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 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], [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. 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) [28], UndefinedBehaviorSanitizer (UBSan) [2], and MemorySanitizer (MSan) [29].

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 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 [37] removes likely redundant ASan and UBSan checks through dynamic profiling. FuZZan [15] 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.

```
1  _TIFFfree(*read_ptr);
2  ...
3  read_buff = *read_ptr;
4  if (!read_buff)
5  {
6      read_buff = limitMalloc(buffsize);
7  }
8  else
9  {
10     if (prev_readsize < buffsize)
11     {
12         new_buff =
13             _TIFFrealloc(read_buff, buffsize);
14         if (!new_buff)
15         {
16             free(read_buff);
17             read_buff = limitMalloc(buffsize);
18         }
19         else
20             read_buff = new_buff;
21     }
22 }
23
24 read_buff[buffsize] = 0;
```

Figure 1: A simplified Use-after-Free bug from libtiff in CVE-2023-26965. Line 24 triggers the bug because the freed buffer in line 1 is not re-allocated neither in line 6 nor in lines 13 and 17.

0. † **Equal contribution**

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 validity. We now raise this question: *Can we decide whether an input triggers a bug without truly executing a sanitizer-enabled program?* Theoretically, it seems paradoxical and infeasible as only by executing an input 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 24. Normal and most execution paths are {1 ! 3 ! 6 ! 24}, {1 ! 3 ! 12 ! 16 ! 17 ! 24}, or {1 ! 3 ! 12 ! 20 ! 24}, where the freed buffer `read_buff` in line 1 is correctly re-allocated. However, when the execution path is {1 ! 3 ! 24}, the freed buffer `read_buff` is incorrectly used in line 24. This buggy execution has a unique path not seen in other normal executions.

Since triggering such control-flow sensitive bugs 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 into sanitizer-enabled programs to reduce overall sanitization overhead. Conceptually, our intuition can effectively tackle control-flow sensitive bugs, e.g., Use-after-Free bugs, since triggering such kinds of bugs needs to exercise a unique execution path, e.g., from the free point to the use point. For other bugs, such as Buffer-Overflow and Use-of-Uninitialized-Memory, they are more "data sensitive". For example, triggering a Buffer-Overflow bug often requires significantly changing the buffer offset value. Nevertheless, such data-flow changes can often result from or be reflected by control-flow information [13]. As our illustration in Section 2.3 shows, Buffer-Overflow bugs are often control-flow related, such as unusual loop iterations, and thus can be identified via unique execution paths. In fact, previous studies [16], [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 ! 3 ! 6 ! 24}, we re-execute the input on the sanitizer-enabled binary. The result is that this input does not trigger any bug. For all future inputs that have the same execution path, we will not validate them on the sanitizer-enabled binary. When we first encounter an input that has the execution path {1 ! 3 ! 24}, 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 our evaluation in Section 4 will show, only less than 2% of inputs on average have unique execution paths, and more than 96% 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 fine-grained execution information like execution path is too costly to be practical [33], [10]. In our implementation, we tackle this problem by designing an efficient and scalable execution pattern to approximate the execution path. Execution pattern discards the order information in the execution path as a trade-off for efficiency. For instance, for the execution path {1 ! 3 ! 24}, its execution pattern is {1; 3; 24} meaning that code regions 1; 3 and 24 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++ [9] 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 12 real-world programs widely used by the fuzzing community to evaluate SAND. Our evaluation shows that in 24 hours, compared to traditional 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, 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.

2. Observation and Illustration

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

TABLE 1: 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. 7 indicates a compilation failure.

