SAND: Decoupling Sanitization from Fuzzing for Low Overhead

Ziqiao Kong†
ETH Zurich and
Nanyang Technological University

Shaohua Li†
ETH Zurich
shaohua.li@inf.ethz.ch

Zhendong Su
ETH Zurich
zhendong.su@inf.ethz.ch

† ***Equal contribution***

***Abstract*—Sanitizers provide robust test oracles for various software vulnerabilities. Fuzzing on sanitizer-enabled programs has been the best practice to find software bugs. Since sanitizers need to heavily instrument a target program to insert run-time checks, sanitizer-enabled programs have much higher overhead compared to normally built programs.**

In this paper, we present SAND, a new fuzzing framework that decouples sanitization from the fuzzing loop. SAND performs fuzzing on a *normally built program* and only invokes sanitizer-enabled programs when input is shown to be interesting. Since most of the generated inputs are not interesting, *i.e.*, not bug-triggering, SAND allows most of the fuzzing time to be spent on the normally built program. To identify interesting inputs, we introduce *execution pattern* for a practical execution analysis on the normally built program.

We realize SAND on top of AFL++ and evaluate it on 12 real-world programs. Our extensive evaluation highlights its effectiveness: on a period of 24 hours, compared to fuzzing on ASan/UBSan-enabled and MSan-enabled programs, SAND respectively achieves 2.6x and 15x throughput and detects 51% and 242% more bugs.

1. Introduction

Fuzzing has been one of the most successful approaches to finding security vulnerabilities [11], [39]. At a high level, fuzzers generate a large number of new inputs and execute the target program on each of them. Fuzzers typically rely on observable test oracles such as crashes to report bugs. However, many security flaws do not always yield crashes and thus are not detectable. Sanitizers are designed to tackle this problem. At compile-time, when sanitizers are enabled, compilers will heavily instrument the target program to insert various checks. At run-time, violations on these checks will result in program crashes. Fuzzing on sanitizer-enabled programs is thus more effective in discovering software bugs. To date, the most widely-used sanitizers include AddressSanitizer (ASan) [26], UndefinedBehaviorSanitizer (UBSan) [1], and MemorySanitizer (MSan) [27].

Sanitizers, despite their extraordinary bug-discovery capability, have two main drawbacks. First, sanitizers bring significant performance overhead to fuzzing. As our evaluation in Section 2.1 will show, ASan, UBSan, and MSan averagely slow down fuzzing speed by a factor of 3.6x, 2.0x, and 46x, respectively. Since fuzzing is computationally intensive, such high sanitizer overheads inevitably impede the performance of fuzzers. Many approaches have been proposed to reduce the run-time overhead of sanitizers. For example, Debloat [37] optimizes ASan checks via sound static analysis. SANRAZOR [35] removes likely redundant ASan and UBSan checks through dynamic profiling. FuZZan [14] designs dynamic metadata structure to improve the performance of ASan and MSan. Notwithstanding these optimization efforts, the overhead imposed by sanitizers remains considerable. For instance, as our evaluation will show, the state-of-the-art Debloat can only reduce less than 10% run-time cost of ASan. Moreover, all these schemes require significant modifications to the existing sanitizer code base.

The second drawback is that some sanitizers are mutually exclusive. For instance, because ASan and MSan maintain the same metadata structure, they can not be used together. Consequently, a fuzzer has to fuzz ASan-enabled program and MSan-enabled program separately.

Our insight. Since sanitizers provide the security oracle for execution, all current fuzzers execute sanitizer-enabled programs on every fuzzer-generated input to verify their validity. We now raise this question: *Can we decide whether or not an input may trigger a bug without truly executing a sanitizer-enabled program?* Theoretically, it seems to be paradoxical and infeasible as only by executing an input can we know if it is bug-triggering. However, our empirical evaluation will substantiate its feasibility. Our key insight is that bug-triggering inputs typically have unique execution characteristics on normally built programs. By analyzing the execution of an input, we can learn whether or not the input is potentially bug-triggering.

The new fuzzing framework. In this paper, we propose a new fuzzing framework that decouples sanitization from the normal fuzzing loop. In our framework, the fuzzer performs fuzzing on the normally built programs and only invokes sanitizer-enabled programs whenever necessary. Specifically, for a newly generated input, (1) the fuzzer executes it on the *normally built program*, *i.e.*, a program built without

TABLE 1: Execution speed of normally built programs and sanitizer-enabled programs. "Overhead" is calculated by dividing the native speed with the sanitizer speed. ✗ indicates a compilation failure.

Programs	Native	ASan		Debloat		UBSan		MSan	
	Speed	Speed	Overhead	Speed	Overhead	Speed	Overhead	Speed	Overhead
imginfo	2,964	869	341%	907	327%	1,968	151%	43	6,836%
infotocap	2,676	685	390%	✗	✗	1,962	136%	43	6,209%
jhead	2,963	859	345%	888	334%	2,652	112%	45	6,614%
mp3gain	1,488	627	237%	634	235%	917	162%	42	3,530%
mp42aac	1,917	472	406%	✗	✗	682	281%	42	4,612%
mujs	1,491	425	351%	440	339%	685	218%	42	3,590%
nm	2,209	586	377%	✗	✗	1,597	138%	43	5,079%
objdump	573	212	270%	✗	✗	250	229%	38	1,506%
pdftotext	410	151	271%	✗	✗	192	214%	35	1,172%
tcpdump	1,754	432	407%	493	356%	561	313%	42	4,196%
tiffsplit	2,093	665	315%	✗	✗	1,247	168%	43	4,868%
wav2swf	2,757	486	604%	517	550%	1,211	252%	43	6,357%
Average	1,941	539	356%	-	-	1,160	196%	42	4,552%

sanitizers; (2) it then analyzes the execution to determine whether or not the input should be sanitized; (3) if no, the fuzzer continues to the next input; otherwise, it invokes the sanitizer-enabled program(s) to verify if the input is bug-triggering.

We propose *execution pattern*, an execution abstraction approach, for practical execution analysis. Our evaluation shows that with *execution pattern* based analysis, averagely more than 98% of inputs do not need to be sanitized, *i.e.*, go to the "no" branch in (3). Since executing an input on the normally built program is significantly more efficient than on the sanitizer-enabled program, our new fuzzing framework can significantly reduce sanitizer overhead. Our solution creates a new paradigm of using sanitizers in fuzzing and is both simple and effective:
(1) *Our solution is simple*: it has no changes to sanitizer implementations and requires only ∼400 lines of extra code in a fuzzer.
(2) *Our solution is yet effective*: it significantly boosts the performance of fuzzing on sanitizer-enabled programs. Furthermore, it supports using multiple sanitizer-enabled programs simultaneously in fuzzing.

We realized our idea on top of AFL++ with around 400 lines of code. We call our tool SAND. We use 12 real-world programs from UNIFUZZ benchmark to evaluate SAND. Our evaluation shows that in a period of 24 hours, compared to fuzzing on ASan/UBSan-enabled and MSan-enabled programs, SAND respectively achieves 2.6x and 15x throughput and detects 51% and 242% more bugs. Compared to fuzzing on normally built programs, SAND can achieve nearly the same level of throughput while covering 258% more bugs. In summary, our contributions are:

- We propose a novel fuzzing framework that decouples sanitization from the fuzzing loop.
- We design *execution pattern* to practically analyze executions for identifying sanitization-required inputs.
- We implement our approach in a tool named SAND. We conduct in-depth evaluations to understand its effectiveness in terms of bug-finding, throughput, and coverage.

TABLE 2: Ratio of bugger-triggering inputs.

Programs	Executions	Bug-triggering	Ratio
imginfo	8.4M	92,345	1.1%
infotocap	14.0M	23,784	0.2%
jhead	16.7M	468,202	2.8%
mp3gain	13.9M	105,349	0.8%
mp42aac	4.6M	16	¡ 0.01%
mujs	13.5M	19,134	0.1%
nm	11.6M	517	0.0%
objdump	10.4M	328,546	3.2%
pdftotext	7.3M	1,358	¡ 0.01%
tcpdump	8.6M	10,980	0.1%
tiffsplit	11.6M	23,017	0.2%
wav2swf	7.7M	563,588	7.3%
Average	10.7M	136,403	**1.3%**

2. Observation and Motivation

In this section, we introduce three key observations on fuzzing, which motivates our design.

