
| | CVE-2017-15225 | 5.5 | Memory Leak | Angora, MOPT |
| | CVE-2017-14940 | 5.5 | Null Pointer Dereference | AFLFast, AFL, Angora, Honggfuzz, MOPT, QSYM, T-Fuzz |
| | CVE-2017-14129 | 5.5 | Buffer Overflow | Angora, MOPT, QSYM |
| | CVE-2017-14932 | 5.5 | Infinite Loop | Angora, MOPT |
| tcpdump | CVE-2017-13005, CVE-2016-7933, CVE-2017-12998, CVE-2016-7924 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2016-7975, CVE-2017-11542, CVE-2016-7930, CVE-2017-13023 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2017-13025, CVE-2017-5485, CVE-2017-12899, CVE-2017-13004 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2017-13035, CVE-2017-13000, CVE-2017-13033, CVE-2017-13020 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2017-12985, CVE-2017-12999, CVE-2017-12986, CVE-2017-12893 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2017-12993, CVE-2017-12902, CVE-2017-5341, CVE-2016-7940 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2017-13052, CVE-2016-7925, CVE-2017-13015, CVE-2017-13038 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2017-13012, CVE-2017-13688, CVE-2017-13031, CVE-2017-12996 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2017-5202, CVE-2017-5483, CVE-2017-12900, CVE-2017-13026 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2017-13017, CVE-2017-13016, CVE-2017-13028, CVE-2016-7985 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2016-7986, CVE-2016-7923, CVE-2016-7973, CVE-2017-12988 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2016-7931, CVE-2017-13022, CVE-2016-7983, CVE-2016-7992 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2016-7927, CVE-2016-7929, CVE-2017-5484, CVE-2016-7974 | 9.8 | Buffer Overflow | QSYM |
| | CVE-2017-12895, CVE-2016-7984 | 9.8 | Buffer Overflow | Angora, MOPT, QSYM |
| | CVE-2016-7939, CVE-2016-7932 | 9.8 | Buffer Overflow | Angora, QSYM |
| | CVE-2016-7993 | 9.8 | Buffer Overflow | MOPT, QSYM |

$r_s \in [-1, +1]$. A positive $r_s$ means that the two variables are positively correlated and vice versa. Figure 2 presents the value of $r_s$ between the number of unique bugs and line coverage, where we observe that most of them are less than 0.60, which means that the correlation between the number of unique bugs and the line coverage is not strong.

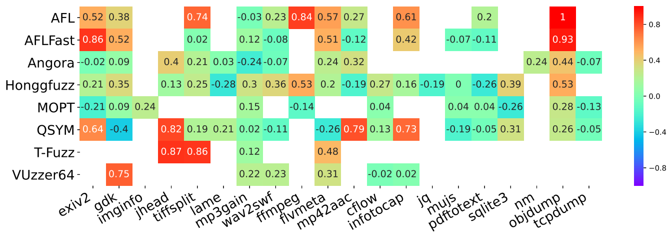

Figure 2: The Spearman's correlation coefficient $r_s$ between the number of unique bugs and line coverage.

# 8 Performance of the Fuzzers Using Different Seed Sets

In order to evaluate a fuzzer with different seed sets, we select four seed sets with different amounts: empty, 10 seeds, 50 seeds, and 100 seeds from the seed corpus collected in Section ??. Each fuzzing experiment is conducted for 5 hours, with 20 repetitions. Figure 3 shows the performance of each fuzzer with different seed sets when applied on `exiv2`, `mp3gain` and `who`, where we have the following observations. (1) One observation is that the results of non-empty seed sets have similar distributions. The comparison results of the fuzzers' performance among the three non-empty seed sets is not significantly different. (2) There is significant difference between the fuzzers' performance on empty seed set and non-empty seed sets. For instance, on `exiv2`, when using non-empty seed sets, most of the fuzzers can find bugs, while using empty seed set, only QSYM and MOPT perform well.

Although Klees et al.[23] suggested using an empty seed to evaluate fuzzers, our evaluation does not support this recommendation unfortunately. First, it is difficult for a fuzzer to generate well-formatted inputs from an empty seed. Second, as shown in Figure 3, the fuzzing performance is less stable when using an empty seed as compared to that using the non-empty seed sets.

