# VisFuzz: Understanding and Intervening Fuzzing with Interactive Visualization

Chijin Zhou*, Mingzhe Wang*, Jie Liang*, Zhe Liu†, Chengnian Sun‡ and Yu Jiang*✉
BNRist, School of Software, Tsinghua University, Beijing, China*
Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China †
Cheriton School of Computer Science,University of Waterloo, Canada‡

***Abstract*—Fuzzing is widely used for vulnerability detection. One of the challenges for an efficient fuzzing is covering code guarded by constraints such as the magic number and nested conditions. Recently, academia has partially addressed the challenge via whitebox methods. However, high-level constraints such as array sorts, virtual function invocations and tree set queries are yet to be handled.**

**To meet this end, we present VisFuzz[1], an interactive tool for better understanding and intervening fuzzing process via real-time visualization. It extracts call graph and control flow graph from source code, maps each function and basic block to the line of source code and tracks real-time execution statistics with detail constraint contexts. With VisFuzz, test engineers first locate blocking constraints, and then learn its semantic context, which helps to craft targeted inputs or update test drivers. Preliminary evaluations are conducted on four real-world programs in Google fuzzer-test-suite. Given additional 15 minutes to understand and intervene the state of fuzzing, the intervened fuzzing outperforms the original pure AFL fuzzing, and the path coverage improvements range from 10.84% to 150.58%, equally fuzzed for 12 hours.**

**Video link: https://youtu.be/opjRKcqOvNs.**

## I. Introduction

Many fuzzers such as American Fuzzy Lop (AFL) [1], libFuzzer [2] have discovered hundreds of high-impact vulnerabilities in widely-used software systems. However, their performance is limited by complex constraints in large programs. One of the reasons is limited coverage — code guarded by constraints such as magic number and nested conditions prevent a fuzzer to explore paths behind.

To tackle the problem, a greybox approach is optimizing the mutation and selection algorithms. For example, AFLFast [3] gives more mutation times to valuable seeds which exercise low-frequency paths. FairFuzz [4] optimizes AFL's mutation algorithm to target rare branches. Those techniques still suffer from insufficient exploration of a program, because they cannot traverse paths beyond complex constraints, e.g., magic value. Another approach leverages whitebox techniques to automatically solving constraints. For example, Driller [5] switches to symbolic execution when AFL reaches a plateau, and switches back to AFL as soon as the complex constraint is bypassed. However, symbolic execution cannot effectively solve high-level constraints (e.g. indirect memory addressing). HaCRS [6] tries to divide hard-to-solve problem into some well-defined sub-tasks and displays them for human assistance. But it only provides symbolic tokens for human, and high level information is lost during the conversion.

High-level constraints are ubiquitous in real-world program, but the automatic techniques above are in lack of support for them (e.g. array sorts, virtual function invocations and tree set queries). On the contrary, manual intervention by human has the potential to solve high-level constraints.

Table I shows that both AFL and KLEE [7] take more than 12 hours to trigger the assertion violation when checking the 18-length array median problem. The straightforward example demonstrates that whitebox techniques suffer from path explosion and greybox techniques fails to cover the branch efficiently, while human can easily construct an input to satisfy the condition given the semantic context.

TABLE I: When automated test techniques meet checking array median problem.

(a) Code of checking array median

```
void test(int *buf, size_t size)
{
  if (size != SORT_SIZE)
   return;
  qsort(buf, SORT_SIZE, sizeof(buf[0]), cmp);
  assert(buf[SORT_SIZE / 2] != MAGIC_NUMBER);
}
```

(b) Time to assertion failure

| sort size | **AFL** | **KLEE** |
|---|---|---|
| 6 | 13min | 10s |
| 10 | >12h | 3min |
| 14 | >12h | 16min |
| 18 | >12h | >12h |

However, real-world programs are much more complex, and analyzing the semantic of the whole program is beyond human's ability. Thus, to enable human-assisted fuzzing practically, we need to solve three challenges: 1) locate the boundary of unexplored regions (where the bottleneck resides); 2) understand the semantic context around the bottleneck; 3) intervene the fuzzing process (e.g. construct targeted input, update test driver) to achieve higher coverage.