2.1. High Overhead of Sanitizers

Despite the fact that sanitizers are highly effective in exposing software bugs, they are initially designed for software developers to conduct in-house testing rather than fuzzing. To benchmark sanitizer overhead in fuzzing, we use all 12 benchmark programs from our evaluation. For

each program, we compile five versions of it, *i.e.*, native program, ASan-enabled program, Debloat-enabled program, UBSan-enabled program, and MSan-enabled program. The native program refers to a normally built program without using any sanitizers. Since Debloat achieves the state-of-the-art optimization for ASan, we include it to understand its performance. We use AFL++ as the default fuzzer. For each program, we follow the below steps:

Step (1) Use AFL++ to fuzz the native program and collect the first 1 million generated inputs to the program. All these inputs are saved into disk[1].

Step (2) Run AFL++ again on the native program to benchmark its running time on the saved 1 million inputs. The AFL++ here is slightly modified to fetch inputs from the disk instead of generating them.

Step (3) Repeat **Step(2)** on four sanitizer-enabled programs to collect their running time on the same set of inputs.

We run the above experiment 10 times and report the average fuzzing speed. All experimental settings are the same as our later evaluation in Section 4.1. Table 1 presents the average speed of different programs. Compared to native programs, ASan, UBSan, and MSan averagely bring 356%, 196%, and 4,552% overhead. Specifically, ASan introduces 237∼604% overhead, UBSan introduces 112∼313% overhead, and MSan introduces 1,172∼6,836% overhead. Even for the best ASan optimization Debloat, its improvement over ASan is rather insignificant compared to the native program. Such huge sanitizer overheads inevitably hinder the fuzzing throughput. Because sanitizers bring fuzzing a significantly stronger bug-detection capability, current fuzzers have to bear the following speed loss. Suppose that we have a way to reduce or even eliminate sanitizers' overhead while still keeping their bug-detection capability, fuzzing would benefit significantly from it.

2.2. Rareness of Bug-triggering Inputs

Fuzzers typically generate a large body of inputs for a target program. It is intuitive that bug-triggering inputs are rarely met during fuzzing. To understand the ratio of bug-triggering inputs to all the generated inputs, we count the total number of bug-triggering inputs and all generated inputs during 24 hours of fuzzing. The experimental data is from our later evaluation in Section 4.2.

Table 2 lists the number of total generated inputs, the number of bug-triggering inputs, and the ratio. We can find that averagely *only 1.3%* of inputs are bug-triggering. For some programs, it is even rarer. For instance, on pdftotext, less than 1 out of 10^4 inputs trigger bugs. We can conclude that *Only a tiny fraction of fuzzer-generated inputs are bug-triggering.*

2.3. Uniqueness of Execution Pattern on Bugs

Current fuzzers run all inputs on sanitizer-enabled programs. It would be very beneficial if only a subset of inputs are executed on sanitizer-enabled programs *while this subset of inputs contains all bug-triggering inputs*. In this paper, we propose to analyze the execution of input to achieve this goal.

1. We use tmpfs [33] to reduce I/O overhead.

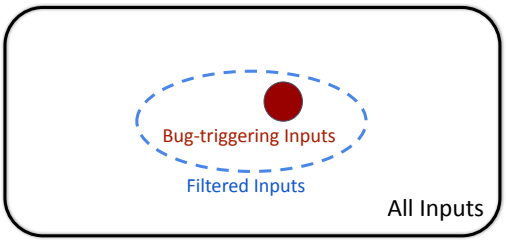


Figure 1: The abstraction filters out very small fraction of inputs from all inputs generated by fuzzers but still covers all bug-triggering inputs

Our hypothesis is that bug-triggering inputs have certain execution characteristics. Figure 1 illustrates our hypothesis. An ideal input filter can cover all bug-triggering inputs. If such an input filter can be obtained at a low cost, we can potentially boost the performance of fuzzing by selectively feeding inputs into sanitizer-enabled programs.

Previous studies have shown that bugs correlate highly to executions. Generally, program execution is a temporal transition sequence of program states. Theoretically, a full program state includes all the execution information, such as register values and memory state, at each time. In practice, we may not need all the execution information and choose different abstractions of program states. For instance, the commonly used code coverage metric is an abstraction of execution where a program state is the code region that the execution reaches. Because code edge is used by most fuzzers (26 out of 27 as reported in [23]), we use code edge as the default coverage metric. In this paper, we use *execution pattern* as the execution abstraction. Formally, we define *execution pattern* as follows.

Definition 2.1 (Execution Pattern)**.** Given an execution $\mathcal{E}$, the *execution pattern* of $\mathcal{E}$ is defined as $\mathcal{T}_{\mathcal{E}} = \{e_1, e_2, \cdots, e_m\}$, where $e_i \neq e_j (i \neq j)$ and e_i is the unique id of the code edge reached by $\mathcal{E}$. Note that, $\mathcal{T}_{\mathcal{E}}$ is a set meaning that ordering is ignored, *e.g.*, $\{e_{i_1}, e_{i_2}, e_{i_3}\} = \{e_{i_2}, e_{i_3}, e_{i_1}\}$.

In other words, *execution pattern* is the set of code edges that an execution covered. To be noted, execution patterns are obtained on normally built programs. We filter out all inputs with unique execution patterns and consider them as interesting. We now need to validate whether or not this filter can be used for filtering bug-triggering inputs in Figure 1.

We use the same set of 12 programs as before. For each round of experiment on a program, we follow the below steps:

Step (1) Use AFL++ to fuzz the *normally built* program.

Step (2) For each generated input, we collect its execution

TABLE 3: Ratios of inputs that have unique execution patterns. "All" refers to the ratio of all generated inputs that have unique execution patterns. "Bug" refers to the ratio of bug-triggering inputs that have unique execution patterns. "Coverage-Bug" refers to the ratio of bug-triggering inputs that are coverage-increasing.

Programs	Execution Pattern		Coverage-Bug
	All	Bug	
imginfo	0.22%	72.8%	39.3%
infotocap	8.11%	98.3%	7.9%
jhead	1.44%	91.6%	13.0%
mp3gain	0.18%	98.2%	76.0%
mp42aac	0.04%	100.0%	66.7%
mujs	5.34%	99.0%	40.1%
nm	0.35%	100.0%	73.0%
objdump	1.30%	100.0%	82.2%
pdftotext	4.82%	100.0%	85.6%
tcpdump	2.09%	100.0%	70.4%
tiffsplit	1.17%	99.2%	27.0%
wav2swf	0.01%	100.0%	67.1%
Average	**2.09%**	**96.58%**	54.03%

pattern. The collection is done by analyzing the coverage bitmap of the execution on the *normally built* program. We then examine whether or not this execution pattern has been observed before by checking its hash value in a hash-table. Technical details about how execution patterns are collected will be presented in Section 3.2.

Step (3) For each generated input, we also run it on ASan- and UBSan-enabled programs to verify if it triggers a bug.

We run the experiments for 24 hours and repeat them 10 times. Because we need to run through sanitizer-enabled programs for every input, to gather a sufficiently large number of inputs, we exclude MSan in **Step (3)** due to its extremely low speed. However, we will include it in our later rigorous end-to-end evaluation in Section 4. With this set of experiments, we want to explore (1) out of all bug-triggering inputs, how many of them are also marked as having unique execution patterns? (2) out of all inputs generated during fuzzing, how many of them are marked as having unique execution patterns?

