icLibFuzzer: Isolated-context libFuzzer for Improving Fuzzer Comparability

Yu-Chuan Liang
National Taiwan University
jasonliang30115@gmail.com

Hsu-Chun Hsiao
National Taiwan University
hchsiao@csie.ntu.edu.tw

***Abstract*—libFuzzer is a powerful fuzzer that has helped find thousands of bugs in real-world programs. However, fuzzers that seek to compare with libFuzzer and its variants face two significant limitations. First, they are restricted to use the time-to-first-crash metric rather than the code-coverage metric because libFuzzer will abort whenever the fuzzing target crashes. Second, even if libFuzzer in the *ignore-crash mode* can continue after finding a crash, it may produce wrong results for programs expecting a clean global context. Thus, fuzzers wishing to compare with libFuzzer are restricted to use carefully modified programs or programs without global-context dependency. To solve this context pollution problem and enhance comparability between libFuzzer and other fuzzers, we present a new libFuzzer mode called *isolated-context mode* (icLibFuzzer) that isolates the contexts of each fuzzer instance and fuzzing target, allowing to reinitialize the fuzzing target's context after each execution efficiently. To implement icLibFuzzer, we modify libFuzzer's in-process infrastructure into a lightweight forkserver infrastructure inspired by AFL's design and propose *structure packing*, which speeds up the fuzzing speed by about 2x. We compare icLibFuzzer with four state-of-the-art fuzzers (AFL, Angora, QSYM, and Honggfuzz) using several real-world programs. The experiment result shows that icLibFuzzer outperforms these four fuzzers in most target programs after 24 hours of fuzzing and maintains the lead from 24 to 72 hours. To demonstrate that we can easily keep up with libFuzzer's updates, we upgrade icLibFuzzer to using the latest libFuzzer (from LLVM9 to LLVM11) with no change to our code base. Our preliminary evaluation hints at icLibFuzzer-LLVM11's promising improvement compared with icLibFuzzer-LLVM9 and AFL++, one of the latest fuzzers in the AFL family. We hope icLibFuzzer can serve as another baseline for fuzzing research. Our source code is available at GitHub.**

I. Introduction

Fuzzing is a popular technique for automatically finding bugs. To fuzz a computer program, which can be a standalone executable or a library function, a fuzzing tool (called a fuzzer) feeds random inputs into the program and reports inputs that cause unexpected behaviors such as hangs or crashes. These interesting inputs can further help uncover bugs in the program.

Workshop on Binary Analysis Research (BAR) 2021
21 February 2021
ISBN 1-891562-69-X
https://dx.doi.org/10.14722/bar.2021.23xxx
www.ndss-symposium.org

AFL [20] and libFuzzer [14] are two of the most successful fuzzers with different design goals and have been evolving separately for several years. AFL aims to continuously fuzz executables, while libFuzzer targets library functions and stops whenever hitting a crash. As a result, fuzzers in the AFL and libFuzzer families mainly consider the code-coverage and time-to-first-crash metrics, respectively. It is challenging to compare them directly with each other over the same metrics.

This comparability issue is solved partially by enabling the *ignore-crash mode* in libFuzzer. The ignore-crash mode spawns multiple fuzzer instances and fuzzes a target executable in different processes parallelly. Whenever a fuzzer instance exits or finds a crash, libFuzzer will restart the fuzzer instance. Although libFuzzer in the ignore-crash mode can fuzz continuously and restart after a crash, the fuzzing results may be incorrect when previous non-crash runs pollute the global context because of libFuzzer's in-process infrastructure. Therefore, most fuzzers were compared with libFuzzer only on few manually tuned datasets (e.g., fuzzer-test-suite [9]) that allow both fuzzer families to operate correctly.

Due to the lack of comparability, the advances in libFuzzer seem to be left unnoticed in the state-of-the-art academic papers, which predominantly consider the code-coverage metric. For example, libFuzzer has integrated advanced instrumentation features such as CMP tracing since 2017, but similar features were unavailable in AFL-based fuzzers until 2019 in RedQueen [2] (which is later integrated into AFL++ [7]). To this end, we believe a *bridge* between these two fuzzer families is needed to foster knowledge accumulation and sharing in the community.

In this work, we present *isolated-context libFuzzer (icLibFuzzer)*, a new libFuzzer mode that bridges the two fuzzer families and allows them to be compared over commonly seen metrics on arbitrary programs. To solve the context pollution problem, icLibFuzzer runs in the AFL-like forkserver architecture and correctly manages the global context between executions. As speed is a crucial requirement of fuzzers, we further optimize icLibFuzzer's implementation via *structure packing*, achieving similar performance to other forkserver-based implementation while preserving libFuzzer's advanced instrumentation features.

To demonstrate that icLibFuzzer improves fuzzer comparability, we compare icLibFuzzer with four state-of-the-art fuzzers (AFL, Angora, Honggfuzz, and QSYM) using the code-coverage metric on several real-world programs. icLibFuzzer outperforms all of the tested fuzzers in most target programs after 24 hours of fuzzing. Moreover, icLibFuzzer maintains the lead and continues to increase the code coverage from 24 to 72 hours[1].

To incentivize adoption, we provide a compiler wrapper to compile standalone programs without modifying the source code. As libFuzzer is under active development and new features may appear in the future, we also provide a guideline for adding its new features into icLibFuzzer. To demonstrate that we can easily keep up with libFuzzer's updates, we upgrade icLibFuzzer to using the latest libFuzzer (i.e., upgrading from LLVM9 to LLVM11) with no change to our code base. Our preliminary evaluation hints at icLibFuzzer-LLVM11's promising improvement compared with icLibFuzzer-LLVM9 and AFL++, one of the latest fuzzers in the AFL family that integrates multiple enhanced techniques.

This paper makes the following contributions:

1) We design and implement icLibFuzzer, a new libFuzzer mode that isolates the contexts of each fuzzing target and fuzzer instance, enabling continuous fuzzing on arbitrary programs while inheriting libFuzzer's advanced instrumentation features and analysis techniques.
2) We propose the structure packing technique to improve the running speed of icLibFuzzer by roughly 2x.
3) We evaluate icLibFuzzer using several real-world programs and find that icLibFuzzer outperforms all of the tested fuzzers in most target programs, implying that icLibFuzzer is a competitive fuzzing tool and may serve as another evaluation baseline for fuzzing research.
4) We release our source code at https://github.com/csienslab/icLibFuzzer.

