Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling

Chijin Zhou
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China
zcj18@mails.tsinghua.edu.cn

Mingzhe Wang
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China
wmzhere@gmail.com

Jie Liang
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China
liangjie.mailbox.cn@gmail.com

Zhe Liu
Computer Science and Technology
NUAA
Nanjing, China
zhe.liu@nuaa.edu.cn

Yu Jiang*
KLISS, BNRist, School of Software
Tsinghua University
Beijing, China
jiangyu198964@126.com

ABSTRACT

Coverage-guided fuzzing is one of the most popular software testing techniques for vulnerability detection. While effective, current fuzzing methods suffer from significant performance penalty due to instrumentation overhead, which limits its practical use. Existing solutions improve the fuzzing speed by decreasing instrumentation overheads but sacrificing coverage accuracy, which results in unstable performance of vulnerability detection.

In this paper, we propose a coverage-sensitive tracing and scheduling framework `Zeror` that can improve the performance of existing fuzzers, especially in their speed and vulnerability detection. The `Zeror` is mainly made up of two parts: (1) a self-modifying tracing mechanism to provide a zero-overhead instrumentation for more effective coverage collection, and (2) a real-time scheduling mechanism to support adaptive switch between the zero-overhead instrumented binary and the fully instrumented binary for better vulnerability detection. In this way, `Zeror` is able to decrease collection overhead and preserve fine-grained coverage for guidance.

For evaluation, we implement a prototype of `Zeror` and evaluate it on Google fuzzer-test-suite, which consists of 24 widely-used applications. The results show that `Zeror` performs better than existing fuzzing speed-up frameworks such as Untracer and INSTRIM, improves the execution speed of the state-of-the-art fuzzers such as AFL and MOPT by 159.80%, helps them achieve better coverage (averagely 10.14% for AFL, 6.91% for MOPT) and detect vulnerabilities faster (averagely 29.00% for AFL, 46.99% for MOPT).

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

*Yu Jiang is the corresponding author.

ASE '20, September 21–25, 2020, VirtualEvent, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416572

KEYWORDS

Coverage-guided Fuzzing, Coverage-Sensitive Tracing, Scheduling

ACM Reference Format:
Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed Up Fuzzing with Coverage-sensitive Tracing and Scheduling. In *35th IEEE/ACM International Conference on Automated SoftwareEngineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416572

1 INTRODUCTION

Coverage-guided fuzzing is one of the most popular software testing techniques for bug detection. In the past few years, it has gained significant traction in academic research as well as in industry practice. Most notably, Google's OSS-Fuzz [18] adopts American Fuzzy Lop (AFL) [25], honggfuzz [21] and libFuzzer [34] to continuously test open source applications. Over 16,000 bugs in 250 open source projects are discovered by OSS-Fuzz.

A coverage-guided fuzzer feeds a program with random test cases, collects coverage-increasing test cases (such test cases are called interesting seeds), and generates new test cases by mutating those seeds. The key goal of coverage-guided fuzzers is to maximize coverage and explore deeper paths as fast as possible. Many fuzzing optimizations have been proposed to maximize coverage, including the ones that improve seed selection strategy [5, 14, 41, 42] or mutation strategy [6, 28, 29, 36], the ones that integrate multiple fuzzing optimizations [9, 30, 32], and the ones that leverage taint analysis [2, 7, 8, 41], symbolic execution [40, 46, 49, 52, 53], human knowledge [1, 45, 54], or machine learning [10, 16, 44] to assist fuzzing.

While those above optimizations greatly improve performance, especially in coverage improvements, they do not take fuzzing overhead into consideration, which may hinder them from achieving better scalability. For example, the overhead caused by coverage collection is costly. We conduct experiments on AFL using real-world programs of Google fuzzer-test-suite [17] to investigate the overhead of collecting coverage. To our surprise, AFL spends an average of 71.85% and up to 98.5% of its runtime to trace coverage. Some related works try to decrease overheads from instrumentation. INSTRIM [22] reduces instrumentation cost by instrumenting

a part of basic blocks and reconstructing coverage information. Untracer [39] avoids tracing coverage of non-coverage-increasing test cases by removing visited instrumentation points. They can effectively decrease overhead but cannot preserve fine-grained coverage guidance, which limits their vulnerability detection.

To speed up fuzzing and further improve vulnerability discovery, the main challenge is to keep a good balance between instrumentation overheads and the granularity of the collected coverage. Those existing overhead reduction methodologies decrease the overhead with sacrificing coverage accuracy. For example, our experiments demonstrate that compared with AFL, although improves the speed by 155.75%, Untracer decreases coverage by 8.31%, which results in an unstable ability of vulnerability discovery. Therefore, it is not easy to keep a good balance between overhead reduction and coverage accuracy.

In this paper, we propose a coverage-sensitive tracing and scheduling framework `Zeror` , which aims at increasing fuzzing speed with diversely-instrumented binaries. The main idea is switching to a self-modifying based zero-overhead-instrumented binary for fuzzing when the normal instrumented binary fails to make better progress. `Zeror` is mainly made up of two parts: (1) A self-modifying tracing mechanism to provide a zero-overhead instrumentation for coverage collection. The self-modifying tracing mechanism reduces the coverage collection overhead by restricting coverage tracing to only coverage-increasing test cases. (2) A real-time scheduling mechanism to support adaptive switch between the zero-overhead instrumented binary and the fully instrumented binary. To choose the optimal binary, it estimates the probabilities of discovering interesting seeds for each binary by Bayesian inference. Instead of doing a tradeoff between fuzzing speed and coverage accuracy within a single binary, the scheduler helps fuzzers achieve both by taking advantages of diversely-instrumented binaries.

We implemented the prototype of `Zeror` and applied it to several state-of-the-art fuzzers, including AFL [25] and MOPT [36]. We evaluated them on Google fuzzer-test-suite, which consists of 24 widely-used real-world applications. The evaluation results demonstrate that `Zeror` performs better than existing fuzzing speed up frameworks such as Untracer and INSTRIM. Compared with Untracer, it covers 20.84% more branches with almost the same execution time. Compared with INSTRIM, it covers 6.82% more branches with 50.72% less execution time. It improves the execution speed of original AFL instrumentation, which is also adopted in MOPT, by 159.80%, helps them achieve better coverage (averagely 10.14% for AFL, 6.91% for MOPT) and exposure vulnerabilities faster (averagely 29.00% for AFL, 46.99% for MOPT).

In summary, this paper makes following contributions:

- We propose a coverage-sensitive tracing and scheduling framework, which integrates diversely-instrumented binaries and supports adaptive switch between them, to speed up fuzzing as well as maintain the vulnerability detection ability.
- We propose a self-modifying tracing mechanism to reduce coverage collection overhead. By using this mechanism, fuzzers will be sensitive to edge-level coverage granularity and only trace coverage of coverage-increasing test cases.
- We propose a real-time scheduling mechanism, which is able to dynamically choose a proper instrumented binary for fuzzing execution to achieve both speed and accuracy.
- We implemented the prototype of `Zeror`, which could be applied to most of the state-of-the-art fuzzers such as AFL and MOPT. The results show that `Zeror` could help boost execution speed and discover vulnerabilities faster than the existing speed-up framework such as Untracer and INSTRIM.

This paper is organized as follows: Section 2 introduces the background of coverage-guided fuzzing and coverage tracing. Section 3 illustrates the motivation of this work through an empirical study on efficiencies of different coverage collection methods. Section 4 elaborates the idea and design of `Zeror`. Section 5 presents the implementation and evaluation. Section 6 shows some related works and the main differences, and we get the conclusion in Section 7.

2 BACKGROUND

2.1 Coverage-guided Fuzzing

Coverage-guided fuzzing is currently one of the most effective and efficient vulnerability discovery solution. It aims to automatically generate proof of concept (PoC) exploits by maximizing code coverage. AFL [25], libFuzzer [34] and honggfuzz [21] are some well-recognized coverage-guided fuzzers.