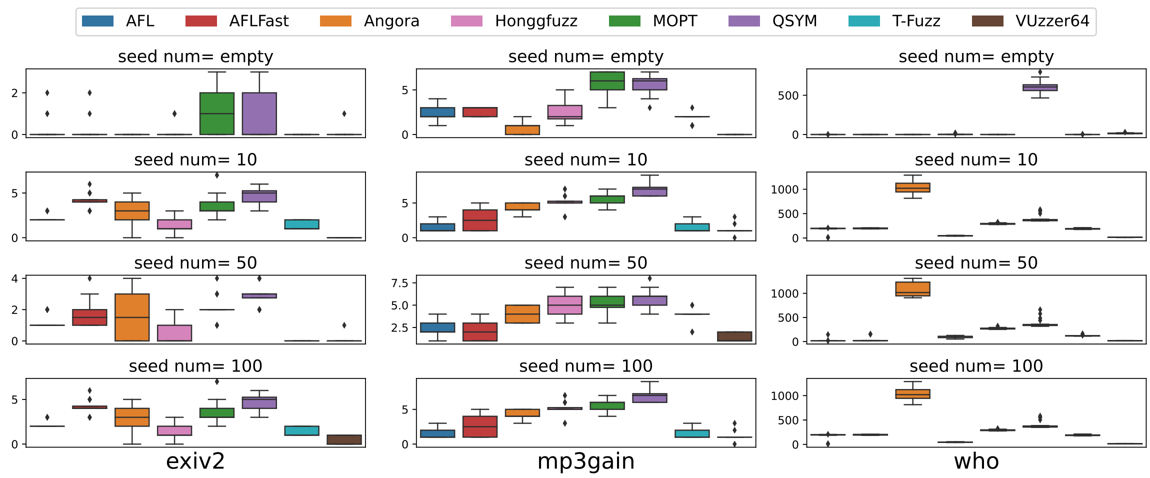


Figure 3: The number of unique bugs varying with different seed sets.

# 9 Fuzzing for a Long Period of Time

Since fuzzers may have late-development advantage, i.e., they may perform better for a longer fuzzing period. We conduct further evaluations on `ffmpeg`, `objdump` and `pdftotext` by fuzzing for 10 days, with three repetitions. QSYM discovers the most unique bugs on program `pdftotext`. Honggfuzz and MOPT discover the most unique bugs on program `ffmpeg` and `pdftotext`, respectively. We further show the growth curve of the number of unique bugs in Figure 4, where we observe that a longer fuzzing time is likely to result in more stable fuzzing performance. In addition, we put the median values of the number of unique bugs discovered in 10 days together with the results in 24 hours for easy comparison, and the result is presented in Table 5. For 24 groups of the fuzzing experiments

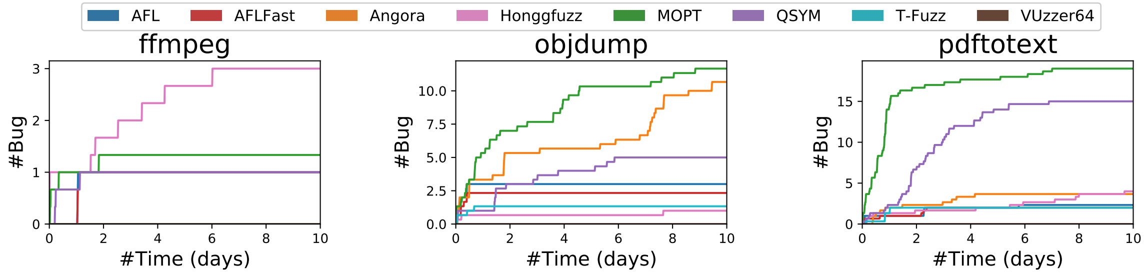


Figure 4: The growth curve of the number of unique bugs varying with time.

Table 5: The median of the number of unique bugs discovered by the fuzzers in 24 hours and 10 days.

| | AFL | | AFLFast | | Anogra | | Honggfuzz | | MOPT | | QSYM | | T-Fuzz | | VUzzer64 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 24h | 10 days | 24h | 10 days | 24h | 10 days | 24h | 10 days | 24h | 10 days | 24h | 10 days | 24h | 10 days | 24h | 10 days |
| `ffmpeg` | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 3 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| `objdump` | 1 | 3 | 1 | 3 | 3 | 11 | 1 | 1 | 6 | 10 | 1.5 | 5 | 1 | 1 | 0 | 0 |
| `pdftotext` | 0 | 2 | 1 | 2 | 1 | 3 | 1 | 4 | 5.5 | 23 | 2 | 14 | 0 | 2 | 0 | 0 |

(eight fuzzers with three benchmark programs), there are almost 1/3 experiments where the fuzzer does not find more unique bugs in 10 days compared to that in 24 hours. For instance, on `objdump`, Honggfuzz, T-Fuzz and VUzzer64 do not make any process in finding new bugs from the second day to the 10th day fuzzing. This observation reveals that the feedback mechanism of the three fuzzers may not be efficient in guiding to find bugs on `objdump`.