Table 3 shows the result. The second column shows that on average, only 2% inputs have unique execution patterns. For more than half of the programs, the ratio is even below 1%. We can thus conclude that *Only a small fraction of inputs have unique execution patterns.* Surprisingly, the third column shows that more than 96% of bug-triggering inputs have unique execution patterns. This means that if we only pass inputs with unique execution patterns to sanitizer-enabled programs, we will be able to capture most bugs. In fact, since we did not deduplicate all crashes here, many of the bug-triggering inputs are duplicates. Our evaluation in Section 4.2 will show that all bugs are captured by unique execution patterns. From these experimental results, we can conclude that *execution pattern is a practical instance as the input filter for bug-triggering inputs.*

Advantages of execution pattern. There are two main advantages of the designed execution pattern:

- Effortless: CGF fuzzers such as AFL++ by default collect a bitmap for code edges to calculate coverage. We can simply obtain execution patterns by reusing the bitmap.
- Effectiveness: The above empirical evaluation has shown that our execution pattern can be used as an input filter to cover more than 96% of bug-triggering inputs.

Other execution abstraction as an input filter. Theoretically, there are an infinite number of abstraction methods for program executions. Finding an abstraction that meets all the requirements indicated in Figure 1 is not easy. Here, we discuss two other possible alternatives.

(1) Coverage-increasing as an input filter. Because coverage plays a vital role in all CGF fuzzers, it is intuitive to use the coverage-increasing property as an input filter. To understand whether or not coverage-increasing can be a good abstraction, we redo the experiments by also recording if a bug-triggering input increases coverage. The last column "Coverage-Bug" in Table 3 shows the result. We can find that more than 54% of the bug-triggering inputs are not coverage-increasing. In some cases, the ratio is even below 10%. In other words, using the coverage-increasing property as an input filter can miss a lot of bugs, which is unacceptable in fuzzing.

(2) Execution trace as an input filter. Our proposed execution pattern only considers the set of reached code edges. One may also use a temporal sequence of executed code edges for a finer-grained execution abstraction. However, in order to collect such information, one has to change the instrumentation logic of fuzzers because current fuzzers only collect coverage bitmap where temporal information is not available. In contrast, our proposed execution pattern can reuse the coverage bitmap without touching any instrumentation logic. Moreover, as our empirical evaluation in this section has shown and our later evaluation will show that our current execution pattern has achieved a perfect performance. Even if we adopt a finer-grained execution abstraction, there would be little gains.

Summary. Researchers have proposed many execution abstraction methods for fuzzing feedback such as N-gram [31] and program invariant [9]. Some of them might also be useful for input filtering. As has been analyzed before, they all require significant changes to the instrumentation logic of current fuzzers. Given that our *execution pattern* can achieve significant performance, it would be very hard for them to bring extra benefits. We consider the exploration of other alternative execution abstractions as interesting future work.

3. Our Approach

This section introduces the design and implementation of our new fuzzing framework SAND.

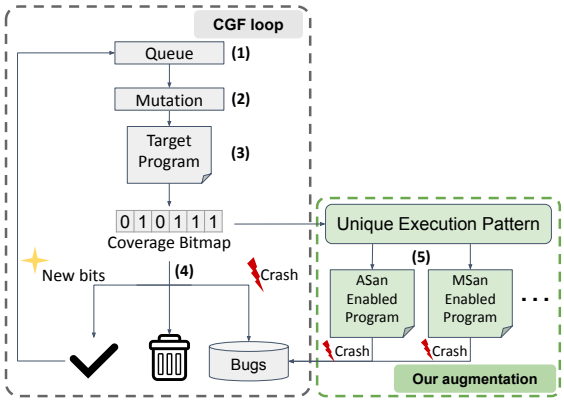


Figure 2: SAND fuzzing loop

3.1. Fuzzing framework

Before introducing our new fuzzing framework, we first explain the general workflow of a coverage-guided fuzzer (CGF). The grey area in Figure 2 outlines the high-level fuzzing sketch of a CGF. *Before fuzzing starts, the fuzzer compiles the target program with fuzzer instrumentation and/or sanitizers.* Then, it fuzzes the target program following the below steps:

(1) Seed selection. Select one seed from the seed pool according to predefined strategies.

(2) Mutation. Mutate the seed to generate new test inputs.

(3) Executing on the target program. Execute a test input on the target program.

(4) Coverage and execution analysis. Collect coverage feedback from the execution. If the execution increases coverage, save it to the seed pool; if the execution results in a crash, report the corresponding input as bug-triggering and save it to the disk; otherwise, discard it.

As one can see, CGFs rely on the execution result of the target program to detect bugs. In order to maximize bug detection capability, current CGFs usually compile the target program with sanitizers enabled. This routine significantly slows down fuzzing speed due to the high overhead of sanitizers.

In this paper, based on our observation in Section 2.3, we propose a new fuzzing framework to decouple sanitization from the normal fuzzing loop. Our approach can reduce the extra overhead introduced by sanitizers while maintaining the same bug-detection capability. The green part in Figure 2 highlights our approach. Before fuzzing starts, the fuzzer compiles multiple versions of the same program: (1) a normally built program without any sanitizer enabled (denoted as $\mathcal{P}_{fuzz}$), *on which the fuzzer performs fuzzing*, and (2) a set of sanitizer-enabled programs, *e.g.*, ASan-enabled program ($\mathcal{P}_{ASan}$) and MSan-enabled program ($\mathcal{P}_{MSan}$). The fuzzer follows the same steps as a CGF to fuzz the *normally built program*. But, after each execution of the target program, we introduce a new step:

(5) Conditional sanitization. Extract *execution pattern* from the available coverage bitmap. If the execution pattern has been observed before, *i.e.*, not unique, discard it. Otherwise, the current input is identified as *sanitization-required*, then the fuzzer executes it on each of the sanitizer-enabled programs and reports any discovered crashes.

Algorithm 1: The New Fuzzing Loop of SAND

Input: Seed pool $\mathcal{S}$.

```
1  while ¬Abort() do
2      s ← SelectSeed(S) // Seed selection
3      s' ← Mutate(s) // Generate a new input
4      ret, cov ← Execution(s', P_fuzz) // Execute
5
6      T_E ← GetExecutionPattern(cov)
7      if IsUnique (T_E) then
8          foreach P_san ∈ {P_ASan, P_MSan, ···} do
9              ret_san ← SanExecution(s', P_san)
10             if ret_san == crash then
11                 ret = crash
                                   // Our augmentation
12
13     if ret == crash then // Any execution crash
14         save s' to disk
15     if cov covers new code then // New coverage
16         add s' to S
```

This new step does not alter the normal fuzzing logic. The execution pattern is obtained from the already-available coverage bitmap collected from the normally built target program. To determine whether or not an execution pattern has been observed before, we use a hash-table to store all observed execution patterns (see details in the next section §3.2).

Algorithm 1 sketches the implementation pseudo-code of our new fuzzing framework. In each fuzzing loop (line 1), the fuzzer firstly selects a seed s and mutates it to generate a new input s' (lines 2-3). Next, it executes the normally built program $\mathcal{P}_{fuzz}$ on the input to collect its execution return ret and coverage bitmap cov (line 4). Then, the fuzzer extracts the execution pattern from cov (line 6) and determines whether or not this execution pattern has been observed (line 7). If this is a new execution pattern, the fuzzer labels it as sanitization-required and executes each of the available sanitizer-enabled programs $\mathcal{P}_{san}$ on the input s' (lines 8-9). If any execution crashes, meaning that the input s' triggers a bug, the fuzzer sets the return status to *crash* (lines 10-11). Finally, the fuzzer continues the original procedure: save the new input as bug-triggering if the execution return status is *crash* (lines 13-14); or queue

it to the seed pool if it increases coverage (lines 15-16).