Programs	Native	ASan		Debloat		UBSan		MSan	
	Speed	Speed	Slowdown	Speed	Slowdown	Speed	Slowdown	Speed	Slowdown
imginfo	2,964	869	3.4	907	3.3	1,968	1.5	43	68.4
infotocap	2,676	685	3.9	7	7	1,962	1.4	43	62.1
jhead	2,963	859	3.5	888	3.3	2,652	1.1	45	66.1
mp3gain	1,488	627	2.4	634	2.4	917	1.6	42	35.3
mp42aac	1,917	472	4.1	7	7	682	2.8	42	46.1
mujs	1,491	425	3.5	440	3.4	685	2.2	42	35.9
nm	2,209	586	3.8	7	7	1,597	1.4	43	50.8
objdump	573	212	2.7	7	7	250	2.3	38	15.1
pdftotext	410	151	2.7	7	7	192	2.1	35	11.7
tcpdump	1,754	432	4.1	493	3.6	561	3.1	42	42.0
tiffsplit	2,093	665	3.2	7	7	1,247	1.7	43	48.7
wav2swf	2,757	486	6.0	517	5.5	1,211	2.5	43	63.6
Average	1,941	539	3.6	-	-	1,160	2.0	42	45.5

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%**

2.1. 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 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* 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 one 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 ran the above experiment 10 times and reported the average fuzzing speed. All experimental settings are the same as our later evaluation in Section 4.1. Table 1 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 3.6x, 2.0x, and 45.5x, respectively. Specifically, ASan reduces the speed by 2.4x 6.0x, UBSan by 1.1x 3.1x, and MSan by 11.7x 68.4x. 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 then 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

1. We use tmpfs [35] to reduce I/O overhead.

```
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 }
```

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

```
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 }
```

Figure 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.

inputs during 24 hours of fuzzing. The experimental data is from our later evaluation in Section 4.3.

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*. Blindly sanitizing all of them is thus a huge waste of resources.

2.3. Illustrative Examples

In this section, we present examples of real-world bugs to demonstrate that inputs that trigger bugs follow distinct execution paths not seen with normal inputs, further reinforcing the rationale behind our approach.

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

```
1  char *
2  _nc_tic_expand(const char *src, bool tic_format)
3  {
4      static char *buffer;
5      ...
6      buffer = typeRealloc(char, length, buffer);
7      int bufp = 0;
8      if (ch == '%' && REALPRINT(str + 1))
9         buffer[bufp++] = *str++;
10     else if (ch == 128)
11        buffer[bufp++] = '\\';
12     } else if (ch == '\033')
13        buffer[bufp++] = '\\';
14     else
15         bufp += 4;
16     str++;
17     ...
18     return buffer;
19 }
20
21 int fmt_entry(...) {
22     ...
23     char *tmp = _nc_tic_expand(boxchars, 1);
24     while (*tmp != '\0') {
25         ...
26     }
27 }
```

Figure 4: A simplified Use-of-Uninitialized-Memory bug from infotocap in CVE-2019-17595. The buffer `*tmp` in line 24 is uninitialized when the execution jumps from line 7 to line 15.

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 ! 4 ! 7 ! 8 ! 9 ! 8 ! 9 ! ! 4 ! 7 ! 8 ! 9 ! }. In practice, we observe that data-sensitive bugs like Buffer-Overflow often accompany control-flow changes that normal executions do not exercise. Thus, using unique execution path 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. Although integer overflow is a locally data-sensitive bug type, it can be identified through unusual control-flow visits. In this example, the overflowed value in `numSyms` subsequently causes a small allocated buffer in line 14, which leads to buffer overflow and dramatic control-flow changes in the rest of the execution.

Use-of-Uninitialized-Memory. Figure 4 shows a Use-of-Uninitialized-Memory bug in line 24, where the `while` loop is conditioned on an uninitialized memory pointed to by `tmp`. The buffer `tmp` is obtained via the function call to `_nc_tic_expand()`, which returns an allocated buffer. Normally, `buffer` will be initialized either in line 9 or in lines 11 and 13. The corresponding (partial) execution paths are {7 ! 9 ! 18 ! 24}, {7 ! 11 ! 18 ! 24}, and

{7 ! 13 ! 18 ! 24}. Buggy executions, however, jump from line 7 to line 15 without initializing buffer . The corresponding buggy execution path is {7 ! 15 ! 18 ! 24}, which is different from all normal executions.