II. Background

libFuzzer supports several advanced instrumentation features that have not been fully adopted by other fuzzers, but it is challenging to compare libFuzzer with others due to its in-process design. We briefly introduce libFuzzer in this section and explain the comparability issues in the next section.

A. libFuzzer Overview

Similar to AFL, libFuzzer is a *coverage-guided* fuzzer, which collects coverage information through *instrumentation*, selects interesting inputs (called *seeds*) based on the collected information, and mutates seeds to generate new inputs.

For example, to collect program branches covered by inputs, code-coverage instrumentation inserts instructions to the target binary to record the current program counter right after each conditional branch, exemplified by the following C code:

```
    if (a > 0) {
       bitmap[current_program_ctr] += 1;
       if (b > 0) {
          bitmap[current_program_ctr] += 1;
          do_something();
       }
    }
```

[1]72 hours is the longest time budget that we can afford in our experiments.

After the program terminates, the `bitmap` array records the branches reached by the input. Thus, given a set of inputs, we can measure a fuzzer's branch coverage (a type of code-coverage metrics) by accumulating the information recorded in the bitmap.

While AFL mainly collects code-coverage information and keeps coverage-increasing inputs as seeds, libFuzzer collects additional information such as the branch condition via instrumentation and keeps inputs that make a variety kinds of "progress" (e.g., closer to pass a conditional branch), as we will see in the next subsection.

B. libFuzzer's Advanced Instrumentation Features

libFuzzer supports several advanced instrumentation features to assist fuzzing. Some of these features are unavailable in other fuzzers or appear years later. For example, libFuzzer implemented CMP tracing in 2017, but similar ideas were unavailable in other fuzzers until 2019 in RedQueen [2].

Below we choose and describe four advanced instrumentation features in libFuzzer which may impact code coverage significantly, and have also been ported to our icLibFuzzer: CMP tracing, value profiling, compare function, and dataflow trace. We also provide a guideline in Section VII to port new libFuzzer features to icLibFuzzer.

1) CMP tracing: Due to their random nature, fuzzers often struggle to discover inputs satisfying complex branch conditions, such as `x==65536` or `pwd=="secr3t"`. Instead of random guessing, libFuzzer supports a salient feature called CMP tracing, which hooks all of the `CMP` instructions (in the LLVM IR instruction set) at compile time and stores the two operands, indexed by their xored value. Take the following C code snippet as an example:

```
1       short int magic;
2       read(0, &magic, sizeof(short int));
3       if (magic == 0x1234)
4          bug();
```

The `if` branch in Line 3 will be compiled into a `CMP` instruction, so libFuzzer inserts instrumentation code to save the compared targets:

```
1       short int magic;
2       read(0, &magic, sizeof(short int));
3       save_compared_arguments();
4       if (magic == 0x1234)
5          bug();
```

When executing Line 3, the instrumentation code will store `magic` and `0x1234`, both indexed by `magic` $\oplus$ `0x1234`. These values will then be used when mutating seeds into new inputs, such as replacing `magic` in input into `0x1234`.

2) Value profiling: When value profiling is enabled, libFuzzer collects additional information to quantify the progress made toward passing each branch condition at runtime. Take the following code as an example:

```
initialize();
while TRUE do
    input = mutate();
    LLVMFuzzerTestOneInput(input, inputSize);  ▷ User should implement this function.
end
```

Fig. 1: libFuzzer's in-process fuzzing

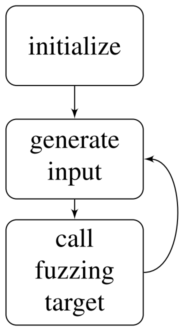


Fig. 2: libFuzzer's in-process fuzzing workflow

```
1       short int magic;
2       read(0, &magic, sizeof(short int));
3       if (magic == 0x1234)
4           bug();
```

In this example, though all `magic` values except `0x1234` will fail the branch condition at Line 3, some values are closer to passing the condition than others. Finding an input closer to passing the condition should be considered progress and saved as a new seed. For example, assuming that distance between two values is defined as the number of different bits, then if we mutate `magic` from `0x1237` to `0x1235`, `0x1235` should be saved as a seed because `0x1235` differs from the target (`0x1234`) by one bit, while `0x1237` has a two-bit difference.

When enabling value profiling, libFuzzer uses the program counter as an index to store the distance information calculated based on the xored value of the compared operands. For example, the distance can be defined as the number of zero bits in the xored value. If the distance indexed by the same program counter reduces, which means that we are closer to solving the branch, the input will become a new seed.

3) Compare function: Besides CMP tracing for the `CMP` instruction, libFuzzer also hooks functions related to string comparison (e.g., `strstr`, `memcmp`, and `strncasecmp`), and stores the function parameters for mutation later.

4) Dataflow trace: Hooking the compare instruction and string comparison functions allows libFuzzer to retrieve the compared values, but does not reveal the relationship between the input bits and the bits of the compared value. Dataflow trace taints an input to uncover the mapping between the input and compared value by checking the taint label.

C. libFuzzer's in-process infrastructure

libFuzzer adopts an *in-process* infrastructure for achieving high fuzzing speed (i.e., executions per second). In an in-process infrastructure, the fuzzer instance (libFuzzer) and the fuzzing target (LLVMFuzzerTestOneInput) work in the same context and share the same virtual memory space, as shown in Figure 1 and Figure 2. By contrast, AFL uses a *forkserver infrastructure*, in which each fuzzer instance and fuzzing target has an isolated context.

Despite the higher speed, the in-process infrastructure is more fragile and subject to the *context pollution problem* because the fuzzer instance may behave incorrectly due to bugs triggered in the fuzzing target, and the fuzzing target may behave incorrectly due to global context polluted in previous runs of fuzzing target.

The original libFuzzer can tolerate context pollution for two main reasons. First, it targets library functions (such as `png_sig_cmp` in `libpng`), and the fragility amplifies the impact of silent, non-crash bugs (e.g., memory leak), which are fatal in library functions.[2] Second, the impact of polluted context is limited because libFuzzer stops whenever encountering a crash, and the libFuzzer user is expected to fix the bug before finding the next one.