Our new fuzzing framework decouples sanitization from normal fuzzing logic. It has the following main advantages:

- ***Orthogonal to CGFs.*** We only introduce an orthogonal step to execute sanitizer-enabled programs on inputs with unique execution patterns. In theory, any AFL-family fuzzers can be augmented by our approach without modifying their main fuzzing logic.
- ***Sanitizer inclusive.*** Normally, some sanitizers are mutually exclusive such as ASan and MSan, meaning that they cannot be enabled on the same program. Current fuzzers can only perform fuzzing on a program with only one of such sanitizers enabled. In our new framework, multiple sanitizer-enabled programs can be used for sanitization simultaneously.
- ***Effective.*** Our evaluation will show that only a small fraction (averagely $\leq 2\%$) of inputs have unique execution patterns and requires sanitization, thus significantly improving fuzzing throughput. We will late formalize the theoretical benefit of our approach in Section 3.4.
- ***Simple.*** First, sanitizers are directly used for instrumentation and thus we do not need to change their code base. Second, the only modification we applied to a fuzzer is the augmented new step after each execution. Other parts of the fuzzer are not touched. For instance, in our current implementation based on AFL++, our implementation takes only around 400 lines of code.

In the following sections, we will introduce more details on how unique execution patterns are determined and how sanitizer-enabled programs are invoked. Lastly, we will formalize the theoretical benefit of our approach.

3.2. Execution Pattern Analysis

For an execution, CGF fuzzers typically use a bitmap to record the hit counts of each code region. According to Definition 2.1, the execution pattern of an execution is the set of all code regions it visited. Thus, we can reuse the coverage bitmap to obtain execution patterns by simply setting all non-zero hit counts to 1[2]. To identify unique execution patterns, we calculate checksums of all observed execution patterns and use a hash-map to store them. Algorithm 2 details the pseudocode of the process.

Note that, the hash-table HashTable is initialized to all zeros at the start of fuzzing. In our implementation, we use XXH32 hashing algorithm[8] because of its fast speed and a 32-bit hash-table to support a maximum of $4,294M$ different checksums. Our evaluation in Section 4.5 will demonstrate that the cost associated with hashing is negligible and no instances of hash collision are observed.

3.3. Sanitizer-Enabled Programs

In our new fuzzing framework, the target program $\mathcal{P}_{fuzz}$ is instrumented by the fuzzer to include the necessary instrumentation code for coverage collection. Since all the sanitizer-enabled programs $\mathcal{P}_{san}$ are used for sanitization only, no such instrumentation is needed. Thus, we directly use the LLVM compiler to compile the program with different sanitizers. Because some sanitizers such as ASan and MSan are mutually exclusive, multiple sanitizer-enabled programs can be compiled in this stage. By default, we use two sanitizer-enabled programs, *i.e.*, ASan/UBSan-enabled program and MSan-enabled program. Since ASan and UBSan can be used together, we use one $\mathcal{P}_{ASan/UBSan}$ to support both of them. To reduce the burden of invoking each $\mathcal{P}_{san}$ from the fuzzer, we utilize the *forkserver*[34] mode, which is also used by default in many CGFs.

2. In our current implementation, we use the `simplify_trace()` function in AFL++ to achieve this goal.

Algorithm 2: Identify unique execution patterns

```
1 IsUnique(T_E):
2     cksum ← Hash(T_E)
3     if HashTable[cksum] ≠ 1 then
4         HashTable[cksum] = 1
5         return True;
6     return False;
```

3.4. Theoretical Benefits

In this section, we analyze the theoretical benefits of our approach in terms of the speedup rate compared to the current fuzzing framework. Intuitively, the speedup rate of our new fuzzing framework is related to the sanitizer overhead and the frequency of invoking sanitizer-enabled programs. Before analyzing the speedup rate, we first introduce *sanitizer overhead ratio* and *unique execution pattern ratio*.

Sanitizer overhead ratio. Given t_{norm} as the time of executing the normally built program and t_{san} as the time of executing sanitizer-enabled program(s), the overhead ratio of the sanitizer K_{san} is defined as:

$$K_{san} = \frac{t_{san}}{t_{norm}} \tag{1}$$

As shown in Section 2.1, if we only consider a single sanitizer-enabled program such as ASan, the sanitizer overhead ratio K_{ASan} can range from 237% to 604%. If we use multiple sanitizers such as all ASan, UBSan, and MSan, the average ratio can be as high as 5,104%.

Unique execution pattern ratio. Given N_{total} as the total number of generated inputs during fuzzing and N_{unique} as the number of inputs that have unique execution patterns, the unique pattern ratio R_{unique} is defined as:

$$R_{unique} = \frac{N_{unique}}{N_{total}} \tag{2}$$

Speedup rate. Assume that we have a CGF fuzzer that performs fuzzing on a sanitizer-enabled program. Because running an execution on the sanitizer-enabled program takes

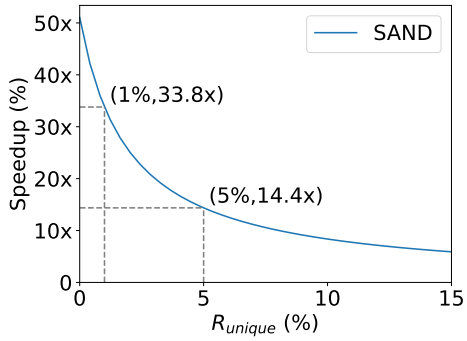


Figure 3: The speedup of SAND while the R_{unqiue} varies. K_{san} is set to the average overhead of all sanitizers calculated from table 1.

t_{san}, fuzzing on all N_{total} inputs takes in total $T_{CGF} = N_{total} \cdot t_{san}$. For our fuzzing framework, since only N_{unqiue} inputs will invoke the sanitizer-enabled program, the total time of fuzzing on all N_{total} inputs can be calculated by $T_{our} = N_{total} \cdot t_{norm} + N_{unique} \cdot t_{san}$. Thus, compared to the existing CGF fuzzer, the speedup rate of our framework can be calculated as:

$$S = \frac{T_{CGF}}{T_{our}} = \frac{N_{total} \cdot t_{san}}{N_{total} \cdot t_{norm} + N_{unique} \cdot t_{san}} \quad (3)$$

$$= \frac{1}{\frac{t_{norm}}{t_{san}} + \frac{N_{unique}}{N_{total}}} \quad (4)$$

$$= \frac{1}{\frac{1}{K_{san}} + R_{unique}} \quad (5)$$

The equation highlights that the speedup rate S correlates negatively with the sanitizer overhead K_{san} and positively with the unique execution pattern ratio R_{unique}. As our observation in Section 2.1 has shown, the average overhead for ASan, UBSan, and MSan are respectively 356%, 196%, and 4,552%. Assuming that we use all three sanitizers in our fuzzing framework, we can set K_{san} =(356+196+4552)%=5,104%. Based on these data, in Figure 3, we visually illustrate the relationship between the speedup rate S and the unique pattern ratio R_{unique}. As the figure suggests, when R_{unique} is small, *e.g.*, less than 1%, we can have more than 30x speedup rate. When the ratio R_{unique} grows to 5%, we can still observe a 14x speedup. Our evaluation in the next section will show that in practice, most of the unique pattern ratios R_{unqiue} are less than 2%. Thus, our new fuzzing framework can significantly accelerate fuzzing on sanitizer-enabled programs.

4. Evaluation

We implemented SAND based on AFL++-4.05c [10], the latest version at the time of implementation. AFL++ is the state-of-the-art grey-box fuzzer and has been widely used as the baseline fuzzer in many previous study [19], [21], [7]. Our integration consists of in total 476 lines of code, a rather small effort compared to the large code base, more than 100k lines of code in AFL++. Since SAND is orthogonal to the baseline fuzzer, to show its general applicability, we also apply SAND to MOpt-AFL++ [20], which proposes an efficient mutation strategy for fuzzing.

TABLE 4: All 12 real-world programs from UNIFUZZ used in the evaluation.