In this paper, we present VisFuzz. VisFuzz helps the test engineer to: 1) drill down into the bottleneck from function level, basic block level to statement level; 2) learn semantic context from basic blocks and source code; 3) construct targeted inputs or update the test driver to increase coverage. VisFuzz is implemented as a LLVM plugin for obtaining call graphs, control-flow graph and coverage, a modified AFL for runtime statistics, and a Python script for visualization web server. To the best of our knowledge, VisFuzz is the first tool for visualizing fuzzing process.

Preliminary evaluations are conducted on four real-world programs in Google fuzzer-test-suite. Given additional 15 minutes to understand and intervene the state of fuzzing, the intervened versions outperform the original AFL versions, and the path coverage improvements range from 10.84% to 150.58%, equally fuzzed for 12 hours.

## II. Demonstration Scenario

Figure 1 summarizes the procedure for understanding and intervening fuzzing with VisFuzz. 1) The test engineer monitors the charts and statistics provided by VisFuzz, and checks

[1]The tool can be download at: https://github.com/ChijinZ/VisFuzz

for the evidence of plateau. 2) If the situation needs intervention, then the test engineer navigates to the potential intervention points by the call graph and control flow graph. The graphs are enhanced with real-time coverage, and the search of intervention points is greatly accelerated. After locating the intervention points, VisFuzz loads the corresponding code snippet, and helps understanding the bottleneck by providing the semantic context for the test engineer. 3) Next, the test engineer performs intervention, such as constructing a targeted seed, or updating the test driver and target program.

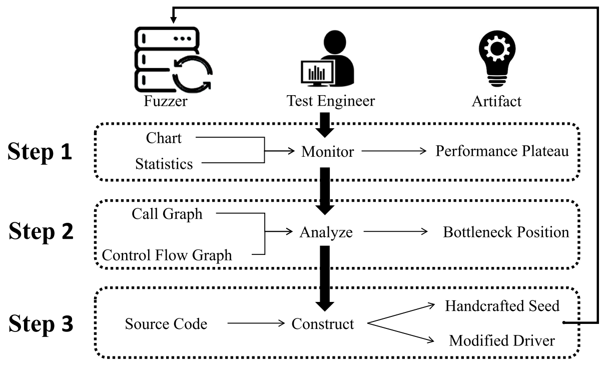


Fig. 1: Interaction between VisFuzz and test engineer: 1) monitor; 2) locate; 3) intervene.

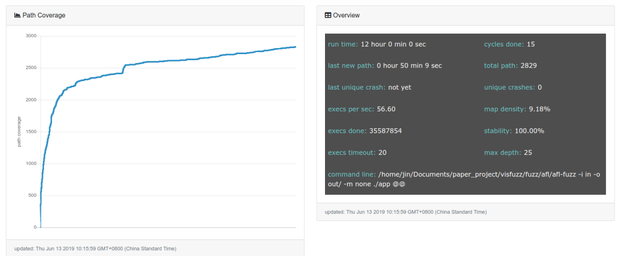
The procedure above is generic, and can be applied to all projects for fuzzing visualization. Here we demonstrate the usage of VisFuzz on re2, a well-tested program in Google fuzzer-test-suite. The first step is compilation. As Table II presents, VisFuzz follows the normal clang-like build procedure for cost-effective adaption. The next step is fuzzing re2, and the details are presented below.

TABLE II: VisFuzz and normal build procedure

| | VisFuzz | Normal build |
|---|---|---|
| Configure | ./configure | ./configure |
| Build lib | **make** -j | **make** -j |
| Build driver | **clang++** target.cc -c | **clang++** target.cc -c |
| Link | **vis-clang++** target.o \ lib.a -o app | **clang++** target.o \ lib.a -o app |
| Run | **vis-fuzz** -i in -o out ./app | ./app |

**Step 1: monitor charts and statistics**. The starting point is judging whether fuzzing has reached a plateau via analyzing the charts and statistic. As Fig. 2(a) shows, after 12 hours, path coverage in the chart has reached a plateau. The statistics also show that it has been 50 minutes since finding the last new path. The phenomenon implies that a bottleneck has been encountered, and users should follow Step 2 to find the bottleneck function.

