MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs

Hongxu Chen, *University of Science and Technology of China and Nayang Technological University;* Shengjian Guo, *Baidu Security;* Yinxing Xue, *University of Science and Technology of China;* Yulei Sui, *University of Technology Sydney;* Cen Zhang and Yuekang Li, *Nanyang Technological University;* Haijun Wang, *Ant Financial Services Group;* Yang Liu, *Nanyang Technological University*

https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

This paper is included in the Proceedings of the 29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs

Hongxu Chen[§][†] Shengjian Guo[‡] Yinxing Xue[§][*] Yulei Sui[¶]
Cen Zhang[†] Yuekang Li[†] Haijun Wang[#] Yang Liu[†]
[†]*Nanyang Technological University* [‡]*Baidu Security* [¶]*University of Technology Sydney*
[§]*University of Science and Technology of China* [#]*Ant Financial Services Group*

Abstract

Grey-box fuzz testing has revealed thousands of vulnerabilities in real-world software owing to its lightweight instrumentation, fast coverage feedback, and dynamic adjusting strategies. However, directly applying grey-box fuzzing to input-dependent multithreaded programs can be extremely inefficient. In practice, multithreading-relevant bugs are usually buried in the sophisticated program flows. Meanwhile, existing grey-box fuzzing techniques do not stress thread-interleavings that affect execution states in multithreaded programs. Therefore, mainstream grey-box fuzzers cannot adequately test problematic segments in multithreaded software, although they might obtain high code coverage statistics.

To this end, we propose MUZZ, a new grey-box fuzzing technique that hunts for bugs in multithreaded programs. MUZZ owns three novel thread-aware instrumentations, namely coverage-oriented instrumentation, thread-context instrumentation, and schedule-intervention instrumentation. During fuzzing, these instrumentations engender runtime feedback to accentuate execution states caused by thread interleavings. By leveraging such feedback in the dynamic seed selection and execution strategies, MUZZ preserves more valuable seeds that expose bugs under a multithreading context.

We evaluate MUZZ on twelve real-world multithreaded programs. Experiments show that MUZZ outperforms AFL in both multithreading-relevant seed generation and concurrency-vulnerability detection. Further, by replaying the target programs against the generated seeds, MUZZ also reveals more concurrency-bugs (e.g., data-races, thread-leaks) than AFL. In total, MUZZ detected eight new concurrency-vulnerabilities and nineteen new concurrency-bugs. At the time of writing, four reported issues have received CVE IDs.

1 Introduction

Multithreading has been popular in modern software systems since it substantially utilizes the hardware resources to boost software performance. A typical computing paradigm of multithreaded programs is to accept a set of inputs, distribute computing jobs to threads, and orchestrate their progress accordingly. Compared to sequential programs, however, multithreaded programs are more prone to severe software faults. On the one hand, the non-deterministic thread-interleavings give rise to *concurrency-bugs* like data-races, deadlocks, etc [32]. These bugs may cause the program to end up with abnormal results or unexpected hangs. On the other hand, bugs that appear under specific inputs and interleavings may lead to *concurrency-vulnerabilities* [5, 30], resulting in memory corruptions, information leakage, etc.

There exist a line of works on detecting bugs and vulnerabilities inmultithreaded programs. Static concurrency-bug predictors [2, 40, 45, 50] aim to approximate the runtime behaviors of a program without actual concurrent execution. However, they typically serve as a complementary solution due to the high percentage of false alarms [19]. Dynamic detectors detect concurrency-violations by reasoning memory read/write and synchronization events in a particular execution trace [5, 12, 21, 41, 42, 49, 58]. Several techniques like ThreadSanitizer (a.k.a., TSan) [42] and Helgrind [49] have been widely used in practice. However, these approaches by themselves do not automatically *generate new test inputs* to exercise different paths in multithreaded programs.

Meanwhile, grey-box fuzzing is effective in generating test inputs to expose vulnerabilities [34, 36]. It is reported that grey-box fuzzers (GBFs) such as AFL [63] and libFuzzer [31] have detected more than 16,000 vulnerabilities in hundreds of real-world software projects [16, 31, 63].

Despite the great success of GBFs in detecting vulnerabilities, *there are few efforts on fuzzing user-space multithreaded programs*. General-purpose GBFs usually cannot explore thread-interleaving introduced execution states due to their unawareness of multithreading. Therefore, they cannot effectively detect concurrency-vulnerabilities inherently buried in sophisticated program flows [30]. In a discussion in 2015 [64], the author of AFL, Michal Zalewski, even suggests that "*it's generally better to have a single thread*". In fact, due

[*]Corresponding Author.

to the difficulty and inefficiency, the fuzzing driver programs in Google's continuous fuzzing platform OSS-fuzz are all tested in *single-threaded* mode [15]. Also, by matching unions of keyword patterns "race*", "concurren*" and "thread*" in the MITRE CVE database [48], we found that only 202 CVE records are relevant to concurrency-vulnerabilities out of the 70438 assigned CVE IDs ranging from CVE-2014-* to CVE-2018-*. In particular, we observed that, theoretically, at most 4 CVE records could be detected by grey-box fuzzers that work on user-space programs.

As a result, there are no practical fuzzing techniques to test *input-dependent user-space multithreaded programs* and detect bugs or vulnerabilities inside them. To this end, we present a dedicated grey-box fuzzing technique, MUZZ, to reveal bugs by exercising *input*-dependent and *interleaving*-dependent paths. We categorize the targeted *multithreading-relevant bugs* into two major groups:

- **concurrency-vulnerabilities** (V_m): they correspond to memory corruption vulnerabilities that occur in a multi-threading context. These vulnerabilities can be detected during the *fuzzing* phase.
- **concurrency-bugs** (B_m): they correspond to the bugs like data-races, atomicity-violations, deadlocks, etc. We detect them by *replaying* the seeds generated by MUZZ with state-of-the-art concurrency-bug detectors such as TSan.

Note that B_m may not be revealed during *fuzzing* since they do not necessarily result in memory corruption crashes. In the remaining sections, when referring to *multithreading-relevant bugs*, we always mean the combination of concurrency-bugs and concurrency-vulnerabilities, i.e., $V_m \cup B_m$.

We summarize the contributions of our work as follows:
1) We develop three novel thread-aware instrumentations for grey-box fuzzing that can distinguish the execution states caused by thread-interleavings.
2) We optimize seed selection and execution strategies based on the runtime feedback provided by the instrumentations, which help generate more effective seeds concerning the multithreading context.
3) We integrate these analyses into MUZZ for an effective bug hunting in multithreaded programs. Experiments on 12 real-world programs show that MUZZ outperforms other fuzzers like AFL and MOPT in detecting concurrency-vulnerabilities and revealing concurrency-bugs.
4) MUZZ detected 8 new concurrency-vulnerabilities and 19 new concurrency-bugs, with 4 CVE IDs assigned. Considering the small portion of concurrency-vulnerabilities recorded in the CVE database, the results are promising.

2 Background and Motivation

2.1 Grey-box Fuzzing Workflow

Algorithm 1 presents the typical workflow of a grey-box fuzzer [3, 34, 63]. Given a target program $\mathbb{P}_o$ and the input

Algorithm 1: Grey-box Fuzzing Workflow

```
input  : program $\mathbb{P}_o$, initial seed queue $Q_S$
output : final seed queue $Q_S$, vulnerable seed files $T_C$
1  $\mathbb{P}_f \leftarrow$ instrument($\mathbb{P}_o$) ;                  // instrumentation
2  $T_C \leftarrow \emptyset$;
3  while True do
4      t ← select_next_seed($Q_S$) ;                  // seed selection
5      $\mathbb{M} \leftarrow$ get_mutation_chance($\mathbb{P}_f$, t) ;   // seed scheduling
6      for $i \in 1 \ldots \mathbb{M}$ do
7          $t' \leftarrow$ mutated_input(t) ;                // seed mutation
8          res ← run($\mathbb{P}_f$, t', $\mathbb{N}_c$);            // repeated execution
9          if is_crash(res) then                     // seed triaging
10             $T_C \leftarrow T_C \cup \{t'\}$ ; // report vulnerable seeds
11         else if cov_new_trace(t', res) then
12             $Q_S \leftarrow Q_S \oplus t'$ ; // preserve "effective" seeds
```

seeds Q_S, a GBF first utilizes instrumentation to track the coverage information in $\mathbb{P}_o$. Then it enters the fuzzing loop: 1) *Seed selection* decides which seed to be selected next; 2) *Seed scheduling* decides how many mutations $\mathbb{M}$ will be applied on the selected seed t; 3) *Seed mutation* applies mutations on seed t to generate a new seed t'; 4) During *repeated execution*, for each new seed t', the fuzzer executes against it $\mathbb{N}_c$ times to get its execution statistics; 5) *Seed triaging* evaluates t' based on the statistics and the coverage feedback from instrumentation, to determine whether the seed leads to a vulnerability, or whether it is "effective" and should be preserved in the seed queue for subsequent fuzzing. Here, steps 3), 4), 5) are continuously processed $\mathbb{M}$ times. Notably, $\mathbb{N}_c$ times of repeated executions are necessary since a GBF needs to collect statistics such as average execution time for t', which will be used to calculate mutation times $\mathbb{M}$ for seed scheduling in the next iteration. In essence, the effectiveness of grey-box fuzzing relies on the feedback collected from the instrumentation. Specifically, the result of *cov_new_trace* (line 11) is determined by the *coverage* feedback.

2.2 The Challenge in Fuzzing Multithreaded Programs and Our Solution

Figure 1 is an abstracted multithreaded program that accepts a certain input file and distributes computing jobs to threads. Practically it may behave like compressors/decompressors (e.g., *lbzip2*, *pbzip2*), image processors (e.g., *ImageMagick*, *GraphicsMagick*), encoders/decoders (e.g., *WebM*, *libvpx*), etc. After reading the input content `buf`, it does an initial validity check inside the function `check`. It exits immediately if the buffer does not satisfy certain properties. The multithreading context starts from function `compute` (via `pthread_create` at lines 24-25). It contains shared variables `s_var` (passed from `main`) and `g_var` (global variables), as well as the mutex primitive `m` to exclusively read/write shared variables (via `pthread_mutex_lock` and `pthread_mutex_unlock`).

```
int g_var = -1;
void modify(int *pv) { *pv -= 2;}                    // ⑨

void check(char * buf) {
  if (is_invalid(buf)) { exit(1); }
  else { modify((int*)buf); }
}

char* compute(void *s_var) {
  g_var += 1;                                        // ①
  g_var *= 2;                                        // ②
  if ((int*)s_var[0]<0)                              // ③
     modify((int*)s_var);                            // ④
  pthread_mutex_lock(&m);                            // ⑤
  modify(&g_var);                                    // ⑥
  pthread_mutex_unlock(&m);                          // ⑦
  return (char*)s_var;                               // ⑧
}

int main(int argc, char **argv) {
  char * buf = read_file_content(argv[1]);
  check(buf);
  pthread_t T1, T2;
  pthread_create(T1,NULL,compute,buf);
  pthread_create(T2,NULL,compute,buf+128);
  ......
}
```

Figure 1: Code segments abstracted from real-world programs. The shadow lines denote "suspicious interleaving scope" introduced in §4.1.

With different inputs, the program may execute different segments. For example, based on the condition of statement ③, which is purely dependent on the input content (i.e., different results of `buf` provided by seed files), it may or may not execute ④. Therefore, different seed files need to be generated to exercise different paths in multithreading context — in fact, this is the starting point that we use fuzzing to generate seed files to test multithreaded programs.

Meanwhile, in the presence of thread-interleavings, `g_var` (initialized with -1) may also have different values. Let us focus on different seeds' executions at two statements: ①:"`g_var+=1`", and ②: "`g_var*=2`". Suppose there are two threads: T1, T2; and T1:① is executed first. Then there are at least three interleavings:

i) T1:①→T2:①→T2:②→T1:② `g_var=4`
ii) T1:①→T2:①→T1:②→T2:② `g_var=4`
iii) T1:①→T1:②→T2:①→T2:② `g_var=2`

After the second ② is executed, the values of `g_var` may be different (4 and 2, respectively). Worse still, since neither ① nor ② is an *atomic* operation in the representation of the actual program binary, many more interleavings can be observed and `g_var` will be assigned to other values.

The challenge. To reveal multithreading-relevant bugs, a GBF needs to generate diverse seeds that execute different paths in multithreading context (e.g., paths inside `compute`). However, *existing GBFs even have difficulties in generating seeds to reach multithreading segments*. For example, if `check` is complicated enough, most of the seeds may fail the check and exit before entering `compute` — this is quite common due to the low quality of fuzzer-generated seeds [34, 61]. Meanwhile, *even if a seed indeed executes multithreading code, it may still fail to satisfy certain preconditions to reach the problematic context.* For example, suppose `modify` contains a vulnerability that can only be triggered when `g_var` is 2. If the fuzzer has occasionally generated a seed that executes `compute` and the condition of ③ is true, with no awareness of thread-interleavings, it will not distinguish different schedules between i), ii) and iii). As a result, subsequent mutations on this seed will miss important feedback regarding `g_var`, making it difficult to generate seeds that trigger the vulnerability.

To summarize, the challenge of fuzzing multithreaded programs is, existing GBFs have difficulties in generating seeds that execute multithreading context and keep thread-interleaving execution states.

Our solution. *We provide fine-grained* thread-aware feedback *for seed files that execute multithreading context and distinguish more such execution states.* According to §2.1, the preservation of seeds is based on the feedback; then we can expect that the fuzzer will preserve more distinct seeds that execute multithreading code segments in the seed queue. This means that the multithreading-relevant seeds are *implicitly prioritized.* Since these seeds have already passed the validity checking, the overall quality of the generated seeds is higher. The "Matthew Effect" helps keep the quality of seed generations for subsequent fuzzing. Essentially, this provides a biased coverage feedback on multithreading code segments (more explanations on this are available in §5.3.

Now let us investigate what instrumentations can be improved to existing fuzzers for *thread-aware feedback.*

2.3 Thread-aware Feedback Improvements

2.3.1 Feedback to Track Thread-interleavings and Thread-context

The state-of-the-art GBFs, such as AFL, instrument the entry instruction of each basicblock *evenly* as the basicblock's *deputy*. We refer to this selection strategy over deputy instructions as AFL-Ins. AFL-Ins provides coverage feedback during the dynamic fuzzing phase to explore more paths. During repeated execution (line 8 in Algorithm 1), AFL labels a value to each *transition* that connects the deputies of two consecutively executed basicblocks [63]. By maintaining a set of *transitions* for queued seeds, AFL-Ins *tracks* the "coverage" of the target program. *cov_new_trace* (line 11 in Algorithm 1) checks whether a transition indicates a new path/state.

Figure 2b depicts the transitions upon executing the functions `compute` and `modify` in Figure 1. For brevity, we use *source code* to illustrate the problem and use *statements* to represent *instructions* in assembly or LLVM IR [28].

AFL-Ins works perfectly on single-threaded programs: the

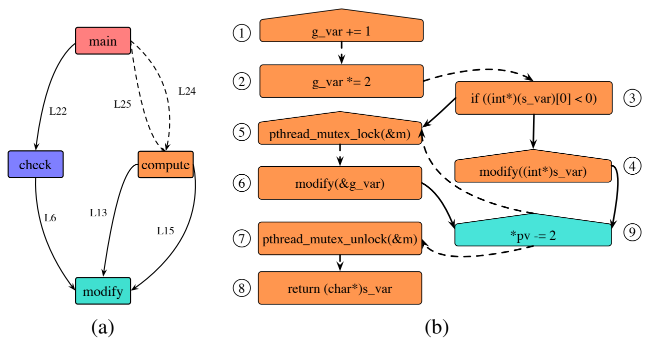

Figure 2: (a) thread-aware callgraph of Figure 1; (b) its edge transitions across `compute` and `modify`. In (b), the arrows denote the transitions between statements. The pentagons denote basicblocks' entry statements; the other statements are represented by rectangles. Their colors are consistent with function nodes in (a). Since AFL-Ins only tracks branches' entry statements, only branching edges (③→④ and ③→⑤) and function call edges (④→⑨ and ⑥→⑨) are recorded — these transitions are marked as solid arrows.

kept transitions can reflect both branching conditions (e.g., ③→④ and ③→⑤) and function calls (e.g., ④→⑨ and ⑥→⑨). However, AFL-Ins cannot capture these differences among schedules i), ii) and iii) (c.f. §2.2). In fact, it can only observe there is a transition ①→①; thus it will not prioritize this path for subsequent mutations, compared to other paths that do not even execute `compute`. The root cause of this defect lies in that AFL only tracks entry statements of basicblocks *evenly*, and does not record thread identities. Therefore, we can add *more deputy instructions* within multithreading-relevant basicblocks to provide more interleaving feedback, and add *thread-context information* to distinguish different threads.

2.3.2 Schedule-intervention Across Executions

During a GBF's repeated execution procedure (line 8 in Algorithm 1), a seed may exhibit *non-deterministic behaviors*: it executes different paths of the target program across executions due to randomness. In this scenario, AFL (and other GBFs) will execute against such a seed more times than a seed with deterministic behaviors [63]. For the non-deterministic behaviors caused by scheduling-interleaving in multithreaded programs, since the execution is *continuously* repeated $\mathbb{N}_c$ times, the system level environment (e.g., CPU usage, memory consumption, I/O status) is prone to be similar [23, 26]. This will decrease the diversities of schedules, and consequently reduce the overall effectiveness. For example, during a repeated execution with $\mathbb{N}_c = 40$, schedules i) and iii) might occur 10 and 30 times respectively, while schedule ii) do not occur at all; in this scenario, the execution states corresponding to ii) will not be observed by the fuzzer. Ideally, we would like the fuzzer to observe as many distinct interleavings as possible during repeated execution since that marks the potential states a seed can exercise. In the case of statements ① and ②, we hope schedules i), ii), iii) can all occur. Therefore, it is favorable to provide *schedule interventions* to diversify the actual schedules.

3 System Overview

Figure 3 depicts the system overview of MUZZ. It contains four major components: Ⓐ static thread-aware analysis guided instrumentations, Ⓑ dynamic fuzzing, Ⓒ vulnerability analysis, Ⓓ concurrency-bug revealing.

During Ⓐ:*instrumentation* (§4), for a multithreaded program $\mathbb{P}_o$, MUZZ firstly computes thread-aware interprocedural control flow graph (ICFG) and the code segments that are likely to interleave with others during execution [11, 45], namely *suspicious interleaving scope*, in §4.1. Based on these results, it performs three instrumentations inspired by §2.3.

1) *Coverage-oriented instrumentation* (§4.2) is one kind of stratified instrumentation that assigns more deputies to suspicious interleaving scope. It is the major instrumentation to track thread-interleaving induced coverage.
2) *Thread-context instrumentation* (§4.3) is a type of lightweight instrumentation that distinguishes different thread identities by tracking the context of threading functions for thread-forks, locks, unlocks, joins, etc.
3) *Schedule-intervention instrumentation* (§4.4) is a type of lightweight instrumentation at the entry of a thread-fork routine that dynamically adjusts each thread's priority. This complementary instrumentation aims to diversify interleavings by intervening in the thread schedules.

During Ⓑ:*dynamic fuzzing* (§5), MUZZ optimizes *seed selection* and *repeated execution* to generate more multithreading-relevant seeds. For seed selection (§5.1), in addition to the new coverage information provided by coverage-oriented instrumentation, MUZZ also prioritizes those seeds that cover new thread-context based on the feedback provided by thread-context instrumentation. For repeated execution (§5.2), owing to the schedule-intervention instrumentation, MUZZ adjusts the repeating times $\mathbb{N}_c$, to maximize the benefit of repetitions and track the interleaved execution states.

Ⓒ:*Vulnerability analysis* is applied to the crashing seeds found by dynamic fuzzing, which reveals vulnerabilities (including V_m). Ⓓ:*concurrency-bug revealing* component reveals B_m with the help of concurrency-bug detectors (e.g., TSan [42], Helgrind [49]). These two components will be explained in the evaluation section (§6).

4 Static Analysis Guided Instrumentation

This component includes the thread-aware static analysis and the instrumentations based on it.

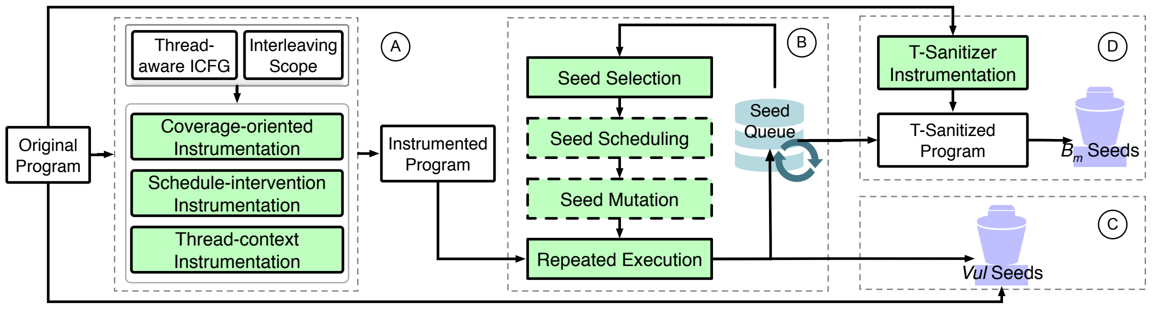

Figure 3: Overview of MUZZ. Inputs are the original program and initial seeds (in seed queue); outputs are the seeds with vulnerabilities or concurrency-bugs. It contains four components. Ⓐ (left area) does static analysis and applies thread-aware instrumentations; Ⓑ (center area) contains the flows that proceed with dynamic fuzzing (seed scheduling and seed mutation [34] are the same as typical GBF flows, thus are marked dashed); Ⓒ (right-bottom) denotes the vulnerability analysis applied on vulnerable seeds; and Ⓓ (right-top) is the replaying component used to reveal concurrency-bugs from the seed queue.

4.1 Thread-aware Static Analysis

The static analysis aims to provide lightweight thread-aware information for instrumentation and runtime feedback.

4.1.1 Thread-aware ICFG Generation

We firstly apply an inclusion-based pointer analysis [1] on the target program. The points-to results are used to resolve the def-use flow of thread-sharing variables and indirect calls to reconstruct the ICFG. By taking into account the semantics of threading APIs (e.g., POSIX standard Pthread, the OpenMP library), we get an ICFG that is aware of the following multithreading information:

1) *TFork* is the set of program sites that call thread-fork functions. This includes the explicit call to *pthread_create*, the *std::thread* constructor that internally uses *pthread_create*, or the "parallel pragma" in OpenMP. The *called* functions, denoted as F_{fork}, are extracted from the semantics of these forking sites.
2) *TJoin* contains call sites for functions that mark the end of a multithreading context. It includes the call sites of the pthread APIs such as *pthread_join*, *pthread_exit*, etc.
3) *TLock* is the set of sites that call thread-lock functions such as *pthread_mutex_lock*, *omp_set_lock*, etc.
4) *TUnLock* is the set of sites that call thread-unlock functions like *pthread_mutex_unlock*, *omp_unset_lock*, etc.
5) *TShareVar* is the set of variables shared among different threads. This includes global variables and those variables that are passed from thread-fork sites (e.g., *TFork*).

4.1.2 Suspicious Interleaving Scope Extraction

Given a program that may run simultaneously with multiple threads, we hope the instrumentation to collect execution states to reflect the interleavings. However, instrumentation introduces considerable overhead to the original program, especially when it is applied intensively throughout the whole program. Fortunately, with the static information provided by the thread-aware ICFG, we know that thread-interleavings may only happen on some specific program statements; therefore, the instrumentation can stress these statements. We hereby use L_m to denote the set of these statements and term it as *suspicious interleaving scope*. L_m is determined according to the following three conditions.

C1 The statements should be executed after one of *TFork*, while *TJoin* is not encountered yet.
C2 The statements can only be executed before the invocation of *TLock* and after the invocation of *TUnLock*.
C3 The statements should read or write at least one of the shared variables by different threads.

C1 excludes the statements irrelevant to multithreading. These statements can be prologue code that does the validity check (e.g., `check` in Figure 1), or the epilogue that post-processes the inputs or deals with error handlings. **C2** prevents the statements that are protected by certain locks from being put into L_m. **C3** is necessary since the interleavings will not affect the shared states if the segment involves no shared variables. This condition is determined by observing whether the investigated statement contains a variable data dependent on *TShareVar* (based on pointer analysis). We provide a separate preprocessing procedure to exclude cases where there are only *read* operations on shared variables.

Note that L_m is used to emphasize multithreading-relevant paths via instrumentations for state exploration during fuzzing. Therefore the conditions are different from the constraints required by static models (e.g., may-happen-in-parallel [11, 45]) or dynamic concurrency-bug detection algorithms (e.g., happens-before [12] or lockset [41]).

In Figure 1, according to the call `pthread_create` at Lines 24 and 25, $F_{fork} = \{$`compute`$\}$. MUZZ then gets all the functions that may be called by functions inside F_{fork}, i.e., $\{$`modify`,`compute`$\}$ and according to **C1** the scope L_m comes from Lines 1, 2, 10−17. Inside these functions, we check the statements that are outside `pthread_mutex_lock` and `pthread_mutex_unlock` based on C2: Line 15 should be excluded from L_m. According to C3, we exclude the statements that do not access or modify the shared variables `g_var`, `s_var`, which means Lines 14 and 16 should also be excluded. In the end, the scope is determined as $L_m = \{1, 2, 10, 11, 12, 13, 17\}$. Note that although `modify` can be called in a single-threading site inside `check` (Line 6), we still conservatively include it in L_m. The reason is that it *might be* called within multithreading contexts (Line 13 and Line 15) — `modify` is protected by mutex `m` at Line 15 while unprotected at Line 13. It is worth noting that line 15, although protected by `m`, may still happen-in-parallel [11,45] with lines 10 and 11. However, since lines 10 and 11 have already been put in L_m, we consider it sufficient to help provide more feedback to track thread-interleavings, with line 15 excluded from L_m.

Overall, the static analysis is lightweight. For example, the pointer analysis is flow- and context-insensitive; extraction of thread-aware results such as F_{fork} (in C1) and *TShareVar* (in C3) are over-approximated in that the statically calculated sets may be larger than the actual sets; C2 may aggressively exclude several statements that involve interleavings. The benefit, however, is that it makes our analysis scalable to large-scale real-world programs.

4.2 Coverage-oriented Instrumentation

With the knowledge of L_m, we can instrument more deputy *instructions* (corresponding to statements in source code) inside the scope than the others, for exploring new transitions. However, it is still costly to instrument on *each instruction* inside L_m since this may significantly reduce the overall execution speed of the target programs. It is also *unnecessary* to do so — although theoretically, interleavings may happen everywhere inside L_m, many interleavings are not important because they do not change the values of shared variables in practice. This means that we can skip some instructions for instrumentation, or equivalently instrument them *with a probability*. We still instrument, despite less, on segments outside L_m for *exploration* purposes [34]. For example, in Figure 1, we apply instrumentation on `check`, just in case the initial seeds are all rejected by the validity check and no intermediate feedback are available at all, making the executions extremely difficult to even enter `compute`. Similarly, we can also selectively instrument some instructions outside L_m.

4.2.1 Instrumentation Probability Calculation

The goal of calculating instrumentation probabilities is to strike a balance between execution overhead and feedback effectiveness by investigating code segments' complexity of the target programs. First of all, MUZZ calculates a *base instrumentation probability* according to *cyclomatic complexity* [35], based on the fact that bugs or vulnerabilities usually come from functions with higher cyclomatic complexity [9,43]. For each function f, we calculate the complexity value: $M_c(f) = E(f) - N(f) + 2$ where $N(f)$ is the number of nodes (basicblocks) and $E(f)$ is the number of edges in the function's control flow graph. Intuitively, this value determines the complexity of the function across its basicblocks. As 10 is considered to be the preferred upper bound of M_c [35], we determine the base probability as:

$$P_e(f) = \min\left\{\frac{E(f) - N(f) + 2}{10},\ 1.0\right\} \tag{1}$$

We use P_s as the probability to selectively instrument on the entry instruction of a basicblock that is *entirely outside* suspicious interleaving scope, i.e., none of the instructions inside the basicblock belong to L_m. Here, P_s is calculated as:

$$P_s(f) = \min\left\{P_e(f),\ P_{s0}\right\} \tag{2}$$

where $0 < P_{s0} < 1$. Empirically, MUZZ sets $P_{s0} = 0.5$.

Further, for each basicblock b inside the given function f, we calculate the total number of instructions $N(b)$, and the total number of memory operation instructions $N_{\mathbf{m}}(b)$ (e.g., load/store, *memcpy*, *free*). Then for the instructions within L_m, the instrumentation probability is calculated as:

$$P_m(f, b) = \min\left\{P_e(f) \cdot \frac{N_{\mathbf{m}}(b)}{N(b)},\ P_{m0}\right\} \tag{3}$$

where P_{m0} is a factor satisfying $0 < P_{m0} < 1$ and defaults to 0.33. The rationale of $\frac{N_{\mathbf{m}}(b)}{N(b)}$ is that vulnerabilities usually result from memory operation instructions [34], and executions on more such operations deserve more attention.

4.2.2 Instrumentation Algorithm

The coverage-oriented instrumentation algorithm is described in Algorithm 2. It traverses functions in the target program P. For each basicblock b in function f, MUZZ firstly gets the intersection of the instructions inside both b and L_m. If this intersection $L_m(b)$ is empty, it instruments the entry instruction of b with a probability of $P_s(f)$. Otherwise, 1) for the entry instruction in b, MUZZ always instruments it (i.e., with probability 1.0); 2) for the other instructions, if they are inside L_m, MUZZ instruments them with a probability of $P_m(f, b)$. We will refer to our selection strategy over deputy instructions as M-Ins. As a comparison, AFL-Ins always instruments evenly at the entry instructions of all the basicblocks.

For the example in Figure 1, since the lines 21-25 and line 5 are out of L_m, we can expect M-Ins to instrument fewer entry statements on their corresponding basicblocks. Meanwhile, for the statements inside L_m, M-Ins may instrument other statements besides the entry statements. For example, ① is the entry statement thus it must be instrumented; statement ② may also be instrumented (with a probability) — if so, transition ①→② can be tracked.

Algorithm 2: Coverage-oriented Instrumentation

```
input  : target program $P$, and suspicious interleaving scope $L_m$
output : program $P$ instrumented with M-Ins deputies
for $f \in P$ do
    for $b \in f$ do
        $L_m(b) = L_m \cap b$;
        if $L_m(b) \, != \emptyset$ then
            for $i \in b$ do
                if $is\_entry\_instr(i, b)$ then
                    P ← instrument_cov(P, $i$, 1.0);
                else if $i \in L_m$ then
                    P ← instrument_cov(P, $i$, $P_m(f,b)$);
        else
            for $b \in f$ do
                $i$ = get_entry_instr($b$);
                P ← instrument_cov(P, $i$, $P_s(f)$);
```

4.3 Threading-context Instrumentation

We apply threading-context instrumentation to distinguish thread identities for additional feedback. This complements *coverage-oriented instrumentation* since the latter is unaware of thread IDs. The context is collected at the call sites of $F_{ctx} = \{TLock, TUnLock, TJoin\}$, each of which has the form $TC = \langle Loc, N_{ctx} \rangle$, where Loc is the labeling value of deputy instruction executed before this call site, and N_{ctx} is obtained by getting the value of the key identified by current thread ID from the "thread ID map" collected by the instrumented function F_S (to be explained in §4.4). Given an item F in F_{ctx}, we keep a sequence of context $\langle TC_1(F), \ldots, TC_n(F) \rangle, F \in F_{ctx}$. At the end of each execution, we calculate a hash value $H(F)$ for item F. The tuple $S_{ctx} = \langle H(TLock), H(TUnLock), H(TJoin) \rangle$ is a *context-signature* that determines the overall thread-context of a specific execution. Essentially, this is a *sampling* on threading-relevant APIs to track the thread-context of a specific execution. As we shall see in §5.1, the occurrence of S_{ctx} determines the results of *cov_new_mt_ctx* during seed selection.

In Figure 1, each time when `pthread_mutex_lock`$\in$ *TLock* is called, MUZZ collects the deputy instruction prior to the corresponding call site (e.g., ③) and the thread ID label (e.g., T1) to form the tuple (e.g., $\langle ③, T1 \rangle$); these tuples form a sequence for *TLock*, and a hash value $H(TLock)$ will be calculated eventually. Similar calculations are applied for `pthread_mutex_unlock` and `pthread_join`.

4.4 Schedule-intervention Instrumentation

When a user-space program does not specify any scheduling policy or priority, the operating system determines the actual schedule dynamically [23, 26]. Schedule-intervention instrumentation aims to diversify the thread-interleavings to collaborate with coverage-oriented and thread-context instrumentations. This instrumentation should be general enough to work for different multithreaded programs and extremely lightweight to keep runtime overhead minimal.

POSIX compliant systems such as Linux, FreeBSD usually provide APIs to control the low-level process or thread schedules [23, 26]. In order to intervene in the interleavings during the execution of the multithreading segments, we resort to the POSIX API *pthread_setschedparam* to adjust the thread priorities with an instrumented function named F_S that will be invoked during fuzzing. This function does two tasks:

a) During repeated execution (§5.2), whenever the thread calls F_S, it updates the scheduling policy to *SCHED_RR*, and assigns a ranged random value to its priority. This value is *uniformly distributed random* and diversifies the actual schedules across different threads. With this intervention, we try to approximate the goal in §2.3.2.
b) For each newly mutated seed file, it calls *pthread_self* in the entry of F_{fork} to collect the thread IDs. It has two purposes: 1) it informs the fuzzer that the current seed is multithreading-relevant; 2) based on the invocation order of F_S, each thread can be associated with a unique ID N_{ctx} starting from $1, 2, \ldots$, which composes "thread ID map" and calculates thread-context in §4.3.

Algorithm 3: *select_next_seed* Strategy

```
input  : seed queue $Q_S$, seed $t$ at queue front
output : whether $t$ will be selected in this round
if $has\_new\_mt\_ctx(Q_S)$ or $has\_new\_trace(Q_S)$ then
    if $cov\_new\_mt\_ctx(t)$ then
        return $true$;
    else if $cov\_new\_trace(t)$ then
        return $select\_with\_prob(P_{ynt})$;
    else
        return $select\_with\_prob(P_{ynn})$;
else
    return $select\_with\_prob(P_{nnn})$;
```

5 Dynamic Fuzzing

The dynamic fuzzing loop follows the workflow of a typical GBF described in Algorithm 1. To improve the feedback on multithreading context, we optimize seed selection (§5.1) and repeated execution (§5.2) for fuzzing multithreaded programs, based on the aforementioned instrumentations.

5.1 Seed Selection

Seed selection decides which seeds to be mutated next. In practice, this problem is reduced to: when traversing seed queue Q_S, whether the seed t at the queue front will be selected for mutation. Algorithm 3 depicts our solution. The intuition is that we prioritize those seeds *with new (normal) coverage* or *covering new thread-context*.

In addition to following AFL's strategy by using *has_new_trace(Q_S)* to check whether there exists a seed, s, in Q_S that covers a new transition (i.e., *cov_new_trace(s)*==true), MUZZ also uses *has_new_mt_ctx(Q_S)* to check whether there exists a seed in Q_S with a new thread-context (S_{ctx}). If either is satisfied, it means there exist some "interesting seeds" in the queue. Specifically, if the current seed t covers a new thread-context, the algorithm directly returns true. If it covers a new trace, it has a probability of P_{ynt} to be selected; otherwise, the probability is P_{ynn}. On the contrary, if no seeds in Q_S are interesting, the algorithm selects t with a probability of P_{nnn}. Analogous to AFL's seed selection strategy [63], MUZZ sets $P_{ynt} = 0.95$, $P_{ynn} = 0.01$, $P_{nnn} = 0.15$.

As to the implementation of *cov_new_mt_ctx(t)*, we track the thread-context of calling a multithreading API in $F_{ctx} = \{TJoin, TLock, TUnLock\}$ (c.f. §4.3) and check whether the context-signature S_{ctx} has been met before — when S_{ctx} is new, *cov_new_mt_ctx(t)*=true; otherwise, *cov_new_mt_ctx(t)*=false. Note that *cov_new_trace(t)*==true does not imply *cov_new_mt_ctx(t)==true*. The reason is that (1) we cannot instrument inside the body of threading API functions (as they are "external functions") inside F_{ctx}, hence *cov_new_trace* cannot track the transitions; (2) *cov_new_mt_ctx* also facilitates the thread IDs that *cov_new_trace* is unaware of.

5.2 Repeated Execution

Multithreaded programs introduce non-deterministic behaviors when different interleavings are involved. As mentioned in §2.3.2, for a seed with non-deterministic behaviors, a GBF typically repeats the execution on the target program against the seed for more times. With the help of F_S (c.f. §4.4), we are able to tell whether or not the exhibited non-deterministic behaviors result from thread-interleavings. In fact, since we focus on multithreading only, based on the thread-fork information kept by F_S, the fuzzer can distinguish the seeds with non-deterministic behaviors purely by checking whether the executions exercise multithreading context. Further, if previous executions on a seed induce more distinct values of S_{ctx} (the number of these values for a provided seed t is denoted as C_m (t)), we know that there must exist more thread-interleavings. To determine the repeating times $\mathbb{N}_c$ applied on t, we rely on $C_m(t)$. In AFL, the repeating times on t is:

$$\mathbb{N}_c(t) = N_0 + N_v \cdot B_v, \quad B_v \in \{0,1\} \tag{4}$$

where N_0 is the initial repeating times, N_v is a constant as the "bonus" times for non-deterministic runs. B_v=0 if none of the N_0 executions exhibit non-deterministic behaviors; otherwise B_v=1. We augment this to fit for multithreading setting.

$$\mathbb{N}_c(t) = N_0 + \mathtt{min}\{N_v, N_0 \cdot C_m(t)\} \tag{5}$$

In both AFL and MUZZ, $N_0 = 8$, $N_v = 32$. For all the $\mathbb{N}_c$ executions, we track their execution traces and count how many different states it exhibits. The rationale of adjusting $\mathbb{N}_c$ is that, in real-world programs the possibilities of thread-interleavings can vary greatly for different seeds. For example, a seed may exhibit non-deterministic behaviors when executing `compute` in Figure 1 (e.g., races in `g_var`), but it exits soon after failing an extra check inside `compute` (typically, exit code >0). For sure, it will exhibit fewer non-deterministic behaviors than a seed that is concurrently processed and the program exits normally (typically, exit code =0).

5.3 Complementary Explanations

Here we provide some explanations to show why MUZZ's static and dynamic thread-aware strategies help to improve the overall fuzzing effectiveness.

1) Mutations on multithreading-relevant seeds are more valuable for seed mutation/generation. Multithreading-relevant seeds themselves have already passed validity checks of the target program. Compared to a seed that cannot even enter the thread-fork routines, it is usually much easier to generate a multithreading-relevant seed mutant from an existing multithreading-relevant seed. This is because the mutation operations (e.g., bitwise/bytewise flips, arithmetic adds/subs) in grey-box fuzzers are rather random and it is rather difficult to turn an invalid seed to be valid. Therefore, from the *mutation's perspective*, we prefer multithreading-relevant seeds to be mutated.

2) MUZZ can distinguish more multithreading-relevant states. For example, in Figure 1, it can distinguish transitions ①→①→②→② and ①→②→①→②. Then when *two different seeds* exercise the two transitions, MUZZ is able to preserve both seeds. However, other GBFs such as AFL cannot observe the difference. Conversely, when we provide less feedback for seeds that do not involve multithreading, MUZZ can distinguish less of these states and put less multithreading-*irrelevant* seeds in the seed queue.

3) Large portions of multithreading-relevant seeds in the seed queue benefit subsequent mutations. Suppose at some time of fuzzing, both MUZZ and AFL preserve 10 seeds (N_{all} =10), and MUZZ keeps 8 multithreading-relevant seeds (N_{mt} =8) while AFL keeps 6 (N_{mt} =6). Obviously, the probability of picking MUZZ generated multithreading-relevant seeds (80%) is higher than AFL's (60%). After this iteration of mutation, more seed mutants in MUZZ are likely multithreading-relevant. The differences of seed quality (w.r.t. relevance to multithreading) in the seed queue can be amplified with more mutation iterations. For example, finally MUZZ may keep 18 multithreading-relevant seeds (N_{mt} =18), and 10 other seeds, making N_{all} =28 and $\frac{N_{mt}}{N_{all}} = \frac{18}{28} = 64.3\%$; while AFL keeps 12 multithreading-relevant seeds (N_{mt} =12) and 14 other seeds, making N_{all} =26 and $\frac{N_{mt}}{N_{all}} = \frac{12}{26} = 46.2\%$.

Properties **1)**, **2)** and **3)** collaboratively affects the fuzzing effectiveness in a "closed- loop". Eventually, both N_{mt} and

$\frac{N_{mt}}{N_{all}}$ in MUZZ are likely to be bigger than those in AFL. Owing to more multithreading-relevant seeds in the queue and property **1)**, we can expect that:

a) concurrency-vulnerabilities are more likely to be detected with the new proof-of-crash files mutated from multithreading-relevant files from the seed queue.
b) concurrency-bugs are more likely to be revealed with the (seemingly normal) files in the seed queue that violate certain concurrency conditions.

Providing more feedback for multithreading-relevant segments essentially provides a biased coverage criterion to specialize fuzzing on multithreaded programs. Other specialization techniques, such as the *context-sensitive instrumentation* used by Angora [7], or the *typestate-guided instrumentation* in UAFL [52], provide similar solutions and achieve inspiring results. The novelty of MUZZ lies in that we facilitate the multithreading-specific features as feedback to innovatively improve the seed generation quality. It is worth noting that our solution only needs lightweight thread-aware analyses rather than deep knowledge of multithreading/concurrency; thus, it can scale to real-world software.

6 Evaluation

We implemented MUZZ upon SVF [46], AFL [63] , and ClusterFuzz [16]. The thread-aware ICFG construction leverages SVF's inter-procedural value-flow analysis. The instrumentation and dynamic fuzzing strategies lay inside AFL's LLVM module. The vulnerability analysis and concurrency-bug replaying components rely on ClusterFuzz's crash analysis module. We archive our supporting materials at `https://sites.google.com/view/mtfuzz`. The archive includes initial seeds for fuzzing, the detected concurrency-vulnerabilities and concurrency-bugs, the implementation details, and other findings during evaluation.

Our evaluation targets the following questions:

RQ1 Can MUZZ generate more effective seeds that execute multithreading-relevant program states?
RQ2 What is the capability of MUZZ in detecting concurrency-vulnerabilities (V_m)?
RQ3 What is the effect of using MUZZ generated seeds to reveal concurrency-bugs (B_m) with bug detectors?

6.1 Evaluation Setup

6.1.1 Settings of the grey-box fuzzers

We use the following fuzzers during evaluation.

1) **MUZZ** is our full-fledged fuzzer that applies all the thread-aware strategies in §4 and §5.
2) **MAFL** is a variant of **MUZZ**. It differs from MUZZ only on the coverage-oriented instrumentation — MAFL uses AFL-Ins while MUZZ uses M-Ins. We compare MAFL with MUZZ to demonstrate the effectiveness of M-Ins, and compare MAFL with AFL to stress other strategies.
3) **AFL** is by far the most widely-used GBF that facilitates general-purpose AFL-Ins instrumentation and fuzzing strategies. It serves as the baseline fuzzer.
4) **MOPT** [33] is the recently proposed general-purpose fuzzer that leverages adaptive mutations to increase the overall fuzzing efficiency. It is claimed to be able to detect 170% more vulnerabilities than AFL in fuzzing (single-thread) programs.

6.1.2 Statistics of the evaluation dataset

The dataset for evaluation consists of the following projects.

1) Parallel compression/decompression utilities including *pigz*, *lbzip2*, *pbzip2*, *xz* and *pxz*. These tools have been present in GNU/Linux distributions for many years and are integrated into the GNU tar utility.
2) *ImageMagick* and *GraphicsMagick* are two widely-used software suites to display, convert, and edit image files.
3) *libvpx* and *libwebp* are two WebM projects for VP8/VP9 and WebP codecs. They are used by popular browsers like Chrome, Firefox, and Opera.
4) *x264* and *x265* are the two most established video encoders for H.264/AVC and HEVC/H.265, respectively.

All these projects' single-thread functionalities have been intensively tested by mainstream GBFs such as AFL. We try to use their latest versions at the time of our evaluation; the only exception is *libvpx*, which we use version v1.3.0-5589 to reproduce the ground-truth vulnerabilities and concurrency-bugs. Among the 12 multithreaded programs, *pxz*, *GraphicsMagick*, and *ImageMagick* use OpenMP library, while the others use native PThread.

Table 1 lists the statistics of the benchmarks. The first two columns show the benchmark IDs and their host projects. The next column specifies the command-line options. In particular, four working threads are specified to enforce the program to run in multithreading mode.

The rest of the columns are the static statistics. Column "Binary Size" calculates the sizes of the instrumented binaries. Column T_{pp} records the preprocessing time of static analysis (c.f. §4.1). Among the 12 benchmarks, *vpxdec* takes the longest time of approximately 30 minutes. Columns N_b, N_i, and N_{ii} depict the number of basicblocks, the number of total instructions, and the number of deputy instructions for M-Ins (c.f. §4.2), respectively. Recall that AFL-Ins instruments evenly over entry instructions of all basicblocks, hence N_b also denotes the number of deputy instructions in AFL, MAFL, and MOPT. The last column, $\frac{N_{ii}-N_b}{N_b}$, is the ratio of more instructions MUZZ instrumented versus AFL (or MAFL, MOPT). This ratio ranges from 6.0% (*pbzip2-c* or *pbzip2-d*) to 288.9% (*x265*). Fortunately, in practice, this does not proportionally increase the runtime overhead. Many aspects can affect this metric, including the characteristics of the target programs, the precision of the applied static analysis, and the empirically specified thresholds P_{s0} and P_{m0}.

Fuzzing Configuration The experiments are conducted on

Table 1: Static statistics of the 12 evaluated benchmarks; meanings of the columns are explained in §6.1.2.

ID	Project	Command Line Options	Binary Size	T_{pp}	N_b	N_i	N_{ii}	$\frac{N_{ii}-N_b}{N_b}$
lbzip2-c	lbzip2-2.5	lbzip2 -k -t -9 -z -f -n4 FILE	377K	7.1s	4010	24085	6208	54.8%
pbzip2-c	pbzip2-v1.1.13	pbzip2 -f -k -p4 -S16 -z FILE	312K	0.9s	2030	8345	2151	6.0%
pbzip2-d	pbzip2-v1.1.13	pbzip2 -f -k -p4 -S16 -d FILE	312K	0.9s	2030	8345	2151	6.0%
pigz-c	pigz-2.4	pigz -p 4 -c -b 32 FILE	117K	5.0s	3614	21022	5418	49.9%
pxz-c	pxz-4.999.9beta	pxz -c -k -T 4 -q -f -9 FILE	42K	1.2s	3907	30205	7877	101.6%
xz-c	XZ-5.3.1alpha	xz -9 -k -T 4 -f FILE	182K	8.4s	4892	34716	8948	82.9%
gm-cnvt	GraphicsMagick-1.4	gm convert -limit threads 4 FILE out.bmp	7.6M	224.4s	63539	383582	98580	55.1%
im-cnvt	ImageMagick-7.0.8-7	convert -limit thread 4 FILE out.bmp	19.4M	434.2s	128359	778631	200108	55.9%
cwebp	libwebp-1.0.2	cwebp -mt FILE -o out.webp	1.8M	56.3s	12117	134824	33112	173.3%
vpxdec	libvpx-v1.3.0-5589	vpxdec -t 4 -o out.y4m FILE	3.8M	431.6s	31638	368879	93400	195.2%
x264	x264-0.157.2966	x264 –threads=4 -o out.264 FILE	6.4M	1701.0s	38912	410453	103926	167.1%
x265	x265-3.0_Au+3	x265 –input FILE –pools 4 -F 2 -o	9.7M	78.3s	22992	412555	89408	288.9%

four Intel(R) Xeon(R) Platinum 8151 CPU@3.40GHz workstations with 28 cores, each of which runs a 64-bit Ubuntu 18.04 LTS; the evaluation upon a specific benchmark is conducted on one machine. To make fair comparisons, MUZZ, MAFL and AFL are executed in their "fidgety mode" [65], while MOPT is specified with `-L 0` to facilitate its "pacemaker mode" [33]. The CPU affinity is turned off during fuzzing to avoid multiple threads being bound to a single CPU core. During fuzzing, we run each of the aforementioned fuzzers *six times* against all the 12 benchmark programs, with a time budget of 24 hours. Since all the evaluated programs are set to run with four working threads and the threads are mapped to different cores, it takes *each fuzzer* approximately $12 \times 6 \times 24 \times 4 = 6912$ CPU hours.

6.2 Seed Generation (RQ1)

Table 2 shows the overall fuzzing results in terms of newly generated seeds. We collect the total number of generated seeds (N_{all}) and the number of seeds that exercise the multithreading context (N_{mt}). In AFL's jargon, N_{all} corresponds to the distinct paths that the fuzzer observes [63]. The multithreading-relevant seeds are collected with a separate procedure, based on the observations that they at least invoke one element in *TFork*. Therefore, N_{mt} tracks the different multithreading execution states during fuzzing — a larger value of this metric suggests the fuzzer can keep more effective thread-interleaving seeds. We sum up those seed files across all six fuzzing runs to form N_{all} and N_{mt} in Table 2. The $\frac{N_{mt}}{N_{all}}$ column shows the percentage of N_{mt} over N_{all}. $\frac{N_{mt}}{N_{all}}$ determines the probability of picking a multithreading-relevant seed during seed selection, which greatly impacts the overall quality of the generated seeds. Obviously, the most critical metrics are N_{mt} and $\frac{N_{mt}}{N_{all}}$.

MUZZ surpasses MAFL, AFL, and MOPT in both metrics. First, MUZZ exhibits superiority in generating multithreading-relevant seeds — in all the benchmarks MUZZ achieves the highest N_{mt}. For example, in *pbzip2-d*, despite that all the $\frac{N_{mt}}{N_{all}}$ are relatively small, MUZZ generated 297 multithreading-relevant seeds, which is 178 more than MAFL (119), 229 more than AFL (68), and 235 more than MOPT (62). Moreover, for larger programs such as *im-cnvt* (binary size 19.4M), N_{mt} of MUZZ (12987) is still better than the others (MAFL: 10610, AFL: 7634, MOPT: 8012). Second, the value of $\frac{N_{mt}}{N_{all}}$ in MUZZ is more impressive — MUZZ wins the comparison over all the benchmarks. For example, in *pbzip2-d*, MUZZ's result of $\frac{N_{mt}}{N_{all}}$ is higher — MUZZ: 14.9%, AFL: 7.0% MAFL: 4.1%, and MOPT: 3.8%. For the benchmark where AFL has already achieved a decent result, e.g., 89.3% for *x264*, MUZZ can even improve it to 96.5%. Meanwhile, although MAFL has the largest N_{mt} for *x265* (10890), the value of its $\frac{N_{mt}}{N_{all}}$ (78.6%) is less than that of MUZZ (82.6%).

It is worth noting that MAFL also outperforms AFL and MOPT w.r.t. N_{mt} and $\frac{N_{mt}}{N_{all}}$ in all the benchmarks. For example, in *pxz-c*, the number of generated multithreading-relevant seeds in MAFL is 3401, which is more than AFL (2470) and MOPT (2634). Correspondingly, the percentage of multithreading-relevant seeds in MAFL is 60.3%; for AFL and MOPT, they are 46.1% and 47.2%, respectively. Considering MAFL, AFL, MOPT apply coverage-oriented instrumentation (M-Ins), we can conclude that other strategies in MAFL, including thread-context instrumentation, schedule-intervention instrumentation, and the optimized dynamic strategies, also contribute to effective seed generation.

Answer to RQ1: MUZZ has advantages in increasing the number and percentages of multithreading-relevant seeds for multithreaded programs. The proposed three thread-aware instrumentations and dynamic fuzzing strategies benefit the seed generation.

6.3 Vulnerability Detection (RQ2)

For vulnerability detection, we denote the total number of proof-of-crash (POC) files generated during fuzzing as N_c. The vulnerability analysis component (right-bottom area as Ⓒ in Figure 3) analyzes the POC files and categorizes them into different vulnerabilities. This basically follows ClusterFuzz's practice [16]: if two POC files have the same last N lines of backtraces and the root cause is the same (e.g., both

Table 2: Fuzzing results on MUZZ, MAFL, AFL and MOPT, in terms of generated seeds. N_{all}: total number of new seeds; N_{mt}: number of new multithreading-relevant seeds; $\frac{N_{mt}}{N_{all}}$: the percentage of multithreading-relevant seeds among all the generated seeds. **Bold data entries** mark the best results among the fuzzers, in terms of N_{mt} and $\frac{N_{mt}}{N_{all}}$. The numbers in parentheses (for N_{all} and $\frac{N_{mt}}{N_{all}}$) denote the differences between MUZZ and the others; for example, "(+1850)" is the more multithreading-relevant seeds generated by MUZZ: 5127 than MAFL: 3277.

ID	MUZZ			MAFL			AFL			MOPT		
	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$	N_{all}	N_{mt}	$\frac{N_{mt}}{N_{all}}$
lbzip2-c	8056	**5127**	**63.6%**	6307	3277(+1850)	52.0%(+11.7%)	5743	2464(+2663)	42.9%(+20.7%)	6033	2524(+2603)	41.8%(+21.8%)
pbzip2-c	381	**126**	**33.1%**	340	91(+35)	26.8%(+6.3%)	272	69(+57)	25.4%(+7.7%)	279	71(+55)	25.4%(+7.6%)
pbzip2-d	1997	**297**	**14.9%**	1706	119(+178)	7.0%(+7.9%)	1650	68(+229)	4.1%(+10.8%)	1623	62(+235)	3.8%(+11.1%)
pigz-c	1406	**1295**	**92.1%**	1355	1189(+106)	87.7%(+4.4%)	1298	1098(+197)	84.6%(+7.5%)	1176	982(+313)	83.5%(+8.6%)
pxz-c	7590	**5249**	**69.2%**	5637	3401(+1848)	60.3% (+8.8%)	5357	2470(+2779)	46.1% (+23.0%)	5576	2634(+2615)	47.2% (+21.9%)
xz-c	2580	**1098**	**42.6%**	2234	767(+331)	34.3%(+8.2%)	1953	581(+517)	29.7%(+12.8%)	1845	566(+532)	30.7%(+11.9%)
gm-cnvt	15333	**13774**	**89.8%**	14031	10784(+2990)	76.9%(+13.0%)	12453	8290(+5484)	66.6%(+23.3%)	12873	8956(+4818)	69.6%(20.3%)
im-cnvt	14377	**12987**	**90.3%**	12904	10610(+2377)	82.2%(+8.1%)	9935	7634(+5353)	76.8%(+76.8%)	10203	8012(+4975)	78.5%(+11.8%)
cwebp	11383	**7554**	**66.4%**	10389	6868(+686)	66.1% (+0.3%)	9754	5874(+1680)	60.2% (+6.1%)	9803	5869(+1685)	59.9%(+6.5%)
vpxdec	28892	**25593**	**88.6%**	27735	22507(+3086)	81.2%(+7.4%)	24397	18936(+6657)	77.6%(+11.0%)	27119	20896(+4697)	77.1%(11.5%)
x264	15138	**14611**	**96.5%**	14672	13413(+1198)	91.4% (+5.1%)	13211	11801(+2810)	89.3% (+7.2%)	12427	11202(+3409)	90.1%(+6.4%)
x265	12965	10704	**82.6%**	13858	**10890**(-186)	78.6% (+4.0%)	12980	9957(+747)	76.7% (+5.9%)	13142	10154 (+550)	77.3%(+5.3%)

exhibit as *buffer-overflow*), they are treated as one vulnerability. Afterwards, we manually triage all the vulnerabilities into two groups based on their relevance with multithreading: the concurrency-vulnerabilities V_m, and the other vulnerabilities that do not occur in multithreading context V_s. The number of these vulnerabilities are denoted as N_v^m and N_v^s, respectively.

We mainly refer to N_c^m, N_v^m in Table 3 to evaluate MUZZ's *concurrency-vulnerability* detection capability.

The number of multithreading-relevant POC files, N_c^m, is important since it corresponds to different crashing states when executing multithreading context [27, 34]. It is apparent that MUZZ has the best results of N_c^m in all the benchmarks that have V_m vulnerabilities (e.g., for *im-cnvt*, MUZZ: 63, MAFL: 23, AFL: 6, MOPT: 6). Moreover, MAFL also exhibits better results than AFL and MOPT (e.g., for *pbzip2-c*, MUZZ: 6, MAFL: 6, AFL: 0, MOPT: 0). This suggests that MUZZ's and MAFL's emphasis on multithreading-relevant seed generation indeed helps to exercise more erroneous multithreading-relevant execution states.

The most important metric is N_v^m since our focus is to detect concurrency-vulnerabilities (V_m). Table 3 shows that MUZZ has the best results: MUZZ detects 9 concurrency-vulnerabilities, while MAFL, AFL and MOPT detects 5, 4, 4, respectively. Detected V_m can be divided into three groups.

1) V_m caused by concurrency-bugs. We term this group of vulnerabilities as V_{cb}. The 4 vulnerabilities in *im-cnvt* all belong to this group — the misuses of caches shared among threads cause the data races. The generated seeds may exhibit various symptoms such as *buffer-overflow* and *memcpy-param-overlap*. MUZZ found all the 4 vulnerabilities, while the others only found 2. We also observed that for the 2 vulnerabilities that are detected by all these fuzzers, MAFL's detection capability appears *more stable* since it detects both in all its six fuzzing runs, while the others can only detect them at most in five runs (not depicted in the table). **2) V_m triggered in multithreading only but not induced by concurrency-bugs**. For example, the vulnerability in *pbzip2-d* stems from a *stack-overflow* error when executing a function concurrently. This crash can never happen when *pbzip2-d* works in single-thread mode since it does not even invoke that erroneous function. In our evaluation, MUZZ detected this vulnerability while the other fuzzers failed. Another case is the vulnerability in *pbzip2-c*, which was detected by MUZZ and MAFL, but not by AFL or MOPT. 3) **Other concurrency-vulnerabilities**. The characteristics of these V_m are that their crashing backtrace contains multithreading context (i.e., *TFork* is invoked), however, the crashing condition might also be occasionally triggered when only one thread is specified. The V_m vulnerabilities detected in *vpxdec* and *x264* belong to this category. In particular, MUZZ detects 2 vulnerabilities in *vpxdec* while MAFL, AFL, and MOPT only find 1.

We consider the reason behind the differences w.r.t. N_c^m and N_v^m among the fuzzers to be that, MUZZ keeps more "deeper" multithreading-relevant seeds that witness different execution states, and mutations on some of them are more prone to trigger the crashing conditions.

Columns N_c, N_c^s, N_v^s are metrics less focused. But we can still observe that 1) according to N_c, MUZZ (and MAFL) can exercise more total crashing states; 2) despite that the values of N_c^s from MUZZ are usually smaller, MUZZ can still find all the (categorized) V_s detected by other fuzzers.

From the 12 evaluated benchmarks, we reported the 10 new vulnerabilities (sum of MUZZ's results in columns N_v^m and N_v^s except for row *vpxdec*; 7 of them belong to V_m), all of them have been confirmed or fixed, 3 of which have already been assigned CVE IDs. Besides, we also conducted a similar evaluation on *libvpx* v1.8.0-178 (the git HEAD version at the time of evaluation). MUZZ detected a 0-day concurrency-vulnerability within 24 hours (among six fuzzing runs, two of them detected the vulnerability in 5h38min and 16h07min,

Table 3: Fuzzing results on MUZZ, MAFL, AFL and MOPT, in terms of crashes and vulnerabilities. Some projects (e.g., *lbzip2-c*) are excluded since there were no crashes/vulnerabilities detected by any of the fuzzers. N_c: number of proof-of-crash (POC) files; N_c^m: number of multithreading-relevant POC files; N_v^m: number of concurrency-vulnerabilities. N_c^s: number of POC files irrelevant with multithreading; N_v^s: number of vulnerabilities irrelevant to multithreading. **Bold data entries** mark the best results for N_c^m and N_v^m. The numbers in parentheses denote the differences between MUZZ and others.

ID	MUZZ					MAFL					AFL					MOPT				
	N_c	N_c^m	N_v^m	N_c^s	N_v^s	N_c	N_c^m	N_v^m	N_c^s	N_v^s	N_c	N_c^m	N_v^m	N_c^s	N_v^s	N_c	N_c^m	N_v^m	N_c^s	N_v^s
pbzip2-c	6	**6**	**1**	0	0	6	0(+6)	**1**(0)	0	0	0	0(+6)	0(+1)	0	0	0	0(+6)	0(+1)	0	0
pbzip2-d	15	**15**	**1**	0	0	0	0(+15)	0(+1)	0	0	0	0(+15)	0(+1)	0	0	0	0(+15)	0(+1)	0	0
im-cnvt	87	**63**	**4**	24	1	49	23(+40)	2(+2)	26	1	29	6(+57)	2(+2)	23	1	32	6(+57)	2(+2)	26	1
cwebp	19	0	0	19	1	27	0(0)	0(0)	27	1	14	0(0)	0(0)	14	1	15	0(0)	0(0)	15	1
vpxdec	523	**347**	**2**	176	2	495	279(+68)	1(+1)	216	2	393	205(+142)	1(+1)	188	2	501	301(+46)	1(+1)	200	2
x264	103	**103**	**1**	0	0	88	88(+15)	**1**(0)	0	0	78	78(+25)	**1**(0)	0	0	66	66(+37)	**1**(0)	0	0
x265	43	0	0	43	1	52	0(0)	0(0)	52	1	62	0(0)	0(0)	62	1	59	0(0)	0(0)	59	1

respectively), while MAFL, AFL and MOPT failed to detect it in 15 days (360 hours) in all their six fuzzing runs. The newly detected vulnerability has been assigned with another CVE ID. The vulnerability details are available in Table 5.

Given the fact that there are extremely few CVE records caused by concurrency-vulnerabilities (e.g., 202 among 70438, based on records from CVE-2014-* to CVE-2018-*) [48], MUZZ demonstrates the high capability in detecting concurrency-vulnerabilities.

Answer to RQ2: MUZZ demonstrates superiority in exercising more multithreading-relevant crashing states and detecting concurrency-vulnerabilities.

6.4 Concurrency-bug Revealing (RQ3)

The fuzzing phase only detects the vulnerabilities caused by crashes, but the *seemingly normal* seed files generated during fuzzing may still execute paths that trigger concurrency-violation conditions like *data-races*, *deadlocks*, etc. We detect concurrency-bugs in concurrency-bug revealing component (Ⓓ, right-top in Figure 3). It is worth noting that our goal is *not* to improve the capabilities of concurrency-bug detection over existing techniques such as TSan [42], Helgrind [49], or UFO [21]. Instead, we aim to *reveal as many bugs as possible within a time budget, by replaying against fuzzer-generated seeds with the help of these techniques*. In practice, this component feeds the target program with the seeds that were generated during fuzzing as its inputs, and facilitate detectors such as TSan to reveal concurrency-bugs. During this evaluation, we compiled the target programs with TSan and replayed them against the fuzzer-generated multithreading-relevant seeds (corresponding to N_{mt} in Table 2). We did not replay with *all the generated seeds* (corresponding to N_{all} in Table 2) since seeds not exercising multithreading context will not reveal concurrency-bugs.

We limit our replay time budget to two hours; in §6.5.4 we discuss the rationale of this configuration. The next is to *determine the replay pattern* per seed to reveal more concurrency-bugs within this budget. This is necessary since TSan may fail to detect concurrency-bugs in a few runs when it does not *observe concurrency violation conditions* [12, 42, 49]. Meanwhile, as the time budget is limited, we cannot exhaustively replay against a given seed to see whether it may trigger concurrency-violations — in the worst case, we may waste time in executing against a seed that never violates the conditions. We provide two replay patterns.

$\mathbb{P}1$ It executes against each seed in the queue *once per turn* in a round-robin way, until reaching the time budget.

$\mathbb{P}2$ It relies on $\mathbb{N}_c$ in repeated execution (c.f., §5.2): each seed is executed $\frac{\mathbb{N}_c}{N_0}$ *times per turn* continuously in a round-robin way. According to Equation 4, we replay 5 times per turn (40/8) for AFL generated multithreading-relevant seeds; for MUZZ and MAFL, it is determined by Equation (5), with candidate values 2, 3, 4, 5.

It is fair to compare replay results w.r.t. $\mathbb{P}1$ and $\mathbb{P}2$ in that the time budget is fixed. The difference between the two patterns is that seeds' execution orders and accumulated execution time spent on them can be rather different.

Table 4 depicts the results for concurrency-bug revealing with $\mathbb{P}1$ and $\mathbb{P}2$. N_e^m is the number of *observed concurrency-violation executions* and N_B^m is the number of concurrency-bugs (B_m) according to their root causes. For example, it only counts one concurrency-bug (N_B^m=1) even when the replaying process observes 10 data-race pairs across executions (N_e^m=10), as long as the root cause of the races is unique. We analyze this table from two perspectives.

First, MUZZ *demonstrates superiority in concurrency-bug detection regardless of replay patterns.* This is observed based on the "best results" for each metric in each pattern. MUZZ achieves the best results for most projects. For example, when *x264* is replayed with N_e^m, 1) MUZZ's found the most violations — the values of N_e^m are, MUZZ: 68, MAFL: 46, AFL: 28, MOPT: 30; 2) the best result of N_B^m also comes from MUZZ, MUZZ: 8, MAFL: 6, AFL: 4, MOPT: 5. Similar results can also be observed with $\mathbb{P}2$ for *x264*, where MUZZ has the biggest N_e^m(91) and biggest N_B^m(9). The only project where MAFL achieves the best is *pigz-c*, where it is slightly

Table 4: Comparisons of replay patterns $\mathbb{P}1$ and $\mathbb{P}2$ on MUZZ, MAFL, AFL and MOPT, in terms of concurrency violations (N_e^m) and concurrency-bugs (N_B^m). The best results of N_e^m and N_B^m are underlined / **bold** for $\mathbb{P}1$ / $\mathbb{P}2$ respectively.

ID	$\mathbb{P}1$								$\mathbb{P}2$							
	MUZZ		MAFL		AFL		MOPT		MUZZ		MAFL		AFL		MOPT	
	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m	N_e^m	N_B^m
lbzip2-c	469	1	447	1	386	1	435	1	**493**	**1**	483	**1**	421	**1**	458	**1**
pigz-c	793	1	803	1	764	1	789	1	856	**1**	**862**	**1**	727	**1**	742	**1**
gm-cnvt	93	5	79	4	45	2	55	3	**133**	**5**	83	4	54	3	57	3
im-cnvt	92	3	84	3	58	2	56	2	**118**	**3**	117	**3**	65	2	59	2
vpxdec	31	3	17	1	23	1	18	1	**42**	**3**	22	1	25	1	22	1
x264	68	8	46	6	28	4	30	5	**91**	**9**	52	6	25	4	28	4

better than MUZZ.

Second, as to MUZZ *and* MAFL, $\mathbb{P}2$ *is probably better than* $\mathbb{P}1$. It is concluded based on the fact that $\mathbb{P}2$'s "best results" are all better than $\mathbb{P}1$'s. For example, as to N_e^m in *x264*, the best result of N_e^m is achieved with $\mathbb{P}2$ ($\mathbb{P}1$: 68, $\mathbb{P}2$: 91); similarly, the best result of N_B^m also comes from $\mathbb{P}2$ ($\mathbb{P}1$: 8, $\mathbb{P}2$: 9). In the meantime, there seems to be no such implication inside AFL or MOPT. Besides the numbers of concurrency-violations or concurrency-bugs, §6.5.3 provides a case study on `gm-cnvt` that demonstrates $\mathbb{P}2$'s advantages over $\mathbb{P}1$ w.r.t. *time-to-exposure* of the concurrency-bugs.