Type	Program	Version	Arguments
Image	imginfo	0.26	@@
	jhead	3.00	@@
	tiffsplit	libtiff 3.9.7	@@
Audio	mp3gain	1.5.2-r2	@@
	wav2swf	swftools 0.9.2	-o /dev/null @@
Video	mp42aac	Bento4 1.5.1-628	@@ /dev/null
Text	infotocap	ncurses 6.1	-o /dev/null @@
	mujs	1.0.2	@@
	pdftotext	xpdf 4.00	@@
Binary	nm	binutils-5279478	-A -a -I -S -s \ –special-syms \ –synthetic \ –with-symbol. \ -D @@
	objdump	binutils 2.28	-S @@
Network	tcpdump	4.8.1 + libpcap 1.8.1	-e -vv -nr @@

4.1. Experimental Setup

Benchmark. We use real-world programs from the benchmarking test platform UNIFUZZ for our evaluation. We use the provided seeds for all fuzzing campaigns. Some programs cannot be fuzzed with sanitizers because they would crash on all seed inputs when sanitizers are enabled. Because our evaluation target is to compare against fuzzing on sanitizer-enabled programs, we exclude such programs. This selection gives us a total of 12 real-world programs. Table 4 lists the details. These programs cover a diverse range of input types, including image (*e.g.*, imginfo), audio (*e.g.*, wav2swf), video (*e.g.*, mp42aac), text (*e.g.*, infotocap), binary (*e.g.*, objdump), and network packet (*e.g.*, tcpdump).

Baseline fuzzers. We choose the three most widely-used sanitizers, *i.e.*, ASan, UBSan, and MSan. Because ASan and UBSan are compatible with each other, we use them together when compiling a program. We equip SAND with all three sanitizers. That is, for each program, we compile a normally built version $\mathcal{P}_{fuzz}$ and two sanitizer-enabled versions, *i.e.*, $\mathcal{P}_{ASan/UBSan}$ and $\mathcal{P}_{MSan}$.

We choose AFL++ as the baseline fuzzer and use it to fuzz (1) normally built programs (denoted as "AFL++-Native"), (2) ASan/UBSan-enabled programs (denoted as "AFL++-ASan/UBSan"), and (3) MSan-enabled programs (denoted as "AFL++-MSan"). All fuzzers and programs are

TABLE 5: Mean number of unique bugs across repetitions. ✗ indicates a compilation failure.

Programs	AFL++-				SAND$_{p\text{-val}}$
	Native	ASan/ UBSan	Debloat/ UBSan	MSan	
imginfo	0	7.9	7.7	0.4	8.3 $_{0.18}$
infotocap	0.3	5.1	✗	1.7	6.4 $_{0.03}$
jhead	0.8	5.7	4.8	6.4	7.1 $_{0.03}$
mp3gain	4.5	8.1	7.9	1.1	7.8 $_{0.71}$
mp42aac	0	2	✗	0	3 $_{0.00}$
mujs	0	7.9	8.1	2.7	7.8 $_{0.75}$
nm	0	1.3	✗	0	3.5 $_{0.00}$
objdump	1	4	✗	1.2	3.3 $_{1.00}$
pdftotext	3.1	2.5	✗	1	3.7 $_{0.09}$
tcpdump	0	6.2	6.6	3.5	12.3 $_{0.00}$
tiffsplit	4.3	7.3	✗	2	13.7 $_{0.00}$
wav2swf	7.2	12.3	12.8	4	16.8 $_{0.03}$
Average	1.8	5.9	-	2.0	**7.8**

built with LLVM-14, the latest stable version at the time of implementation and evaluation.

To understand if SAND can surpass the existing sanitizer optimization schemes, we also choose the state-of-the-art ASan optimization technique, Debloat. Because Debloat optimizes ASan, it can also be used together with UBSan. To maximize its bug detection capability, we let AFL++ to fuzz on Debloat/UBSan-enabled program (denoted as "AFL++-Debloat/UBSan"). All programs instrumented with Debloat are built with LLVM-12 because this is the highest LLVM version that Debloat supports. Since compiling infotocap, mp42aac, nm, objdump, and pdftotextwith Debloat results in compilation failures, we exclude them for AFL++-Debloat/UBSan.

Hardware and Setup. We conduct all experiments on a machine equipped with an AMD 3990x CPU and 256G memory running Ubuntu 22.04. Following Klee's [16] standard we repeated all experiments 10 times and ran all fuzzing campaigns for 24 hours. We apply the Mann-Whitney U-test [22] to understand the statistical significance of our results.

4.2. Bug-Finding Capability

Finding bugs is the ultimate goal of fuzzing. In this section, we evaluate the bug-finding capability of all fuzzers. In particular, we would like to answer the following two questions:

- **Q1** Does SAND find *more bugs* compared to other baseline fuzzers?
- **Q2** Does SAND *miss any bugs* found by any baseline fuzzers?

To answer these questions, we collect all crashes found by each fuzzers. We triage all crashes according to their root causes to precisely quantify the number of unique bugs found by each fuzzer. We first deduplicate all crashes according to the last 3 stack frames with the assistance of GDB [18] and then manually analyze the root cause of the crashes.

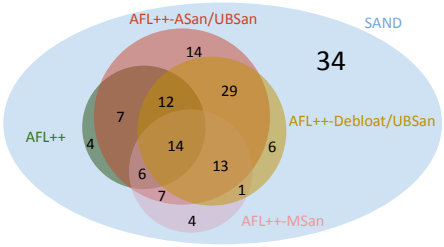


Figure 4: Unique bugs found by fuzzers.

Total Number of Unique Bugs. To understand the overlaps of bugs found by different fuzzers, we collect all unique bugs found by each fuzzer in each repetition. Figure 4 illustrates the unique bug sets of different fuzzers through a Venn diagram. It shows that SAND *covers all bugs* discovered by all other fuzzers. Moreover, SAND even finds **34** more bugs that can not be covered by all other fuzzers. Table 6 further breaks down the total number of bugs discovered by each fuzzer. Compared to every other fuzzers, SAND finds significantly more bugs on all programs. The second-to-last column "All" lists the total number of bugs found by *all other fuzzers together*. SAND again finds more bugs in each program. On some programs, SAND can even find $2x \sim 3x$ more bugs. For instance, SAND discovers 2x on mp42aacand 3.5x on nm. In summary, we can answer **Q1** and **Q2**: SAND *does not miss any bugs and can find significantly more bugs.*

Mean Number of Unique Bugs. We report the average number of unique bugs found by each fuzzer across repetitions in Table 5. On average, SAND finds 32% more bugs than the second-best fuzzer (AFL++-ASan/UBSan). On 7 out of 12 programs, SAND can discover more bugs than any other fuzzers with statistical significance (*i.e.*, p-value ¡ 0.05). On some programs, SAND can cover significantly more (¿ 50%) bugs. For instance, on tcpdump, SAND can averagely find 12.3 bugs while the next-best fuzzer AFL++-Debloat/UBSan only finds 6.6 bugs. For the rest 5 programs, SAND finds statistically the same number of unique bugs. For instance, although AFL++-ASan/UBSan averagely finds 8.1 bugs and SAND finds 7.8 bugs, there is no statistical difference between them (p-val $= 0.707$). In summary, our result confirms the answer to **Q1**: SAND *has a significantly stronger bug-finding capability than all other fuzzers.*

Compared to AFL++-Native, ASan, as well as Debloat and UBSan, has positive effects on 11 out of 12 programs. MSan finds fewer bugs on 4 programs, which is due to its extremely low fuzzing speed. An interesting case is pdftotext, where AFL++-Native can detect more bugs than other fuzzers except for SAND. The main reason is that

TABLE 6: Total number of unique bugs. ✗indicates a compilation failure. "All" refers to the number of unique bugs found by all other AFL++-based fuzzers except for SAND.

Programs	AFL++-				All	SAND
	Native	ASan/ UBSan	Debloat/ UBSan	MSan		
imginfo	0	10	10	1	11	11
infotocap	1	8	✗	3	9	10
jhead	5	9	8	10	10	12
mp3gain	10	12	12	2	12	14
mp42aac	0	2	✗	0	2	4
mujs	0	9	9	3	9	10
nm	0	2	✗	0	2	7
objdump	1	4	✗	2	4	4
pdftotext	5	3	✗	3	5	5
tcpdump	0	12	15	9	23	36
tiffsplit	8	9	✗	2	11	18
wav2swf	13	22	21	10	22	23
Sum	43	102	-	45	120	**154**

all bugs in pdftotextcan be triggered without sanitizers and AFL++-Native has higher throughput than other fuzzers. Nevertheless, the overall result confirms the necessity of using sanitizers during fuzzing.

