


Schfuzz: Detecting Concurrency Bugs with Feedback-Guided Fuzzing

Hiomasa Ito², Yutaka Matsubara² and Hiroaki Takada^{1,2} ^a

¹*Institute of Innovation for Future Society, Nagoya University, Aichi, Japan*

²*Graduate School of Informatics, Nagoya University, Aichi, Japan*

Keywords: Fuzzing, Concurrency Testing, Concurrency Bug Detection, Feedback-Guided Fuzzing, Memory-Access-Guided Fuzzing.

Abstract: It is challenging to detect concurrency bugs with fuzzing. There are two main reasons for this. First, manifesting them by exploring input space is inefficient because they only occur under specific interleavings. Second, re-giving an input detected a bug in a fuzzing campaign does not necessarily reproduce the bug because typical runtimes do not schedule threads deterministically. This research proposes Schfuzz, a novel approach for detecting concurrency bugs with feedback-guided fuzzing. This approach executes programs under test deterministically based on test cases generated by fuzzers. In addition, it feeds back dynamic memory-access orders to aid fuzzers in detecting concurrency bugs more efficiently and effectively. We evaluate Schfuzz with a hand-made motivating example and four benchmark programs from SCTBench (Thomson et al., 2016). The result shows that it can detect concurrency bugs more efficiently and effectively than traditional feedback-guided fuzzing.

1 INTRODUCTION

System developers have to design concurrent systems because it is challenging to increase the density of semiconductor integration further in the post-Moore era. Although concurrent systems enable us to use computational resources effectively, developers often suffer from *concurrency bugs*. They are errors in concurrent systems manifesting when the systems schedule threads in particular interleavings. We need an effective method to detect concurrency bugs because it is more challenging to detect them than sequential ones.

Although *fuzzing* is one of the most successful automated software testing methods, detecting concurrency bugs with fuzzing is not so straightforward. There are two main reasons for this. First, exploring input space is inefficient for hunting concurrency bugs because most inputs may run under the same interleaving. Second, bugs detected in fuzzing campaigns are hard to reproduce because typical runtimes schedule threads non-deterministically.

This research proposes *Schfuzz*, a novel method for detecting concurrency bugs with feedback-guided fuzzing. This approach has two key points. First, it executes programs under test deterministically based on test cases generated by fuzzers. Second, it feeds

back dynamic memory-access orders to aid fuzzers in detecting concurrency bugs more efficiently and effectively.

The contributions of the research are as follows:


- We propose Schfuzz, a novel approach to detect concurrency bugs with feedback-guided fuzzing.
- We implement the prototypes of Schfuzz and statistically compare it with traditional fuzzing methods based on quantitative metrics retrieved from the experimental results.

2 BACKGROUND AND MOTIVATION

2.1 Concurrency Bug

A concurrency bug (e.g., data race, deadlock, and atomicity violation) is one of the errors that can occur in concurrent systems. If a system has latent concurrency bugs, the bugs can occur when the system schedules threads in particular interleavings. The occurred bug can propagate and finally fail the system.

Even if a system has latent concurrency bugs, it is not easy to manifest them artificially. In order to manifest them, the system must run under specific interleavings. The number of such interleavings is tiny

^a  <https://orcid.org/0000-0003-3544-2397>

relative to the size of the interleaving space. In addition, the input which manifested the bug does not necessarily reproduce the bug because most runtimes do not schedule threads deterministically. Therefore, there is a need for a novel method that can 1) efficiently explore interleaving space, 2) effectively detect concurrency bugs, and 3) reproduce detected bugs without extra effort.

2.2 Fuzzing

Fuzzing is one of the software testing methods. Generally, it repeats the following three steps:

1. A test case generator (fuzzer) generates a test case and feeds it to a program under test (PUT).
2. The PUT runs with the test case.
3. The fuzzer checks the PUT's liveness (e.g., status code).

Although the concept of fuzzing proposed by Miller et al. (1990) is so simple, it detected many bugs in the UNIX utility programs. In recent years, many researchers have studied how to detect various bugs more efficiently and effectively with fuzzing (Bohme et al., 2017; Rawat et al., 2017; Stephens et al., 2016; Yun et al., 2018).

Feedback-guided fuzzing feeds back runtime information in the fuzzing loop. A feedback-guided fuzzer evaluates a test case using the runtime information gathered when the PUT ran with the test case. It stores highly-rated test cases as seeds and uses them to generate beneficial descendant test cases. For example, a code-coverage-guided fuzzer uses test cases with improved code coverage in a fuzzing campaign as seeds and generates new ones by mutating the seeds. American Fuzzy Lop¹ (AFL) and libFuzzer² are typical code-coverage-guided fuzzers that have detected numerous bugs and vulnerabilities in real-world programs.