Figure 1 shows the general workflow of a coverage-guided fuzzer. Given a target program and initial inputs, fuzzing works as follows: (1) compile target program into target binary, where coverage instrumentation are injected; (2) execute the binary and spawn target process; (3) queue initial inputs into seeds generator; (4) generate test cases as input; (5) trace coverage to evaluate the test case; (6) save the test case to corpus if there is coverage growth (i.e. the test case is interesting), and goto step 4. During the fuzzing execution loop, performance is highly impacted by execution speed during runtime. Fuzzer's runtime consists of two parts, coverage tracing and fuzzer's internal logic (including child process establishment, seed selection and mutation, coverage comparison, etc.). A simple-but-practical optimization for fuzzer's internal logic is AFL persistent mode, where a long-live process can be reused to try out multiple test cases, eliminating the need for repeated fork() calls and the associated OS overhead [26].

2.2 Coverage Tracing

Coverage-guided fuzzers utilize coverage information to guide fuzzing. They track coverage of each execution, compare the coverage with preserved coverage, and check whether current test case is coverage-increasing. The most common approach to gain coverage information for fuzzing is instrumentation, which is taken variously by different fuzzers. For OS kernel fuzzing, Syzkaller [47] and kAFL [43] instrument target kernel by hardware-assisted mechanisms (e.g. Intel PT [23]). For blackbox (source-unavailable) applications fuzzing, VUzzer [41] uses PIN [35] to dynamically instrument black-box binaries. For whitebox (source-available) applications fuzzing, libFuzzer and honggfuzz use SanitizerCoverage [19] instrumentation method provided by Clang compiler, and AFL implements instrumentation by hardcoding basic-block keys into the assembly file of target programs.

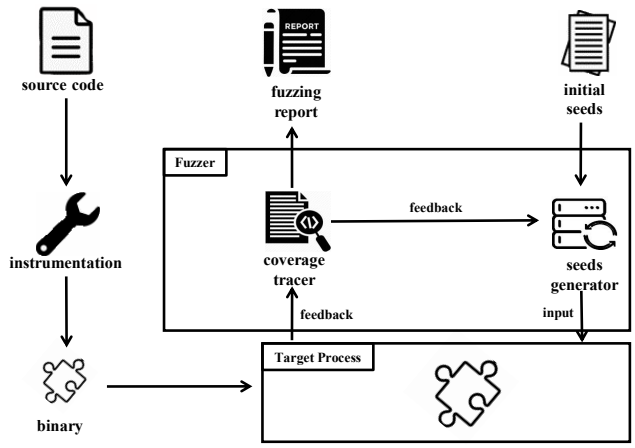


Figure 1: The general workflow of coverage-guided fuzzing

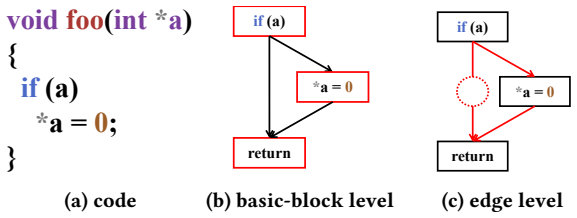


Figure 2: Different coverage granularities provided by SanitizerCoverage. Basic-block level focuses on the coverage of each node, while edge level focuses on the coverage of the edge. Furthermore, an empty "dummy" block is inserted to denote a critical edge between two basic blocks.

Different instrumentation mechanisms provide different coverage granularities. SanitizerCoverage and AFL instrumentation method are two most widely-used coverage instrumentation mechanisms. SanitizerCoverage offers basic-block level and edge level instrumentation. Figure 2 illustrates the mechanisms in a brief example. Basic blocks are the nodes of program's control-flow graph, denoting a piece of straight line code (i.e. there is no jump in or out of the middle of a block). SanitizerCoverage extracts control-flow graph of target program and instruments each basic block in LLVM IR when the basic-block level instrumentation is activated. To enhance instrumentation from basic-block level to edge level, SanitizerCoverage adds "dummy" blocks to denote critical edges, which is neither the only edge leaving its source block, nor the only edge entering its destination block. Unlike SanitizerCoverage, AFL instrumentation method tracks edge coverage directly. It assigns random keys to target program's basic blocks during static instrumentation, dynamically calculates edge keys through *previous* basic-block keys and *current* basic-block keys, and tracks edge counters in a 64K hash table by edge keys [14, 25]. AFL is also compatible with SanitizerCoverage [26].

3 MOTIVATIONS

Different coverage collection mechanisms trace different coverage granularities. The more accurate information gains through tracing coverage, the more overheads fuzzing faces. However, it is unclear how granularity relates to tracing coverage and overhead. An intuitive impression is that, fuzzers guided by different coverage granularities have different strengths when fuzzing different target programs. To verify our hypothesis, we conducted a preliminary experiment on different coverage granularities to evaluate each granularity's efficiency. Three different coverage collection instrumentation mechanisms are chosen in our experiment:

- **AFL (edge)**: the fuzzer is AFL and target programs are instrumented by original AFL's edge level instrumentation.
- **AFL (basic-block)**: the fuzzer is AFL and target programs are instrumented by SanitizerCoverage, using basic-block level instrumentation.
- **AFL (coarse-basic-block)**: the fuzzer is AFL and the target programs are instrumented by Untracer [39], which decreases time on handling discarded test cases but only obtains coarse basic-block level coverage without accumulating hit count.

We run above three mechanisms on Google fuzzer-test-suite [17] for 6 hours and select partial results for preliminary illustration (all experiment settings are in line with Section 5.1). From the result of Figure 3 and Table 1, we have the following observations:

Observation 1: tracing accurate coverage is costly. As illustrated in Section 2.1, coverage tracing and internal logic execution are two constituent part of fuzzer's runtime. We record AFL internal logic execution time during each iteration, and calculate edge level coverage tracing time by comparing each test case's execution time in instrumented version and non-instrumented version. As Figure 3 shows, time spent in tracing coverage accounts for averagely 71.85% of AFL's whole runtime. The ratio is even up to 98.5% when fuzzing `openssl-1.0.1f`.

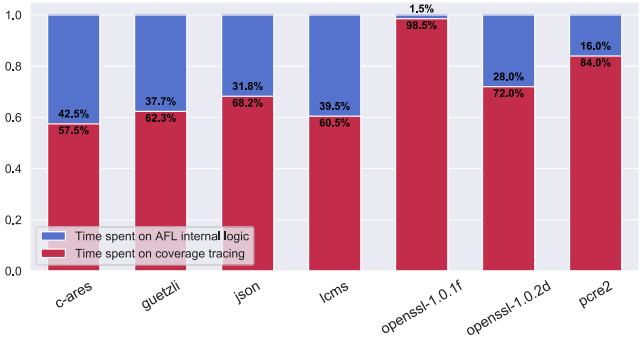


Figure 3: Percentage of internal logic execution time and edge level coverage tracing time in AFL.

Observation 2: the efficiency of each coverage granularity varies with target programs. We record the time spent in triggering known vulnerabilities for each mechanism, and the result is shown in Table 1. Due to the limitation of Dyninst [13], Untracer is incompatible with some projects (denote as N/A). From Table 1, we can see that: AFL (edge) exposes known vulnerabilities

faster than others on `openssl-1.0.1f` and `openssl-1.0.2d`; AFL (coarse-basic-block) exposes known vulnerabilities faster than others on `guetzli`. AFL (basic-block) exposes known vulnerabilities faster than others on `lcms`, `pcre2`.

Table 1: Time taken to trigger known bugs for fuzzers guided by different coverage granularities. ∞ denotes the fuzzer cannot expose known bugs in 6 hours. N/A denotes compatibility issues of Untracer on specific programs.

Project	Average Reaching Time (seconds)		
	AFL (edge)	AFL (basic-block)	AFL+Untracer (coarse-basic-block)
c-ares	**5**	**5**	842
guetzli	∞	∞	**16257**
json	**5**	6	**5**
lcms	20679	**4084**	11827
openssl-1.0.1f	**19**	31	N/A
openssl-1.0.2d	**8716**	10407	N/A
pcre2	822	**413**	6095

Focus of this Paper: From the observation 1, we find that tracing coverage is costly. In search for coverage-increasing test cases, fuzzing is based on genetic algorithm, which makes its effectiveness highly impacted by execution speed. Thus, we focus on improving fuzzing efficiency by reducing the coverage collection overhead. We propose a novel self-modifying tracing mechanism to eliminate needless coverage collection. Besides, inspired by the observation 2, instead of doing a tradeoff between fuzzing speed and coverage accuracy, we propose a scheduling scheme, which helps fuzzers achieve both goals by integrating diversely-instrumented binaries.