4.3. Fuzzing Throughput

As our theoretical analysis in Section 3.4 indicates, SAND can potentially improve fuzzing speed by a significant factor. We now analyze the end-to-end fuzzing throughput, *i.e.*, the total number of inputs generated and executed during fuzzing. Figure 5 shows the average throughput of each fuzzer.

Compared to sanitizers-enabled fuzzers. SAND achieves an average of 2.6x, 2.1x, and 150x throughput than AFL++-ASan/UBSan, AFL++-Debloat/UBSan, and AFL++-MSan. Moreover, SAND has significantly higher throughput *on all programs*. On some of the programs, the speedup rate is even higher. For instance, on nm, SAND executes 4x and 303x inputs than AFL++-ASan/UBSan and -MSan, respectively. To be noted, SAND is equipped with all three sanitizers including the slowest MSan. All other fuzzers, on the other hand, only support one or two sanitizers.

Compared to AFL++-Native. Overall, SAND achieves 75% of AFL++-Native's throughput. On 4 out of 12 programs, SAND achieves more than 90% of AFL++-Native's throughput. *The result shows that* SAND *successfully increases the speed of fuzzing on sanitizer-enabled programs to a near-native level.*

Unique execution pattern ratio. According to our theoretical analysis in Section 3.4, the high speedup ratio is mainly due to a relatively small unique execution pattern ratio R_{unique}. Intuitively, a small R_{unique} indicates that only a small fraction of inputs require sanitization and thus most of the sanitization costs can be saved. We track the R_{unique} every 40 seconds during fuzzing. Figure 6 plots the results. As expected, at the start of fuzzing, R_{unique} is relatively high because many execution patterns are new. With the fuzzing going on, more and more inputs have duplicate execution patterns and thus the ratio R_{unique} becomes significantly smaller. This tendency is analogous to the saturation situation of code coverage. Overall, SAND has less than 2% ratio on 10 out of 12 programs, meaning that only less than 2% of inputs are fed into sanitizer-enabled programs for sanitization. Although a slightly higher ratio is observed on infotocap, and mujs, these ratios are eventually all less than 5%. According to the throughput data in Figure 5, SAND has indeed relatively lower speedup ratios on these two programs. Such correlation is coherent with our theoretical analysis in Section 3.4.

4.4. Code Coverage

We use the afl-showmap utility in AFL++ to collect the code coverage. Table 7 presents the average code coverage achieved by each fuzzer on each program.

Compared to sanitizers-enabled fuzzers. SAND achieves statistically higher code coverage on 9 out of 12 programs. For the rest 3 programs, SAND has higher code coverage but with no statistical significance. Intuitively, since SAND has a much higher throughput than all other sanitizer-enabled fuzzers, SAND executes more inputs and thus achieves higher code coverage.

Compared to AFL++-Native. On 6 out of 12 programs, there is no significant coverage difference between AFL++-Native and SAND. For the rest 6 programs, SAND achieves almost the same code coverage as AFL++-Native, with an average of 0.53% less coverage. As analyzed before, SAND can achieve 75% throughput of AFL++-Native, which accounts for the coverage drop in some programs. Since bug-finding capability is the golden measuring metric for fuzzing, although AFL++-Native can achieve relatively higher code coverage, it has the worst bug-finding rate and thus is less favorable in the realm of fuzzing.

4.5. Hash in SAND

As the Algorithm 1 indicates, SAND needs to calculate the hash checksum for every execution pattern. All these checksums are checked and indexed in a hash-table. In this section, we evaluate the hash overhead of SAND and the potential hash collision risk in the used hash-table.

Hash overhead. To precisely evaluate the hash overhead, we modified SAND to two versions. First, NoHash, where lines 6-11 in Algorithm 1 are removed so that no hash operations are performed. Second, SAND-Hash, where lines 8-11 in Algorithm 1 are removed so that hash operations are performed as the normal SAND but no sanitizer-enabled programs are invoked. We use the same random

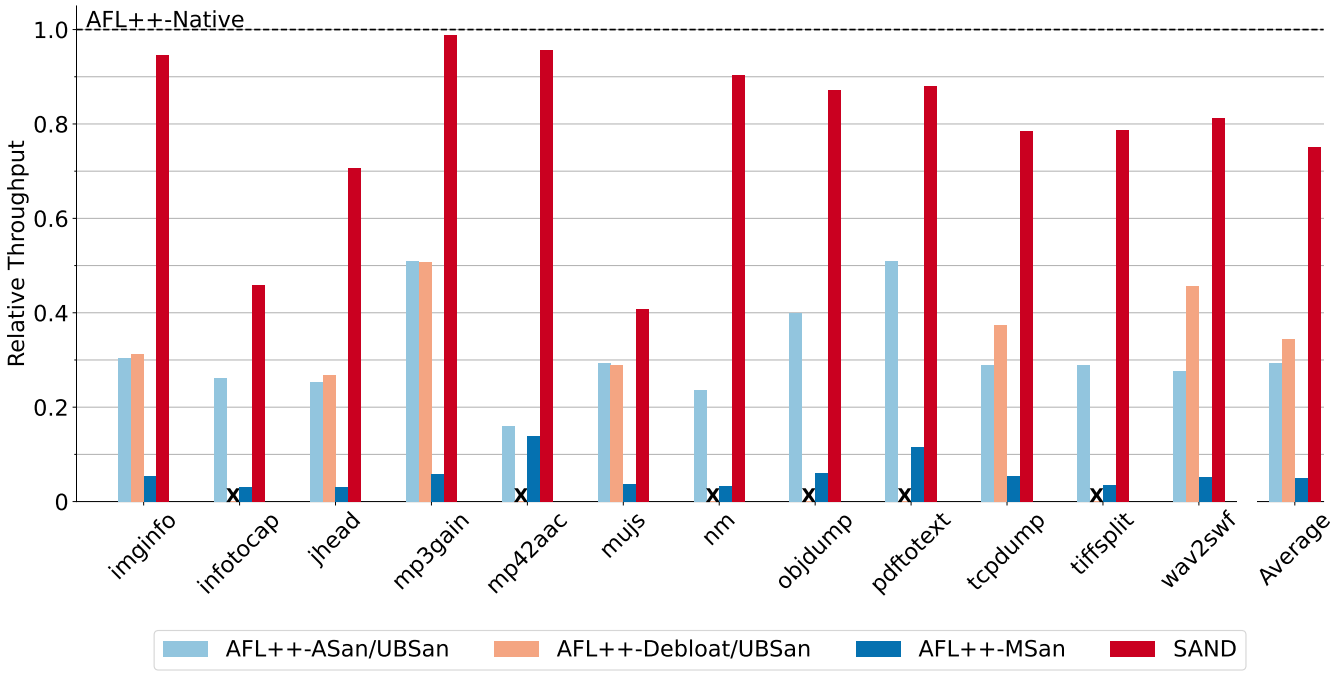


Figure 5: Relative throughput normalized to AFL++-Native. ✗indicates a compilation failure. "Average" refers to the average throughput of all programs.