Although existing fuzzing methods and various fuzzers have detected many sequential bugs, detecting concurrency bugs is not straightforward. First, exploring input space is inefficient for detecting concurrency bugs. Even though various inputs indirectly explore interleaving space because PUTs can run under various interleavings with them, it is less efficient than direct exploration. Second, even if an input in a fuzzing campaign manifests a bug, the input does not necessarily reproduce the bug due to the non-determinism of thread scheduling. Therefore, there is a need for a novel approach to detecting concurrency bugs with fuzzing.

¹<https://lcamtuf.coredump.cx/afl/>

²<https://llvm.org/docs/LibFuzzer.html>

```

1 void
2 main_task(intptr_t exinf)
3 {
4     while (1) {
5         char c; // character input
6         if (!read_data(&c, sizeof(char))) {
7             continue;
8         }
9         if ((c & 0b00001111) == 0b1111) {
10            consume_time(1000);
11
12            // acquire lock
13            loc_cpu();
14            lock = 1;
15            unl_cpu();
16
17            // do nothing
18
19            // release lock
20            loc_cpu();
21            lock = 0;
22            unl_cpu();
23
24            consume_time(1000);
25        } else {
26            consume_time(1000);
27        }
28    }
29 }

```

Figure 1: Main task of the motivating example.

2.3 Motivating Example

Figures 1 and 2 show the source codes of an application program of TOPPERS/ASP3³, an open-source real-time operating system that runs on GR-PEACH⁴, an RZ/A1H SoC development board. The application consists of a task and an interrupt service routine (ISR).

Figure 1 shows the code for the main task of the example. It consists of a single infinite loop. In the loop, it repeatedly reads input, and if its last four bits are all one, it acquires and releases the lock represented by the variable `lock`.

Figure 2 shows the code for the ISR of the example. It checks the state of the lock, and if its value is one, it increments the value of the global variable `shm`. Because the access to `shm` is not synchronized, other threads (i.e., ISR instances) can overwrite its value between its increment and its reload for assertion checking. In other words, data race can occur because the ISR is not reentrant.

In order to manifest the bug, threads must interleave to satisfy the following two conditions:

³<https://toppers.jp/asp3-kernel.html>

⁴<https://www.renesas.com/us/en/products/gadget-renesas/boards/gr-peach>

```

1 void
2 intno1_isr(intptr_t exinf)
3 {
4     intno1_clear();
5
6     unsigned int lock_state = 0;
7
8     // check the lock state
9     iloc_cpu();
10    lock_state = lock;
11    iunl_cpu();
12
13    // iff. the task have the lock, increment
14    shm.
15    if (lock_state) {
16        unsigned int buf = 0;
17        buf = shm;
18        shm++;
19        assert(shm == (buf + 1));
20    }

```

Figure 2: Interrupt service routine (ISR) of the motivating example.

1. Firing the first interrupt when the value of `lock` is one.
2. Firing the second interrupt between incrementing `shm` and reloading it for assertion checking.

The number of such interleavings is minimal relative to the size of the interleaving space. In addition, since input rarely contributes to interleaving space exploration in this example, it is hard to satisfy condition 2 with traditional fuzzing methods. In order to efficiently explore the latent bug in this example, fuzzers should also explore the interleaving space.

3 SCHFUZZ

This research proposes Schfuzz, a novel method to detect concurrency bugs with feedback-guided fuzzing. There are two critical points of the method.

- It executes PUTs deterministically based on test cases generated by fuzzers.
- It feeds back dynamic memory-access orders to fuzzers.

Figure 3 shows an overview of Schfuzz. A *scheduler* receives a test case generated by a fuzzer and interprets it as a series of *schedules*. A *runtime environment* (e.g., a system emulator) executes a PUT under the schedule, records orders of shared-memory access, and feeds them back to the fuzzer.

Algorithm 1 shows how schedulers and runtime environments execute PUTs deterministically based

on test cases generated by fuzzers. If a bug occurs with a test case, running the PUT with the test case can reproduce the bug. The algorithm repeats until it exhausts data in a test case (lines 2 - 9). A *schedule* consists of *addr*, *maxHitCount*, and *maxInstCount* and a scheduler retrieves it from the test case (line 4). A runtime environment executes a PUT under the schedule and pauses the execution when the value of the program counter matches *addr*'s value the number of times as *maxHitCount*'s value (line 6). The value of *maxInstCount* is the upper bound of the number of instructions executed before pausing. After that, the runtime environment saves the current thread context (line 7) and gets the next available thread (line 8). Note that the function `transformToValidAddr` validates the value of *addr* because it is likely to be unreachable if *addr* is allowed to take any value. If *addr* takes an invalid value, `transformToValidAddr` transforms its value into a valid one (line 5). Testers must determine the valid value range because it depends on PUTs.