III. Comparability Issues

This section highlights two comparability issues when prior work compares libFuzzer with other fuzzers and explains why existing solutions are insufficient. These issues motivate our design and implementation of icLibFuzzer.

A. Lack support of common evaluation metrics

Fuzzers are often evaluated by how much code they covered in a fixed time, how many bugs they found in a fixed time, and how fast they found all intended bugs. However, it is hard to measure libFuzzer's progress using these metrics on arbitrary programs because libFuzzer terminates immediately whenever the fuzzing target crashes. Moreover, the crash may be triggered by a polluted global context rather than the program's bug. Thus, previous work is either limited to using the time-to-first-crash metric [21], compared on fine-tuned datasets [1], [9], or has to enable libFuzzer's ignore-crash mode, which restarts the fuzzer right after each crash [5].

Enabling the ignore-crash mode solves this comparability issue partially but also introduces additional problems. First, restarting a fuzzer is time-consuming, thus slowing down the fuzzing process. Worst yet, restarts may frequently happen because fuzzing often triggers the same crash repeatedly. Second, the context pollution problem is less tolerable when we are interested in metrics such as code coverage and the number of bugs found because the polluted context may result

[2]A memory leak bug may be tolerable in a standalone program with a short lifetime because the operating system will handle memory leaks after the program terminates. However, a memory leak is fatal in library functions since the memory leak will persist and exhaust memory eventually.

in an incorrect measurement of these metrics. Simply restarting the fuzzer after a crash is insufficient to avoid a broken context caused by non-crash bugs, which we will explain in the next subsection.

Some fuzzers [6], [21] avoid the context pollution problem by limiting their comparisons to a small set of manually crafted targets (e.g., Google's fuzzer-test-suite) or programs without global-context dependency, but again this limits the comparison scope.

B. *Incorrect results due to context pollution*

Even if libFuzzer in the ignore-crash mode can continue after finding a crash input to the fuzzing target, it may produce wrong results for targets expecting a clean global context. We identify two cases in which the ignore-crash mode may behave wrongly during fuzzing due to context pollution.

a) Programs depending on global variables: The first case is fuzzing from a *main* function that assumes correctly-initialized global variables before being called.

A C/C++ program can safely assume that global variables with no initial value will be set to zero. However, when fuzzing such a program, libFuzzer treats and calls the fuzzing target like a function, so the global variables are not reinitialized when libFuzzer fuzzes the target again. Thus, if the fuzzing target modifies these global variables during its execution, the fuzzing target may act incorrectly after running a few times.

For example, `getopt` is a widely used function parsing the command-line arguments. It relies on a global variable `optind`, which indicates the next element's index in the command-line arguments:

```
int getopt(int argc, char *const argv[],
   const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

If the fuzzing target calls `getopt` and libFuzzer fuzzes the fuzzing target twice without reinitializing the global variable `optind`, `getopt` may behave unpredictably.

LibKluzzer [12] proposes to solve this problem by memorizing and replacing global variables every time the fuzzing target reruns. However, shared contexts other than the global variables, e.g., heap, are still polluted and will not be reset. Also, since their tool is not publicly available, it is unclear how LibKluzzer handles every possible global variable initialization, including some that can only be correctly reinitialized by calling the corresponding functions. Take the following C code as an example:

```
1  class Foo {
2  public:
3     Foo() {x = new char[10];}
4     ~Foo() {delete[] x;}
5     char *x;
6  };
7
8  Foo bar;
9
10 int LLVMFuzzerTestOneInput(uint8_t *data,
11     size_t size) {
12     memcpy(bar.x, data, 10);
13     parse(bar.x);
14     delete bar;
15 }
```

`bar` is a global variable. When LibKluzzer resets `bar` according to the previous snapshot after the function `LLVMFuzzerTestOneInput` exits, `bar.x` will be reinitialized to a memory address that has already been freed at line 14. Moreover, LibKluzzer cannot handle programs with memory leak, explained below.

b) Programs with memory leak: Another form of context pollution occurs when fuzzing programs that have memory leak bugs. Since the fuzzing target runs in the same context as the fuzzer instance, if the fuzzing target contains a memory leak, the memory leak will persist and keep consuming the memory until libFuzzer crashes and restarts.

IV. ICLIBFUZZER

This section presents icLibFuzzer's design and implementation. icLibFuzzer resolves the two comparability issues while achieving speed comparable to AFL. The key idea is to isolate each run of the fuzzing target and the fuzzer instance and implement this context isolation efficiently by changing libFuzzer's in-process infrastructure into a forkserver infrastructure. Additionally, we propose structure packing to improve fuzzing speed, set CPU affinity to ensure comparability, and provide a compiler wrapper to compile a standalone program without implementing the `LLVMFuzzerTestOneInput` function.

A. *Efficient context isolation*

There are two possible approaches to isolate context between each run of fuzzing target and fuzzing instance.

1) **Naive forkserver**: A straightforward yet expensive approach to ensuring a clean context would be forking from the fuzzer instance and running the fuzzing target every time. Since the `fork` call will spawn another process, the global context between each run of fuzzing target and fuzzing instance will be separated. However, this approach will introduce unnecessary overhead because many complicated logic and data structures in the fuzzer instance are unnecessary for the fuzzing target, and the main memory size impacts the forking speed significantly according to our experiment in Section V-D.
2) **Lightweight forkserer**: Inspired by AFL's infrastructure, instead of forking directly from the fuzzer instance, we can first `fork` and `execve` to create a forkserver, and `fork` from the forkserver. Because this approach requires less writable memory, it is faster than the native forkserver approach. The speed comparison is shown in Table V in Section V-C

Thus, we take the lightweight forkserver approach and separate the fuzzer instance into two components—*main controller* and *forkserver*. The pseudocode in Figure 3 and Figure 4