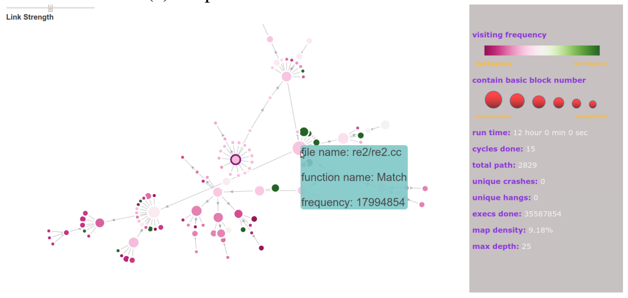
**Step 2: analyze call graphs and control flow graphs**. To find the bottleneck, users should investigate which function is seldom hit. As Fig. 2(b) shows, each node represents a function, and each edge represents the call relation between two functions. The color of a node denotes the hit frequency of the function. The size of a node denotes the number of basic blocks the function contains. When the mouse hovers on a specific node, the corresponding function information


Path Coverage
path coverage
3000
2500
2000
1500
1000
500
0
updated: Thu Jun 13 2019 10:15:59 GMT+0800 (China Standard Time)
Overview

```
run time: 12 hour 0 min 0 sec          cycles done: 15
last new path: 0 hour 50 min 9 sec     total path: 2829
last unique crash: not yet             unique crashes: 0
execs per sec: 56.60                   map density: 9.18%
execs done: 35587854                   stability: 100.00%
execs timeout: 20                      max depth: 25
command line: /home/jin/Documents/paper_project/visfuzz/fuzz/afl/afl-fuzz -i in -o
out/ -m none ./app @@
```

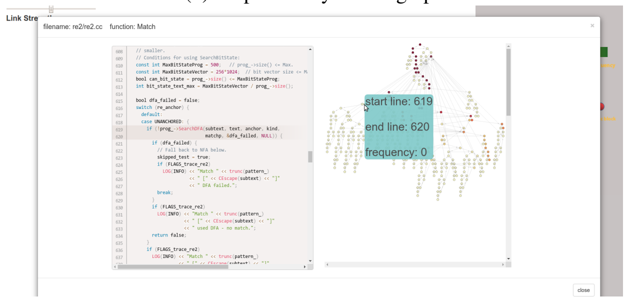
updated: Thu Jun 13 2019 10:15:59 GMT+0800 (China Standard Time)

(a) Step 1: monitor chart and statistics



```
run time: 12 hour 0 min 0 sec
cycles done: 15
total path: 2829
unique crashes: 0
unique hangs: 0
execs done: 35587854
map density: 9.18%
max depth: 25
```

(b) Step 2: analyze call graph



```
608   // smaller.
609   // Conditions for using SearchBitState:
610   const int MaxBitStateProg = 500;   // prog_->size() <= Max.
611   const int MaxBitStateVector = 256*1024;  // bit vector size <= M
612   bool can_bit_state = prog_->size() <= MaxBitStateProg;
613   int bit_state_text_max = MaxBitStateVector / prog_->size();
614
615   bool dfa_failed = false;
616   switch (re_anchor) {
617     default:
618     case UNANCHORED: {
619       if (!prog_->SearchDFA(subtext, text, anchor, kind,
620                             matchp, &dfa_failed, NULL)) {
621         if (dfa_failed) {
622           // Fall back to NFA below.
623           skipped_test = true;
624           if (FLAGS_trace_re2)
625             LOG(INFO) << "Match " << trunc(pattern_)
626                       << " [" << CEscape(subtext) << "]"
627                       << " DFA failed.";
628           break;
629         }
630         if (FLAGS_trace_re2)
631           LOG(INFO) << "Match " << trunc(pattern_)
632                     << " [" << CEscape(subtext) << "]"
633                     << " used DFA - no match.";
634         return false;
635       }
636       if (FLAGS_trace_re2)
637         LOG(INFO) << "Match " << trunc(pattern_)
```