Schfuzz feeds back shared-memory access orders to fuzzers for detecting concurrency bugs effectively and efficiently. Typical feedback-guided fuzzing feeds back basic-block level paths and prefers test cases that improve code coverage. However, this strategy is not suited for detecting concurrency bugs. There are two main reasons for this. First, a context switch can occur in concurrent programs in the middle of a basic block. Because of this feature, static-basic-block level paths imprecisely represent PUT's executions. Likewise, the number of dynamic-basic-block level paths increases exponentially to the number of instructions and threads. Second, a concurrency bug occurs when threads access shared memories in particular orders. Generally, improving code coverage helps to detect latent concurrency bugs. Meanwhile, even if a test case improves the code coverage of a fuzzing campaign, it does not contribute to concurrency bug detection if the newly covered codes do not access shared memory. Therefore, Schfuzz feeds back access orders to shared memory and prefers test cases that access the memory in an order not yet observed in a fuzzing campaign.

For the evaluation described in section 4, we have implemented two prototypes of Schfuzz. They use American Fuzzy Lop (ver. 2.52b) as a feedback-guided fuzzer and Unicorn (ver. 1.0.3) and Qiling (ver. 1.4.3) as fine-controllable runtime environments based on schedules. It depends on runtime environments how to control executions and gather runtime information. In order to run fuzzing for our motivating example, we have implemented the first proto-

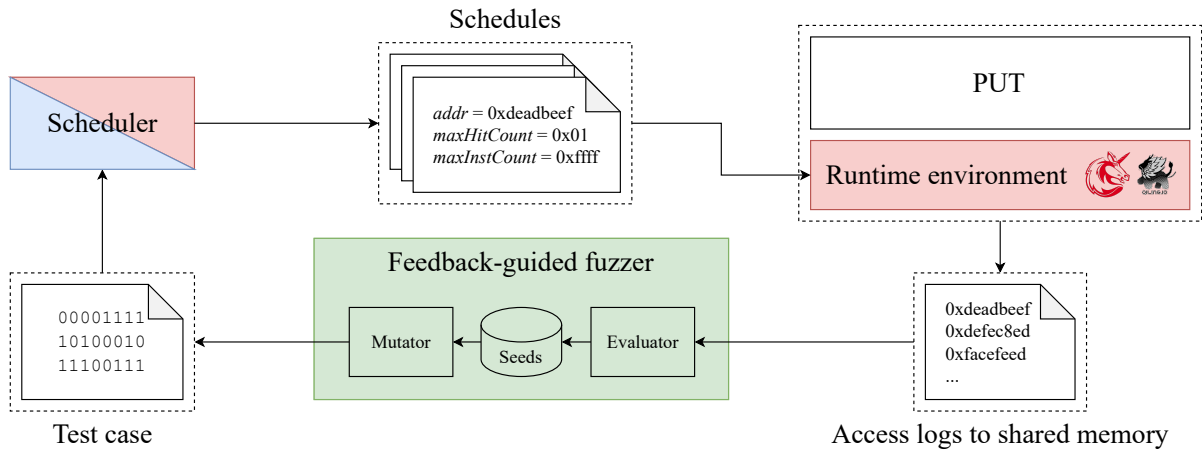


Figure 3: Overview of Schfuzz. For prototyping, we use American Fuzzy Lop (AFL) as a feedback-guided fuzzer and Unicorn and Qiling as runtime environments for PUTs. For the first prototype, we have implemented the scheduler in the code of the test driver. On the other hand, for the second prototype, we have integrated the scheduler into Qiling’s code.

```

Input: testCase, mainThread
1: currentThread ← mainThread
2: while testCase is not EOF do
3:   ctx ← loadContext(currentThread)
4:   addr, maxHitCount, maxInstCount ← getSchedule(testCase)
5:   validAddr ← transformToValidAddr(addr)
6:   new_ctx ← run(ctx, validAddr, maxHitCount, maxInstCount)
7:   saveContext(new_ctx)
8:   currentThread ← getNextThread()
9: end while
    
```

Algorithm 1: Deterministic execution based on a test case.

type with Unicorn⁵, a multi-architecture CPU emulation framework. We use its APIs to implement Algorithm 1 because they enable us to control and execute PUTs at the instruction level. In addition, they enable us to hook various runtime events (e.g., memory access). The prototype records runtime information with the APIs. Although the first prototype enables us to run fuzzing for the motivating example, we should apply Schfuzz to more various PUTs for evaluation. Many real-world concurrent programs run on the Linux/ptthread environment, but it is not straightforward to run Linux/ptthread software on vanilla Unicorn because it is just a CPU emulator (i.e., an instruction-set simulator). Hence we have implemented the second prototype with Qiling⁶, a binary emulation framework that uses Unicorn as its backend. Although we cannot control PUT’s execution via Qiling as finely as Unicorn, we modified it to enable test-case-based deterministic execution. The second prototype records runtime information with the APIs of Qiling as those of Unicorn.