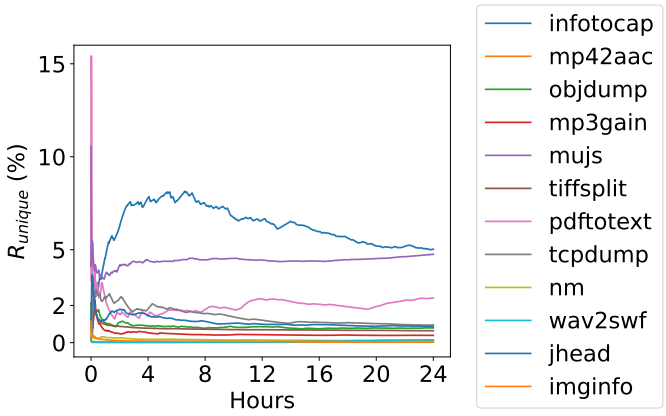


Figure 6: The trend of R_{unqiue} over time during fuzzing.

seed for both modified fuzzers such that they can generate an identical set of inputs on the same initial seed pool. We run both fuzzers on each program 10 times and record the total fuzzing time on the first 1,000,000 inputs. The second and third columns in Table 8 report the average speed of both fuzzers. On 10 out of 12 programs, both fuzzers do not have statistically differentiable (p-val ¡ 0.05) speed. Only on mujsand pdftotext, SAND-HASH is slightly slower at a rate of 1.6% and 0.7% while averagely SAND-HASH only incurs 0.7% penalty. We can then conclude that *hashing operations in* SAND *have negligible overhead to fuzzing.*

Hash collision. A *hash collision* can happen when two different execution patterns either have the same hash checksum or result in the same index in the hash-table. Because the effectiveness of SAND relies on the ability to identify unique execution patterns, hash collision may potentially have a negative impact on the performance. To evaluate the hash collision rate of SAND, we use the SAND-HASH and expand it to also save all execution patterns (rather than checksums) to disk. When an execution pattern is marked as observed, we compare this execution pattern with the saved execution pattern byte to byte. Any difference in the comparison signifies a hash collision. The last column in Table 8 lists the number of hash collisions for the first 1,000,000 inputs. Surprisingly, *none hash collision* is detected. The main reason is that unique execution patterns are rare (averagely 2% according to Section 2.2) and thus the hash-table is very sparse.

4.6. General Applicability

As depicted in Figure 2, SAND only augments the baseline CGF fuzzer with an extra sanitization step. Since no other part of the fuzzer is modified, SAND is in principle generally applicable to other CGF or AFL-family fuzzers. To demonstrate its general applicability, we port SAND to an alternative fuzzer MOpt[20], which implements a new and effective mutation strategy. We denote the new fuzzer as SAND-MOpt. We conduct the same evaluation on the 12 programs and compare them against MOpt-Native, MOpt-ASan/UBSan, and MOpt-MSan.

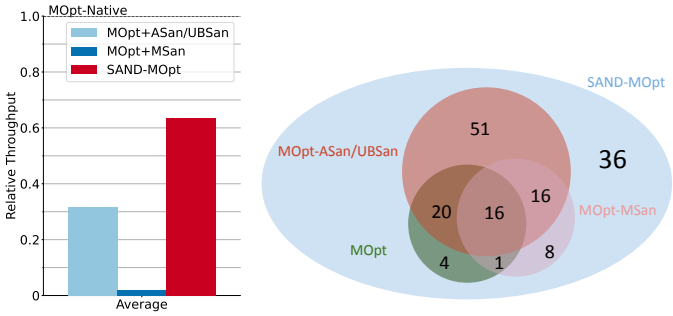
Figure 7a shows the normalized throughput of each fuzzer averaged across all programs. Same as SAND-

TABLE 7: Code coverage of fuzzers. ✗ indicates a compilation failure. "Diff" is the difference compared to AFL++-Native.

Programs	AFL++-							SAND	
	Native	ASan/UBSan		Debloat/UBSan		MSan			
	Cov	$\text{Cov}_{p\text{-val}}$	Diff	$\text{Cov}_{p\text{-val}}$	Diff	$\text{Cov}_{p\text{-val}}$	Diff	$\text{Cov}_{p\text{-val}}$	Diff
imginfo	13.36%	11.89% $_{0.00}$	-1.47%	11.36% $_{0.00}$	-2.00%	8.88% $_{0.00}$	-4.48%	12.94% $_{0.04}$	-0.42%
infotocap	19.42%	17.85% $_{0.03}$	-1.57%	✗	✗	14.05% $_{0.03}$	-5.37%	18.74% $_{0.27}$	-0.68%
jhead	14.96%	14.94% $_{0.37}$	-0.02%	14.90% $_{0.00}$	-0.06%	14.85% $_{0.37}$	-0.11%	14.96% $_{1.00}$	0.00%
mp3gain	41.57%	38.52% $_{0.00}$	-3.05%	38.60% $_{0.00}$	-2.97%	34.47% $_{0.00}$	-7.10%	39.88% $_{0.00}$	-1.69%
mp42aac	7.15%	6.80% $_{0.00}$	-0.35%	✗	✗	6.78% $_{0.00}$	-0.37%	7.04% $_{0.16}$	-0.11%
mujs	27.97%	21.04% $_{0.00}$	-6.93%	20.79% $_{0.00}$	-7.18%	21.18% $_{0.00}$	-6.79%	27.26% $_{0.00}$	-0.71%
nm	7.80%	7.27% $_{0.00}$	-0.53%	✗	✗	6.69% $_{0.00}$	-1.11%	7.53% $_{0.01}$	-0.27%
objdump	6.98%	5.10% $_{0.00}$	-1.88%	✗	✗	6.60% $_{0.00}$	-0.38%	6.67% $_{0.00}$	-0.31%
pdftotext	16.69%	13.07% $_{0.00}$	-3.62%	✗	✗	14.77% $_{0.00}$	-1.92%	15.18% $_{0.00}$	-1.51%
tcpdump	18.36%	16.77% $_{0.21}$	-1.59%	17.25% $_{0.38}$	-1.11%	12.20% $_{0.21}$	-6.16%	17.33% $_{0.47}$	-1.03%
tiffsplit	20.96%	17.93% $_{0.00}$	-3.03%	✗	✗	13.34% $_{0.00}$	-7.62%	20.59% $_{0.47}$	-0.37%
wav2swf	2.04%	1.96% $_{0.00}$	-0.08%	1.89% $_{0.00}$	-0.15%	1.60% $_{0.00}$	-0.44%	2.00% $_{0.37}$	-0.04%
Average	-	-	-2.01%	-	-	-	-3.49%	-	**-0.60%**

TABLE 8: The hash overhead and collision in SAND.

Programs	NoHash	SAND-Hash		
	Speed	Speed_p	Overhead	Collision
imginfo	3,114	3,100 $_{0.65}$	100.47%	0
infotocap	3,011	2,982 $_{0.15}$	100.96%	0
jhead	3,327	3,327 $_{0.94}$	100.00%	0
mp3gain	1,929	1,915 $_{0.43}$	100.73%	0
mp42aac	1,702	1,688 $_{0.21}$	100.87%	0
mujs	1,841	1,812 $_{0.01}$	101.58%	0
nm	2,421	2,389 $_{0.26}$	101.36%	0
objdump	1,279	1,266 $_{0.36}$	101.05%	0
pdftotext	408	405 $_{0.00}$	100.66%	0
tcpdump	2,018	2,004 $_{0.52}$	100.73%	0
tiffsplit	2,335	2,334 $_{0.94}$	100.02%	0
wav2swf	2,830	2,817 $_{0.36}$	100.48%	0
Average	2,185	2,170	**100.74%**	**0**



(a) Average throughput. (b) Unique bugs breakdown.

Figure 7: The overall performance of SAND-MOpt.

AFL++, SAND-MOpt has achieved 2.2x and 16x throughput than MOpt-ASan/UBSan and MOpt-MSan, respectively. We also count the total number of unique bugs found by each fuzzer in Figure 7b. With no surprise, SAND-MOpt still discovers much more bugs than any other fuzzers. In particular, SAND-MOpt covers **36** extra bugs that are not found by any other fuzzers. These extraordinary results highlight the effectiveness and general applicability of SAND.

5. Discussion

Compatibility to other fuzzer advances. SAND does not touch the main fuzzing logic of a CGF. It is orthogonal to many other fuzzer advances. For example, new mutation strategies [2], [20], effective seed scheduling schemes [6], [4], and hybrid fuzzing techniques [13], [28] can be normally integrated into a CGF fuzzer, which SAND can further build upon.

Alternative test oracles. Sanitizers essentially provide test oracles for executions. These test oracles are customized for security vulnerabilities. In practice, there exist many other test oracles for different application scenarios. For instance, when fuzzing Javascript JIT compilers [3], a semantic correctness test oracle is usually provided. Differential test oracles are applied to find incorrect outputs such as wrong implementations [25] and platform-dependent divergences [36]. All these test oracles are usually expensive. Applying our idea to selectively feed inputs into the costly test oracle checkers could be beneficial and requires further verification and exploration.