We have illustrated that both control- ow-related and data-sensitive bugs can result from/in unusual execution paths. This observation motivates us to utilize the execution path for determining whether the expensive sanitizer checks should be invoked.

3. Our Approach

This section introduces the design of our new fuzzing framework SAND. Section 3.1 de nes execution path and its proxy approximation, execution pattern. Section 3.2 describes our proposed fuzzing framework. Section 3.3 clari es the technical details of the implementation.

3.1. Preliminary: Execution Path and its Proxy

Our illustrative bugs exemplify that bug-triggering inputs have unique execution paths. We formally de ne the execution path as follows:

De nition 3.1 (Execution Path) Given an execution E, the execution path of E is de ned as $_E = [e_1; e_2; \ ; e_n]$, where e_i is the unique id of the code edge executed by E. Note that, $_E$ 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- ow visits. For instance, the buggy execution in Figure 4 {7 ! 15 ! 18 ! 24} has the execution path as [7; 15; 18; 24]. Execution path is order-sensitive meaning that [7; 15; 18; 24] 6= [15; 7; 18; 24]. Unfortunately, obtaining the execution path of execution is too expensive to be practical in fuzzing [33], [10]. Since throughput is a key factor in fuzzing effectiveness, we cannot directly use execution path in fuzzer design. In this paper, we propose to use execution pattern as an approximate yet accurate proxy for the execution path. We de ne execution pattern as follows:

De nition 3.2 (Execution Pattern) Given an execution E, the execution pattern of E is de ned as $T_E = \{e_1; e_2; \ ; e_m\}$, where $e_i \neq e_j$ $(i \neq j)$ and e_i is the unique id of the code edge reached by E. Note that, T_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 instance, the buggy execution in Figure 4 {7 ! 15 ! 18 ! 24} has the execution pattern as {7; 15; 18; 24}. Execution pattern is order-insensitive, e.g., {7; 15; 18; 24} = {15; 7; 18; 24}. Intuitively, the execution pattern inevitably loses precision in approximating the execution path. But it still captures the uniqueness of buggy executions. For the bug in Figure 4, normal execution patterns are {7; 9; 18; 24}, {7; 11; 18; 24}, and {7; 13; 18; 24}. The buggy execution pattern {7; 15; 18; 24}, although containing no ordering information, is distinct from normal ones. Our evaluation in Section 4.2 will use extensive experiments to demonstrate that execution patterns can precisely encapsulate buggy executions.

Figure 5: Executions (left) are monitored through bitmaps (middle), which are used in AFL++ to update the coverage map (right). Our execution patterns (bottom right) can be derived from these bitmaps.

An essential bene t of the execution pattern is its ease of acquisition during fuzzing. Fuzzers like AFL++, by default, utilize an ef cient data structure, bitmap, to collect visited code edges of an execution. Figure 5 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. 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 bottom right in Figure 5. Although the execution pattern might lose precision in modeling the execution path, we view it as a trade-off for performance. Furthermore, our evaluation will highlight that execution pattern is accurate enough in encapsulating buggy executions.

3.2. Sanitization-decoupled Fuzzing

Based on the above formalization to execution patterns, we introduce our new fuzzing framework design. We rst describe the general work ow of a coverage-guided fuzzer (CGF). The gray area in Figure 6 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 as follows:

(1) Seed selection. Select one seed from the seed pool according to prede ned 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.

Figure 6: SAND fuzzing loop.

(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. 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 6 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 P_{fuzz}), on which the fuzzer performs fuzzing and (2) a set of sanitizer-enabled programs, e.g., ASan-enabled program (P_{ASan}) and MSan-enabled program (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 of the execution from its 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 executes this input on each sanitizer-enabled program (P_{ASan}, P_{MSan}, etc.) and reports any discovered crashes.

Example. Figure 7 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 input 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, we only need to sanitize one buggy execution from the same set of unique execution paths to identify the bug. In this example, the first time we sanitize the execution with pattern $\{6, 4, 2, 3, 5\}$, we can discover the bug. For all future executions with the same pattern, they are likely to trigger the same bug and do not need sanitization. Our evaluation in Section 4.3 will support this claim.