⁵<https://github.com/unicorn-engine/unicorn>

⁶<https://github.com/qilingframework/qiling>

4 EVALUATION

4.1 Experimental Setup

We experimentally evaluate Schfuzz. It is clear that Schfuzz has better reproducibility of bugs than existing fuzzing methods because it deterministically executes PUTs based on test cases generated by fuzzers. In the experiment, we evaluate the effectiveness of using access orders to shared memory as runtime feedback. We run fuzzing campaigns many times for five PUTs using three different runtime feedback and discuss the effectiveness of Schfuzz based on quantitative metrics of the results. All of the experiments run on Ubuntu 18.04 (Linux 5.4.0) on Intel(R) Core(TM) i9-9960X CPU @ 3.10GHz and 32 GiB RAM.

Research Question. There are two research questions in the experiment.

RQ1. Does Schfuzz improve the *efficiency* of detecting concurrency bugs compared to existing fuzzing methods?

RQ2. Does Schfuzz improve the *effectiveness* of detecting concurrency bugs compared to existing fuzzing methods?

The time and the number of generated test cases to detect bugs index the efficiency, and bug-detection rates index the effectiveness.

PUT. In the experiment, we use five concurrent programs as PUT. One is the motivating example shown in section 2.3, and we run 100 10-minute fuzzing campaigns for it with each method described in the paragraph below. The others (reorder_10_bad, reorder_20_bad, safestack, and twostage_100_bad) are benchmark programs in SCTBench (Thomson et al., 2016), a set of buggy concurrent software run on the Linux/pthread environment. Thomson et al. (2016) have evaluated various methods to detect concurrency bugs with SCTBench. In their evaluation, the number of methods that detected the bugs in the four programs is fewer than those that detected bugs in the others. In other words, they are the four most challenging programs in SCTBench to detect bugs. They all have assertion checks, and their checks can fail when they run under specific interleavings. We run 30 6-hour fuzzing campaigns for them with each method.

Method. We compare three methods, each of which uses the following runtime information:

- Shared-memory access orders (Schfuzz).
- Basic-block level paths (code-coverage-guided fuzzing).
- None (dumb fuzzing).

As shown in Algorithm 1, a test case is interpreted as a series of schedules. However, when the PUT is the motivating example, it is interpreted as a series of inputs and schedules because we must give specific inputs to manifest the bug. Table 1 shows the size of each element that makes up a schedule and an input.

Limitations of the Prototypes. There are two limitations of the prototypes.

1. A tester must provide shared memory addresses to be monitored.
2. A tester must provide a range for the function transformToValidAddr in Algorithm 1.

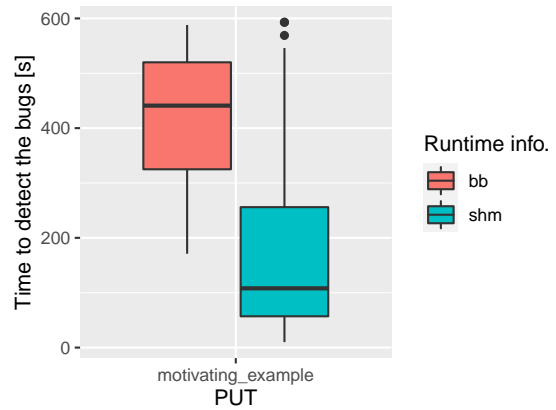


Figure 4: Time distributions to detect the bug in the motivating example.

Table 2 shows the shared variables to be monitored in the experiments for each PUT. The range of transformToValidAddr in the experiments is the address range of their application codes. Note that we modified the source code of safestack because of limitation 1. Since safestack dynamically allocates the memory to be monitored and the prototypes cannot monitor memory whose address is determined at runtime, we modified the code to allocate it statically.

4.2 Result

Table 3 shows each method’s time to detect the latent bugs in each PUT. In the fuzzing campaigns for the motivating example, Schfuzz detected the bug on average 2.36 times faster than code-coverage-guided fuzzing. Figure 4 shows the box plot of the time distributions to detect the bug in the motivating example. The p-value calculated by the Brunner-Munzel test (Brunner and Munzel, 2000) for these distributions is less than $2.2e^{-16}$, which means significant differences between them.

Table 4 shows the number of test cases each method generates to detect the latent bugs in each PUT. Note that the values are estimated by integrating the number of test cases executed per second. In the fuzzing campaigns for the motivating example, Schfuzz detected the bug in 2.94 times fewer test cases on average than code-coverage-guided fuzzing. Figure 5 shows the box plot of the distributions of the estimated number of test cases generated to detect the bug in the motivating example. The p-value calculated by the Brunner-Munzel test (Brunner and Munzel, 2000) for these distributions is $6.6e^{-16}$, which means significant differences between them.

Table 5 shows the bug detection rate of each method in each PUT. Schfuzz has a higher bug detection rate than the other methods for the motivating

Table 1: Size of each element that makes up a schedule and an input.

PUT	Size [byte]				
	<i>numberOfInput</i>	<i>input</i>	<i>addr</i>	<i>maxHitCount</i>	<i>maxInstCount</i>
motivating example	1	<i>numberOfInput</i>	4	1	2
reorder_10_bad	0	0	4	1	2
reorder_20_bad	0	0	4	1	2
safestack	0	0	4	1	2
twostage_100_bad	0	0	4	1	2

Table 2: Shared variables to be monitored in the experiments for each PUT.