4 ZEROR DESIGN

Figure 4 depicts the basic work flow and main components of `Zeror`. Different from traditional coverage-guided fuzzing, `Zeror` will choose a proper binary as fuzzing target (i.e. the running program for fuzzing) among diversely-instrumented binaries. `Zeror` consists of two main components : *coverage tracer* and *binary-switching scheduler*. (1) *Coverage tracer* collects coverage information from fuzzing target, stores seeds into corpus if the seeds are interesting and sends statistical data to *binary-switching scheduler*. It will self-adjust when fuzzing target changes: when fuzzing AFL-instrumented binaries, *coverage tracer* will read coverage from edge-counters hash table; when fuzzing the binaries instrumented by self-modifying tracing, *coverage tracer* will monitor the status of child process and modify the instructions of child process. (2) *Binary-switching scheduler* records the statistical data from *coverage tracer*, estimates efficiency of each instrumented binary based on the statistical data and choose the optimal binary as fuzzing target when time to switch binary. Specially, we leverage empirical Bayesian method to estimate efficiency in a cost-effective way and adopt exponential smoothing to smooth the time-varying efficiency.

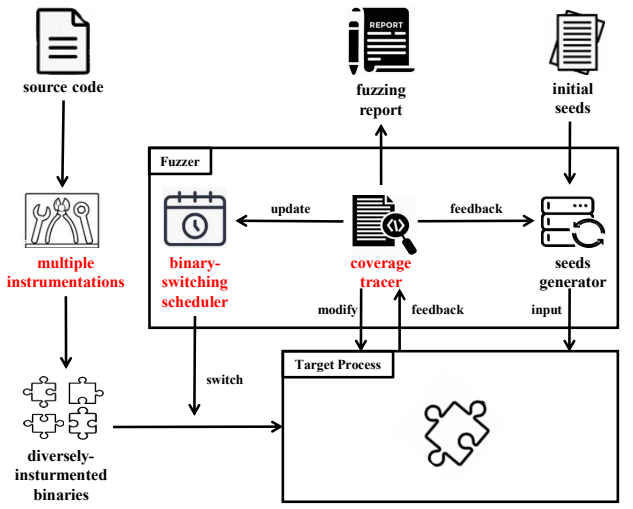


Figure 4: Overview of `Zeror`, which mainly includes the self-modifying tracing mechanism implemented with multiple instrumentation and coverage tracer, and the real-time scheduling mechanism implemented with the binary-switching scheduler. Multiple instrumentation means the self-modifying tracing based instrumentation and the full instrumentation of the integrated original fuzzer.

4.1 Self-modifying Tracing

As aforementioned, coverage-guided fuzzing spends the majority of its runtime in collecting coverage. It is intuitive that restricting coverage tracing to only coverage-increasing test cases will significantly reduce the overhead. However, how to sense coverage-increasing seeds and ignore discarded test cases is still an open problem. Different with static binary rewriting technique used in Untracer [39], which is coverage-inaccurate, time-consuming and not scalable on many complex programs, our solution, namely self-modifying tracing, adopts self-modifying code technique to address the problem. With the assistance of self-modifying tracing, fuzzers could (1) dynamically remove visited instrumentation points during fuzzing process; (2) sense fine-grained coverage; (3) barely introduce new overhead.

Self-modifying code (SMC) refers to the code that can modify its own instructions during the execution of the program. It is widely used in many of software systems to support runtime code generation [27, 37] and optimization [3], minimize the code size [11], and reinforce dynamic code encryption and obfuscation [24]. There are several advantages in SMC, such as fast paths establishment, repetitive conditional branches reduction and algorithmic efficiency improvement. To apply SMC to coverage tracing, we need to obtain the addresses of instrumentation points at compilation stage, and self-modifying the addresses at runtime stage. A step-by-step example is shown in Figure 5 to elaborate how our solution performs self-modifying tracing with compilation stage and runtime stage.

At compilation stage, we need to generate a zero-overhead binary and obtain the addresses of instrumentation points. However,

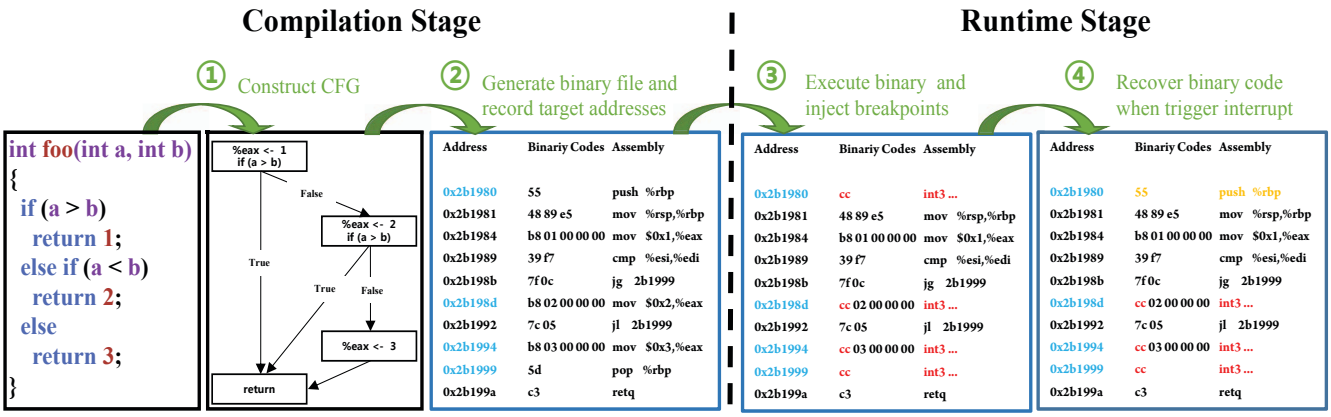


Figure 5: A step-by-step demonstration of self-modifying tracing. It eliminates needless overhead spent in tracing coverage of non-coverage-increasing test cases with two stages. It first instruments target programs, obtains addresses of instrumentation points and generates a non-instrumented executable binary file at compilation stage. Then, it does fuzz testing on the binary, detects whether instrumentation points are triggered and removes visited instrumentation points at runtime stage. (The segments in blue rectangles is the text segments of the program's memory layout, the addresses of instrumentation points are highlighted in blue, the modified instructions are highlighted in red, the recovered instructions are highlighted in orange.)

there are two challenges to be addressed: (1) *How to inject instrumentation points into target program?* Blackbox instrumentation will obtain redundant and less-accurate coverage information, which impair fuzzing performance. While instrumenting programs in a white-box way like AFL instrumentation [25] or SanitizerCoverage [19] will introduce costly overhead. Besides, using self-modifying code based on AFL instrumentation also obtains coarse-grained coverage because AFL only injects instrumentation points into basic blocks. Thus, an instrumentation approach which obtains fine-grained coverage and introduces less overhead is demanded. (2) *How to track the addresses of instrumentation points?* Compilers will deactivate some code optimizations as soon as any address of basic block is obtained, and the un-optimized binary will be executed at a low speed. Thus, we need to track the addresses of instrumentation points in a proper way.

To generate a zero-overhead binary and track the addresses of instrumentation points, it works as follows to compile a program from source code to object file:

- *Inject instrumentation points.* Before the compiler starts performing platform-independent code optimizations, we construct control flow graph and inject an instrumentation point, i.e. a `CALL` instruction to invoke callback function, at the start of each basic block. Note that, similar with SanitizerCoverage, the instrumentation could be enhanced from basic-block level to edge level by adding "dummy" blocks to denote critical edges as Section 2.2 illustrates. Instrumenting before code optimizations allows control flow graph to preserve semantics of source code so that coverage information is collected accurately.
- *Record & Clear.* We record the corresponding basic block symbols of injected `CALL` instructions and erase all the injected `CALL` instructions after compiler finishes platform-independent code optimizations at intermediate representation (IR) level. In this way, the generated IR could be non-instrumented while the recorded basic block symbols inherit the fine-grained coverage information from instrumentation points.
- *Emit addresses.* We obtain addresses of instrumentation points through the recorded basic block symbols, allocate a memory in the generated object file and emit the addresses into the memory after compiler finishes platform-dependent code optimizations at machine-specific intermediate representation (MIR) level. Note that, the addresses are a series of offsets in object file and will be relocated to absolute addresses when a linker generates executable binary. In this way, the addresses of instrumentation points are written in generated binary and could be accessed to perform self-modify tracing during runtime.