```
fork();
if inChild then
    execv(forkserver);
end
while TRUE do
    input = mutate();
    tellForkServer(input);

    waitForForkServer();

end
```

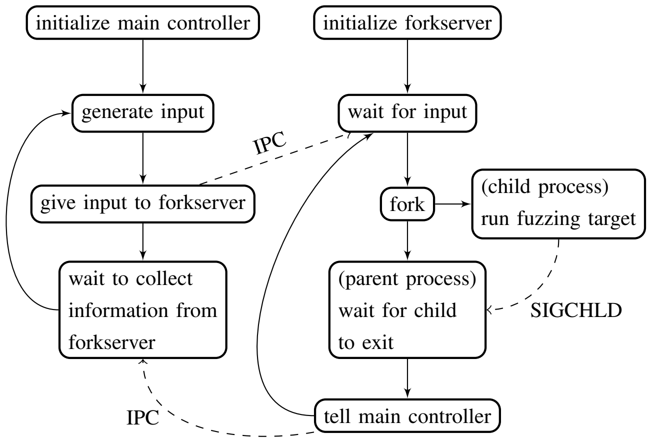
Fig. 3: icLibFuzzer: main controller

```
while TRUE do
    input =
      FromMainCon-
      troller();

    fork();
    if inChild then
        call(fuzzingTarget,
          input);
        exit(0);
    else
        waitForChild();
    end
    tellMainController();

end
```

Fig. 4: icLibFuzzer: forkserver



initialize main controller
generate input
give input to forkserver
wait to collect information from forkserver
initialize forkserver
wait for input
fork
(child process) run fuzzing target
(parent process) wait for child to exit
tell main controller
IPC
SIGCHLD
IPC

Fig. 5: icLibFuzzer's workflow.
The dashed lines represent the communication channel.

represent the *main controller* and *forkserver*, respectively, and Figure 5 illustrates how icLibFuzzer works.

The forkserver is spawned (`execve` after `fork`) from the main controller, and they communicate with each other through Inter-Process Communication (IPC). The main controller component is responsible for analyzing information collected from the instrumented program and generating inputs according to the collected information. For each input received from the main controller, the forkserver forks a child process to run the fuzzing target and collects instrumentation information. This design ensures that the fuzzing target always starts running from a clean context, free from the pollution of the previous runs, and is much faster than forking directly from fuzzing instance.

Moreover, we use shared memory as a fast IPC approach to pass instrumentation information back to the main controller from the forkserver. To achieve this, we move the data structure that stores instrumentation information into shared memory.

B. Structure Packing

Although this forkserver infrastructure is faster than forking directly from the *main controller*, it is still much slower than the original libFuzzer due to the resource-consuming fork procedure.

We observe that a primary factor affecting the forking speed is the writable memory size—the memory space that may be written to in the fuzzing target. Therefore, we propose *structure packing* to reduce the memory size and increase the forking speed.

The idea of structure packing is to pack all data collected from instrumentation into one compact structure to minimize the wasted space. For example, the following code snippet shows several data structures used in libFuzzer to store instrumentation data. The arrays `MemMemWords` and `Table` store the fuzzing target's information, collected through instrumentation at runtime.

```
template<size_t kMaxSize>
class FixedWord {
   public:
      ...
   private:
      uint8_t Size = 0;
      uint8_t Data[kMaxSize];
};
typedef FixedWord<64> Word;
struct Pair {
   Word A, B;
};
Word MemMemWords[1024];
struct Pair Table[32];
```

As these data structures store instrumentation information, they may be modified during the fuzzing target's execution. Instead of allocating space to them separately, icLibFuzzer packs all data into a compact structure in the shared memory to minimize the unused space:

```
Total = mmap(...);
MemMemWords = Total;
// ensure MemMemWords and Table
// don't overlap
Table = Total + Offset;
```

As shown in Section V-D, structure packing can double the running speed in several real-world binaries as we minimize the total memory pages that may be written to in the fuzzing target, allowing icLibFuzzer to fuzz at speed comparable to AFL.

This structure packing technique can be applied to speed up other forkserver-based fuzzers that use multiple data structures to pass information between the forkserver and main controller.

C. Set CPU Affinity

We set CPU affinity to each icLibFuzzer instance. Namely, we bind each fuzzer instance to a CPU/core. There are three reasons for doing this. First, binding a process to one CPU will improve cache locality and reduce cache miss for better performance. Second, since AFL binds its process to one CPU, most of its successors and family members do this too. Therefore, we eliminate a potential variation when comparing icLibFuzzer with fuzzers in the AFL family by following this convention. Third, binding a process to a CPU reduces interference between different fuzzing instances. Since one fuzzer instance can use at most one CPU, a fuzzer instance consisting of several processes (e.g., icLibFuzzer) will not interfere with other instances if we want to fuzz in parallel.

D. Compiler wrapper

icLibFuzzer comes with a compiler wrapper to automatically compile a standalone program for icLibFuzzer.

Recall that libFuzzer requires users to implement a `LLVMFuzzerTestOneInput` function as a callback in the fuzzer instance to identify the fuzzing target. This task can be laborious because users have to find out every input function in the fuzzing target and may accidentally introduce additional bugs. It is even more challenging when the fuzzing target reads inputs through non-standard methods or performs complex operations.

Take the following code snippet as an example:

```
read(0, buf1, size1);
read(0, buf2, size2);
```

One may implement the desired `LLVMFuzzerTestOneInput` function as follows.