(c) Step 2: analyze control flow graph

Fig. 2: Main workflow and interfaces of VisFuzz

will be displayed. Users leverage those information to find a caller of seldom-hit functions. In this case, function `Match` in `re2.cc` is a potential bottleneck. Next, users should drill down into the function and figure out the seldom-hit basic block, and further than that, the corresponding lines in the source code. When double clicking the node in call graph, the control flow graph and the corresponding source code of this function pops up as Fig. 2(c) shows. Similar to the call graph, the nodes and edge in control flow graph represents basic block and relation between two basic blocks. When the mouse hovers on a specific node, the corresponding basic block information will be displayed. In this case, users find `re_anchor` is never a `UNANCHORED` state in `re2.cc:618`. The semantic context guides the users to find a misconfigured regular expression engine: the default test driver only accepts fully-matched pattern, and the automaton is prevented from entering the `UNANCHORED` state.

**Step 3: intervene fuzzing**. After pinpointing the bottleneck statement and figuring out why the statement is blocked, users should intervene fuzzing process. In this case, users update the test driver to enable partially matched regular expressions and continue fuzzing. Experiments shows it gains 20.85% higher path coverage. See section IV for details.

## III. VisFuzz Design

VisFuzz includes three folds. First, VisFuzz extracts call graph and control flow graph from source code and maps each function and basic block to the line of source code. Second, VisFuzz tracks real-time execution statistics. Third, VisFuzz provides a level-of-detail visualization to help test engineer pinpoint the bottleneck constraints. The three folds are accomplished in three stages, as presented in Figure 3.

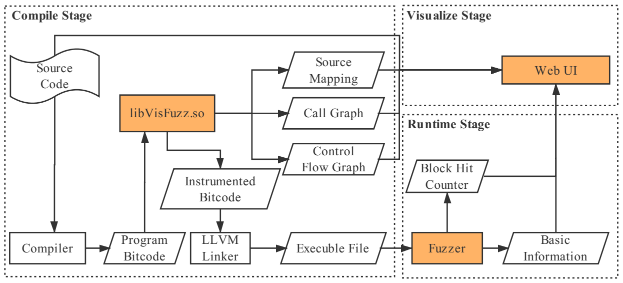


Fig. 3: The design of VisFuzz

### A. Compile Stage

Modern compilation systems separates compilation units and utilizes parallelism to accelerate compilation. However, a global view of the program is the prerequisite of obtaining the calling relations. In this stage, VisFuzz leverages the LLVM platform to combine all compilation modules into a whole-program bitcode. Graph extraction and instrumentation are integrated in the compiled library file `libVisFuzz.so`.

**Function summary extraction**. VisFuzz first uniquely labels each function, and generates a summary for it. The summary contains the file name and line range of the given function. This is obtained by scanning the debug metadata of the basic blocks inside the function.

**Graph construction**. With the unique labels for functions available, VisFuzz constructs the call graph. This is done by scanning the basic blocks of each function and checking for function invocation instructions inside the basic block. Moreover, use of function pointers are also recorded for further use. Control flow graph is constructed in a similar method.

**Basic block instrumentation**. After constructing the graphs above, VisFuzz instruments basic blocks. Similar to AFL, VisFuzz updates the coverage statistics indexed by hashing the random identifiers of the previous block and current block. More than AFL, VisFuzz introduces an array for storing the hit counts for each basic block, which is used by the visualization part. For accurate counting, the instrumentation not only increases the counter, but also saturates to prevent wrapping on overflow.

### B. Runtime Stage

**Shared memory initialization**. The runtime part of VisFuzz is a static library linked into the target program. It registers in the initialization section of the compiled executable file to perform early initialization. When the target program runs, the initialization code attaches to the XSI shared memory owned by the fuzzer. Two shared memories regions are attached — one is the bitmap for fuzzing, and another is the block hit counts.