Figure 7: Out of all eight consecutive executions from top to bottom, three are identified as unique and require sanitization.

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
7
8
9
10
11                                  // 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
```

This newly introduced conditional sanitization does not alter the standard fuzzing logic. The execution pattern is obtained from the already-available bitmap collected on 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 §3.3).

Algorithm 1 sketches the implementation pseudo-code of our new fuzzing framework. In each fuzzing loop (line 1),

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;
```

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 P_{fuzz} on the input to collect its execution return ret and bitmap $bitmap$ (line 4). Then, the fuzzer extracts the execution pattern T_E from bitmap (line 6) and determines whether or not this execution pattern has been observed (line 7). If T_E is new, the fuzzer labels it as sanitization-required and executes each of the available sanitizer-enabled programs P_{san} on the input s' (lines 8-9). Meanwhile, T_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 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 standard 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 like ASan and MSan are mutually exclusive, 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. We will provide additional technical details in Section 3.3 to explain how we support multiple sanitizers.

Effective. Our evaluation will show that only a small fraction (averagely 2%) of inputs have unique execution patterns and require sanitization, thus significantly improving fuzzing throughput.

Practical. 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.

3.3. 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 and implementation allow us to efficiently get execution patterns during fuzzing. 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. The hash table `HashTable` is initialized to all zeros at the start of fuzzing. In our implementation, we use XXH32 hashing algorithm[1] because of its fast speed. The size of `HashTable` is set to 32-bit, which supports a maximum of $4,294M$ different checksums. Our evaluation in Section 4.6 demonstrates that the cost of hashing is negligible, and no instances of hash collision are observed.

Note that normal executions can also have unique execution patterns, such as the first and third executions in Figure 7. Our approach is efficient as long as the overall ratio of unique execution patterns during fuzzing is low. As our evaluation in Section 4.4 shows, the average ratio is less than 2%.

Program Instrumentation in SAND. The fuzz target 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 the program with different sanitizers. Because ASan and MSan are mutually exclusive, we combine them in SAND. By default, we use two sanitizer-enabled programs, i.e., ASan/UBSan-enabled program ($P_{ASan=UBSan}$) and MSan-enabled program (P_{MSan}). To reduce the burden of invoking $P_{ASan=UBSan}$ and P_{MSan}, we utilize the forkserver [36] mode to create one forkserver to communicate with all sanitizer-enabled programs efficiently during fuzzing.

4. Evaluation

We implemented SAND based on AFL++-4.05c [9], the latest version at the time of implementation. AFL++ is the state-of-the-art gray-box fuzzer and has been widely used as the baseline fuzzer in many previous study [20], [22], [8]. Our evaluation first evaluates the accuracy of execution pattern (§4.2) in encapsulating buggy executions and then extensively evaluates the end-to-end fuzzing performance of SAND in terms of bug-finding (§4.3), throughput (§4.4), and code coverage (§4.5).

4.1. Experimental Setup

Benchmark. We use real-world programs from the benchmarking test platform UNIFUZZ 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. Due to the compatibility issue of MSan, we failed to instrument 8 out of 20 programs with MSan. We thus exclude them from our evaluation.

TABLE 3: All 12 real-world programs from UNIFUZZ used in the evaluation.

Type	Program	Type	Program
Image	imginfo	Text	infotocap
	jhead		mujs
	tiffsplit		pdftotext
Audio	mp3gain	Binary	nm
	wav2swf		objdump
Video	mp42aac	Network	tcpdump

Our full evaluation is done on the remaining 12 programs. These programs Table 3 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. Since ASan and UBSan are compatible with each other, we combine them together when building binaries. For each program fuzzed in UNIFUZZ, we compile a normally built binary P_{fuzz} and two sanitizer-enabled binaries i.e., $P_{ASan=UBSan}$ and 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 built with LLVM-14, the latest stable version at the time of implementation.

To understand if UNIFUZZ 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 infotocap, mp42aac, nm, objdump, and pdftotext with 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 [17] standard we repeated all experiments 10 times and ran all fuzzing campaigns for 24 hours. We apply the Mann-Whitney U-test [23] to our results to understand their statistical significance.

4.2. Effectiveness of 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 on the 12 programs:

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

TABLE 4: 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.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
imginfo	0.22%	72.8%
infotocap	8.11%	98.3%
jhead	1.44%	91.6%
mp3gain	0.18%	98.2%
mp42aac	0.04%	100.0%
mujs	5.34%	99.0%

Programs	All	Bug
nm	0.35%	100.0%
objdump	1.30%	100.0%
pdftotext	4.82%	100.0%
tcpdump	2.09%	100.0%
tiffsplit	1.17%	99.2%
wav2swf	0.01%	100.0%
Average	2.09%	96.58%

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- and UBSan-enabled programs to test if it triggers a bug.

We ran the experiments for 24 hours and repeated them ten times. Because we need to run through sanitizer-enabled programs for every input in order to gather a sufficiently large number of inputs, we exclude MSan due to its deficient speed. With this set of experiments, we want to answer: Q1: out of all executions in Step (2), how many of them are marked as having unique execution patterns? Q2: 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 first question Q1 can tell us the ratio of unique execution patterns during fuzzing. A smaller ratio indicates that sanitization is required less frequently, resulting in higher speed. The second question Q2 can inform us how effective the unique execution pattern is in encapsulating bug-triggering inputs. Table 4 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 fuzzer-generated inputs have unique execution patterns.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 can successfully sanitize 96% of the buggy inputs. 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, 96% accuracy can already ensure, with a high probability, that no bugs will be missed in practice. Our upcoming evaluation will provide extensive end-to-end evaluation results to confirm this high precision.

Figure 8: The number of unique bugs detected by fuzzers across repetitions. indicates a compilation failure.

TABLE 5: The mean number of unique bugs across repetitions. The indicates a compilation failure. The largest mean numbers are highlighted in green.

Programs	AFL++- Native	AFL++- ASan/ UBSan	AFL++- Debloat/ UBSan	AFL++- MSan	SAND	p-val
imginfo	0	7.9	7.7	0.4	8.3	0.18
infotocap	0.3	5.1	7	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	7	0	3	0.00
mujs	0	7.9	8.1	2.7	7.8	0.75
nm	0	1.3	7	0	3.5	0.00
objdump	1	4	7	1.2	3.3	1.00
pdftotext	3.1	2.5	7	1	3.7	0.09
tcpdump	0	6.2	6.6	3.5	12.3	0.00
tiffsplit	4.3	7.3	7	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	

TABLE 6: Accumulative number of unique bugs. indicates a compilation failure. "All" refers to the number of unique bugs found together by all other fuzzers except for SAND.

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

4.3. Bug-Finding Capability

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

Q1 Does SAND nd 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 nds. Our deduplication is done with both the stack frame information from GDB [19] and manual analysis.

Number of Unique Bugs. We plot the number of unique bugs in every repetition in Figure 8. Table 5 aggregates the results and reports the mean number of unique bugs. On average, SAND nds 32% more (7.8) bugs than the second-best fuzzer (5.9 in AFL++-ASan/UBSan). On 7 out of 12 programs, SAND found more bugs than all other fuzzers with statistical signi cance (i.e., p-value < 0.05). On some programs, SAND can cover signi cantly more (> 50%) bugs. For instance, on tcpdump, SAND can averagely nd 12.3 bugs, while the next-best fuzzer AFL++-Debloat/UBSan only nds 6.6 bugs. For the remaining ve programs, SAND nds statistically the same (p-value > 0.05) number of unique bugs. In summary, our result answers Q1: SAND has a signi cantly stronger bug- nding 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 nds fewer bugs on four 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 all bugs in pdftotext can be triggered without sanitizers, while AFL++-Native has higher throughput than all other fuzzers. Nevertheless, the overall result con rms the necessity of using sanitizers during fuzzing.

Figure 9: Unique bugs found by fuzzers.

Figure 10: The distribution of diverse types of bugs found by SAND.

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. Figure 9 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 finds 34 additional bugs that all other fuzzers can not cover. Table 6 breaks down the total number of bugs discovered by each fuzzer. Compared to other fuzzers, SAND finds more bugs on all programs. The second-to-last column "All" lists the total number of bugs found by all other fuzzers together. Even compared to "All", SAND can still find more bugs in each program. On some programs, SAND can even cover 2x – 3x more bugs. For instance, SAND discovers 2x more bugs on mp42aac and 3.5x more bugs on nm. In summary, we can answer Q1 and Q2: SAND does not miss any bugs and can find significantly more bugs.

Number of Unique Bugs Reported by Sanitizer-enabled Programs. Of all the 154 unique bugs identified by SAND, more than 75% of them are not detectable on the normally built programs. These bugs are reported after invoking sanitizer-enabled programs in SAND.

Bug Types. Our approach relies on the control-flow information to identify sanitizer-required inputs. Our observation, as illustrated with bug examples in Section 2.3, is that data-sensitive bugs like buffer overflow and integer overflow usually result from/in control-flow changes. Figure 10 reports the types of bugs found by SAND. Firstly, SAND can indeed find many control-flow-related bugs, such as the 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 [19]. Secondly, SAND can find a large number of data-sensitive bugs, such as heap buffer overflow (32 bugs) and integer overflow (41 bugs). This outstanding result confirms our assumption that data-sensitive bugs can be captured by control-flow information. Given that SAND does not miss any bugs and finds both control-flow and data-sensitive bugs, we can conclude that our approach generally applies to all kinds of sanitizer-detectable bugs.

False Negatives. Despite the outstanding performance of SAND, we have no 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 5, where SAND has a slightly lower mean number of bugs (7.8) than AFL++-ASan/UBSan (7.9) and AFL++-Debloat/UBSan (8.1). The reason is that one bug in mujs was not triggered in all repetitions of SAND, but in all repetitions in the other two fuzzers. The execution pattern for this bug can sometimes be seen in normal executions, which leads to the 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 of SAND, we believe that the moderate false negative issue is acceptable and does not impede its general effectiveness.

4.4. Fuzzing Throughput

We now analyze the end-to-end fuzzing throughput, i.e., the total number of inputs generated and executed during fuzzing. Figure 11 shows the average throughput of each fuzzer normalized to AFL++-Native.

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, respectively. 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 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. 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.

Figure 11: Relative throughput normalized to AFL++-Native. "/" indicates a compilation failure. "Average" refers to the average throughput of all programs.

Unique Execution Pattern Ratio. Intuitively, a smaller unique execution pattern ratio means that sanitizer-enabled programs are less frequently invoked and, consequently, have a higher throughput. To understand how this ratio changes during fuzzing, we track the unique execution pattern ratio every 40 seconds. Figure 12 plots the results. As expected, at the start of fuzzing, the ratio 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 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 11, SAND has indeed relatively lower speedup rates on these two programs.

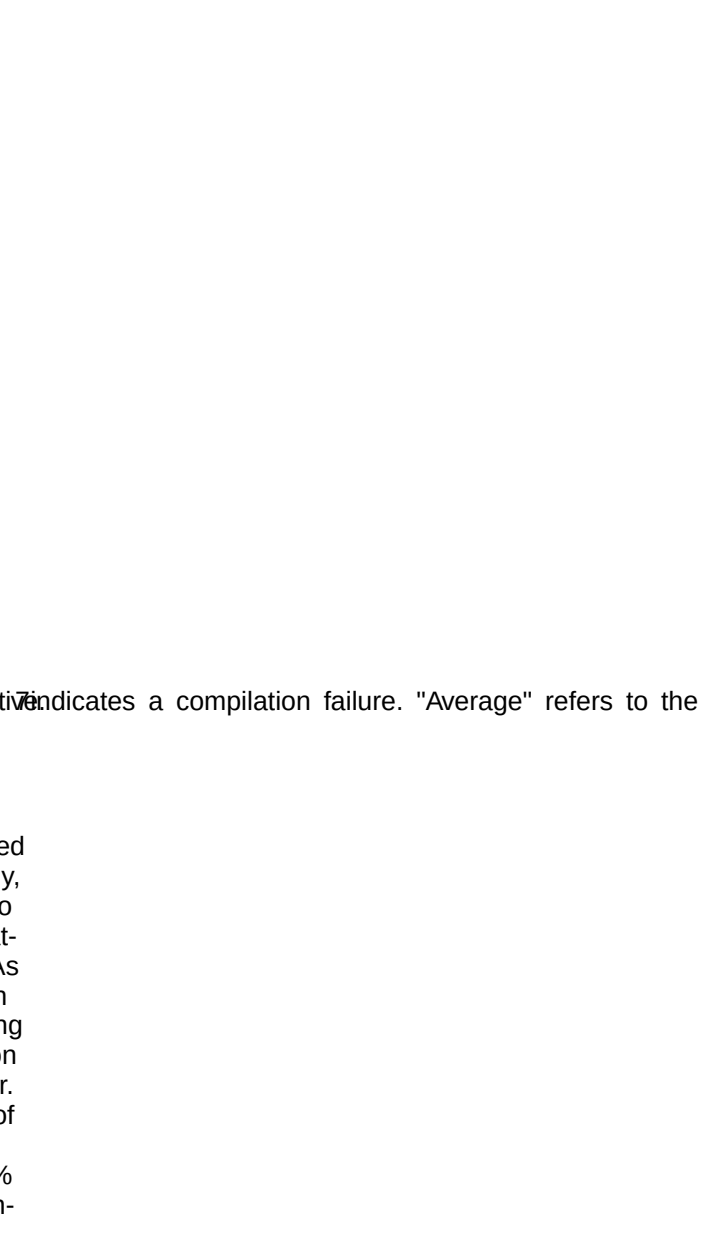

Figure 12: The trend of R_{unqiue}, i.e., the unique execution pattern ratio, over time during fuzzing.

4.5. 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 other three 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 remaining six 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 practice.

4.6. Hash in SAND

As the Algorithm 1 indicates, SAND utilizes a hash table to store hash checksums for each execution pattern. In this section, we evaluate the hash overhead of SAND and the potential hash collision risk in the hash table.

Hash Overhead. We modified SAND to two versions to precisely evaluate the hash overhead. First, SAND-NHASH, where lines 6-11 in Algorithm 1 are removed so that no hash operations are performed. Second, SAND-HASH, where lines

TABLE 7: Code coverage (%) of fuzzers. indicates a compilation failure. "Diff" is the difference compared to AFL++-Native. The highest code coverage compared to AFL++-Native is highlighted green.

Programs	AFL++ -Native	AFL++- ASan/UBSan		AFL++- Debloat/UBSan		AFL++- MSan		SAND	
		$Cov_{p\text{-}val}$	Diff	$Cov_{p\text{-}val}$	Diff	$Cov_{p\text{-}val}$	Diff	$Cov_{p\text{-}val}$	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	7	7	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	7	7	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	7	7	6.69 $_{0.00}$	-1.11	7.53 $_{0.01}$	-0.27
objdump	6.98	5.10 $_{0.00}$	-1.88	7	7	6.60 $_{0.00}$	-0.38	6.67 $_{0.00}$	-0.31
pdftotext	16.69	13.07 $_{0.00}$	-3.62	7	7	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	7	7	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

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 seed for both modified fuzzers to generate an identical set of inputs on the same initial seed pool. We run both fuzzers on each program ten times and record the total fuzzing time on the first one million 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 mujs and 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.

TABLE 8: The hash overhead and collision in SAND.