PUT	Vars to be monitored
motivating example	lock, shm
reorder_10_bad	a, b
reorder_20_bad	a, b
safestack	stack_items, stack
twostage_100_bad	data1Value, data2Value

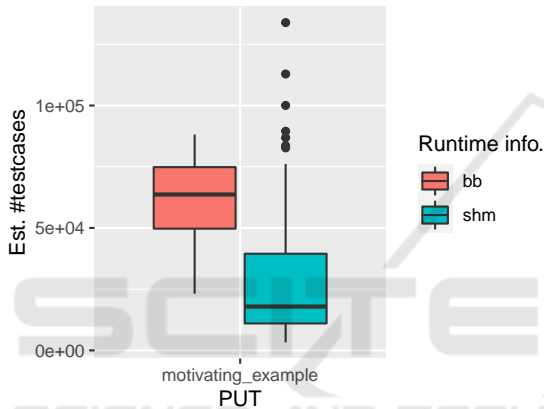


Figure 5: Distributions of the estimated number of test cases generated to detect the bug in the motivating example.

example, reorder_10_bad and reorder_20_bad. On the other hand, it could not detect the bugs in safestack and twostage_100_bad as the other methods. Code-coverage-guided fuzzing detected only the bug in the motivating example in 33% of the fuzzing campaigns. Regarding dumb fuzzing, it could not detect any bugs in the PUTs.

4.3 Discussion

We discuss the following three topics based on the results:

1. The bugs which Schfuzz could not detect.
2. The answers to the research questions.
3. The threats to validity.

Undetected Bugs. We should discuss the bugs which Schfuzz could not detect. It could not detect the bugs in safestack and twostage_100_bad with 6-hour fuzzing campaigns. Although the bug in safes-

tack is not false-positive, it is known to be challenging to detect. No method could detect it in the evaluation by Thomson et al. (2016). The most recent work on concurrency bug detection (Wen et al., 2022) also failed to detect the bug. twostage_100_bad concurrently runs 100 threads, which access the shared variables. A large number of threads has two adverse effects on Schfuzz. First, it slows down fuzzing speed. In other words, PUTs run with fewer test cases. Second, it makes detecting bugs more difficult because it increases the size of interleaving space.

Answers to Research Questions. We should answer the research questions based on the results.

A1. Yes. Schfuzz can detect concurrency bugs more efficiently than the existing fuzzing methods.

In the experiment, Schfuzz detected the bugs in the three PUTs faster and with fewer test cases than the other methods. In addition, there are significant differences among the distributions of time and the number of test cases to detect the bugs.

A2. Yes. Schfuzz can detect concurrency bugs more effectively than the existing fuzzing methods.

Schfuzz showed a higher bug-detection rate in the experiment than the existing method for the three PUTs.

Threats to Validity. We should discuss the threats to the validity:

- Fuzzing has randomness. Therefore, it is impossible to get the same result as this. In order to reduce the effect of the randomness, we have analyzed the statistics based on many-times fuzzing campaigns.
- The efficiency and effectiveness of Schfuzz depend on the property and amount of knowledge given by testers. For example, testers give knowledge about shared-memory addresses to be monitored and the range of the function transformTo-

Table 3: Statistics on time to detect the bugs. “Runtime info.” indicates the methods. “shm” denotes Schfuzz, “bb” denotes code-coverage-guided fuzzing, and “none” denotes dumb fuzzing. “#samples” means the number of the fuzzing campaigns that detected the bugs. In the columns of the statistical values, “NA” means no values.

PUT	Runtime info.	#samples	Mean [s]	S.D. [s]	Median [s]	Min [s]	Max [s]
motivating example	shm	93	176.96	165.14	108	10	593
	bb	33	418.28	127.31	441	171	588
	none	0	NA	NA	NA	NA	NA
reorder_10_bad	shm	20	8546.10	5147.28	7554	1043	19593
	bb	0	NA	NA	NA	NA	NA
	none	0	NA	NA	NA	NA	NA
reorder_20_bad	shm	21	10240.33	5953.48	8574	1843	19050
	bb	0	NA	NA	NA	NA	NA
	none	0	NA	NA	NA	NA	NA
safestack	shm	0	NA	NA	NA	NA	NA
	bb	0	NA	NA	NA	NA	NA
	none	0	NA	NA	NA	NA	NA
twostage_100_bad	shm	0	NA	NA	NA	NA	NA
	bb	0	NA	NA	NA	NA	NA
	none	0	NA	NA	NA	NA	NA

Table 4: Statistics on the estimated number of test cases generated to detect the bugs. “Runtime info.” indicates the methods. “shm” denotes Schfuzz, “bb” denotes code-coverage-guided fuzzing, and “none” denotes dumb fuzzing. “#samples” means the number of the fuzzing campaigns that detected the bugs. In the columns of the statistical values, “NA” means no values.

PUT	Runtime info.	#samples	Mean	S.D.	Median	Min	Max
motivating example	shm	93	30788.32	27817.70	17895	3250	133884
	bb	33	61217.48	18446.95	63653	23165	88163
	none	0	NA	NA	NA	NA	NA
reorder_10_bad	shm	20	10134.85	5949.11	9276	1196	22791
	bb	0	NA	NA	NA	NA	NA
	none	0	NA	NA	NA	NA	NA
reorder_20_bad	shm	21	6769.76	3721.73	6693	1189	12216
	bb	0	NA	NA	NA	NA	NA
	none	0	NA	NA	NA	NA	NA
safestack	shm	0	NA	NA	NA	NA	NA
	bb	0	NA	NA	NA	NA	NA
	none	0	NA	NA	NA	NA	NA
twostage_100_bad	shm	0	NA	NA	NA	NA	NA
	bb	0	NA	NA	NA	NA	NA
	none	0	NA	NA	NA	NA	NA

ValidAddr. The property and amount of knowledge would affect the result.

- Concurrency bugs must be observable to detect them with Schfuzz. Although we can observe the bugs in the experimental PUTs as assertion failures, some concurrency bugs, such as deadlocks and starvations, are hard to generalize how to observe because their definition depends on system requirements.
- Although we evaluated Schfuzz with the five PUTs, more experiments with more realistic PUTs would allow us to evaluate it more precisely.

5 RELATED WORK

5.1 Concurrency Bug Detection

Many researchers have published works to detect concurrency bugs. According to Fu et al. (2018), we can classify them into the following three categories:

- Random delay disturbance (Park et al., 2009; Chew and Lie, 2010).
- Thread scheduling/switching (Musuvathi et al., 2008; Musuvathi and Qadeer, 2007; Yu et al., 2012; Burckhardt et al., 2010).
- Fuzzing (Sen, 2008; Joshi et al., 2009; Cai et al., 2014).

Table 5: Detection rates of the bugs. “Runtime info.” indicates the methods. “shm” denotes Schfuzz, “bb” denotes code-coverage-guided fuzzing, and “none” denotes dumb fuzzing.

PUT	Runtime info.	Rate [%]
motivating example	shm	93.0
	bb	33.0
	none	0.0
reorder_10_bad	shm	66.7
	bb	0.0
	none	0.0
reorder_20_bad	shm	70.0
	bb	0.0
	none	0.0
safestack	shm	0.0
	bb	0.0
	none	0.0
twostage_100_bad	shm	0.0
	bb	0.0
	none	0.0

Random delay disturbance (Park et al., 2009; Chew and Lie, 2010) is a category of methods that randomly insert delays in PUT’s execution. Delays induce various interleavings and can manifest latent concurrency bugs. Stress testing is a common practice in this category. Nevertheless, methods in this category are considered inadequate because they tend to execute PUTs under similar interleavings.

Thread scheduling/switching (Musuvathi et al., 2008; Musuvathi and Qadeer, 2007; Yu et al., 2012; Burckhardt et al., 2010) is a category of methods that control thread scheduling to execute PUTs under various interleavings. Schfuzz is classified into this category because it controls thread scheduling based on test cases generated by fuzzers. This category has two sub-categories, systematic and non-systematic. Systematic methods (Musuvathi et al., 2008; Musuvathi and Qadeer, 2007) explore interleaving space systematically (i.e., exhaustively). However, it is infeasible to explore interleaving spaces of large-scale real-world programs because the size of interleaving space increases exponentially to the number of threads and instructions in PUTs. Hence they often have some budget limits (e.g., a time limit for exploration) in practice. Although Thomson et al. (2016) applied some systematic methods to the four PUTs we used in our experiments to detect the bugs, no methods succeeded in detecting the bugs within a schedule bound of 100,000. Non-systematic methods (Burckhardt et al., 2010) do not exhaustively explore interleaving space. Schfuzz is non-systematic because fuzzing has randomness and does not guarantee to explore the space exhaustively. Although it is similar to random schedule generators, we can expect it can detect concurrency bugs more efficiently and effectively than random schedulers when we target complicated

programs because it is feedback-guided. In the experiment by Thomson et al. (2016), a random scheduler failed to detect the bugs in the four PUTs we used in our experiments.

Fuzzing (Sen, 2008; Joshi et al., 2009; Cai et al., 2014) is a category of methods to manifest potential bugs reported by other methods, such as data-race detectors. Although Fu et al. (2018) named the category “fuzzing,” note that its meaning differs from that used in the context of bug detection in sequential programs. The efficiency and effectiveness of the methods in this category to manifest bugs depend on methods to report potential bugs.

Our research aims to apply feedback-guided fuzzing to concurrency bug detection. Although it is also an exciting research topic to compare Schfuzz to them with quantitative metrics, that is currently out of the scope.

5.2 Fuzzing Concurrent Software

As mentioned in section 2.2, though most fuzzing studies aim to detect sequential bugs, some works target concurrency bugs like Schfuzz:

- ConAFL (Liu et al., 2018).
- MUZZ (Chen et al., 2020).
- AutoInter-fuzzing (Ko et al., 2022).
- CONZZER (Jiang et al., 2022).
- ConFuzz (Vinesh and Sethumadhavan, 2020).

Table 6 shows compare them to Schfuzz in summary. Unfortunately, it is almost impossible to verify them and to fairly compare them to Schfuzz with quantitative metrics because their tools, except ConAFL⁷, are publicly unavailable now⁸. In addition, ConAFL targets programs that process some input and run on Linux/pthread, so our motivating example and the programs in SCTBench (Thomson et al., 2016) are out of the scope.

ConAFL (Liu et al., 2018) runs fuzzing campaigns for PUTs, which are instrumented to run under various interleavings and to induce to manifest potential concurrency vulnerabilities. There are two main differences between ConAFL and Schfuzz:

- ConAFL uses test cases generated by fuzzers only for input-space exploration and explores interleaving space by randomly varying threads’ priorities. On the other hand, Schfuzz uses them to explore interleaving space. It deterministically decides when to switch contexts based on test cases.

⁷<https://github.com/Lawliar/ConAFL>

⁸Feb. 06, 2023.

Table 6: Comparing Schfuzz with the existing fuzzing studies. ✓* means partially yes because AutoInter-fuzzing controls thread scheduling when it runs fuzzing in its interleaving mode.

Property	ConAFL	MUZZ	AutoInter-fuzzing	CONZZER	ConFuzz	Schfuzz
Directly exploring interleaving space				✓		✓
Handling non-determinism in fuzzing			✓*	✓		✓
Reproducibility of detected bugs	✓		✓			✓

- ConAFL is a typical code-coverage-guided fuzzer. It prefers test cases that improve the code coverage in a fuzzing campaign.

MUZZ (Chen et al., 2020) runs fuzzing campaigns for PUTs, which are instrumented to run under various interleavings and to record runtime information about thread interleavings. There are three main differences between MUZZ and Schfuzz:

- MUZZ does not exclude the non-determinism of thread scheduling, and Chen et al. (2020) do not mention the reproducibility of detected bugs.
- MUZZ induces various interleavings by randomly varying threads’ priorities. This strategy corresponds to random delay disturbance in the classification of Fu et al. (2018) and is commonly inefficient.
- MUZZ uses test cases to explore input space as ConAFL.

AutoInter-fuzzing (Ko et al., 2022) prefers test cases with which PUTs access shared memory from many dispersed threads. There are two main differences between AutoInter-fuzzing and Schfuzz:

- AutoInter-fuzzing does not control thread scheduling when it runs fuzzing in its normal mode, so its non-determinism affects fuzzing campaigns. Therefore, it can pick up a useless test case as a seed by chance, resulting in inefficient and ineffective exploration.
- As ConAFL and MUZZ, AutoInter-fuzzing uses test cases to explore input space.

CONZZER (Jiang et al., 2022) explores data races by mutating pairs of function calls concurrently executable (concurrent call pair). There are two main differences between CONZZER and Schfuzz:

- Although CONZZER tries to manifest data races by executing various concurrent call pairs, it might waste time exploring codes that have nothing to do with data races because it does not consider whether an executed concurrent call pair can be involved in the data race (i.e., whether it accesses shared memory). On the other hand, Schfuzz prefers to explore codes involved in concurrency bugs because it employs test cases that access shared memory in an unobserved order as seeds.

- As MUZZ, Jiang et al. (2022) do not mention the reproducibility of detected bugs.

ConFuzz (Vinesh and Sethumadhavan, 2020) statically assigns weights to each basic block based on distances from thread function call sites (e.g., mutex lock/unlock). The weight of a test case is the sum of the weights of executed basic blocks, and ConFuzz prefers high-weight test cases. There are two main differences between ConFuzz and Schfuzz:

- ConFuzz does not handle the non-determinism of thread scheduling. As MUZZ and CONZZER, Vinesh and Sethumadhavan (2020) do not mention the reproducibility of detected bugs.
- As ConAFL, MUZZ, and AutoInter-fuzzing, ConFuzz uses test cases to explore input space.

6 CONCLUSION

This research proposed Schfuzz, a novel method to detect concurrency bugs with feedback-guided fuzzing. It has two key points. First, it controls PUTs’ execution with test cases generated by fuzzer and guarantees the reproducibility of detected bugs. Second, it feeds back dynamic orders of shared-memory access to explore interleaving space efficiently and effectively. For evaluation, we ran fuzzing campaigns for the five concurrent programs with Schfuzz, code-coverage-guided fuzzing, and dumb fuzzing. Schfuzz detected the bugs in three of the five PUTs most effectively and efficiently. Although Schfuzz failed to detect the bugs in the two PUTs, the other methods also failed to detect them. This result shows that Schfuzz can detect concurrency bugs more effectively and efficiently than the other methods.