# 10 Execution Speed Analysis

Table 6 shows the median of execution speed (i.e., execution times per second) of the fuzzers among 30 repetitions, where we get the following observations. (1) The execution speed of VUzzer64 is much slower than other fuzzers. The reason may be that VUzzer64 leverages Intel PIN [27] to implement dynamic binary instrumentation, which makes the execution speed slower than other fuzzers that use compile-time instrumentation. Using dynamic binary instrumentation does not need the source code of the target program, which is an advantage of fuzzers like VUzzer64. Nevertheless, the slow execution speed might be the main reason to explain the performance limitation of VUzzer64. Thus, improving the execution speed might be a solution to improve the performance of binary instrumentation based fuzzers. (2) Although slower execution speed might be the bottleneck of a fuzzer, by comparing to the experimental results in Section 4 of the UNIFUZZ paper [26], fuzzers that achieve higher execution speed do not necessarily have better performance in finding bugs or exploring paths. For instance, T-Fuzz achieves relatively a higher execution speed among the fuzzers, while it does not find more bugs. (3) Fuzzers that achieve higher execution speed do not necessarily take more computing resources. For instance, AFL, AFLFast and MOPT have a high execution speed, but they do not use more memory during the fuzzing process.

Table 6: The median of execution times per second of the fuzzers among 30 repetitions.

| | AFL | AFLFast | Angora | Honggfuzz | MOPT | QSYM | T-Fuzz | VUzzer64 |
|---|---|---|---|---|---|---|---|---|
| `exiv2` | 389.9 | 283.13 | 132.08 | 8.75 | 388.35 | 5.23 | 723.6 | 0.24 |
| `gdk` | 167.33 | 172.45 | 240.72 | 18.26 | 193.41 | 36.03 | 557.01 | 0.49 |
| `imginfo` | 12.41 | 12.62 | 63.63 | 39.96 | 367.03 | 47.08 | 44.87 | 0.18 |
| `jhead` | 399.37 | 306.68 | 566.98 | 78.98 | 500.77 | 101.4 | 1660.81 | 1.02 |
| `tiffsplit` | 699.18 | 500.1 | 466.85 | 0.97 | 767.93 | 25.38 | 2214.38 | 1.13 |
| `lame` | 83.125 | 81.52 | 24.745 | 6.56 | 59.92 | 19.22 | 103.82 | 0.58 |
| `mp3gain` | 289.22 | 303.65 | 328.69 | 67.38 | 208.58 | 132.64 | 207.18 | 0.53 |
| `wav2swf` | 366.49 | 572.43 | 118.56 | 260.12 | 740.5 | 11.28 | 1448.50 | 1.31 |
| `ffmpeg` | 4.105 | 4.41 | 75.33 | 1.625 | 7.39 | 16.24 | n.a. | 0 |
| `flvmeta` | 695.33 | 689.46 | 625.32 | 244.51 | 835.5 | 9.03 | 1503.48 | 1.38 |
| `mp42aac` | 496.02 | 506.16 | 189.27 | 165.64 | 800.28 | 121.33 | 1376.33 | 0.91 |
| `cflow` | 45.055 | 44.58 | 39.42 | 27.59 | 130.89 | 24.85 | 26.9 | 0.01 |
| `infotocap` | 681.92 | 706.19 | 701.49 | 30.87 | 784.36 | 30.42 | 517.58 | 0.8 |
| `jq` | 82.945 | 81.79 | 134.88 | 92.62 | 91.53 | 32.15 | 107.82 | 0.67 |
| `mujs` | 303.59 | 262.73 | 127.46 | 128.84 | 319.58 | 83.8 | 385.85 | 0.67 |
| `pdftotext` | 104.79 | 98.99 | 43.55 | 10.01 | 149.92 | 66.7 | 67.62 | 0.39 |
| `sqlite3` | 79.18 | 58.8 | 73.06 | 1.38 | 227.29 | 38.33 | 424.64 | n.a. |
| `nm` | 130.42 | 118.38 | 136.76 | 46.86 | 306.81 | 64.74 | 1466.31 | 0.6 |

## References


[1] Clang static analyzer. https://clang-analyzer.llvm.org/.