```
int LLVMFuzzerTestOneInput(uint8_t*
  Data, size_t Size) {
  if (Size >= size1) {
     memcpy(buf1, Data, size1);
     if (Size >= size1 + size2)
        memcpy(buf2, Data + size1,
           size2);
     else
        memcpy(buf2, Data + size1,
           Size - size1);
  }
  else
     memcpy(buf1, Data, Size)
}
```

As the fuzzing target becomes more complex, the implementation of `LLVMFuzzerTestOneInput` may become harder and human errors may occur occasionally.

To simplify the procedure and avoid human errors, we provide a *clang* wrapper, *clang-fast*, which handles everything—just like *afl-clang-fast* in AFL. All a user needs to do is compile the target project using *clang-fast*; no need to modify any source code manually.

TABLE I: Fuzzers used in the evaluation and their versions

	version (or last-commit date)
AFL	2.52b (5 Nov 2017)
Angora	25 May 2019
Honggfuzz	22 May 2019
libFuzzer-LLVM9	LLVM 9 (22 April 2019)
libFuzzer-LLVM11	LLVM 11 (27 Sep 2020)
QSYM	15 October 2019
AFL++	11 Dec 2020
icLibFuzzer-LLVM9/11	follow libFuzzer-LLVM9/11

V. Evaluation

To demonstrate that icLibFuzzer improves fuzzer comparability, we compare icLibFuzzer with state-of-the-art fuzzers based on code coverage in real-world programs. The results will unveil how effective libFuzzer's advanced features are in comparison with other fuzzers. We also investigate icLibFuzzer's speed: how much is traded for comparability and how much we gain back by infrastructural and implementation optimizations.

Specifically, we ask the following three research questions:

RQ-1: Code-coverage comparison. Can icLibFuzzer achieve better code coverage compared with other state-of-the-art fuzzers?

RQ-2: Execution speed difference. What is icLibFuzzer's execution speed compared with AFL, libFuzzer, and a naive forkserver implementation?

RQ-3: The effectiveness of optimization. Does structure packing improve fuzzing speed, and by how much?

A. Settings

a) Fuzzers: Table I lists the fuzzers and their versions used in our experiments. We choose AFL and Honggfuzz because they are the core fuzzers (in addition to libFuzzer) used in OSS-Fuzz [16]. We also choose two strong fuzzers, QSYM and Angora, released in 2019. QSYM is a hybrid fuzzer that defeats AFL and previous hybrid fuzzers. It uncovers bugs in `ffmpeg` which has already been fuzzed by OSS-Fuzz for years. Angora uses multiple techniques, including taint tracking, to solve branch conditions without using symbolic execution. It shows notable improvement over AFL.

Our icLibFuzzer uses the same LLVM version as libFuzzer. Throughout the evaluation, we use LLVM9 by default (i.e., icLibFuzzer-LLVM9) unless specified otherwise. We include icLibFuzzer-LLVM11 (released in late 2020) mainly to demonstrate that one can effortlessly update icLibFuzzer whenever newer versions of LLVM and libFuzzer are available. Additionally, we include AFL++, whose latest version was released in December 2020. AFL++ assembles multiple optimization techniques and integrates them into the latest AFL. It shows a significant performance boost compared with the original AFL after enabling multiple optimizations simultaneously. However, due to the limited time before the submission, we only compare icLibFuzzer-LLVM9, icLibFuzzer-LLVM11, and AFL++ with each other, not with the other fuzzers.

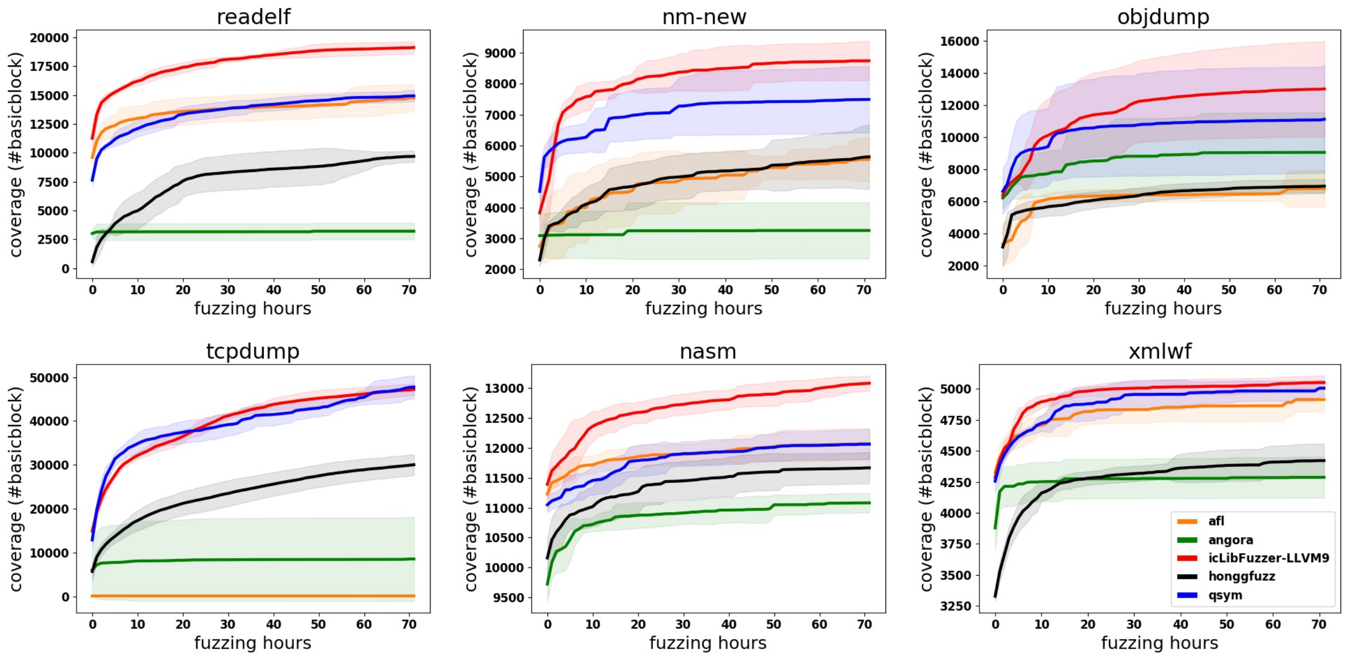


Fig. 6: Comparison between fuzzers. All the legend color mappings are the same as labeled in xmlwf.

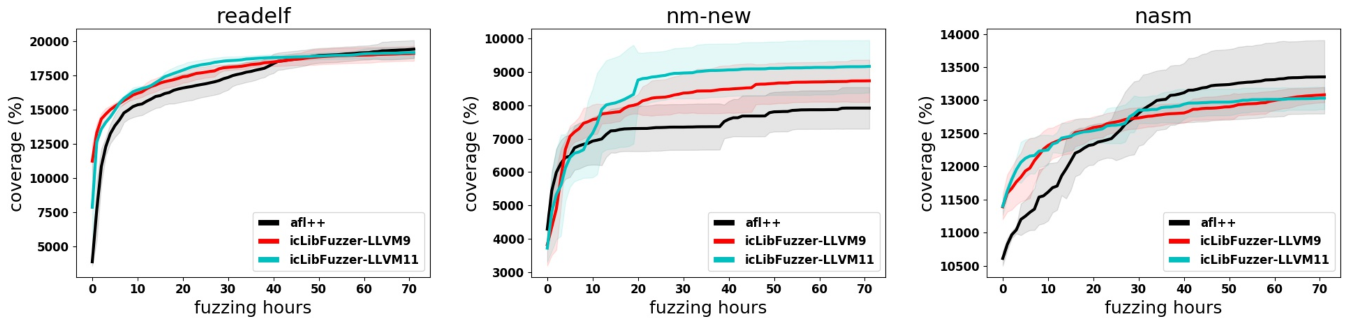


Fig. 7: Comparison among latest icLibFuzzer-LLVM11, state-of-the-art AFL++ and icLibFuzzer-LLVM9.

b) Datasets: We use two datasets: one contains real-world programs (objdump-2.35, readelf-2.35, nm-new-2.35, tcpdump-4.9.2, xmlwf-2.2.9, nasm-2.16rc0) and the other is the fuzzer-test-suite. The real-world program dataset is for coverage and speed comparison with icLibFuzzer, while the fuzzer-test-suite is only used for speed comparison with libFuzzer.

c) Time budget: We run every binary eight times and each for 72 hours, which is the longest we can do given our resource budget. We report the average code coverage and standard deviation. Because fuzzing in the real world often lasts long (e.g., more than one month), investigating the trend after 24 hours of fuzzing can hint at the fuzzer's ability to find new code coverage continuously.