There are some future works derived from this research. First, how to improve the efficiency and effectiveness of Schfuzz? It failed to detect the bugs in the two PUTs in the experiment. This result means we can further improve Schfuzz. Second, how to improve the applicability of Schfuzz? Source code or binary analysis may enable Schfuzz to automatically retrieve information about PUTs currently given by testers. Finally, can we extend the results of this research to general? We evaluated Schfuzz with the hand-made motivating example and the four most challenging

programs in SCTBench (Thomson et al., 2016) to detect bugs. Experiments with various real-world software will let us evaluate its generality more precisely.

REFERENCES

- Bohme, M., Pham, V.-T., and Roychoudhury, A. (2017). Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering*, 45(5):489–506.
- Brunner, E. and Munzel, U. (2000). The Nonparametric Behrens-Fisher Problem: Asymptotic Theory and a Small-Sample Approximation. *Biometrical Journal*, 42(1):17–25.
- Burckhardt, S., Kothari, P., Musuvathi, M., and Nagarakatte, S. (2010). A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA. Association for Computing Machinery.
- Cai, Y., Wu, S., and Chan, W. K. (2014). ConLock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 491–502, New York, NY, USA. Association for Computing Machinery.
- Chen, H., Guo, S., Xue, Y., Sui, Y., Zhang, C., Li, Y., Wang, H., and Liu, Y. (2020). MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *Proceedings of the 29th USENIX Security Symposium*, SEC’20, pages 2325–2342, USA. USENIX Association.
- Chew, L. and Lie, D. (2010). Kivati: Fast detection and prevention of atomicity violations. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys’10, pages 307–320, New York, NY, USA. Association for Computing Machinery.
- Fu, H., Wang, Z., Chen, X., and Fan, X. (2018). A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. *Software Quality Journal*, 26(3):855–889.
- Jiang, Z.-M., Bai, J.-J., Lu, K., and Hu, S.-M. (2022). Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium*, NDSS 2022, Reston, VA. Internet Society.
- Joshi, P., Park, C.-S., Sen, K., and Naik, M. (2009). A randomized dynamic program analysis technique for detecting real deadlocks. *ACM SIGPLAN Notices*, 44(6):110–120.
- Ko, Y., Zhu, B., and Kim, J. (2022). Fuzzing with automatically controlled interleavings to detect concurrency bugs. *Journal of Systems and Software*, 191:111379.
- Liu, C., Zou, D., Luo, P., Zhu, B. B., and Jin, H. (2018). A Heuristic Framework to Detect Concurrency Vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 529–541, New York, NY, USA. Association for Computing Machinery.
- Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44.
- Musuvathi, M. and Qadeer, S. (2007). Iterative context bounding for systematic testing of multithreaded programs. *ACM SIGPLAN Notices*, 42(6):446–455.
- Musuvathi, M., Qadeer, S., Nainar, P. A., Ball, T., Basler, G., and Neamtiu, I. (2008). Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI’08, pages 267–280, USA. USENIX Association.
- Park, S., Lu, S., and Zhou, Y. (2009). CTrigger: Exposing atomicity violation bugs from their hiding places. *ACM SIGARCH Computer Architecture News*, 37(1):25–36.
- Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., and Bos, H. (2017). VUzzer: Application-aware Evolutionary Fuzzing. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, NDSS 2017, Reston, VA. Internet Society.
- Sen, K. (2008). Race directed random testing of concurrent programs. *ACM SIGPLAN Notices*, 43(6):11–21.
- Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2016). Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, NDSS 2016, Reston, VA. Internet Society.
- Thomson, P., Donaldson, A. F., and Betts, A. (2016). Concurrency testing using controlled schedulers: An empirical study. *ACM Transactions on Parallel Computing*, 2(4):1–37.
- Vinesh, N. and Sethumadhavan, M. (2020). ConFuzz—A Concurrency Fuzzer. In *Proceedings of the 1st International Conference on Sustainable Technologies for Computational Intelligence*, volume 1045 of *Advances in Intelligent Systems and Computing*, pages 667–691, Singapore. Springer Singapore.
- Wen, C., He, M., Wu, B., Xu, Z., and Qin, S. (2022). Controlled concurrency testing via periodical scheduling. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE 2022, pages 474–486, Pittsburgh Pennsylvania. Association for Computing Machinery.
- Yu, J., Narayanasamy, S., Pereira, C., and Pokam, G. (2012). Maple: A coverage-driven testing tool for multithreaded programs. *ACM SIGPLAN Notices*, 47(10):485–502.
- Yun, I., Lee, S., Xu, M., Jang, Y., and Kim, T. (2018). QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium*, SEC’18, pages 745–761, USA. USENIX Association.