Incompatibility to coverage-guided tracing. Our current execution pattern is collected from the coverage bitmap. There are some research efforts trying to reduce coverage collection overhead such as HexCite [23] and UnTracer [24]. Such approaches break the coverage map and thus cannot be used together with SAND. However, we would like

to highlight that the coverage collection overhead is much smaller compared to sanitizers. Researchers [32] have shown that the latest coverage collection approach used in AFL++ only brings a median of 15% overhead. Sanitizers like ASan and MSan can incur 237∼6,836% overhead. Therefore, even if these approaches can fully eliminate all coverage tracing overhead, the overall benefit when sanitizers are used is small.

Limitations. Despite that SAND brings significant improvement to fuzzing, it also comes with a few limitations. The first limitation is the gap between the unique execution pattern ratio and the bug-triggering input ratio. Our empirical evaluation in Section 2.3 has shown that many bug-triggering input ratios are below 0.5%, which is lower than the average unique execution pattern ratio of 2%. This gap indicates that there is still space for improvement. Designing more effective execution abstraction is an interesting future work. The second limitation is that although our evaluation has confirmed that SAND did not miss any bugs, we can not provide a theoretical guarantee. It would be interesting and useful to explore sound execution analysis to further eliminate this concern.

6. Related work

Reducing sanitizer overhead. Some research efforts exist to reduce sanitizer overhead. ASAP [29] removes sanitizer checks to meet a required performance budget. FuZZan [14] dynamically selects the most optimal metadata structure for both ASan and MSan to reduce sanitization overhead fuzzing. SanRazor [35] and Debloat [37] remove redundant sanitization checks via either static or dynamic analysis. SanRazor supports both ASan and UBSan while Debloat only supports ASan. All of these techniques require significant modifications to sanitizer implementations, which inevitably hinders their practical adoption. SAND, on the other hand, uses sanitizers normally. This feature further brings the orthogonality of SAND to these efforts. For instance, we can replace the ASan-enabled program with Debloat-enabled program to benefit from the improvement of Debloat.

Bug pattern. Igor [15] observes that all bug-triggering inputs have some unusual execution behavior. For specific bug types, UAFL [30] intuitively prioritizes memory operations of longer sequences to effectively detect User-After-Free bugs. Dowser [12] selectively checks instructions that access arrays in a loop for discovering buffer overflow bugs. ParmeSan [40] leverages the knowledge from sanitizer instrumentations to discover certain types of bugs faster. PGE [17] finds that bug-triggering executions correlate with execution prefixes. At a high level, the findings or insights behind these approaches share similar motivations to our execution pattern, *i.e.*, bug-triggering inputs tend to have unique execution features.

Improving fuzzing performance. Since the success of AFL [34], the fuzzing community has seen a broad range of new fuzzer developments. In particular, coverage-guided grey-box fuzzers such as AFL++ [10], AFLFast [6], and AFLGo [5] are the most widely adopted and studied fuzzing techniques. Researchers have also put great efforts into optimizing various aspects of fuzzing, such as seed scheduling [4], mutation strategies [20], [2], and path explorations [28], [13]. In theory, all these improvements are not related to sanitizer-enabled programs and therefore orthogonal to us. In practice, AFL++ has internally integrated many advances such as RedQueen [2] and MOpt [20]. We have demonstrated that our current implementation of SAND atop AFL++ can successfully support the default and MOpt modes. Extending our approach to other modes from AFL++ is inherently supported.

Reducing coverage collection overhead. Some researchers point out that coverage collection in fuzzing brings extra overhead. Untracer [24] and HexCite [23] remove instrumentation code in visited code edges to reduce coverage collection overhead. Zeror [38] shifts between diversely-instrumented binaries to achieve low coverage collection overhead on most executions. Odin [32] dynamically recompiles a binary when the instrumentation requirement changes. Because all these approaches need to modify the coverage bitmap, our approach is not compatible with them. As has been analyzed in Section 5, coverage collection cost is rather small compared to sanitizer overhead, and thus our approach is more beneficial.

7. Conclusion

We have presented a new fuzzing framework, SAND, to decouple sanitization from fuzzing loop. SAND performs fuzzing on the normally built program and only executes sanitizer-enabled programs when input is identified as sanitization-required. SAND utilizes the fact that most of the fuzzer-generated inputs do not need sanitization, which enables it to spend most of the fuzzing time on the normally built program. To identify sanitization-required inputs, we have designed a practical and useful execution analysis via execution pattern.

We have evaluated SAND on 12 real-world programs. Compared to the current fuzzing on sanitizer-enabled programs, SAND can significantly improve fuzzing performance by achieving 16x throughput and finding 30% more bugs. Our work represents an exciting research direction toward the overhead-free adoption of sanitizers in fuzzing.

References

[1] UndefinedBehaviorSanitizer — Clang 17.0.0git documentation. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. Accessed: June 7, 2023.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[3] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. Jit-picking: Differential fuzzing of javascript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–364, 2022.

[4] Marcel Böhme, Valentin JM Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 678–689, 2020.

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

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

[7] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzolic: mixing fuzzing and concolic execution. *Computers & Security*, 108:102368, 2021.

[8] Yann Collet. xxhash: Extremely fast hash algorithm. *GitHub https://github. com/Cyan4973/xxHash (2012-2022)*, 2016.

[9] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, August 2021.

[10] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research.

[11] Google. ClusterFuzz. https://google.github.io/clusterfuzz/, 2022. Accessed: June 7, 2023.

[12] Istvan Haller, Asia Slowinska, and Matthias Neugschwandtner. Dowsing for overflows: A guided fuzzer to find buffer boundary violations.

[13] Heqing Huang, Peisen Yao, Rongxin Wu, Qingkai Shi, and Charles Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1613–1627. IEEE, 2020.

[14] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. FuZZan: Efficient Sanitizer Metadata Design for Fuzzing.

[15] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. Igor: Crash Deduplication Through Root-Cause Clustering. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3318–3336, Virtual Event Republic of Korea, November 2021. ACM.

[16] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, Toronto Canada, October 2018. ACM.

[17] Shaohua Li and Zhendong Su. Accelerating Fuzzing through Prefix-Guided Execution. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):1–27, April 2023.

[18] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers.

[19] Sihang Liu, Suyash Mahar, Baishakhi Ray, and Samira Khan. Pmfuzz: Test case generation for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, 2021.

[20] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, and Yu Song. MOPT: Optimized Mutation Scheduling for Fuzzers.

[21] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. Fuzzing with data dependency information. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, pages 286–302, 2022.

[22] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini encyclopedia of psychology*, pages 1–1, 2010.

[23] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–365, 2021.

[24] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. Same Coverage, Less Bloat: Accelerating Binary-only Fuzzing with Coverage-preserving Coverage-guided Tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–365, Virtual Event Republic of Korea, November 2021. ACM.

[25] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *2017 IEEE Symposium on security and privacy (SP)*, pages 615–632. IEEE, 2017.

[26] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker.

[27] Evgeniy Stepanov and Konstantin Serebryany. MemorySanitizer: Fast detector of uninitialized memory use in C++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55, February 2015.

[28] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[29] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, pages 866–879. IEEE, 2015.

[30] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 999–1010, Seoul South Korea, June 2020. ACM.

[31] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19)*, 2019.

[32] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. Odin: on-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1010–1024, 2022.

[33] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328, 2017.

[34] Michal Zalewski. American fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2014. Accessed: March 7, 2022.

[35] Jiang Zhang, Shuai Wang, Manuel Rigger, Pingjia He, and Zhendong Su. SANRAZOR: Reducing Redundant Sanitizer Checks in C/C++ Programs.

[36] Qian Zhang, Jiyuan Wang, and Miryung Kim. Heterofuzz: Fuzz testing to detect platform dependent divergence for heterogeneous applications. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 242–254, 2021.

[37] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating Address Sanitizer.

[38] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. Zeror: speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 858–870, Virtual Event Australia, December 2020. ACM.

[39] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[40] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing.