SAND: Decoupling Sanitization from Fuzzing for Low Overhead

Ziqiao Kong[†‡]
ETH Zurich and
Nanyang Technological University
ziqiao001@e.ntu.edu.sg

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

Heqing Huang
City University of Hong Kong
heqhuang@cityu.edu.hk

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

***Abstract*—Sanitizers provide robust test oracles for various vulnerabilities. Fuzzing on sanitizer-enabled programs has been the best practice to find software bugs. Since sanitizers require heavy program instrumentation 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 the *sanitizer-enabled program* 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. We further introduce *execution pattern* to practically and effectively identify interesting inputs.

We implement SAND on top of AFL++ and evaluate it on 20 real-world programs. Our extensive evaluation highlights its effectiveness: in 24 hours, compared to all the baseline fuzzers, SAND significantly discovers more bugs while not missing any.

I. INTRODUCTION

Fuzzing has been one of the most successful approaches to finding security vulnerabilities [10], [41]. 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. When sanitizers are enabled at compile-time, compilers heavily instrument the target program to insert various checks. At run-time, violations of these checks result in program crashes. Fuzzing on such *sanitizer-enabled programs* is thus more effective in discovering software bugs. To date, the most widely-used sanitizers include AddressSanitizer (ASan) [27], UndefinedBehaviorSanitizer (UBSan) [1], and MemorySanitizer (MSan) [28].

Problems. Sanitizers, despite their extraordinary bug-finding capability, have two main drawbacks. ***First***, sanitizers bring significant performance overhead to fuzzing. As our evaluation in Section II-A will show, ASan, UBSan, and MSan averagely slow down fuzzing speed by a factor of 3.3x, 2.0x, and 45x, respectively. Since fuzzing is computationally intensive, such high sanitizer overheads inevitably impede both the performance of fuzzers and the adoption of sanitizers. Many approaches have been proposed to reduce the run-time overhead of sanitizers. For example, Debloat [39] optimizes ASan checks via sound static analysis. SANRAZOR [38] 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 effort, 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, which hinders its compatibility with diverse infrastructures. ***Second***, 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 programs and MSan-enabled programs separately to maximize its bug-finding capability.

```
1  _TIFFfree(*read_ptr);
2  ...
3  read_buff = *read_ptr;
4  if (!read_buff) {
5      read_buff = limitMalloc(buffsize);
6  } else {
7      if (prev_readsize < buffsize) {
8          new_buff = _TIFFrealloc(read_buff, buffsize);
9          if (!new_buff) {
10             free(read_buff);
11             read_buff = limitMalloc(buffsize);
12         } else
13             read_buff = new_buff;
14     }
15 }
16
17 read_buff[buffsize] = 0;
```

Fig. 1: A simplified Use-after-Free bug from libtiff in CVE-2023-26965. Line 17 triggers the bug because the freed buffer in line 1 is reallocated neither in line 5 nor in lines 8 and 11.

Key insight. Since sanitizers provide the security oracle, all current fuzzers execute sanitizer-enabled programs on every fuzzer-generated input to verify validity. We now raise this question: *Can we effectively filter bug-triggering inputs without executing a sanitizer-enabled program?* Theoretically, it seems paradoxical and infeasible as only by executing an input

§† **Equal contribution**
§‡ **This work was done when Ziqiao Kong was a master student at ETH Zurich**

can we know if the input is bug-triggering. However, our empirical evaluation will substantiate its feasibility. *The key insight is that bugs are strongly connected to execution paths.* For instance, Figure 1 shows a simplified code snippet from CVE-2023-26965, which contains a Use-after-Free bug in line 17. Normal and most execution paths are $\{1 \rightarrow 3 \rightarrow 5 \rightarrow 17\}$, $\{1 \rightarrow 3 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 17\}$, or $\{1 \rightarrow 3 \rightarrow 8 \rightarrow 13 \rightarrow 17\}$, where the freed buffer `read_buff` in line 1 is correctly reallocated. However, when the execution path is $\{1 \rightarrow 3 \rightarrow 17\}$, the freed buffer `read_buff` is incorrectly used in line 17. This buggy execution has a unique path not seen in other non-bug-triggering executions.

Since triggering this bug requires exercising unique execution paths, our intuition is that we can encapsulate inputs with unique execution paths by executing them on normally built programs, then only feed these inputs with unique execution paths into sanitizer-enabled programs to reduce overall sanitization overhead. More interestingly, our empirical evaluation in Section II-C will show that *nearly all bugs can be accurately captured by unique execution paths.* In fact, previous studies [15], [32] have also implicitly shown that bugs correlate highly to executions.

Our approach. Inspired by this observation, we propose a new fuzzing framework that decouples sanitization from the fuzzing loop for acceleration. In the framework, the fuzzer (1) performs fuzzing on a binary that is built normally, *i.e.*, without enabling sanitizers, then (2) selects inputs that have unique execution paths and runs them on the sanitizer-enabled binaries to check if they trigger any bugs. Take the program shown in Figure 1 as an example: During fuzzing, when we first encounter an input that has the execution path $\{1 \rightarrow 3 \rightarrow 5 \rightarrow 17\}$, we re-execute the input on the sanitizer-enabled binary. The result is that this input does not trigger any bug. Thus, we will not validate all future inputs that have the same execution path on the sanitizer-enabled binary. When we first encounter an input that has the execution path $\{1 \rightarrow 3 \rightarrow 17\}$, similarly, we re-execute the input on the sanitizer-enabled binary. The result is that this input triggers a Use-after-Free bug. As long as only (1) a small fraction of inputs have unique execution paths and (2) the buggy execution reliably has a unique execution path, we can significantly reduce the sanitization overhead during fuzzing. As Section IV-C will show, only 3.8% of all inputs on average have unique execution paths, and more than 91% of buggy executions have a unique execution path.

There is a key challenge in our approach: ***How to efficiently obtain execution path during fuzzing?*** Previous research has demonstrated that obtaining a fine-grained execution path is too costly to be practical [33], [9]. We tackle this problem by introducing *execution pattern* to approximate *execution path*. Execution pattern discards the order information in execution paths as a trade-off for efficiency. For instance, for the execution path $\{1 \rightarrow 3 \rightarrow 17\}$, its execution pattern is $\{1, 3, 17\}$ meaning that code regions $1, 3$ and 17 are executed. Although such approximation may cause imprecision in theory, our evaluation will demonstrate that this design is accurate enough to identify unique execution paths.

Our idea is generally applicable to the gray-box fuzzer family. Since AFL++ [8] is the most popular gray-box fuzzer in both academia and industry, we realized our idea on top of it and implemented a tool named SAND. We use 20 real-world programs widely used by the fuzzing community to evaluate SAND. Our evaluation shows that in 24 hours, compared to all the baseline fuzzers, SAND finds more bugs with statistical significance while not missing any bugs. In summary, we make the following contributions:

- We identify that bugs are strongly connected with unique execution paths and further design an approximate yet accurate execution pattern to efficiently obtain execution path information during fuzzing.
- We propose a novel fuzzing framework that decouples sanitization from the fuzzing loop by selectively feeding fuzzer-generated inputs into sanitizers.
- We implement our idea in a tool named SAND. We conduct in-depth evaluations to understand its effectiveness in terms of bug-finding, throughput, and coverage.

II. Observation and Illustration

In this section, we first show our observations on the high overhead of sanitizers and the rarity of bug-triggering inputs. Then, we use two real-world bug examples to illustrate the strong connections between bugs and execution paths.

A. High Overhead of Sanitizers

To benchmark sanitizer overhead in fuzzing, we use all 20 benchmark programs from our evaluation section. 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 [39] achieves the state-of-the-art optimization for ASan, we include it to understand the significance of its improvement. We use AFL++ as the default fuzzer. For each program, we:

Step (1) Use AFL++ to fuzz the native program and collect the first *one million* fuzzer-generated inputs. All these inputs are saved into disk. We use tmpfs [35] to reduce I/O overhead.

Step (2) Run AFL++ again on the native program to benchmark its running time on the saved one million inputs. AFL++ is adapted to fetch inputs from the disk instead of generation.

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

We ran the above experiment 10 times and reported the average fuzzing speed. All experimental settings are the same as our later evaluation in Section IV-A. Table I presents the average speed, *i.e.*, number of executions per second, of each program. Compared to native programs, ASan, UBSan, and MSan averagely reduce the speed by 326%, 196%, and 4,552%, respectively. Even for the best ASan optimization

TABLE I: Execution speed, *i.e.*, number of executions per second, of native programs and sanitizer-enabled programs. The column "Slowdown" refers to the ratio of the native speed to the sanitizer speed. It is calculated by dividing the native speed by the sanitizer speed. ✗ indicates a compilation failure. "-" indicates the incompatibility with the sanitizer.

Programs	Native	ASan		Debloat		UBSan		MSan	
	Speed	Speed	Slowdown	Speed	Slowdown	Speed	Slowdown	Speed	Slowdown
cflow	360	147	245%	164	220%	-	-	-	-
exiv2	298	92	324%	✗	✗	225	133%	-	-
ffmpeg	14	9	161%	✗	✗	3	422%	-	-
gdk-pixbuf-pixdata	237	68	348%	67	354%	227	104%	-	-
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%
jq	214	55	389%	53	401%	-	-	-	-
sqlite3	2,495	770	324%	✗	✗	1,401	178%	-	-
lame	101	64	157%	74	136%	-	-	-	-
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%
flvmeta	4,100	1,356	302%	1,351	304%	4,028	102%	-	-
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	**1,556**	**452**	**326%**	**508**	**322%**	**1,165**	**196%**	**42**	**4,552%**

TABLE II: Ratio of bugger-triggering inputs. "#Exec." shows the number of all executions. "#Trig." shows the number of bug-triggering executions.

Programs	#Exec.	#Trig.	Ratio
cflow	13.2M	1.2K	0.0%
exiv2	10.2M	278	0.0%
ffmpeg	2.7M	179	0.0%
gdk-pixbuf.	8.9M	246K	2.7%
imginfo	7.8M	65K	0.8%
infotocap	15.3M	31K	0.2%
jhead	16.1M	404K	2.5%
jq	4.7M	425	0.0%
sqlite3	17.4M	350K	2.1%
lame	4.4M	875	0.0%

Programs	#Exec.	#Trig.	Ratio
mp3gain	14.3M	104K	0.7%
mp42aac	3.6M	13	0.0%
mujs	13.4M	19K	0.1%
nm	10.1M	486	0.0%
flvmeta	23.8M	531K	2.2%
objdump	7.8M	196K	2.1%
pdftotext	5.8M	1K	0.0%
tcpdump	9.0M	13K	0.1%
tiffsplit	11.2M	26K	0.2%
wav2swf	8.3M	451K	6.3%
Average	**10.4M**	**122K**	**1.0%**

Debloat, its improvement over ASan is rather insignificant compared to the native programs. Such huge sanitizer overheads inevitably hinder the fuzzing throughput.

B. 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 number of all generated inputs as well as bug-triggering ones during 24 hours of fuzzing. The experimental data is from our later evaluation in Section IV-B. Table II shows the result. We can find that averagely *only 1%* of inputs are bug-triggering. For some programs, it is even rarer. For instance, on `pdftotext`, less than 2 out of 10^4 inputs trigger bugs. We can conclude that *Only a tiny fraction of fuzzer-generated inputs are bug-triggering*. Blindly sanitizing all of them is thus a huge waste.

```
1  int wav_convert2mono(struct WAV *dest, int rate)
2  {
3      ...
4      for(i=0; i < src->size; i += channels) {
5        int j;
6        int pos2 = ((int)pos)*2;
7        for(j=0;j < fill; j += 2) {
8          dest->data[pos2+j+0] = 0;
9          dest->data[pos2+j+1] = src->data[i]+128;
10       }
11       pos += ratio;
12     }
13     ...
14 }
```

Fig. 2: A simplified Buffer-Overflow bug from wav2swf in CVE-2017-11099. Line 8 triggers a buffer overflow when the `for` loops significantly change the buffer offset "`pos2+j`".

C. Illustrative Examples

In Figure 1, we have illustrated a Use-after-Free bug that has a unique execution path. In this section, we present motivating examples of different bug types.

Buffer-Overflow. A Buffer-Overflow bug is triggered when buffer access exceeds the allocated range of stack or heap

```
1  void JBIG2Stream::
2  readTextRegionSeg(Guint segNum, ...)
3  {
4      ...
5      numSyms = 0;
6      for (i = 0; i < nRefSegs; ++i) {
7        if ((seg = findSegment(refSegs[i]))) {
8          if (seg->getType()==jbig2SegSymbol) {
9             numSyms += seg->getSize();
10         } else if (seg->getType() == jbig2Se) {
11               codeTables->append(seg);
12         }
13     ...
14     syms = (JBIG2Bitmap **)gmallocn(numSyms);
15     ...
16 }
```

Fig. 3: A simplified Integer-Overflow bug from xpdf (containing `pdftotext`) in CVE-2022-38171. Line 9 triggers an integer overflow in `numSyms` when the `if` branch in line 8 is evaluated to `True` many times.

memory. Figure 2 shows a real-world Buffer-Overflow bug from wav2swf in CVE-2017-11099. The buggy buffer access is located in line 8. When the two `for` loops iterate a significant number of times, the offset `pos2+j` exceeds the buffer range of `dest->data`. The original cause is from the large values of both `src->size` and `fill`. But these unusual data subsequently lead to a unique execution path, *i.e.*, a long chain of executions $\{1 \to 4 \to 7 \to 8 \to 9 \to 8 \to 9 \to \cdots \to 4 \to 7 \to 8 \to 9 \to \cdots\}$. *In practice, we observe that Buffer-Overflow bugs often accompany execution path changes that normal executions do not exercise.* Thus, using unique execution paths as the indicator for sanitization can help us encapsulate bug-triggering inputs. The many Buffer-Overflow bugs identified by SAND in our evaluation will further confirm this rationale.

Integer-Overflow. Figure 3 shows an Integer-Overflow bug in line 9, where the variable `numSyms` overflows its valid range when the `if` guard in line 8 is frequently evaluated to true. This bug leads to an unusual execution path. The overflowed value in `numSyms` subsequently causes a small allocated buffer in line 14, which leads to buffer overflow and *dramatic path changes in the rest of the execution*.

III. Our Approach

This section introduces the design of our new fuzzing framework SAND. Section III-A defines *execution path* and its proxy approximation, *execution pattern*. Section III-B describes the fuzzing framework. Section III-C clarifies technical details.

A. Preliminary: Execution Path and its Proxy

Our approach is backed by the intuition that bug-triggering inputs have unique execution paths. We formally define the execution path as follows:

Definition III.1 (Execution Path)**.** Given an execution $\mathcal{E}$, the *execution path* of $\mathcal{E}$ is defined as $\Pi_{\mathcal{E}} = [e_1, e_2, \cdots, e_n]$, where e_i is the unique id of the code edge executed by $\mathcal{E}$. Note that, $\Pi_{\mathcal{E}}$ is ordered meaning that e_i is executed before e_j if $i < j$.

Execution path is a temporal transition sequence of all executed code when executing an input on the target program. It contains the full information on control-flow visits. For instance, suppose a buggy execution $\{1 \to 2 \to 4 \to 3\}$. It has the execution path as $[1, 2, 4, 3]$. Execution path is order-sensitive meaning that $[1, 2, 4, 3] \neq [1, 4, 2, 3]$. Unfortunately, obtaining the *execution path* of execution is too expensive to be practical in fuzzing [33], [9]. Since throughput is a key factor in fuzzing effectiveness, we cannot directly use *execution path* in our design. In this paper, we propose to use *execution pattern* as an approximate yet accurate proxy for the execution path. We define *execution pattern* as follows:

Definition III.2 (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 order-insensitive, *e.g.*, $\{e_1, e_2, e_3\} = \{e_2, e_3, e_1\}$.

Execution pattern records all executed code edges of an execution. For example, the previous buggy execution $\{1 \to 2 \to 4 \to 3\}$ has the execution pattern as $\{1, 2, 4, 3\}$. Execution pattern is order-insensitive, *e.g.*, $\{1, 2, 4, 3\}$ = $\{1, 2, 3, 4\}$. This execution pattern design can effectively approximate the execution path. Below, we discuss the soundness, realization, and alternative design of execution pattern in detail.

Soundness of execution pattern. Since *execution pattern* abstracts over *execution path*, it is theoretically possible that we cannot soundly capture all bug-triggering inputs with unique execution patterns. Take Figure 2 as an example, where inputs taking the loop once or multiple times will have the same "execution pattern". However, this is only true within this specific function. The actual execution pattern considers the whole program, and consequently, the same local execution pattern does not represent the overall execution pattern. At a high level, such loop iteration differences result from or will result in different data (*e.g.*, `src->size` and `pos`), which affect the execution of other parts of the program and eventually lead to divergent execution patterns. To further address this soundness concern, we provide an extensive evaluation in Section IV-C to demonstrate that the proposed execution patterns can precisely filter bug-triggering inputs.

Realization of execution pattern. An essential benefit of the execution pattern is its ease of acquisition during fuzzing. Fuzzers like AFL++ utilize an efficient data structure, *bitmap*, to collect visited code edges of an execution. Figure 4 shows an example of this procedure. The bitmap is initialized to all zeros for a new execution. For the execution path $[5, 3, 9, 7]$, the corresponding positions in the bitmap are marked to 1. This bitmap is then used to update a global coverage map with a logic OR. For the next execution $[1, 7, 9, 2]$, a similar bitmap is initialized and then marked. The coverage map is then cumulatively updated to record all code edges visited by all previous executions. The design of *execution pattern* allows us to obtain it effortlessly from the bitmap of execution, as shown in the middle right in Figure 4.

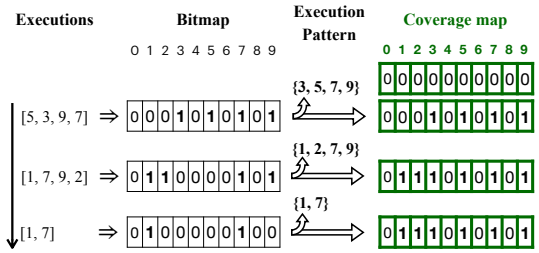

Fig. 4: Executions (left) are recorded by bitmap (middle), which are used in AFL++ to update the coverage map (right). Our execution patterns can be derived from these bitmaps.

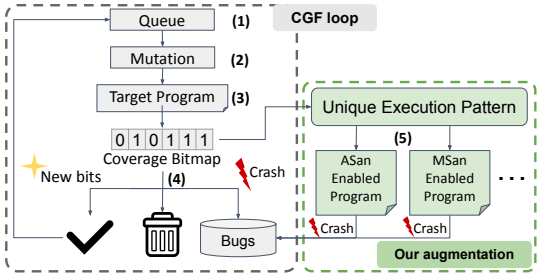

Fig. 5: SAND fuzzing loop.

Alternative design of execution pattern. Theoretically, we can design different execution patterns to abstract execution paths. There are two key requirements to execution pattern design: (1) it should precisely distinguish bug-triggering inputs from normal inputs and (2) it is cheap to obtain during fuzzing. Our current design meets all these requirements. Here, we discuss two possible alternatives. The first one is ***hit count***, where we include the counts of visited code edges into execution patterns. In principle, this design can capture more bug-triggering inputs. However, as our evaluation in Section IV-C will show, more than 13% of the normal inputs also have a unique hit count, which means that *hit count* does not meet the first requirement. Another one is ***coverage***, where we simply use the code coverage as the execution pattern. An input has a unique execution pattern when it increases code coverage. Unfortunately, our evaluation in Section IV-C shows that, on average, 55% of bug-triggering inputs do not increase coverage, causing this design to miss bugs during fuzzing. Our current design offers the first practical execution pattern implementation. Exploring a better design will improve the effectiveness of our approach and is orthogonal to our work.

B. Sanitization-decoupled Fuzzing

Based on the above formalization to execution patterns, we introduce our new fuzzing framework design. We first describe the general workflow of a coverage-guided fuzzer (CGF). The gray area in Figure 5 outlines the high-level sketch of a CGF. *Before fuzzing starts, the fuzzer compiles the target program with fuzzer instrumentation and/or sanitizers.* Then, it fuzzes the target as follows:

(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 a 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, we tackle this problem by decoupling sanitization from the conventional fuzzing loop. The green part in Figure 5 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 the *execution pattern* from the current execution's bitmap. If the execution pattern has been observed before, *i.e.*, not unique, discard it. Otherwise, the current input is identified as *sanitization-required*. The fuzzer then executes this input on each sanitizer-enabled program ($\mathcal{P}_{ASan}$, $\mathcal{P}_{MSan}$, *etc.*) and reports any discovered crashes.

In conditional sanitization, we only consider the unique, *i.e.*, first-seen, execution pattern to be sanitization-required. One may be concerned that bug-triggering inputs do not always have a first-seen execution pattern, and thus, our design may potentially miss many bugs. To address this concern, we provide extensive experiments in Section IV to demonstrate that (1) nearly all bug-triggering inputs (91.7%) have unique execution patterns, and (2) our new fuzzing framework does not miss any bugs. Intuitively, due to the stochastic nature of fuzzing, a bug can be triggered many times. Covering 91.7% of all bug-triggering inputs is sufficient to cover all bugs.

Example. Figure 6 illustrates the process of identifying sanitization-required inputs. Starting from the first execution with pattern $\{3, 5, 7, 9\}$, the fuzzer identifies it as a unique execution pattern and thus sanitization-required. The second execution has the same pattern as before; thus, sanitization is unnecessary. Similarly, the third and fifth executions have unique patterns not seen before and thus require sanitization. Our hypothesis is that all inputs triggering unique bugs also have unique execution patterns. Assuming that the fifth execution $\{6, 4, 2, 3, 5\}$ is buggy, the fuzzer can successfully identify the bug during sanitization. *Since executions holding the same execution path are likely to have similar semantics,* e.g., *exercising the same functionality or triggering the same bug,*

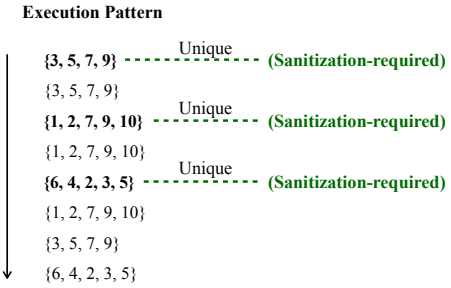

Fig. 6: Out of eight consecutive executions from top to bottom, three are identified as unique and thus require sanitization.

we only need to sanitize the execution with unique execution paths to identify the bug. This newly introduced **conditional sanitization** does not alter the standard fuzzing logic. Execution patterns are obtained from the already-available bitmap collected on the normally built program.

Algorithm 1 sketches the implementation pseudo-code of our new fuzzing framework. In each fuzzing loop (line 1), the fuzzer first 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 $bitmap$ (line 4). Then, it extracts the execution pattern $\mathcal{T}_{\mathcal{E}}$ from $bitmap$ (line 6) and determines whether or not this execution pattern has been observed (line 7). If $\mathcal{T}_{\mathcal{E}}$ is new, 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). Meanwhile, $\mathcal{T}_{\mathcal{E}}$ will be added to the hash table. If any execution crashes, meaning 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 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 standard fuzzing logic. It has the following main advantages:

- ***Orthogonal to CGFs.*** We introduce only an additional step to execute sanitizer-enabled programs on selected inputs. Conceptually, our approach can augment any AFL-family fuzzers without modifying their main fuzzing logic.
- ***Sanitizer inclusive.*** Some sanitizers like ASan and MSan are mutually exclusive, meaning that they cannot be used together on a program. Current fuzzers can only perform fuzzing on a program with only one of such sanitizers enabled. In our SAND, multiple sanitizer-enabled programs can be used for sanitization simultaneously. We will provide additional technical details in Section III-C to explain how we support multiple sanitizers.

C. Implementation

Unique execution pattern analysis. We obtain the execution pattern of an execution from its bitmap. In our implementation, we use the `simplify_trace()` function in AFL++ to achieve this goal. This design allows us to efficiently get execution patterns during fuzzing. To identify unique execution patterns, we calculate checksums of all observed execution

Algorithm 1: The New Fuzzing Loop of SAND

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

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;
```

patterns and use a hash map to store them. Algorithm 2 shows the pseudocode. The hash table HashTable is initialized to all zeros at the start of fuzzing. In our implementation, we use XXH32 hashing algorithm [36] because of its fast speed. Note that the hash function is used to identify unique execution patterns. In theory, any method that can separate unique patterns, such as bit-wise match, can be used here as an alternative solution to the hash function. However, the selected method will not affect the effectiveness of SAND as long as (1) it has negligible overhead and (2) all unique execution patterns can be precisely captured. Our evaluation in Section IV-F will show that the selected hash function meets all requirements.

Program instrumentation in SAND. The fuzz target $\mathcal{P}_{fuzz}$ is instrumented by SAND to include the necessary instrumentation code for coverage collection. Since all the sanitizer-enabled programs are used for sanitization only, no such instrumentation is needed. Thus, we directly use the LLVM compiler to compile $\mathcal{P}_{ASan}$, $\mathcal{P}_{UBSan}$, and $\mathcal{P}_{MSan}$. Because ASan and UBSan are compatible, we combine them as $\mathcal{P}_{ASan/UBSan}$. To reduce the burden of invoking these programs, we utilize the *forkserver* [37] to create one forkserver to communicate with all sanitizer-enabled programs efficiently during fuzzing.

IV. EVALUATION

We implemented SAND based on AFL++-4.05c [8], the latest version at the time of implementation. AFL++ is the

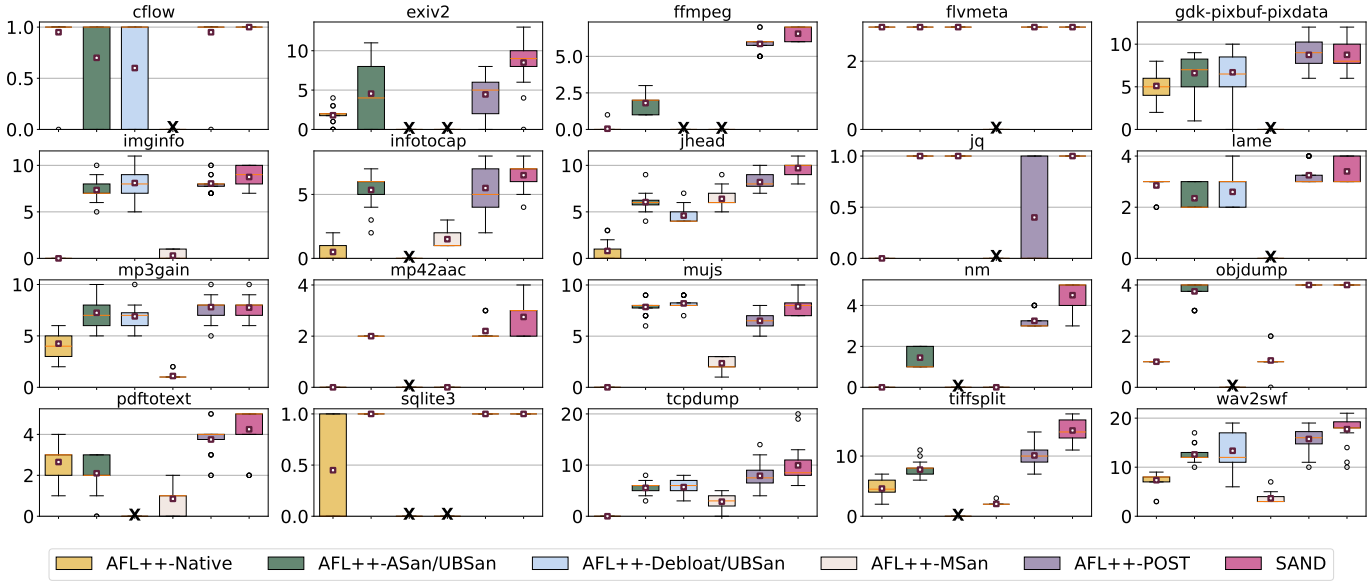

Fig. 7: Number of unique detected bugs across repetitions. ✗indicates a compilation failure or sanitizer incompatibility.

state-of-the-art gray-box fuzzer and has been widely used as the baseline fuzzer in many previous study [6], [19], [21].

A. Experimental Setup

Benchmark. We use real-world programs from the benchmarking test platform UniFuzz [18] for our evaluation. We use the provided seeds from UniFuzz for all fuzzing campaigns. To maximally understand Sand's capability in different sanitizers, we use all three popular sanitizers, *i.e.*, ASan, UBSan, and MSan. We evaluate all 20 programs on ASan and UBSan. Due to the compatibility issue of MSan, we failed to instrument 8 programs with MSan. We thus exclude MSan on their evaluation. For `cflow`, `lame`, and `jq`, all the UniFuzz provided seeds will crash UBSan due to unaligned pointers near program entry points, we thus use ASan instead of ASan/UBSan for these three programs. These programs cover a diverse range of input types, including

- **Image**: imginfo, jhead, tiffsplit, exiv2, gdk-pixbuf-pixdata.
- **Audio**: mp3gain, wav2swf, lame.
- **Video**: mp42aac, ffmpeg, flvmeta.
- **Text**: infotocap, mujs, pdftotext, cflow, jq, sqlite3.
- **Binary**: nm, objdump.
- **Network**: tcpdump.

There are other available benchmarks, such as Magma [12] and Fuzzbench [23]. We did not use them because (1) many sanitizer-reported bugs are not covered by Magma's bug set since Magma uses manually analyzed bugs, (2) Magma contains only 9 projects while UniFuzz contains 20 projects, and (3) Fuzzbench's programs contain too few bugs as its evaluation metric focuses on coverage.

Baseline. Since ASan and UBSan are compatible with each other, we combine them together when building binaries. All fuzzers and programs are built with LLVM-14, the latest stable version at the time of implementation. We compare the performance of Sand against four baseline fuzzers:

1) **AFL++-Native**: Fuzzing normally built programs
2) **AFL++-ASan/UBSan**: Fuzzing $\mathcal{P}_{ASan/UBSan}$.
3) **AFL++-MSan**: Fuzzing $\mathcal{P}_{MSan}$.
4) **AFL++-Debloat/UBSan**: To understand if Sand can surpass the existing sanitizer optimization schemes, we also choose the state-of-the-art ASan optimization technique, Debloat [39]. 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 `exiv2`, `ffmpeg`, `infotocap`, `mp42aac`, `sqlite3`, `nm`, `objdump`, `pdftotext` and `tiffsplit`with Debloat results in compilation or instrumentation failures, we exclude them for AFL++-Debloat/UBSan.
5) **AFL++-POST**: Another workaround to reduce sanitizer overhead is post-processing all the inputs saved in the corpus when fuzzing with AFL++-Native. These inputs increase coverage during fuzzing normally built programs. This baseline can also be viewed as using coverage-increasing information as the execution pattern.

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 20 times and ran all fuzzing campaigns for 24 hours.

B. 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:

TABLE III: Mean number of unique bugs across repetitions. ✗ indicates a compilation failure, and "-" indicates incompatibility. The largest mean numbers are highlighted in green.

Programs	AFL++-					SAND$_{p\text{-val},\hat{A}_{12}}$
	Native	ASan/ UBSan	Debloat/ UBSan	MSan	POST	
cflow	0.95	0.70	0.60	✗	0.95	1.00$_{0.34,0.53}$
exiv2	1.80	4.55	✗	✗	4.45	8.50$_{0.00,0.89}$
ffmpeg	0.05	1.80	✗	✗	5.85	6.55$_{0.00,0.78}$
gdk.	5.10	6.60	6.70	✗	8.75	8.75$_{0.92,0.49}$
imginfo	0.00	7.35	8.10	0.30	8.05	8.75$_{0.04,0.68}$
infotocap	0.50	5.35	✗	1.50	5.50	6.50$_{0.04,0.68}$
jhead	0.80	6.05	4.60	6.40	8.20	9.70$_{0.00,0.86}$
jq	0.00	1.00	1.00	✗	0.40	1.00$_{0.00,0.80}$
sqlite3	0.45	1.00	✗	✗	1.00	1.00$_{1.00,0.50}$
lame	2.85	2.35	2.60	✗	3.25	3.40$_{0.33,0.57}$
mp3gain	4.25	7.25	6.90	1.10	7.80	7.75$_{0.72,0.47}$
mp42aac	0.00	2.00	✗	0.00	2.20	2.75$_{0.01,0.69}$
mujs	0.00	7.85	8.20	2.35	6.50	7.90$_{0.00,0.84}$
nm	0.00	1.45	✗	0.00	3.25	4.50$_{0.00,0.90}$
flvmeta	3.00	3.00	3.00	✗	3.00	3.00$_{1.00,0.50}$
objdump	1.00	3.75	✗	1.05	4.00	4.00$_{1.00,0.50}$
pdftotext	2.65	2.10	✗	0.85	3.75	4.25$_{0.02,0.70}$
tcpdump	0.00	5.55	5.70	2.85	7.90	9.95$_{0.08,0.66}$
tiffsplit	4.60	7.75	✗	2.05	10.10	14.25$_{0.00,0.97}$
wav2swf	7.35	12.60	13.35	3.65	15.75	17.75$_{0.00,0.79}$

Q1 Does SAND find *more bugs* compared to other fuzzers?

Q2 Does SAND *miss any bugs* found by other fuzzers?

To answer these questions, we collect all crashes found by each fuzzer. We triage all crashes according to their root causes to quantify the number of unique bugs each fuzzer finds. Our deduplication is done via both the stack frame information from GDB [18] and manual analysis.

New bugs. Before discussing the evaluation result of SAND, we would like to mention that during the evaluation, we found nine new bugs using SAND. All of these bugs have been confirmed by the developers. Three of them were assigned CVE numbers. Considering all the programs have been heavily fuzzed in both academia and industry, these new bugs reflect the effectiveness of our approach.

Number of Unique Bugs. We plot the number of unique bugs in every repetition in Figure 7. Table III aggregates the results and reports the mean number of unique bugs. On 13 out of 20 programs, SAND found more bugs than all other fuzzers. For the remaining 7 programs, there is no statistical difference between SAND and the second-best fuzzer. Since AFL++-POST is the overall second-best fuzzer, we report the Mann-Whitney U-test [22] (p-val) and Vargha-Delaney [30] effect size ($\hat{A}_{12}$) when comparing SAND with it. The p-val column shows that SAND is statistically different from AFL++-POST (p-val < 0.05) on 12 programs. There is only one case (`mp3gain`) where SAND has a lower mean number of bugs (7.75 v.s 7.80). However, there is no statistical significance (p-val=0.72) between them. Intuitively, the $\hat{A}_{12}$ column measures to what extent or the probability that SAND is better than AFL++-POST. The result shows that 8 programs have $\hat{A}_{12}$ > 0.71 (conventionally large effect size) and 5 programs have $\hat{A}_{12}$ > 0.64 (conventionally medium effect size). Notably, there is no program with $\hat{A}_{12}$ < 0.44, meaning that AFL++-POST cannot surpass SAND on any programs. In summary, our result answers **Q1**: SAND *has a significantly stronger bug-finding capability than all other fuzzers.*

Accumulative Number of Unique Bugs. To understand the overlaps of bugs found by different fuzzers, we accumulate all unique bugs found by each fuzzer in each repetition. Table IV shows the unique number of bugs found by each fuzzer. It shows that SAND *covers the most number of bugs (204)*. The "SAND−" shows that SAND *does not miss any bugs found by any other fuzzers.* The "SAND+" shows that SAND *always finds more bugs than other fuzzers.* Compared to the second-best fuzzer, AFL++-POST, SAND found 30 more bugs, which also confirms that simply post-processing coverage-increasing inputs will miss many bugs. In summary, we can answer **Q1** and **Q2**: SAND *does not miss any bugs and can find significantly more bugs.* We will discuss the potential false negative problem of SAND later.

Number of Unique Bugs Reported by Sanitizer-enabled Programs. Of all the 204 unique bugs identified by SAND, more than 65% are not detectable on normally built programs. These bugs are reported after invoking sanitizer-enabled programs in SAND, highlighting the necessity of using sanitizers.

Bug Types. We now try to understand which types of bugs SAND can cover. Our observation, as illustrated in Section II-C, is that all bug types, like Buffer-Overflow and Integer-Overflow, can result from/in execution path changes. Figure 8 reports the types of bugs found by SAND. The result highlights that SAND can indeed cover all bug types, such as heap Buffer-Overflow (47 bugs), Integer-Overflow (60 bugs), Use-of-Uninitialized-Memory (45 bugs), and Use-after-Free (2 bugs). The small number of Use-after-Free bugs is because they are indeed relatively rare in practice [18].

False Negatives. *Despite the outstanding performance of* SAND*, we have no theoretical guarantee that* SAND *can cover all bugs*. Theoretically, SAND may have false negatives where certain bugs are missed. This false negative impact can be inferred from the `mujs` performance in Table III, where SAND has a slightly lower mean number of bugs (7.9) than AFL++-Debloat/UBSan (8.2). The reason is that one bug in `mujs` was not triggered in every repetition of SAND, but in every repetition in the other fuzzer. The execution pattern for this bug can sometimes be seen in normal executions, which leads to a less frequent discovery in SAND. Due to the stochastic nature of fuzzing, triggering a bug only once is sufficient for SAND to detect it in practice. Given the overall superior performance

TABLE IV: Number of unique bugs found by each fuzzer. "Total" row shows the total number of unique bugs. "SAND−" means the number of bugs missed by SAND. "SAND+" means the number of bugs additionally covered by SAND.

	AFL++-					SAND
	Native	ASan/ UBSan	Debloat/ UBSan	MSan	POST	
Total	70	145	100	54	174	**204**
SAND−	0	0	0	0	0	-
SAND+	134	59	104	150	30	-

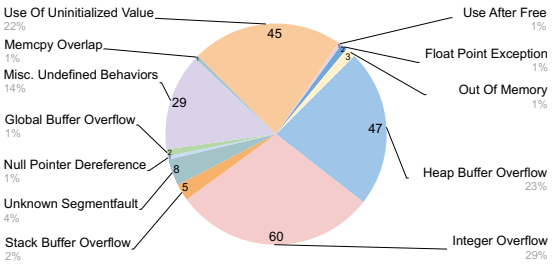

Fig. 8: Distribution of bug types found by SAND.

of SAND, we believe that the moderate false negative issue is acceptable and does not impede its general effectiveness.

C. Effectiveness of Execution Pattern

We now break down SAND to evaluate the effectiveness of its core design, *i.e.*, execution pattern. To understand if unique execution patterns can accurately encapsulate bug-triggering inputs/executions, we extensively analyze all executions during fuzzing. Specifically, we conduct the following experiments:

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

Step (2) For each generated input, we first obtain its execution pattern, then examine whether or not this execution pattern is unique, *i.e.*, has been observed before.

Step (3) For each input, no matter whether or not its execution pattern is unique, we run it on ASan-, UBSan-, and MSan-enabled programs to test if it triggers a bug.

To explore and measure other designs of the execution pattern, we provide the following two alternatives:

Alternative 1: Hit Count. Recall our current execution pattern definition in III.2, where we do not present the hit count of each visited code edge. In this alternative design, we include the hit count of each code edge in the execution pattern.

Alternative 2: Coverage. In this design, the execution pattern is the coverage-increasing information. Whenever an execution increases code coverage, we identify it as sanitizer-required. This design is actually equivalent to the strategy used in AFL++-POST.

We ran the experiments on our current execution pattern as well as two alternative designs for 24 hours and repeated them twenty times. With this set of experiments, for each execution pattern design, we want to answer:

Q3 Out of all executions in **Step (2)**, how many of them are marked as having unique execution patterns?

Q4 Out of all bug-triggering inputs/executions in **Step (3)**, how many of them are also marked as having unique execution patterns in **Step (2)**?

The above **Q3** can tell us the ratio of unique execution patterns during fuzzing. A smaller ratio indicates that sanitization is required less frequently, leading to a higher speed. The following **Q4** can inform us how effective the unique execution pattern is in filtering bug-triggering inputs. Table V shows the result. The "All-SAND" column shows that more than half of all programs have less than 1% unique execution patterns in our current design. The average ratio is only 3.8%. We can thus conclude that *Only a small fraction of fuzzer-generated inputs have unique execution patterns.* The "Bug-SAND" column shows that, on average, 91.7% of bug-triggering inputs have unique execution patterns, which means that if we only pass inputs with unique execution patterns to sanitizer-enabled programs, we can successfully sanitize 91.7% buggy inputs. One abnormal case is `lame` with a 42.7% ratio. The reason is that `lame` is not stable, *i.e.*, executing even the same input multiple times can lead to divergent execution paths. In fact, even vanilla AFL++ discourages fuzzing unstable programs [7]. Note that we do not deduplicate all bug-triggering inputs here; most of them are in fact duplicates. Considering the same bug can be triggered multiple times during fuzzing, 91.7% accuracy can already ensure that no bugs will be missed with a high probability.

The "Bug-Hit" column shows that when we use **Hit Count** as the execution pattern, averagely 97.1% of bug-triggering inputs have unique execution patterns (higher than 91.7% in SAND), but the average ratio of all inputs ("All-Hit" column) is as high as 13.1% (much higher than 3.8% in SAND). This means that 13.1% of inputs will be identified as sanitizer-required, hindering the fuzzing throughput dramatically.

The "Bug-Cov" column shows that when we use **Coverage** as the execution pattern, averagely only 45% of bug-triggering inputs have unique execution patterns. Such a low ratio will cause many bug-triggering inputs to not be sanitized, consequently missing bugs. Our previous bug-finding evaluation of AFL++-POST in Table IV also shows that using coverage will miss 30 out of 204 bugs. Note that since we do not duplicate all bug-triggering inputs here, 45% of bug-triggering inputs does not imply 45% of all bugs.

In summary, compared to the two alternative designs, our current execution pattern design achieves the best balance between the two ratios. However, this does not mean that our current design is perfect. Ideally, the ratio of all inputs should be as low as possible, while the ratio of bug-triggering inputs should remain as high as possible. Exploring better execution pattern design is an exciting future work.

D. Fuzzing Throughput

We analyzed the end-to-end fuzzing throughput, *i.e.*, the total number of inputs executed during fuzzing. Figure 9 shows

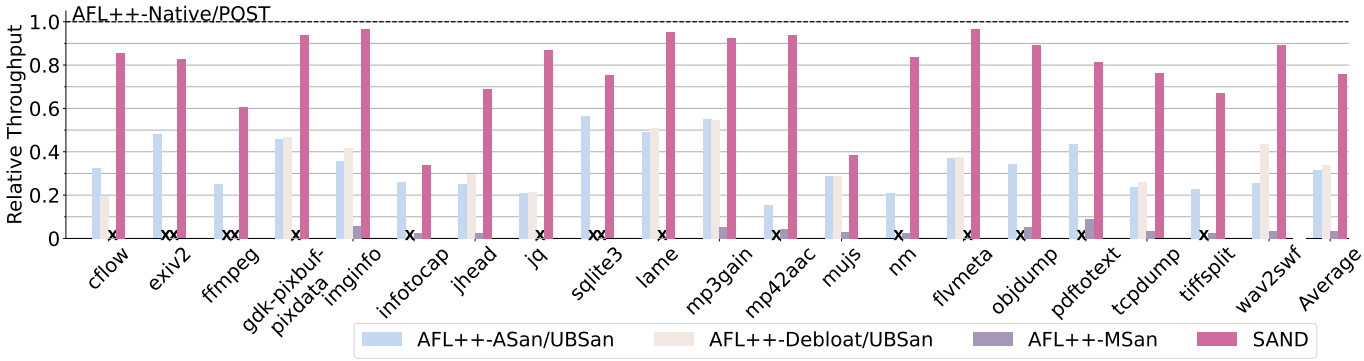

Fig. 9: Relative throughput normalized to AFL++-Native/POST. ✗indicates a compilation failure or an incompatible sanitizer. "Average" refers to the average throughput of all programs.

TABLE V: 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.

Programs	All			Bug		
	Hit	Cov	SAND	Hit	Cov	SAND
cflow	17.7	< 0.1	2.3	100.0	2.8	100.0
exiv2	3.2	< 0.1	0.2	87.4	23.2	75.3
ffmpeg	11.3	< 0.1	0.8	100.0	75.2	99.9
gdk-pixbuf.	2.0	< 0.1	< 0.1	89.2	65.6	86.7
imginfo	0.6	< 0.1	< 0.1	82.8	29.9	60.0
infotocap	20.7	< 0.1	6.5	99.9	7.7	95.3
jhead	8.6	< 0.1	1.0	100.0	13.1	92.3
jq	10.1	< 0.1	3.1	98.3	0.6	60.6
sqlite3	25.4	< 0.1	7.5	100.0	69.3	100.0
lame	72.2	< 0.1	42.7	88.5	80.8	88.5
mp3gain	40.7	< 0.1	0.4	99.3	64.2	89.8
mp42aac	< 0.1	< 0.1	< 0.1	100.0	52.6	100.0
mujs	20.1	< 0.1	4.7	100.0	25.9	98.7
nm	1.0	< 0.1	0.2	100.0	59.5	94.4
flvmeta	0.3	< 0.1	< 0.1	98.6	2.8	97.2
objdump	4.2	< 0.1	0.7	99.9	83.6	99.9
pdftotext	20.2	< 0.1	3.7	100.0	84.6	99.9
tcpdump	2.8	< 0.1	0.8	99.8	70.9	99.7
tiffsplit	1.1	< 0.1	0.5	99.0	18.0	98.5
wav2swf	< 0.1	< 0.1	< 0.1	100.0	70.1	97.7
Average	13.1	0.02	3.8	97.1	45.0	91.7

the average throughput normalized to AFL++-Native. Since AFL++-POST has the same throughput as AFL++-Native, we use AFL++-Native/POST to represent both.

Compared to Sanitizers-enabled Fuzzers. SAND achieves an average of 2.4x, 2.1x, and 20.0x throughput than AFL++-ASan/UBSan, AFL++-Debloat/UBSan, and AFL++-MSan, respectively. Moreover, SAND has significantly higher throughput *on all programs*. On some programs, the speedup rate is even higher. For instance, on `nm`, SAND executes 4.2x and 31.8x more inputs than AFL++-ASan/UBSan and AFL++–MSan, respectively. *It is worth mentioning that* SAND *is equipped with all three sanitizers, including the slowest MSan.*

Compared to AFL++-Native/POST. Overall, SAND achieves 75% of AFL++-Native's throughput. On 6 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.*

E. Code Coverage

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

Compared to Sanitizers-enabled Fuzzers. SAND achieves higher coverage on *all programs*. Intuitively, since SAND has a much higher throughput than other sanitizer-enabled fuzzers, SAND executes more inputs and thus achieves higher coverage.

Compared to AFL++-Native/POST. On 11 out of 20 programs, there is no significant coverage difference between AFL++-Native/POST and SAND. For the remaining 9 programs, SAND achieves almost the same code coverage as AFL++-Native. 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 metric for fuzzing, although AFL++-Native/POST can achieve higher code coverage, it has a worse bug-finding rate and thus is less favorable in practice.

F. Hash Function

We evaluated both the *hash overhead* and *hash collision* to demonstrate that our selected hash function is sufficiently effective to support SAND. Our first evaluation measured the overall time spent on the hash function during fuzzing. The result shows that *hashing operations in* SAND *have an average of 1% overhead, which is negligible in fuzzing*. Our second evaluation measured if there was any hash collision, *i.e.*, two different execution patterns have the same hash value. The result shows that nearly *no hash collision was detected during 24 hours of fuzzing*. These evaluations convincingly demonstrate the effectiveness of our selected hash function.

V. Discussion

Compatibility to Other Advanced Fuzzers. 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 [5], [3],

TABLE VI: Code coverage (%) of fuzzers with statistical p-val. ✗indicates a compilation failure. The highest code coverage compared to AFL++-Native is highlighted in green.

Programs	AFL++ -Native /-POST	AFL++- ASan/ UBSan	AFL++- Debloat/ UBSan	AFL++- MSan	SAND
cflow	24.30	$23.93_{0.00}$	$21.95_{0.00}$	✗	24.26 $_{0.10}$
exiv2	6.02	$3.13_{0.00}$	✗	✗	5.83 $_{0.16}$
ffmpeg	2.70	$0.31_{0.00}$	✗	✗	2.61 $_{0.00}$
gdk-pixbuf.	21.49	$18.54_{0.00}$	$18.13_{0.00}$	✗	21.32 $_{0.67}$
imginfo	13.17	$11.53_{0.00}$	$11.82_{0.00}$	$8.61_{0.00}$	12.75 $_{0.16}$
infotocap	20.33	$11.40_{0.00}$	✗	$13.55_{0.00}$	18.87 $_{0.01}$
jhead	15.31	$9.96_{0.00}$	$9.68_{0.00}$	$15.18_{0.00}$	15.31 $_{1.00}$
jq	31.11	$30.00_{0.00}$	$28.50_{0.00}$	✗	31.14 $_{0.86}$
sqlite3	43.34	$36.81_{0.00}$	✗	✗	38.93 $_{0.00}$
lame	31.96	$31.29_{0.00}$	$28.31_{0.00}$	✗	31.88 $_{0.24}$
mp3gain	41.31	$39.40_{0.00}$	$39.14_{0.00}$	$34.34_{0.00}$	40.38 $_{0.01}$
mp42aac	7.09	$5.11_{0.00}$	✗	$7.50_{0.00}$	7.01 $_{0.13}$
mujs	28.18	$16.08_{0.00}$	$16.51_{0.00}$	$20.35_{0.00}$	27.00 $_{0.00}$
nm	7.82	$3.48_{0.00}$	✗	$6.32_{0.00}$	7.58 $_{0.00}$
flvmeta	5.28	$3.36_{0.00}$	$3.19_{0.00}$	✗	5.28 $_{1.00}$
objdump	6.96	$6.40_{0.00}$	✗	$6.40_{0.00}$	6.87 $_{0.00}$
pdftotext	16.46	$13.18_{0.00}$	✗	$14.92_{0.00}$	15.34 $_{0.00}$
tcpdump	18.76	$9.34_{0.00}$	$8.65_{0.00}$	$11.74_{0.00}$	15.94 $_{0.01}$
tiffsplit	21.06	$18.25_{0.00}$	✗	$12.31_{0.00}$	20.91 $_{0.33}$
wav2swf	1.92	$1.91_{0.00}$	$1.87_{0.00}$	$1.61_{0.00}$	1.89 $_{0.32}$

and hybrid fuzzing techniques [13], [29] can all be normally integrated into a CGF fuzzer, which SAND can build upon. To understand SAND 's general applicability, we port it to an alternative fuzzer MOpt [20], which uses a different mutation scheduling strategy and can be manually turned on in AFL++. We include the details in the supplementary.

Incompatibility to Coverage-guided Tracing. Our current execution pattern is collected from the coverage bitmap. Some research efforts are trying to reduce coverage collection overhead, such as HexCite [24] and UnTracer [25]. 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 [34] 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. Even if these approaches can entirely eliminate 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 II-B has shown that many bug-triggering input ratios are below 0.5%, which is lower than the average unique execution pattern ratio of 3.8%. 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 eliminate this concern.

VI. Related work

Reducing Sanitizer Overhead. ASAP [31] removes sanitizer checks to meet a required performance budget. FuZZan [14] dynamically selects an optimal metadata structure for ASan and MSan. SanRazor [38] and Debloat [39] remove redundant sanitizer 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 without any modification, which further highlights the orthogonality of SAND to these efforts.

Bug pattern. Igor [15] observes that all bug-triggering inputs have unusual execution behaviors. For specific bug types, UAFL [32] prioritizes memory operations of longer sequences to effectively detect User-After-Free bugs. Dowser [11] selectively checks instructions that access arrays in a loop for detecting buffer overflow bugs. ParmeSan [26] 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 [37], the fuzzing community has seen a broad range of new fuzzer developments. In particular, coverage-guided grey-box fuzzers such as AFL++ [8], AFLFast [5], and AFLGo [4] 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 [3], mutation strategies [20], [2], and path explorations [29], [13]. In theory, all these improvements are not related to sanitizer-enabled programs and, therefore, are orthogonal to us.

Reducing Coverage Collection Overhead. Some researchers point out that coverage collection in fuzzing brings extra overhead. Untracer [25] and HexCite [24] remove instrumentation code in visited code edges to reduce coverage collection overhead. Zeror [40] shifts between diversely-instrumented binaries to achieve low coverage collection overhead on most executions. Odin [34] 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 discussed in Section V, coverage collection cost is rather small compared to sanitizer overhead, and thus, our approach is more beneficial.

VII. Conclusion

We have presented a new fuzzing framework, SAND, to decouple sanitization from the fuzzing loop. SAND performs

fuzzing on the normally built program and only executes sanitizer-enabled programs when an 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 normally built programs. We have evaluated SAND on 20 real-world programs to demonstrate its superiority. Our work represents an exciting research direction toward the overhead-free adoption of sanitizers in fuzzing.

REFERENCES

[1] UndefinedBehaviorSanitizer — Clang 18.0.0git documentation. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2022. Accessed: December 7, 2023.

[2] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Proceedings 2019 Network and Distributed System Security Symposium*, NDSS'19, 2019.

[3] Marcel Böhme, Valentin J. M. 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*, ESEC/FSE'20, 2020.

[4] 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*, CCS'17, 2017.

[5] 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*, CCS'16, 2016.

[6] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzolic: Mixing fuzzing and concolic execution. *Computers and Security*, 108, sep 2021.

[7] AFL++ developers. AFLplusplus Performance. https://aflplus.plus/docs/faq/#performance, 2024. Accessed: July 7, 2024.

[8] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, 2020.

[9] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, S&P'18, 2018.

[10] Google. ClusterFuzz. https://google.github.io/clusterfuzz/, 2023. Accessed: November 7, 2023.

[11] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, 2013.

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

[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*, S&P'20, 2020.

[14] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. Fuzzan: Efficient sanitizer metadata design for fuzzing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, ATC'20, 2020.

[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*, CCS'21, Virtual Event Republic of Korea, 2021.

[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*, CCS'18, pages 2123–2138, Toronto Canada, 2018.

[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. In *Proceedings of the 30th USENIX Conference on Security Symposium*, SEC'21, 2021.

[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*, ASPLOS'21, 2021.

[20] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. Mopt: Optimized mutation scheduling for fuzzers. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, 2019.

[21] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. Fuzzing with data dependency information. In *2022 IEEE European Symposium on Security and Privacy*, EuroS&P'22, 2022.

[22] Patrick E McKnight and Julius Najab. Mann-whitney u test. *The Corsini Encyclopedia of Psychology*, 2010.

[23] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE'21, pages 1393—-1403, 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*, CCS'21, 2021.

[25] 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*, CCS'21, Virtual Event Republic of Korea, 2021.

[26] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Parmesan: Sanitizer-guided greybox fuzzing. In *Proceedings of the 29th USENIX Conference on Security Symposium*, SEC'20, 2020.

[27] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ATC'12, 2012.

[28] 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'15, 2015.

[29] 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 *Proceedings 2016 Network and Distributed System Security Symposium*, NDSS'16, 2016.

[30] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[31] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *2015 IEEE Symposium on Security and Privacy*, S&P'15, 2015.

[32] 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*, ICSE'20, Seoul South Korea, 2020.

[33] 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.

[34] 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*, PLDI'22, 2022.

[35] 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*, CCS'17, 2017.

[36] xxHash developers. xxhash: Extremely fast hash algorithm. https://github.com/Cyan4973/xxHash, 2016. Accessed: December 7, 2023.

[37] Michal Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/, 2014. Accessed: March 7, 2022.

[38] Jiang Zhang, Shuai Wang, Manuel Rigger, Pinjia He, and Zhendong Su. SANRAZOR: Reducing redundant sanitizer checks in C/C++ programs. In *15th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'21, 2021.

[39] Yuchen Zhang, Chengbin Pang, Georgios Portokalidis, Nikos Triandopoulos, and Jun Xu. Debloating address sanitizer. In *31st USENIX Security Symposium (USENIX Security 22)*, SEC'22, 2022.

[40] 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*, ASE'20, Virtual Event Australia, 2020.

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