In the text segments after step 2 of Figure 5, we highlight four addresses (`0x2b1980`, `0x2b198d`, `0x2b1994` and `0x2b1999`) in blue to denote the addresses of instrumentation points. For simplicity, we only show basic-block level instrumentation; however, our solution enhances instrumentation from basic-block level to edge level by adding "dummy" blocks to denote critical edges. After compilation stage, a zero-overhead binary is generated and prepared for fuzzing.

At runtime stage, the coverage tracer of `Zeror` will execute the zero-overhead binary, inject breakpoints into it and perform fuzzing on this target. Algorithm 1 details the actions of the coverage tracer. First, as presented in lines 2-8, the fuzzer executes the binary, receives the addresses of instrumentation points, and replaces original instructions with `0xcc`. The corresponding demonstration is shown in step 3 in Figure 5, the binary codes of instrumentation points are replaced with `0xcc` (we highlight the instructions in red). Once the process executes `0xcc`, it will trigger `SIGTRAP` interrupt, and wait for parent process to resume it. After the injection, the fuzzer performs fuzzing on the child process, and monitors the status of it. Once receiving `SIGTRAP` from child process, the fuzzer stores current input as interesting seed for further mutation, recovers the instruction that belongs to the address, and resumes child process,

as presented in lines 11-18. The corresponding demonstration is shown in step 4 in Figure 5.

Algorithm 1: Action of self-modifying coverage tracer

```
   Input  : the target binary b
            Executor Exec
   /* A map to store (address, instruction) pairs */
1  addr.initial()
2  Exec.run(b)
3  addrs = receiveInstrumentedAddrs()
4  unvisitedAddrs = addrs
   /* Inject breakpoints into child process       */
5  foreach addr in addrs do
6      instr = readInstrFromAddr(addr)
7      addrMap.insert(addr, instr)
8      writeInstrIntoAddr(0xcc, addr)
9  end
10 async event loop
11     if receive SIGTRAP from child process then
12         readSeedAndStore()
           /* Recover the instruction              */
13         addr = readRip()
14         instr = addrMap.get(addr)
15         writeInstrIntoAddr(instr, addr)
16         unvisitedAddrs = unvisitedAddrs − {addr}
17         Exec.resume()
18     end
19 end
```

Within the self-modifying tracing, we maintain a set of instrumentation points which have never been visited (`unvisitedAddrs` in Algorithm 1) during fuzzing process. The set will tend to be an empty set as the fuzzer explores target program's states more deeply. Once a instrumentation point is visited, it will be removed and never be collected again. Besides, the self-modifying tracing does not introduce new overhead during fuzzing process. Therefore, along with the fuzzing process, it can theoretically eliminate coverage collection overhead almost down to zero.

4.2 Binary-switching Scheduling

Section 3 reveals that the efficiency of each coverage granularity varies with target programs. Inspired by this, we believe that switching among diversely-instrumented binaries during fuzzing process will improve fuzzing performance. However, estimating efficiencies of diversely-instrumented binaries is challenging, because: (1) *program-dependent efficiency*: the efficiency of each binary varies with target programs, thus we cannot share one static set of parameters configuration among different programs; (2) *time-varying efficiency*: even for testing one target program, the efficiency of each coverage granularity changes over time as the fuzzer explores target program's states more deeply; (3) *cost-effective solution*: the solution should be cost-effective and less-frequent due to the high throughput of fuzzing.

We propose a real-time scheduling mechanism to address above problems. In short, it adaptively switches fuzzing binary among diversely-instrumented binaries at set intervals. During fuzzing process, it collects statistical data (i.e. the number of interesting seeds, the number of executions and the time spent on fuzzing), dynamically monitors the number of interesting seeds each binary could discover, and choose an optimal binary as fuzzing target when the switch time is up. We leverage empirical Bayesian method to estimate efficiency in a cost-effective way and adopt exponential smoothing to smooth the time-varying efficiency.

Estimate efficiency. To simplify the time-varying problem, we discretize continuous time into time periods and assume efficiency is invariant at each time period. For a binary, the efficiency at time period t is defined as

$$\begin{aligned} e_t =& \frac{I_t}{T_t} \\ =& \frac{I_t}{M_t} * \frac{M_t}{T_t} = r_t * s \end{aligned} \tag{1}$$

where I_t denotes the number of discovered interesting seeds during the time period t, T_t denotes the time spent on fuzzing during the time period t, M_t denotes the number of executions during the time period t, r_t denotes the quotient of I_t and M_t (namely, interesting-testcases rate, **ITR**), and s denotes execution speed which can be seen as a constant with respect to binary. Given a binary's statistical data $[I_1, I_2, \cdots, I_t]$, $[T_1, T_2, \cdots, T_t]$ and $[M_1, M_2, \cdots, M_t]$ before current time period t, we aim to estimate ITRs $\hat{r}_t$, and further calculate the estimation of efficiency $\hat{e}_t$ of the binary at current time period t through equation (1).

With empirical Bayesian methods, the integrals over conditional probability distributions are substituted by the empirical statistics in the observed data, which allows us to estimate the posterior probabilities, e.g. a binary's ITRs, by leveraging the information from its statistical data. For each binary, there is an underlying probability distribution of ITR, and at each time period t, the binary's ITR r_t could be regarded as a outcome of the distribution. We use Beta distribution to parameterize the generative process, defined as $Beta(\alpha, \beta)$. Besides, obviously, for each binary at time period t, the number of interesting seeds I_t obeys the Binomial distribution with parameters M_t and r_t. Thus, we have a Beta-Binomial compound distribution for the statistical data. The generative process of our Bayesian model is described as follows:

- Sample $r \sim Beta(\alpha, \beta)$, $p(r|\alpha, \beta) \propto \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} r^{\alpha-1}(1-r)^{\beta-1}$
- Sample $I \sim Binomial(M, r)$, $p(I|M, r) \propto r^I(1-r)^{M-I}$

where Γ is Gamma function. Therefore, the likelihood over all number of interesting seeds is:

$$\begin{aligned} L =& p(I_1, I_2, \cdots, I_t | M_1, M_2, \cdots, M_t, \alpha, \beta) \\ =& \prod_{i=1}^{t} \int_{r_i} p(I_i|M_i, r_i) p(r_i|\alpha, \beta) dr_i \\ \propto& \prod_{i=1}^{t} \int_{r_i} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} r_i^{I_i}(1-r_i)^{M_i-I_i} r_i^{\alpha-1}(1-r_i)^{\beta-1} dr_i \\ =& \prod_{i=1}^{t} \frac{\Gamma(\alpha+\beta)}{\Gamma(M_i+\alpha+\beta)} \frac{\Gamma(I_i+\alpha)}{\Gamma(\alpha)} \frac{\Gamma(M_i-I_i+\beta)}{\Gamma(\beta)} \end{aligned} \tag{2}$$

Then, the maximum likelihood can be calculated through the fixpoint iteration (FPI) [38, 50]:

$$\begin{aligned}\alpha_{x+1} =& \alpha_x \frac{\sum_{i=1}^{t}[\Psi(I_i+\alpha_x)-\Psi(\alpha_x)]}{\sum_{i=1}^{t}[\Psi(M_i+\alpha_x+\beta_x)-\Psi(\alpha_x+\beta_x)]} \\ \beta_{x+1} =& \beta_x \frac{\sum_{i=1}^{t}[\Psi(M_i-I_i+\beta_x)-\Psi(\beta_x)]}{\sum_{i=1}^{t}[\Psi(M_i+\alpha_x+\beta_x)-\Psi(\alpha_x+\beta_x)]}\end{aligned} \tag{3}$$

where $\Psi(x)$ is the digamma function, and can be quickly calculated through Bernardo's algorithm [4].