**Statistics collection**. The fuzzer part of VisFuzz is an extension of AFL. When an input is generated, it sends the input to the target program. After the completion, it reads the bitmap to guide fuzzing just as what AFL does. Additionally, it reads the block counters to update the global counters, then resets the original counters. The operation avoids saturation of the counters: to lower the overhead, the counters should be kept in cache, and the word size of each counter is limited to 16 bits. All fuzzing statistics, including block hit counters and basic fuzzer information (e.g. paths coverage, crash number, etc.), are sent to visualization component.

### C. Visualize Stage

**Context visualization**. We visualize static analysis results (collected in compile stage) along with statistics (collected in runtime stage) in a web application. It uses HTML5 for presentation, Bootstrap for responsive web design, and D3 [8] for data-driven visualization. Specially, we present call graph as a force-directed graph [9] for intuitive presentation. Another optimization is hiding the descents of unexplored functions, which is unrelated to the locating procedure. Besides, we present control flow graph based on Reingold-Tilford tree layout [10] in order to emphasize the hierarchy of basic blocks.

## IV. Evaluation

We evaluate VisFuzz on typical real projects — re2, json, pcre2 and libpng — all selected from Google fuzzer-test-suite. The evaluation is conducted in three stages. In the first trial, AFL is run on the target programs to collect the baseline. Next, a test engineer who is familiar with AFL but without any prior knowledge about the target program is given 15 minutes to understand the performance by VisFuzz. Then the engineer forms a improved version by constructing targeted seeds or updating the driver. In the final trial, AFL is rerun on the improved version. For fair comparison, the final trial is prohibited to reuse seeds and other data from the previous trial, and both trials are given 12 hours to run on a single core (Intel Core i7-8700K @ 3.70GHz).

Table III shows the number of paths, blocks and unique crashes two tool detected. From the second column and the third column, we observe that VisFuzz increases the covered paths by 10.84% on json, 150.58% on libpng, 20.86% on re2, 89.36% on pcre2 compared to AFL. From the forth column and the fifth column, we observe that VisFuzz increases the covered blocks by 1.23% on json, 35.35% on libpng, 14.49% on re2, 65.99% on pcre2 compared to AFL. From the sixth column and the seventh column, we observe that VisFuzz finds 57 more unique crashes than AFL. From these comparisons and statistics, we conclude that VisFuzz helps test engineers spend little time (15 minutes) to gain more coverage, trigger more crashes.

TABLE III: Performance on fuzzer-test-suite

| Project | Paths | | Blocks | | Unique Crashes | |
|---|---|---|---|---|---|---|
| | VisFuzz | AFL | VisFuzz | AFL | VisFuzz | AFL |
| json | 644 | 581 | 1566 | 1547 | 1 | 1 |
| libpng | 651 | 259 | 1206 | 891 | 0 | 0 |
| re2 | 3419 | 2829 | 6888 | 6016 | 0 | 0 |
| pcre2 | 16804 | 8874 | 11259 | 6783 | 106 | 49 |

The reason of performance improvement is that VisFuzz guides the test engineer to locate the bottleneck constraints, also helps him understand semantic context of the bottleneck. Bottleneck constraints found by the test engineer are presented in Table IV. In the follows, we illustrate interventions the test engineer takes in details.

- **json**. The test engineer finds AFL never satisfies the condition in `json.hpp:11044`, which checks whether a string is a unicode with high surrogate followed by a low surrogate. Thus the test engineer constructs a json file containing such a pair of surrogates as initial seeds for a new fuzzing session.
- **linpng**. The test engineer finds AFL always generates invalid seeds which not meet the png specifications because of the CRC check in `pngrutil.c:2069`. Thus the test engineer disables checksum check by setting `PNG_FLAG_CRC_CRITICAL_MASK` for new a fuzzing session.
- **re2**. The test engineer finds `re_anchor` is never a `UNANCHORED` state in deterministic finite automaton in `re2.cc:618` because the default test driver only accepts the fully-matched pattern. Thus the test engineer updates the test driver to enable partially matched regular expressions and continue fuzzing.
- **pcre2**. The test engineer finds the invariable compilation option prevents fuzzer from satisfying condition in `pcre_compile.c:2792`. Thus the test engineer updates the test driver so that the option is altered by inputs.