Programs	NOHASH	SAND-HASH		
	Speed	$Speed_{p\text{-}val}$	Overhead	Collision
imginfo	3,114	3,100 $_{0.65}$	0.47%	0
infotocap	3,011	2,982 $_{0.15}$	0.96%	0
jhead	3,327	3,327 $_{0.94}$	0.00%	0
mp3gain	1,929	1,915 $_{0.43}$	0.73%	0
mp42aac	1,702	1,688 $_{0.21}$	0.87%	0
mujs	1,841	1,812 $_{0.01}$	0.58%	0
nm	2,421	2,389 $_{0.26}$	0.36%	0
objdump	1,279	1,266 $_{0.36}$	0.05%	0
pdftotext	408	405 $_{0.00}$	0.66%	0
tcpdump	2,018	2,004 $_{0.52}$	0.73%	0
tiffsplit	2,335	2,334 $_{0.94}$	0.02%	0
wav2swf	2,830	2,817 $_{0.36}$	0.48%	0
Average	2,185	2,170	0.74%	0

Hash Collision. 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 accurate identification of unique execution patterns, hash collisions may potentially harm the performance. To evaluate the hash collision rate of SAND, we use the SAND-HASH and expand it to 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 one million inputs. The result shows that none hash collision was detected. The main reason is that unique execution patterns are rare (averagely 2% according to Section 2.2), making the hash table sparse and hard to have collisions.

5. 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 [3], [21], effective seed scheduling schemes [7], [5], and hybrid fuzzing techniques [14], [30] can all be formally integrated into a CGF fuzzer, which SAND can further build upon. At a high level, in the sequence of all mutated inputs during fuzzing, SAND's effectiveness depends on the fact that bug-triggering inputs are mostly likely to have unique execution patterns. Therefore, different mutation strategies may affect SAND's performance. To understand SAND's general applicability, we port it to an alternative fuzzer MOpt [21], which uses a different mutation scheduling strategy and can be manually turned on in AFL++. We include the details in the appendix.

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 [4], a semantic correctness test oracle is usually provided. Differential test oracles are applied to find incorrect outputs, such as wrong implementations [27] and platform-dependent differences [38]. 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. 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 AFL++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% to 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 AFL++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 2.2 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 AFL++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.

6. Related work

Reducing Sanitizer Overhead. Some research efforts exist to reduce sanitizer overhead. ASAP [31] removes sanitizer checks to meet a required performance budget. FuZZan [15] dynamically selects the most optimal metadata structure for both ASan and MSan to reduce sanitization overhead in fuzzing. SanRazor [37] and Debloat [39] 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. AFL++SAND, on the other hand, uses sanitizers without any modification. This feature further brings the orthogonality of AFL++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 [16] observes that all bug-triggering inputs have some unusual execution behavior. For specific bug types, UAFL [32] 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 [26] leverages the knowledge from sanitizer instrumentations to discover certain types of bugs faster. PGE [18] 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 [36], the fuzzing community has seen a broad range of new fuzzer developments. In particular, coverage-guided grey-box fuzzers such as AFL++ [9], AFLFast [7], and AFLGo [6] 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 [5], mutation strategies [21], [3], and path explorations [30], [14]. In theory, all these improvements are not related to sanitizer-enabled programs and, therefore, are orthogonal to us. In practice, AFL++ has internally integrated many advances such as RedQueen [3] and MOpt [21]. We have demonstrated that our current implementation of AFL++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 [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 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, AFL++SAND, to decouple sanitization from the fuzzing loop. AFL++SAND performs fuzzing on the normally built program and only executes sanitizer-enabled programs when input is identified as sanitization-required. AFL++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 effective execution analysis via the 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] xxhash: Extremely fast hash algorithm. https://github.com/Cyan4973/xxHash, 2016. Accessed: December 7, 2023.

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

[3] 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.

[4] 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*, CCS'22, 2022.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

[12] 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.

[13] Adrian Herrera, Mathias Payer, and Antony L. Hosking. Dataflow: Toward a data-flow-guided fuzzer. *ACM Trans. Softw. Eng. Methodol.*, 32(5), jul 2023.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

[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] 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*, S&P'17, 2017.

[28] 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.

[29] 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.

[30] 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.

[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.