With equation (3), the $\hat{\alpha}$ and $\hat{\beta}$ could be iteratively estimated, furthermore, the posterior estimation of current time period's ITR could be calculated as $\hat{r}_t = \frac{I_t+\hat{\alpha}}{M_t+\hat{\alpha}+\hat{\beta}}$. To accelerate the convergence speed of the iteration method, we use method of moments [20] to calculate the initial values $\hat{\alpha}_0$ and $\hat{\beta}_0$. Besides, to smooth time-varying observed data, we leverage exponential smoothing [15] to calculate the smoothed number of interesting seeds:

$$I_i = \begin{cases} I'_i & i = 1 \\ \gamma I'_i + (1-\gamma) I_{i-1} & i > 1 \end{cases} \tag{4}$$

where I'_i is the observed number of interesting seeds, I_i is the smoothed number of interesting seeds which is used in equation (3), $\gamma \in (0, 1)$ is the smoothing factor. As time passes the smoothed I_i becomes the exponentially decreasing weighted average of its past observations, in this way, we can capture time relationship between ITRs.

Once the posterior estimation of ITR $\hat{r}_t$ of the binary is estimated, the estimation of efficiency $\hat{e}_t$ could be calculated through equation (1). Thus, at current time period t, we can estimate efficiencies of every diversely-instrumented binaries $[e_t^1, e_t^2, \cdots, e_t^k]$, and form a probability distribution by normalizing these efficiencies:

$$p(X = i) = \frac{e_t^i}{\sum_{j=1}^{k} e_t^j} \tag{5}$$

where e_t^j denotes the efficiency of binary j. When the time to switch, we can select the target binary for fuzzing according to the probability distribution.

Switch among binaries. Based on the efficiency estimation, we can implement the binary-switching scheduler, as detailed in Algorithm 2. First, as presented in lines 1-3, the scheduler randomly chooses several (in line with the configurations) binaries and performs fuzzing on these binaries through executor. For each binary, the executor will `fork` a child process to test them, which is similar to AFL's fork server [26]. Then, the scheduler asynchronously listens events from executor and timer. Executor will periodically report statistics (number of executions, number of interesting seeds, time spent on fuzzing during the time period), and scheduler will record these statistics when receive them from executor as presented in lines 5-8. As presented in lines 10-18, when it is time to switch binary, the executor will stop its child processes, and then, the scheduler will calculate the posterior estimation of each binary's ITR and choose optimal binaries for fuzzing according to the probability distribution of equation (5). Note that, the scheduler supports not only running in single mode (i.e. single-core fuzzing) but also running in parallel mode (i.e. multi-cores fuzzing), which is more common in real industrial practice [31, 33].

Algorithm 2: Action of binary-switching scheduler

```
   Input : List of diversely-instrumented binaries B
           Executor Exec
           Configurations C
1  scheduler.initial(B)
2  targets = scheduler.chooseRandom(C.numCores)
3  Exec.run(targets);
4  async event loop
5      if receive statistics from executor then
6          binary, statistics = Exec.read()
7          scheduler.record(binary, statistics)
8      end
9      if time to switch binary then
10         Exec.stop()
11         foreach b in B do
               /* calculate the posterior estimation
                  of the binary's ITR                  */
12             α0, β0 = scheduler.calByMoment(b)
13             α, β = scheduler.calByFPI(b, α0, β0)
14             r = betaExpectation(α, β)
15             scheduler.update(b, r)
16         end
17         targets = scheduler.chooseOptimal(C.numCores)
18         Exec.run(targets);
19     end
20 end
```

5 EVALUATION

We implemented the framework `Zeror`. The instrumentation mechanism in self-modifying tracing is implemented on the top of LLVM 10.0.0 [48]. The *Record&Clear* procedure is implemented in the initialization of `llvm::MachineModuleInfo` and the *Emit addresses* procedure is implemented in the `EmitBasicBlockStart` method of `llvm::AsmPrinter`. We create a global variable to record the mapping of MBB Symbol (`MCSymbol*` type) and MBB id (`uint32_t` type). The runtime logic of monitoring status of process and modifying instructions of memory in self-modifying tracing is based on `ptrace`. For scalability, the scheduler component contains the self-modifying based zero-overhead binary and the original fully instrumented binary of the integrated fuzzers such as AFL and MOPT. The interval of switching binaries and reporting statistical data are set to 600s and 60s respectively, which barely introduces new overhead and brings best performance after multiple attempts with different values. Inspired by the AFL persistent mode [26], our framework sets up a thread which runs a `ptrace` task to monitor the status of child process. Once the child process triggers a crash or exceeds timeout limit, the thread will terminate and re-spawn the child process.

We evaluated `Zeror` in three aspects. First, we applied `Zeror` to AFL and compared the performance with two state-of-the-art fuzzing speed up frameworks, Untracer [39] and INSTRIM [22], to assess the efficiency. Then, we generalized `Zeror` to MOPT [36], a

Table 2: Fuzzing performances of different AFL-based fuzzing-speed-up methods.

Project	average execution time for each test case (μs)				number of covered branches			
	AFL	AFL+INSTRIM	AFL+Untracer	AFL+Zeror	AFL	AFL+INSTRIM	AFL+Untracer	AFL+Zeror
boringssl	96.69	69.68	N/A	33.05	2661	**2694**	N/A	2549
c-ares	43.34	25.42	13.95	16.32	**57**	**57**	55	**57**
freetype2	44.68	25.17	25.13	20.33	8255	9268	7007	**10059**
guetzli	99.92	67.98	45.80	41.00	4757	4845	4748	**4987**
harfbuzz	149.82	80.36	66.06	55.73	8148	8048	7195	**9168**
json	145.82	100.03	64.33	98.39	1315	1333	1152	**1346**
lcms	97.71	70.92	44.18	63.96	2115	**2244**	1436	2077
libarchive	193.44	112.50	112.90	112.72	1208	1119	1082	**1618**
libjpeg	1469.47	668.96	261.30	337.36	2364	2564	2399	**2857**
libpng	15.34	5.48	5.27	7.54	1092	1096	1029	**1140**
libssh	638.00	340.52	309.62	309.29	**867**	**867**	**867**	**867**
libxml2	268.07	135.05	N/A	88.13	4063	4318	N/A	**4745**
llvm-libcxxabi	137.61	81.61	43.75	42.04	6488	6005	6000	**7012**
openssl-1.0.1f	3418.66	1998.27	N/A	1948.43	4748	6745	N/A	**7372**
openssl-1.0.2d	161.09	92.48	N/A	63.23	1825	**1828**	N/A	1769
openssl-1.1.0c	210.70	89.74	N/A	50.60	**1712**	1711	N/A	1658
openthread	145.51	91.17	64.80	85.16	3561	3537	3279	**3591**
pcre2	199.12	102.21	53.86	49.11	**6890**	6888	6597	**6890**
proj4	23.22	14.24	8.47	7.86	2541	2584	2347	**3886**
re2	640.24	391.97	260.19	235.40	4608	4647	4533	**4725**
sqlite	221.18	160.84	136.01	141.40	1892	**1997**	1986	1972
vorbis	96.14	58.08	36.45	25.48	2035	**2152**	1817	2079
woff2	31.55	20.12	11.80	8.67	2119	2152	1453	**2157**
wpantund	1921.02	2019.62	1544.89	1789.23	7959	7892	7802	**8781**
Zeror improvement	**+159.80%**	**+50.70%**	**-0.46%**		**+10.14%**	**+6.82%**	**+20.84%**	

Table 3: Time to expose known bugs, ∞ denotes the fuzzer cannot expose the known bugs in 6 hours and the projects whose bugs can not be triggered by any fuzzer are removed.

Project	AFL	AFL+INSTRIM	AFL+Untracer	AFL+Zeror
c-ares	**8**	26	842	**8**
guetzli	∞	∞	16257	**6001**
json	**5**	**5**	**5**	**5**
lcms	20679	∞	11827	**10953**
llvm-libcxxabi	788	2197	2347	**709**
openssl-1.0.1f	**19**	**19**	∞	21
openssl-1.0.2d	8716	6877	∞	**6013**
pcre2	822	1375	6095	**439**
re2	∞	∞	∞	**8194**
woff2	3565	**1535**	∞	3260

state-of-the-art fuzzer, to study the scalability. Finally, we evaluated the effectiveness of each component of `Zeror`.