d) Fuzzer configuration: To ensure fair comparisons, we assign no initial seeds to fuzzers because initial seeds may have a significant impact on fuzzing [11]. We discuss the effect of initial seeds in discussion section. We run all fuzzers with two threads to ensure a fair comparison with QSYM (which needs at least two threads). We run our experiment on a machine with one AMD Ryzen Threadripper 2990WX 32-Core Processor and 64GB memory, running Ubuntu 18.04 LTS.

In our evaluation, we enable the first three advanced instrumentation features introduced in Section II but not dataflow trace because dataflow trace requires manual function labeling and may provide an unfair advantage to icLibFuzzer in our comparative evaluation.

B. RQ-1

To answer this question, we evaluate icLibFuzzer's code coverage on the real-world program dataset and compare the results with four state-of-the-art fuzzers (AFL [20], Angora [4], QSYM [19], and Honggfuzz [17]). Note that AFL, Honggfuzz, and libFuzzer have only been compared in the in-process mode (also called the *persistent mode* in AFL and

TABLE II: Comparing each fuzzer's average speed (executions per second) on real-world programs after running 72 hours.

	readelf	nm-new	objdump	tcpdump	nasm	xmlwf
icLibFuzzer	1701	1667	1030	1603	359	2349
AFL	1708	1906	1032	2438	337	2983
qsym	1380	2990	1174	3087	430	4917
angora	1300	681	684	1068	419	1157
honggfuzz	86	96	116	108	76	126

TABLE III: Effect of structure packing. We run for 30 minutes in each cases to obtain a stable execution speed. `RSS` (Resident Set Size) is the RAM size consumed by the process, and `exec/s` stands for executions of fuzzing target per second.

	exec/s		RSS (Bytes)	
	no pack	pack	no pack	pack
objdump	1992	3740	7260K	3636K
readelf	2404	4426	7672K	3356K
nm-new	2083	3991	7544K	3324K
tcpdump	2168	4103	7504K	3492K

TABLE IV: Comparing each fuzzer's speed (executions per second) on the fuzzer-test-suite dataset. Version of libFuzzer and icLibFuzzer here is LLVM9. We run for 30 minutes in each case to get to a relatively stable execution speed.

	libFuzzer	icLibFuzzer	AFL	naive forkserver
boringssl-2016-02-12	26707	1722	1201	136
freetype2-2017	9210	927	1038	128
libpng-1.2.56	8707	2556	3340	69
openssl-1.1.0c	22460	512	560	164
sqlite-2016-11-14	20587	1112	881	63

TABLE V: Comparing naive forkserver with lightweight forkserver. We run for 30 minutes in each case to get to a relatively stable execution speed. `RSS` (Resident Set Size) is the RAM size consumed by the process, and `exec/s` stands for executions of fuzzing target per second.

Benchmark	light-weight forkserver		naive forkserver	
	exec/s	RSS (bytes)	exec/s	RSS (bytes)
boringssl-2016-02-12	1722	9M	136	335M
freetype2-2017	927	9M	128	388M
libpng-1.2.56	2556	6M	69	353M
openssl-1.1.0c	512	11M	164	276M
sqlite-2016-11-14	1112	6M	63	340M

Honggfuzz) in previous work. With icLibFuzzer, these three fuzzers can compare with each other on arbitrary programs.

Code coverage As shown in Figure 6, icLibFuzzer outperforms other fuzzers in most binaries.

We observe two interesting phenomena from the results.

1) **Bad performance of Angora**: Angora shows better performance than AFL in its original paper. However, we can see that Angora does not perform well in our experiment. One plausible reason may be that Angora highly relies on initial seeds, at least for these six programs we tested. Other fuzzers in our experiment seem to work reasonably despite an empty initial seed. It needs further investigation on selecting seeds wisely for Angora.
2) **Code coverage after 24 hours**: As suggested in [11], we should run fuzzers for 24 hours for a fair comparison. We can also see many papers [5], [7], [8] running experiments less than or equal to 24 hours. However, according to our experiment results, icLibFuzzer can still find new code coverage steadily after 24 hours, indicating that fuzzing results within 24 hours may still be insufficient to represent a fuzzer's capability to find new bugs or cover additional code.

Upgrade and compare with the latest fuzzers. To demonstrate that icLibFuzzer can easily update to incorporate newer libFuzzer versions, we update icLibFuzzer to LLVM 11 and compare it with icLibFuzzer-LLVM9 and a state-of-the-art fuzzer, AFL++ [7], released in late 2020.