TABLE IV: Bottlenecks on fuzzer-test-suite

| Project | Bottleneck Constraints | Bottleneck Details |
|---|---|---|
| json | json.hpp:11044 | UTF-16 checker |
| libpng | pngrutil.c:2069 | CRC checker |
| re2 | re2.cc:618 | Unexplored state in automaton |
| pcre2 | pcre_compile.c:2792 | Unexplored mode of compilation |

## V. Related Works

Several approaches have been proposed to versatilely improve fuzzing performance based on AFL. For example, PAFL [11] utilizes guiding information synchronization and task division to extend existing optimizations of single mode to parallel mode. SAFL[12] integrates symbolic execution to help fuzzer with high-quality initial seeds. Besides, several recent techniques[13], [14] integrate multiple fuzzers and obtain significant improvements.

Fuzzing techniques mentioned above focus on automatically increasing testing coverage. A common paradigm of them is that a test engineer writes test driver, the computer performs fuzzing and the test engineer analyzes the results. HaCRS [6] tries to inject human assistants into fuzzing stage through dividing hard-to-solve problems into some game-like tasks. It shifts human-driven fuzzing paradigm to human-assisted fuzzing paradigm.

**Main difference**. VisFuzz follows human-assisted fuzzing paradigm. Comparing all automatic fuzzing techniques, VisFuzz leverages human knowledge to improve performance. VisFuzz is easily integrated with those automatic fuzzing tools because it is orthogonal to them. Comparing HaCRS, VisFuzz provides a more applicable approach for real-world programs. HaCRS is custom designed for CGC binaries, which makes it barely enable to gain enough information to display. In contrast, VisFuzz focuses on binaries built from source code of real-world programs, which contributes to gaining multi-dimension information. Besides, VisFuzz is more applicable because of the cost-effective build procedure. It can be applied to many existing fuzzers such as AFLFast, FairFuzz, PAFL for better performance without extra efforts.

## VI. Conclusion

In this paper, we present VisFuzz to help test engineers to locate the boundary of unexplored regions, understand the semantic context around the bottleneck and intervene the fuzzing process (e.g. construct target input, update test driver) to achieve higher coverage. It is cost-effective to adapt to real-word programs. Experimental results show that VisFuzz helps test engineers spend little time to achieve higher coverage and hunt more vulnerabilities. Our future work would focus on providing some automatic guidelines and templates for the driver generation to pass the bottleneck.

## References

[1] "American fuzzy lop (afl)." [Online]. Available: http://lcamtuf.coredump.cx/afl/

[2] "libfuzzer a library for coverage-guided fuzz testing." [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[3] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 CCS, Vienna, Austria, October 24-28, 2016*, 2016, pp. 1032–1043.

[4] C. Lemieux and K. Sen, "Fairfuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ASE 2018, Montpellier, France, September 3-7, 2018*, 2018, pp. 475–485.

[5] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *23rd NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[6] Y. Shoshitaishvili, M. Weissbacher, L. Dresel, C. Salls, R. Wang, C. Kruegel, and G. Vigna, "Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance," in *Proceedings of the 2017 CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017, pp. 347–362.

[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, 2008, pp. 209–224.

[8] "D3.js - data-driven documents." [Online]. Available: https://d3js.org/

[9] T. M. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[10] E. M. Reingold and J. S. Tilford, "Tidier drawings of trees," *IEEE Trans. Software Eng.*, vol. 7, no. 2, pp. 223–228, 1981.

[11] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "PAFL: extend fuzzing optimizations of single mode to industrial parallel mode," in *Proceedings of the 2018 FSE, Lake Buena Vista, FL, USA, November 04-09, 2018*, 2018, pp. 809–814.

[12] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 61–64.

[13] J. Liang, Y. Chen, M. Wang, Y. Jiang, Z. Yang, C. Sun, X. Jiao, and J. Sun, "Engineering a better fuzzer with synergically integrated optimizations," in *30th ISSRE 2019, Berlin, Germany, Oct 28-31, 2019*, 2019.

[14] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium 2019: Santa Clara, CA, USA*, 2019.