5.1 Experiment Settings

To reveal the practical performance of `Zeror`, the evaluation was conducted on fuzzer-test-suite [17], a widely-used benchmark from Google. This test suite consists of 24 popular real-world applications which have interesting known vulnerabilities, hard-to-find code paths, or other challenges for bug finding tools. The initial seeds were collected from the built-in test suite and each source code inside the test suite was compiled with `-O2` flag. To reduce the side effect caused by AFL's file I/O overhead [51], all fuzzers were running in `tmpfs`. All experiments were performed on a 64-bit machine with 40 cores (Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz), 128 GiB of RAM and Linux 5.5.13. Due to the random effects in fuzzing, we conducted each experiments for six hours and repeated it ten times. And we reported average performance.

In terms of metrics, we evaluate the performance of fuzzers in three aspects, namely execution time, branch coverage and time to expose known bugs. The execution time is the average time the `LLVMFuzzerTestOneInput` function consumed. Different fuzzers are guided by different coverage granularity, for fair comparison, we collect their generated seeds, feed the seeds to original AFL and gather the number of covered branches through AFL `BITMAP`. The time to expose known bugs is the time consumed by the fuzzer to trigger the first crash.

5.2 Efficiency of `Zeror`

We applied `Zeror` to AFL (namely AFL+Zeror) by switching between AFL-instrumented binary and self-modifying tracing instrumented binary based on binary-switching scheduler. We evaluated it on all the 24 programs of Google fuzzer-test-suite and compared it with two state-of-the-art fuzzing speed-up techniques, INSTRIM and Untracer. Specifically, for the baseline AFL, the version used is 2.52b and the compilation tool chain is `afl-clang-fast` [26], which is the most efficient instrumentation method that AFL provide; for INSTRIM, we activate INSTRIM-APPROX mode, which shows best performance in their evaluations [22].

The results are presented in Table 2 and Table 3. The 2-5 columns of Table 2 show the average execution time per test case and the `Zeror` improvement in the last row refers to the execution speed increase. The 6-9 columns of Table 2 show the number of branches covered by each fuzzer and the `Zeror` improvement in the last row refers to branch increase. Table 3 shows the time taken by each fuzzer to expose known bugs, the projects whose bugs cannot be triggered by all the fuzzers in 6 hours are removed from the table. Note that, due to the limitation of Dyninst [13], Untracer is incompatible with some projects (including `boringssl`, `libxml2`, `openssl-1.0.1f`, `boringssl-1.0.2d` and `openssl-1.1.0c`), we denote the corresponding table cell as N/A. From the two tables, we can deduct the following conclusions:

- `Zeror` increases the execution speed of AFL. In Table 2, the average execution time of AFL+Zeror is less than AFL for every benchmark projects. Specifically for `libjpeg`, the average execution time of AFL and AFL+Zeror are 1469.47μs and 337.36μs respectively, which indicates that `Zeror` increases the execution speed of AFL by 335.58%. Averagely, `Zeror` increases the execution speed of AFL by 159.80%.
- `Zeror` helps AFL cover more branches. In Table 2, AFL+Zeror outperforms AFL on 17 out of 24 projects. Specially, AFL+Zeror improves the number of covered branches by 55.27% on `openssl-1.0.1f` and 33.94% on `libarchive`. Averagely, AFL+Zeror increases the number of covered branches of AFL by 10.14%.
- `Zeror` helps AFL expose bugs faster. In Table 3, AFL+Zeror exposes known bugs faster than original AFL on 8 out of 10 projects. Specially, AFL+Zeror is 1.87x faster than AFL in term of triggering the bug in `pcre2`, and exposes the bugs of `re2` and `guetzli`, which cannot be exposed by original AFL in 6 hours.
- `Zeror` shows better performances compared with other fuzzing speed-up techniques. Compared with INSTRIM, `Zeror` is averagely 50.70% faster for each execution, covers 6.82% more branches and spends less time on bugs exposure. Compared with Untracer, `Zeror` covers 20.84% more branches averagely and spends less time on bugs exposure. Because of the real-time scheduling, `Zeror` is averagely 0.46% slower than Untracer, which is almost negligible.

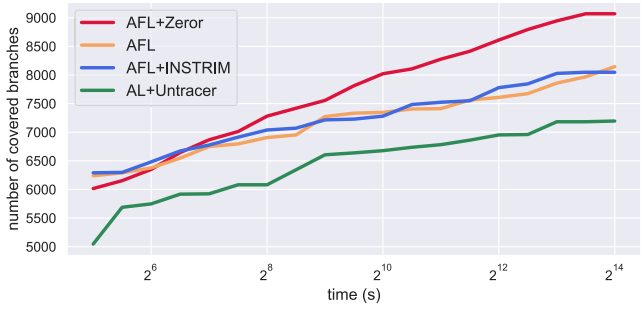


Figure 6: The number of covered branches over time when fuzzing harfbuzz. The x-axis is on a logarithmic scale.

Case study. Figure 6 visualizes the real-time change of covered branches on `harfbuzz` when different fuzzing speed-up methods are applied on AFL. We can observe that AFL+Zeror covers more branches than all the other methods most of the time. Specifically, AFL+Zeror takes 2^{11} seconds to achieve almost the same number of covered branches as AFL and INSTRIM take 2^{14} seconds. Untracer covers less branches most of the time compared with other methods, even compared to the original AFL. As demonstrated in Table 2, Untracer is the fastest for test case execution, but when it deletes almost all the instrumentation points, it will also lose the fine-grained coverage information such as hit count of branches for fuzzing guidance, and will greatly reduce the number of covered branches. INSTRIM makes AFL faster, but not as fast as Untracer and `Zeror`, and it reconstructs the coverage information for guidance with instrumenting a part of basic blocks, to partially maintain the ability to cover more branches.

From the above statistics, it is reasonable to draw the conclusion that: with the aid of `Zeror`, fuzzers are able to gain higher speedup, covers more branches, and exposes bugs faster. In addition, `Zeror` shows better performance of coverage increase and vulnerability discovery compared with other fuzzing speed-up techniques.

5.3 Scalability of `Zeror`

In addition to AFL, we also generalize our experiments to another state-of-the-art fuzzer, MOPT [36], to study the scalability of `Zeror`. MOPT is a fuzzer that improves fuzzing performance by optimizing the efficiency of mutation strategy. We applied `Zeror` to MOPT (namely MOPT+Zeror) in the same way as AFL+Zeror and evaluated it on all the 24 programs of Google fuzzer-test-suite. The results are shown in Figure 7 and Table 4. From Figure 7 we can observe that MOPT+Zeror improves the number of covered branches in 17 out of 24 projects and averagely increases the number of covered branches by 6.91% compared with the original MOPT. Specifically, MOPT+Zeror improves the number by 64.95% on `proj4` and 40.45% on `libarchive`. Table 4 shows the time taken by MOPT and MOPT+Zeror to expose known bugs, those projects whose bugs cannot be triggered by them in 6 hours are removed from the table. From Table 4 we can observe that with the aid of Zeror, MOPT exposes known bugs faster. Specially, Zeror improves the speed of bug exposure by 2.39x on `llvm-libcxxabi`, 2.01x on `pcre2`.

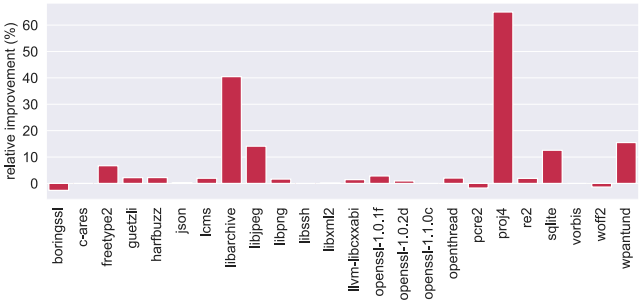


Figure 7: Relative covered branches improvement of MOPT+Zeror compared with MOPT.

Table 4: Time to expose known bugs, and the projects whose bugs cannot be triggered by them in 6 hours are removed.

Project	MOPT	MOPT+Zeror
c-ares	**8**	**8**
json	**5**	**5**
llvm-libcxxabi	1818	**761**
openssl-1.0.1f	31	**21**
openssl-1.0.2d	1633	**1320**
pcre2	1944	**968**
woff2	3767	**3196**

In summary, `Zeror` is applicable to other fuzzing optimizations like MOPT, and more importantly, `Zeror` can further improve fuzzing vulnerability discovery performance on top of them. Although we only use MOPT for illustration in the experiment, it

can be easily applied to other fuzzers such as AFLFast [5] and FairFuzz [28].