As Figure 7 shows, icLibFuzzer-LLVM11 has competitive performance compared with AFL++. icLibFuzzer-LLVM11 performs better than icLibFuzzer-LLVM9 and AFL++ on `nm-new`. All three have similar code coverage on `readelf`, and AFL++ catches up after 24 hours on `nasm`. Due to our time budget, we can only run three binaries two times and 72 hours for each without initial seeds with these three fuzzers. We continue to run these experiments to evaluate the performance of the latest icLibFuzzer.

C. *RQ-2*

To answer this question, we run libFuzzer, icLibFuzzer, and AFL on the fuzzer-test-suite dataset and measure the number of executions per second. Using this dataset ensures libFuzzer can behave correctly.

As Table IV shows, icLibFuzzer is 4x to 50x slower than libFuzzer. However, when we compare icLibFuzzer with AFL, they run at almost the same speed even though icLibFuzzer supports more instrumentation than AFL.

We also compare the execution speed and main memory size of the naive forkserver and lightweight forkserver in Table V, as mentioned in Section IV. Since the naive forkserver approach forks directly from the fuzzer instance, it inherits the large memory space, slowing down the `fork` procedure and fuzzing target, thus the total execution speed of the fuzzer is 3x to 37x slower compared with icLibFuzzer.

D. *RQ-3*

As mentioned in Section IV, structure packing reduces the memory size in the forkserver and increases the speed. We evaluate the execution number per second with and without structure packing on four binaries, and Table III summarizes the results. On average, structure packing halves the memory usage and doubles the execution speed of icLibFuzzer.

VI. Discussion

This section discusses two crucial factors that influence the fuzzing results: initial seeds and the number of fuzzing threads.

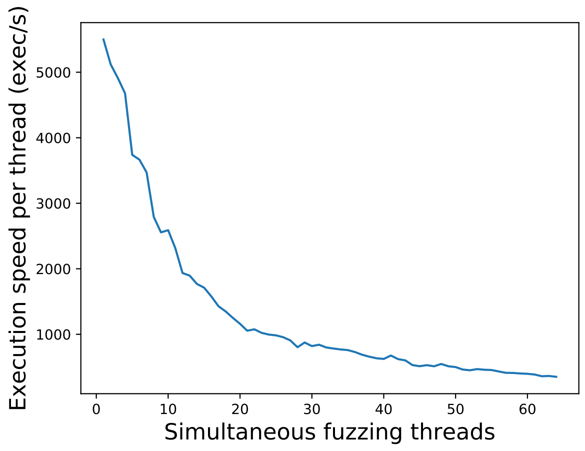


Fig. 8: Running multiple AFL fuzzer instances simultaneously. We run AFL on readelf with 1 to 64 fuzzer instances simultaneously. We spend 30 minutes in each case to wait for the execution speed to become rather stable and report the average speed.

We view them as control variables in the evaluation to ensure fair comparison among fuzzers. Further investigation is needed to understand their impact on fuzzing.

A. *Initial seed selection*

Prior work shows that initial seeds may affect the fuzzing result drastically [11] and recommends that it may also be a good idea to assign no initial seed (i.e., starting a fuzzer with an empty seed pool). It hypothesizes that if one fuzzer can perform well across various programs without initial seeds, it is likely that this fuzzer can also perform well on other datasets without initial seeds. Therefore, in our experiments, we fuzz these real-world programs without assigning any initial seeds. Our experiment results on real-world programs conforms with this hypothesis, but we leave choosing initial seeds wisely and fairly, and evaluating fuzzing with these initial seeds for future work.

B. *The number of fuzzing threads*

Several fuzzers [4], [5], [7], [13], [14] support parallelism to utilize processing power better. Additionally, when conducting fuzzing experiments, one might run multiple fuzzer instances parallelly and compute the averaged results to reduce randomness.

However, we observe from our experiment that when running multiple fuzzers in parallel, the number of concurrent fuzzing threads affects the fuzzing speed. The reason might be that due to the Linux kernel's design nature, some system calls cannot run in parallel, which limits the scalability of parallel fuzzing.

Figure 8 provides an example, showing how AFL's execution speed decrease as number of fuzzer instances increases.

This observation coincides with the results reported in prior work [18] that AFL and LibFuzzer have little performance gain when using more cores in parallel fuzzing; the total execution time per second does not increase linearly with the number of cores used.

Hence, fuzzing experiments should control the number of fuzzer instances running simultaneously. For example, if we would like to run icLibFuzzer on objdump for 30 times and compute the average code coverage, we cannot split the experiment into 10 plus 20 times; otherwise, the execution speed will differ significantly, making the average value meaningless.

C. *Case Study*

Due to our time budget, we did not fuzz many real-world programs to find bugs (e.g., enabling the address sanitizer). Among those we tested, we found one crash on *nasm* after fuzzing it with icLibFuzzer for 15 hours.

The crash happens in the `paste_tokens` function:

```
1  p = buf = nasm_malloc(len + 1);
2  while (tok != next) {
3     p = mempcpy(p, tok_text(tok), tok->len);
4     tok = delete_Token(tok);
5  }
6  *p = '\0';
7  *prev_next = tok = t = tokenize(buf);
8  nasm_free(buf);
9
10 /*
11  * Connect pasted into original stream,
12  * ie A -> new-tokens -> B
13  */
14 while (t->next) {
15    t = t->next;
16 }
```

The object `t` returned by `tokenize` in line 7 is a link list, and in lines 14-16 it tries to find the tail of this link list. However, it fails to check if `t` is `NULL` or not. If `buf` in line 13 is `NULL`, then the returned `t` will be `NULL`, causing line 14 to crash as it tries to dereference a `NULL` pointer.

VII. Guideline for upgrading to newer libFuzzer

As libFuzzer evolves actively, we design icLibFuzzer to be generic enough to work with future libFuzzer updates and provide a guideline to port libFuzzer's new features to icLibFuzzer quickly.

1) **Non-instrumentation features**: Because the change we made in libFuzzer mainly affects how the fuzzer handles data structure that stores instrumentation information, libFuzzer updates that are irrelevant to instrumentation should work seamlessly without any modification to icLibFuzzer.
2) **Instrumentation features**: icLibFuzzer separates the fuzzer instance's context and fuzzing target's context, and we use shared memory to pass the information collected from instrumentation to the main controller. To add new data structure that stores instrumentation information, simply follow the two steps below:
   a) **Increase the allocated memory size**: The first step is increasing the allocated memory size to incorporate

the new instrumentation information, but the increase should be as small as possible to minimize the impact on fuzzing speed. The member method `Initialize` in structure `TracePC` in `FuzzerTracePC.h` will allocate several contiguous shared memory pages storing every data structure related to instrumentation in order to share information between main controller and forkserver, One should try to minimize the required amount of increased memory because the memory size dominates icLibFuzzer's execution speed, as shown in Section V.

b) **Map data structure to shared memory**: After allocating additional shared memory, we point the newly allocated shared memory to the data structure storing information collected from instrumentation. When the instrumentation code in the fuzzing target gathers new information, it will be passed to the main controller by the newly added data structure on the shared memory.

VIII. Related Work

Our work aims to improve comparability between libFuzzer-based fuzzers and other fuzzers in AFL family. In this section, we review related work that enhances, integrates, or compares with libFuzzer.

A. Enhancing libFuzzer

Several projects aim to improve libFuzzer [14], [15]. ParmeSan [21] implements sanitizer-oriented fuzzing on top of libFuzzer, aiming to optimize for bug coverage. FuzzerGym [6] applies reinforcement learning to improve seed mutation, implements the idea by combining OpenAI Gym with libFuzzer, and evaluates using the fuzzer-test-suite dataset.

Entropic [3] improves libFuzzer's seed scheduling algorithm. It quantifies the amount of information gained by each seed and chooses which seed to mutate first according to this value. Entropic is available in the latest libFuzzer (included in LLVM11, released in September 2020). As we show in the evaluation, icLibFuzzer can support this new libFuzzer feature easily, demonstrating the strength of our fuzzer. Moreover, icLibFuzzer-LLVM11 (using the latest libFuzzer) has competitive performance with AFL++ (v3), a powerful AFL-based fuzzer released in December 2020; this reconfirms the need to bridge the libFuzzer and AFL families, allowing new techniques to be shared transparently.

LibKluzzer [12] uses libFuzzer as a part of its hybrid fuzzer engine. To mitigate the context pollution problem, it identifies global variables, snapshots these global variables, and reinitializes these global variables in every fuzzing iteration. However, this approach may face several problems as mentioned in Section IV.

Our work differs from previous work in that we aim to improve libFuzzer's comparability with other fuzzers over a wide range of programs. Thus, we focus on solving the context pollution problem and reducing the manual efforts in modifying the target code. We also pay attention to implementation details to preserve libFuzzer's performance advantage.

B. Integrating or Comparing with libFuzzer

EnFuzz [5] combines fuzzers with diverse capabilities to increase overall code coverage, and one of the included fuzzers is libFuzzer. EnFuzz implements a monitor thread to restart libFuzzer whenever it crashes, which is similar to enabling the ignore-crash mode in libFuzzer. However, as mentioned in Section IV, EnFuzz's approach does not fully solve the comparability issue because the context may still be polluted and produce incorrect results.

OSS-Fuzz [16] is developed to find bugs in real-world programs and employs libFuzzer as one of its components. Instead of modifying libFuzzer, OSS-Fuzz modifies the fuzzing target to ensure libFuzzer works correctly.

IX. Conclusion

Although libFuzzer is under active development and supports advanced techniques such as CMP tracing and value profiling, its progress and success have not been thoroughly compared against other fuzzing tools.

In this work, we propose and implement icLibFuzzer, a new libFuzzer mode that can fuzz programs not designed to run continuously, including many real-world programs, without manually modifying them. icLibFuzzer achieves higher code coverage than several state-of-the-art fuzzers on most of the real-world programs we tested. Additionally, we design icLibFuzzer to be compatible with new features added in the future. We envision icLibFuzzer can serve as another baseline in fuzzing research.

We are currently working on integrating our compiler wrapper with FuzzGen [10], an automated tool to generate *LLVMFuzzerTestOneInput* for a library function, so that we can run icLibFuzzer not only on standalone programs but also library functions.

We are interested in three future research directions: (1) investigating factors critical to fuzzer comparability, such as initial seed selection; (2) improving icLibFuzzer's speed via infrastructural and implementation optimizations; and (3) exploring cache-aware structure packing to eliminate misalignment, a potential side-effect of structure packing.

References

[1] (2020) FuzzBench - Fuzzer benchmarking as a service. [Online]. Available: https://github.com/google/fuzzbench

[2] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[3] M. Böhme, V. J. Manès, and S. K. 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*, 2020, pp. 678–689.

[4] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[5] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1967–1983. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/chen-yuanliang

[6] W. Drozd and M. Wagner, "Fuzzergym: A competitive framework for fuzzing and learning," 07 2018.

[7] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[8] V. Herdt, D. Große, J. Wloka, T. Güneysu, and R. Drechsler, "Verification of embedded binaries using coverage-guided fuzzing with systemc-based virtual prototypes," in *GLSVLSI*, 2020.

[9] G. Inc. (2016) A set of tests (benchmarks) for fuzzing engines (fuzzers). [Online]. Available: https://github.com/google/fuzzer-test-suite/

[10] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou

[11] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[12] H. Le, *LLVM-based Hybrid Fuzzing with LibKluzzer (Competition Contribution)*, 04 2020, pp. 535–539.

[13] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "Pafl: Extend fuzzing optimizations of single mode to industrial parallel mode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 809–814. [Online]. Available: https://doi.org/10.1145/3236024.3275525

[14] libfuzzer@googlegroups.com. (2019) LibFuzzer – a library for coverage-guided fuzz testing. [Online]. Available: http://llvm.org/docs/LibFuzzer.html

[15] K. Serebryany, "Continuous fuzzing with libfuzzer and addresssanitizer," in *2016 IEEE Cybersecurity Development (SecDev)*, 2016, pp. 157–157.

[16] K. Serebryany, "Oss-fuzz - google's continuous fuzzing service for open source software." Vancouver, BC: USENIX Association, Aug. 2017.

[17] R. Swiecki, "Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options," 2017. [Online]. Available: https://github.com/google/honggfuzz

[18] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2313–2328. [Online]. Available: https://doi.org/10.1145/3133956.3134046

[19] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM : A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[20] M. Zalewski. American Fuzzy Lop (AFL). [Online]. Available: http://lcamtuf.coredump.cx/afl/

[21] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided Greybox Fuzzing," in *USENIX Security*, Aug. 2020. [Online]. Available: Paper=https://download.vusec.net/papers/parmesan_sec20.pdf Code=https://github.com/vusec/parmesan