[2] Flawfinder. https://dwheeler.com/flawfinder/.

[3] Rough auditing tool for security (rats). https://code.google.com/archive/p/rough-auditing-tool-for-security/.

[4] AFLPIN. https://github.com/mothran/aflpin, 2015.

[5] dharma: Generation-based, context-free grammar fuzzer. https://github.com/MozillaSecurity/dharma, 2018.

[6] Peach fuzzer. https://www.peach.tech, 2018.

[7] radamsa: a general-purpose fuzzer. https://gitlab.com/akihe/radamsa, 2018.

[8] Fuzzilli. https://github.com/googleprojectzero/fuzzilli, 2019.

[9] jsfunfuzz. https://github.com/MozillaSecurity/funfuzz/tree/master/src/funfuzz/js/jsfunfuzz, 2019.

[10] Jsfuzz: coverage-guided fuzz testing for javascript. https://github.com/fuzzitdev/jsfuzz, 2019.

[11] T-Fuzz. https://github.com/HexHive/T-Fuzz, 2019.

[12] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *Network and Distributed System Security (NDSS)*, 2019.

[13] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2329–2344, 2017.

[14] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1032–1043, 2016.

[15] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[16] Jaeseung Choi, Joonun Jang, Choongwoo Han, and Sang Kil Cha. Grey-box concolic testing on binary code. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 736–747, 2019.

[17] Ivan Fratric. Domato: A DOM fuzzer. https://github.com/googleprojectzero/domato.

[18] Google. honggfuzz. https://google.github.io/honggfuzz/, 2017.

[19] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. QuickFuzz: An automatic random fuzzer for common file formats. In *ACM SIGPLAN Notices*, volume 51, pages 13–20. ACM, 2016.

[20] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *Network and Distributed System Security (NDSS)*, 2019.

[21] Liam Hayes, Hendra Gunadi, Adrian Herrera, Jonathon Milford, Shane Magrath, Maggi Sebastian, Michael Norrish, and Antony L Hosking. Moonlight: Effective fuzzing with near-optimal corpus distillation. *arXiv preprint arXiv:1905.13055*, 2019.

[22] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, pages 45–48. ACM, 2018.

[23] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2123–2138, 2018.

[24] Caca Labs. zzuf - multi-purpose fuzzer. http://caca.zoy.org/wiki/zzuf/, 2017.

[25] Caroline Lemieux and Koushik Sen. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 475–485. ACM, 2018.

[26] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluatingfuzzer. In *Proceedings of the 30th USENIX Security Symposium*, 2021.

[27] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[28] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Weihan Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Security Symposium*, pages 1949–1966, 2019.

[29] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-Fuzz: fuzzing by program transformation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP)*, pages 697–710, 2018.

[30] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. NEZHA: Efficient domain-independent differential testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, pages 615–632, 2017.

[31] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2155–2168, 2017.

[32] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[33] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Network and Distributed System Security (NDSS)*, 2017.

[34] Bhargava Shastry, Markus Leutner, Tobias Fiebig, Kashyap Thimmaraju, Fabian Yamaguchi, Konrad Rieck, Stefan Schmid, Jean-Pierre Seifert, and Anja Feldmann. Static program analysis as a fuzzing aid. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 26–47. Springer, 2017.

[35] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program learning. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP)*, pages 803–817, 2019.

[36] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security (NDSS)*.

[37] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 724–735. IEEE, 2019.

[38] Tai Yue, Yong Tang, Bo Yu, Pengfei Wang, and Enze Wang. LearnAFL: Greybox fuzzing with knowledge enhancement. *IEEE Access*, 7:117029–117043, 2019.

[39] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, pages 745–761, 2018.

[40] Michał Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/, 2017.

[41] Gen Zhang, Xu Zhou, Yingqi Luo, Xugang Wu, and Erxue Min. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access*, 6:37302–37313, 2018.

[42] Xiaogang Zhu, Xiaotao Feng, Tengyun Jiao, Sheng Wen, Yang Xiang, Seyit Camtepe, and Jingling Xue. A feature-oriented corpus for understanding, evaluating and improving fuzz testing. In *Proceedings of the 14th ACM Asia Conference on Computer and Communications Security*, pages 658–663, 2019.