5.4 Evaluation of Individual Components

`Zeror` consists of two main mechanisms: self-modifying tracing and real-time scheduling. To analyze the effects of each individual mechanism, we configure two variants of our framework:

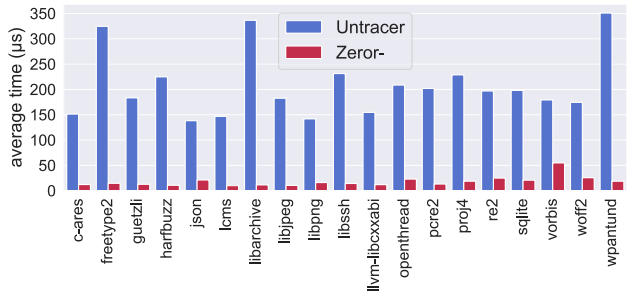
- *Zeror-* represents the fuzzer which adopts AFL as seeds generator and only integrates self-modifying tracing mechanism.
- *Zeror* represents the fuzzer which adopts AFL as seeds generator. Besides, it integrates self-modifying tracing and AFL's instrumentation to collect coverage, and dynamically switches between the two instrumented binaries during fuzzing process based on real-time scheduling mechanism.

Evaluation of self-modifying tracing. Since Untracer [39] shares the similar idea with our self-modifying tracing component, we evaluate our tracing by comparison with Untracer, using 19 projects of fuzzer-test-suite (Untracer is incompatible to the rest 5 projects). For speed improvement, both methods eliminate the coverage-collecting time of non-coverage-increasing test cases by erasing visited instrumentation points, but with different approaches. Figure 8a shows that, when considering erasing instrumentation points, self-modifying tracing saves much more time than Untracer on the average time consumed. Averagely, self-modifying tracing is 13.74x faster than Untracer when erasing instrumentation points. The saved coverage tracing time can be used for efficient binary-switch scheduling. Additionally, self-modifying tracing is edge-aware while Untracer is basic-block-aware. Figure 8b shows the relative covered branches improvement of self-modifying tracing, from which we can conclude that self-modifying tracing mechanism helps fuzzer cover more branches compared with Untracer. Specifically, self-modifying tracing improves the branch coverage by 56.92% on `proj4`, 48.43% on `libarchive`, 43.80% on `lcms`, 42.90% on `freetype2`.

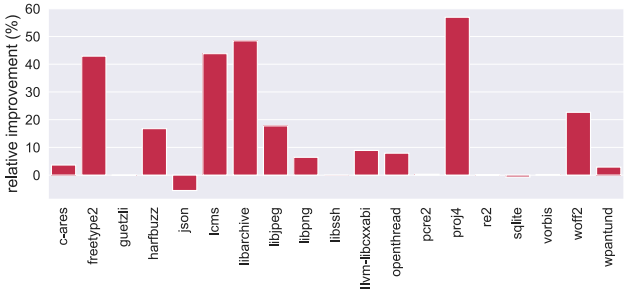
Evaluation of real-time scheduling. Our scheduling mechanism integrates two binaries: the zero-overhead binary instrumented by self-modifying tracing and the original binary instrumented by the integrated fuzzer, and then dynamically switches between them. To study the effectiveness of the scheduler, we compare *Zeror* with *Zeror-* and AFL. The overall result is consistent to Table 2, and for page limitation, we only visualize 2 projects to demonstrate the coverage increase process of different configurations in Figure 9. Both *Zeror-* and *Zeror* cover more branches than AFL, and *Zeror* outperforms *Zeror-*. The visualization indicates that integrating two different instrumented binaries with the real-time scheduling helps fuzzers achieve better performance.

5.5 Discussion

Although binary-switching scheduler is able to integrate multiple diversely-instrumented binaries, we applied `Zeror` to fuzzers by switching only between original instrumented binary and self-modifying tracing instrumented binary in our evaluation, which could not fully excavate `Zeror`'s potentiality, but already demonstrates the effectiveness of tracing and scheduling. Furthermore, even with the scheduling of two binaries, it improves both speed and coverage. Recently, Dinesh [12] proposed a novel approach of instrumentation, we plan to integrate it in the future.

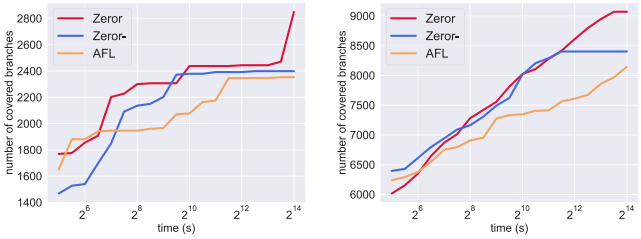


(a) Average time taken for different methods to erase instrumentation points (lower is better).



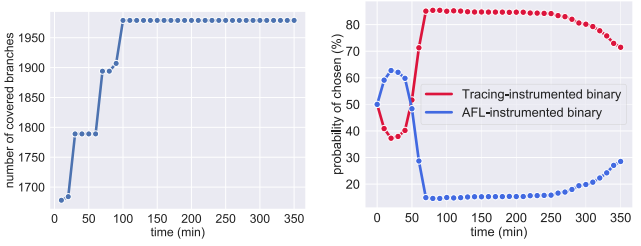
(b) Relative covered branches improvement of Zeror- compared with Untracer.

Figure 8: Comparison between Zeror- and Untracer.



(a) libjpeg **(b) harfbuzz**

Figure 9: Branches covered over time with different configurations. The x-axis is on a logarithmic scale.



(a) Number of covered branches over time. **(b) Chosen probabilities of different binaries over time.**

Figure 10: Case study on sqlite of AFL-Zeror.

Another potential concern is whether the scheduling mechanism can help fuzzer shift into proper binary. Figure 10 is the real-time visualization of covered branches and the chosen probabilities of diversely-instrumented binaries when AFL+Zeror is applied to test `sqlite`. We can observe that the chosen probability of the binary instrumented by AFL is in decline when the number of covered branches reaches the plateau at the time of 30min-60min, and `Zeror` has high probability to shift into the faster binary (instrumented by self-modifying tracing) when the AFL-instrumented binary cannot make any process. The observation indicates that the scheduling scheme do help fuzzer properly choose binary for execution. However, the scheduling scheme only collects execution statistical data, which may not be sufficient enough to fully display its efficiency. It could be further improved by gaining more information from data-flow analysis and control-flow analysis.

6 RELATED WORKS

Optimize fuzzing strategies. Existing optimizations of fuzzing reside in different stages. For the preparation stage, CollAFL [14] provides a solution to collect coverage feedback without bitmap collision, DeepFuzzer [32] leverages symbolic execution to generate qualified initial seeds. For the seed selection stage, AFLFast [5] gives more mutation times to valuable seeds which exercise low-frequency paths, Cerebro [29] prioritizes seeds in corpus on the basis of static analysis and dynamic scoring. For the seed mutation stage, FairFuzz [28] mutates input seeds in a restricted way so that they are more likely to still explore the rarest branch, MOPT [36] finds the optimal selection probability distribution of operators with respect to fuzzing effectiveness. Specially, a number of seed mutation optimizations leverage taint analysis such as REDQUEEN [2], Angora [7] and Matryoshka [8]. REDQUEEN [2] uses a lightweight input-to-state correspondence mechanisms as an alternative to data-flow analysis, Angora [7] adopts byte-level taint analysis and a gradient-descent algorithm for constraint penetration, Matryoshka [8] identifies nesting conditional statements by control flow and taint flow and proposed three strategies for mutating the input to solve path constraints.

Boost fuzzing speed. Xu et al. [51] design three new operating primitives to solve the performance bottlenecks of parallel fuzzing on multi-core machines. INSTRIM [22] reduces instrumentation cost by selectively instrumenting a part of basic blocks and reconstructing coverage information. Untracer [39] avoids tracing coverage of non-coverage-increasing test cases by removing visited instrumentation points.

Main differences. Optimizations of fuzzing strategies are orthogonal to `Zeror`, and most of them could also benefit from `Zeror`. For example, the experiment results show that, with the aid of `Zeror`, MOPT achieves better performance of coverage exploration and vulnerability discovery. Different from INSTRIM and Untracer, our study aims to boost fuzzing speed while preserve fine-grained coverage collection. Although Untracer has a similar idea with our self-modifying tracing component, rather than static binary rewriting, our tracing relies on self-modifying code to erase visited instrumentation points, which barely introduces new overheads and provides more fine-grained coverage collection. With the novel binary-switching scheduler, more improvements can be achieved.

7 CONCLUSION

In this paper, we propose a coverage-sensitive fuzzing framework `Zeror`, which integrates diversely-instrumented binaries to boost fuzzing speed and further improve the vulnerability discovery. `Zeror` is mainly made up of two parts: (1) a self-modifying tracing mechanism to provide a zero-overhead instrumentation for coverage collection; and (2) a real-time scheduling mechanism to select the proper instrumented binary for fuzzing on the basis of empirical Bayesian inference. In the experiments of fuzzing projects from Google fuzzer-test-suite, results show that with the aid of `Zeror`, fuzzers are able to gain higher speedup, cover more branches, and more importantly, expose bugs faster than the existing speed-up techniques. It can be applied to most of the existing fuzzers. In our future work, we plan to complement `Zeror` with other orthogonal fuzzing optimizations.

8 ACKNOWLEDGEMENT

This research is sponsored in part by National Key Research and Development Project (Grant No. 2019YFB1706200), the NSFC Program (No. U1911401, 61802223), the Huawei-Tsinghua Trustworthy Research Project (No. 20192000794), and the Equipment Pre-research Project (No. 61400010107).

REFERENCES

[1] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. 2020. IJON: Exploring Deep State Spaces via Fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1597–1612.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[3] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000*. 1–12. https://doi.org/10.1145/349299.349303

[4] Jose M Bernardo. 1976. Algorithm AS 103: Psi (digamma) function. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 25, 3 (1976), 315–317.

[5] 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, Vienna, Austria, October 24-28, 2016*. 1032–1043. https://doi.org/10.1145/2976749.2978428

[6] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 725–741. https://doi.org/10.1109/SP.2015.50

[7] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 711–725. https://doi.org/10.1109/SP.2018.00046

[8] Peng Chen, Jianzhong Liu, and Hao Chen. 2019. Matryoshka: Fuzzing Deeply Nested Branches. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. 499–513. https://doi.org/10.1145/3319535.3363225

[9] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 1967–1983. https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang

[10] Yuqi Chen, Christopher M. Poskitt, Jun Sun, Sridhar Adepu, and Fan Zhang. 2019. Learning-Guided Network Fuzzing for Testing Cyber-Physical System Defences. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. 962–973. https://doi.org/10.1109/ASE.2019.00093

[11] Saumya K. Debray and William S. Evans. 2002. Profile-Guided Code Compression. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*. 95–105. https://doi.org/10.1145/512529.512542

[12] Sushant Dinesh. 2019. *RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization*. Ph.D. Dissertation. Purdue University Graduate School.

[13] FoRTE-Research. 2020. Illegal pointer to buffer in Dyninst. https://github.com/FoRTE-Research/UnTracer-AFL/issues/5

[14] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 679–696. https://doi.org/10.1109/SP.2018.00040

[15] Everette S Gardner Jr. 1985. Exponential smoothing: The state of the art. *Journal of forecasting* 4, 1 (1985), 1–28.

[16] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*. 50–59. https://doi.org/10.1109/ASE.2017.8115618

[17] Google. 2020. Google fuzzer-test-suite. https://github.com/google/fuzzer-test-suite

[18] Google. 2020. OSS-Fuzz - continuous fuzzing of open source software. https://google.github.io/oss-fuzz/

[19] Google. 2020. SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html

[20] Lars Peter Hansen. 1982. Large sample properties of generalized method of moments estimators. *Econometrica: Journal of the Econometric Society* (1982), 1029–1054.

[21] honggfuzz@googlegroups.com. 2020. hongfuzz - security oriented fuzzer with powerful analysis options. http://honggfuzz.com

[22] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. Instrim: Lightweight instrumentation for coverage-guided fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*.

[23] Intel. 2017. Intel Processor Trace Tools. https://software.intel.com/en-us/node/721535

[24] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. 2003. Exploiting Self-Modification Mechanism for Program Protection. In *27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems, 3-6 November 2003, Dallas, TX, USA, Proceedings*. 170. https://doi.org/10.1109/CMPSAC.2003.1245338

[25] lcamtuf. 2017. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/

[26] lcamtuf. 2017. Fast LLVM-based instrumentation for afl-fuzz. https://github.com/google/AFL/blob/master/llvm_mode/README.llvm

[27] Peter Lee and Mark Leone. 1996. Optimizing ML with Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996*. 137–148. https://doi.org/10.1145/231379.231407

[28] Caroline Lemieux and Koushik Sen. 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, ASE 2018, Montpellier, France, September 3-7, 2018*. 475–485. https://doi.org/10.1145/3238147.3238176

[29] Yuekang Li, Yinxing Xue, Hongxu Chen, Xiuheng Wu, Cen Zhang, Xiaofei Xie, Haijun Wang, and Yang Liu. 2019. Cerebro: context-aware adaptive fuzzing for effective vulnerability detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 533–544. https://doi.org/10.1145/3338906.3338975

[30] Jie Liang, Yuanliang Chen, Mingzhe Wang, Yu Jiang, Zijiang Yang, Chengnian Sun, Xun Jiao, and Jiaguang Sun. 2019. Engineering a Better Fuzzer with Synergically Integrated Optimizations. In *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*. 82–92. https://doi.org/10.1109/ISSRE.2019.00018

[31] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. PAFL: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 809–814. https://doi.org/10.1145/3236024.3275525

[32] Jie Liang, Yu Jiang, Mingzhe Wang, Xun Jiao, Yuanliang Chen, Houbing Song, and Kim-Kwang Raymond Choo. 2019. DeepFuzzer: Accelerated Deep Greybox Fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2019).

[33] Jie Liang, Mingzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2018. Fuzz testing in practice: Obstacles and solutions. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 562–566. https://doi.org/10.1109/SANER.2018.8330260

[34] libfuzzer@googlegroups.com. 2020. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html

[35] Chi-Keung Luk, Robert S. Cohn, Robert Muth, Harish Patil, Artur Klauser, P. Geoffrey Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim M. Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*. 190–200. https://doi.org/10.1145/1065010.1065034

[36] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 1949–1966. https://www.usenix.org/conference/usenixsecurity19/presentation/lyu

[37] Henry Massalin. 1993. Synthesis: An efficient implementation of fundamental operating system services. (1993).

[38] Thomas Minka. 2000. Estimating a Dirichlet distribution.

[39] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 787–802. https://doi.org/10.1109/SP.2019.00069

[40] Awanish Pandey, Phani Raj Goutham Kotcharlakota, and Subhajit Roy. 2019. Deferred concretization in symbolic execution via fuzzing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. 228–238. https://doi.org/10.1145/3293882.3330554

[41] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/

[42] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. 2014. Optimizing Seed Selection for Fuzzing. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. 861–875. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/rebert

[43] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*. 167–182. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo

[44] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 803–817. https://doi.org/10.1109/SP.2019.00052

[45] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the HaCRS: Augmenting Autonomous Cyber Reasoning Systems with Human Assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 347–362. https://doi.org/10.1145/3133956.3134105

[46] 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 *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf

[47] syzkaller@googlegroups.com. 2020. syzkaller – an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller

[48] LLVM team. 2020. The LLVM Compiler Infrastructure. https://llvm.org/

[49] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2018. SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 61–64. https://doi.org/10.1145/3183440.3183494

[50] Xuerui Wang, Wei Li, Ying Cui, Ruofei Zhang, and Jianchang Mao. 2011. Click-through rate estimation for rare events in online advertising. In *Online multimedia advertising: Techniques and technologies*. IGI Global, 1–12.

[51] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing New Operating Primitives to Improve Fuzzing Performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. 2313–2328. https://doi.org/10.1145/3133956.3134046

[52] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[53] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid

Fuzzing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. https://www.ndss-symposium.org/ndss-paper/send-hardest-problems-my-way-probabilistic-path-prioritization-for-hybrid-fuzzing/

[54] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, Chengnian Sun, and Yu Jiang. 2019. VisFuzz: Understanding and Intervening Fuzzing with Interactive Visualization. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 1078–1081. https://doi.org/10.1109/ASE.2019.00106