We have reported all the newly detected 19 concurrency-bugs (excluding the 3 concurrency-bugs in *vpxdec-v1.3.0-5589*) to their project maintainers (c.f., Table 5 for the details).

Answer to RQ3: MUZZ outperforms competitors in detecting concurrency-bugs; the value $\mathbb{N}_c$ calculated during fuzzing additionally contributes to revealing these bugs.

6.5 Further Discussions

This section discusses miscellaneous concerns, issues and observations for MUZZ's design and evaluation.

6.5.1 Constant Parameters

Using empirical constant parameters for grey-box fuzzing is practiced by many fuzzing techniques [6,33,63]. For example, AFL itself has many hard-coded configurations used by default; MOPT additionally has the suggested configuration to control the time to move on to pacemaker mode (i.e., `-L 0`).

In MUZZ, constant parameters are used in two places.

(1) The upper bounds for coverage-oriented instruction: P_{s0} (defaults to 0.5) and P_{m0} (defaults to 0.33). These default values inspire from AFL's "selective deputy instruction instrumentation" strategy to make the instrumentation ratio to be 0.33 when AddressSanitizer is involved during instrumentation. Larger values of P_{s0} and P_{m0} increases the instrumentation ratio only if *the thresholds are frequently reached*. Subsequently, the instrumented program has these symptoms: a) the program size after instrumentation increases; b) the execution state feedback is potentially better; c) the instrumentation-introduced execution speed slowdown is more evident. Therefore, increasing the values of P_{s0} and P_{m0} reflects a tradeoff between precise feedback and its overhead. In our benchmarks, when we assign P_{s0} =0.5, P_{m0} =0.33,

- For *im-cnvt*, the speed slowdown is about 15% compared to default settings, while the capability of detecting concurrency-vulnerabilities and concurrency-bugs are similar; meanwhile, there are a few more multithreading-relevant seeds (N_{mt}) but $\frac{N_{mt}}{N_{all}}$ is slightly smaller.
- For *pbzip2-c*, the differences brought by changes of P_{s0} and P_{m0} from the default settings are all neglectable.

We believe there are no optimal instrumentation thresholds that work for all the projects; therefore MUZZ provides the empirical values as the defaults.

(2) The seed selection probabilities $P_{ynt} = 0.95$, $P_{ynn} = 0.01$, $P_{nnn} = 0.15$ in Algorithm 3. These constants are not introduced by MUZZ, but based on AFL's "skipping probability" to conditionally favor seeds with new coverage [63].

Since the 12 benchmarks that we chose are quite diversified (c.f., §6.1.2), it is considered fair to use default settings for these parameters, when comparing MUZZ, MAFL with other fuzzers such as AFL, MOPT. In practice, we suggest keeping MUZZ's default settings to test other multithreaded programs.

6.5.2 Schedule-intervention Instrumentation

The goal of MUZZ's schedule-intervention is to diversify interleavings during repeated executions in the fuzzing phase. During the evaluation, we did not separately evaluate the effects of *schedule-intervention instrumentation*. However, based on our observation, this instrumentation is important to achieve more stable fuzzing results. Two case studies can support this statement.

a) We turned off schedule-intervention instrumentation in MUZZ and fuzzed *lbzip2-c* six times on the *same* machine. The calculated value of $\frac{N_{mt}}{N_{all}}$ is 54.5% (= 4533/8310), which is lower than the result in Table 2 (63.6% = 5127/8056). Since 54.5% is still greater than the results of AFL (42.9%) and MOPT (41.8%), this also indicates MUZZ's other two strategies indeed benefit the multithreading-relevant seed generation for fuzzing.

Table 5: Newly detected vulnerabilities and concurrency-bugs. This summarizes the *new* vulnerabilities and concurrency-bugs evaluated in Table 3 and Table 4 over the 11 benchmarks (*libvpx-v1.3.0-5589* results are all excluded), and includes one concurrency-vulnerability in *vpxdec-v1.8.0-178* which was mentioned in §6.3.

Bugs	**Project**	**Bug Type**	**Reported Category**	**MUZZ**	**MAFL**	**AFL**	**MOPT**	**Status**
V1	pbzip2	V_m	divide-by-zero	✓	✓	✗	✗	confirmed, not fixed
V2	pbzip2	V_m	stack-overflow	✓	✗	✗	✗	confirmed, not fixed
V3	ImageMagick	V_m	memcpy-param-overlap	✓	✗	✗	✗	CVE-2018-14560
V4	ImageMagick	V_m	buffer-overflow	✓	✓	✓	✓	CVE-2019-15141
V5	ImageMagick	V_m	buffer-overflow	✓	✓	✓	✓	confirmed, fixed
V6	ImageMagick	V_m	buffer-overflow	✓	✗	✗	✗	confirmed, fixed
V7	ImageMagick	V_s	negative-size-param	✓	✓	✓	✓	CVE-2018-14561
V8	x264	V_m	buffer-overflow	✓	✓	✓	✓	confirmed, fixed
V9	libwebp	V_s	failed-to-allocate	✓	✓	✓	✓	confirmed, won't fix
V10	x265	V_s	divide-by-zero	✓	✓	✓	✓	confirmed, not fixed
V11	libvpx-v1.8.0-178	V_m	invalid-memory-read	✓	✗	✗	✗	CVE-2019-11475
C1	lbzip2	B_m	thread-leak	✓	✓	✓	✓	confirmed, not fixed
C2	pigz	B_m	lock-order-inversion	✓	✓	✓	✓	confirmed, fixed
C3-C7	GraphicsMagick	B_m	data-race	5	4	3	2	confirmed, fixed
C8-C10	ImageMagick	B_m	data-race	3	3	2	2	confirmed, fixed
C11-C19	x264	B_m	data-race	9	6	4	4	confirmed, not fixed

b) We turned off schedule-intervention instrumentation in MUZZ and fuzzed *im-cnvt* on a *different* machine. In all the six fuzzing runs it only detects three concurrency-vulnerabilities which is less than the result in Table 3 (N_v^m =4). Meanwhile, when the schedule-intervention instrumentation is re-enabled, MUZZ can still detect four concurrency-vulnerabilities in that machine.

6.5.3 Time-to-exposure for Concurrency-bug Revealing

In §6.4, we demonstrate $\mathbb{P}2$'s advantage over $\mathbb{P}1$ in terms of occurrences of concurrency-violations (N_e^m) and the number of categorized concurrency-bugs (N_B^m). Another interesting metric is the *time-to-exposure* capability of these two replay patterns — given the ground truth that the target programs contain certain concurrency-bugs, the minimal time cost for each pattern to reveal all the known bugs. This metric can further distinguish the two replay patterns' capabilities in terms of revealing concurrency-bug.

We conducted a case study on *gm-cnvt*. From Table 4, it is observable that with both $\mathbb{P}1$ and $\mathbb{P}2$, TSan detected four concurrency-bugs (N_B^m) by replaying the MAFL generated multithreading-relevant seeds (totally 10784) from Table 2; besides, their N_e^m results are also similar ($\mathbb{P}1$: 79, $\mathbb{P}2$: 83). We repeated six times against the 10784 seeds by applying $\mathbb{P}1$ and $\mathbb{P}2$. When a replaying process detects *all* the four different ground-truth concurrency-bugs, we record the total execution time (in *minutes*). Table 6 shows the results.

In Table 6, compared to $\mathbb{P}1$, we can observe that $\mathbb{P}2$ reduces the average time-to-exposure from 66.5 minutes to 34.1 minutes. This fact means, for example, given a tighter replay time budget (say, 60 minutes), $\mathbb{P}1$ has a high chance to miss some of the four concurrency-bugs. Moreover, $\mathbb{P}2$ is more stable since the timing variance is much smaller than that of

Table 6: Time-to-exposure of *gm-cnvt*'s concurrency-bugs during six replays with patterns $\mathbb{P}1$ and $\mathbb{P}2$.

	#1	#2	#3	#4	#5	#6	Avg	Variance
$\mathbb{P}1$	55.3	92.1	21.8	93.7	101.5	34.7	66.5	959.2
$\mathbb{P}2$	33.4	52.2	33.5	37.6	24.7	23.3	34.1	91.0

$\mathbb{P}1$ (91.0 vs. 959.2). This result implicates that, in Table 4, for the concurrency-bug revealing capability of MAFL, the $\mathbb{P}2$'s result in *gm-cnvt* is likely to be much better than $\mathbb{P}1$'s.

The evaluation of time-to-exposure suggests that, given a set of seeds, $\mathbb{P}2$ is prone to expose concurrency-bugs faster and more stable. Since $\mathbb{P}2$ is closely relevant to schedule-intervention instrumentation (§4.4) and repeated execution (§5.2), this also indicates that these strategies are helpful for concurrency-bug revealing.

6.5.4 Time Budget During Replaying

We chose two hours (2h) as the time budget in the reply phase during evaluation. Unlike the fuzzing phase, which aims to generate *new seed* files that exercise multithreading context, the replay phase runs the target program against *existing seeds* (generated during fuzzing). Therefore, the criterion is to 1) minimize the time for replay; 2) ensure that replay phase traverses all the generated seeds. For projects with less generated (multithreading-relevant) seeds (e.g., N_{mt} =126 for *pbzip2-c* when applying MUZZ), traversing the seeds (with both $\mathbb{P}1$ and $\mathbb{P}2$) *once* are quite fast; however for projects with more generated seeds (e.g., N_{mt} =13774 for *gm-cnvt* when applying MUZZ), this requires more time. To make the evaluation fair, we use the fixed time budget for all the 12 benchmarks, where seeds in projects like *pbzip2-c* will be traversed repeatedly until timeout. During the evaluation, we found 2h to be moderate since it can traverse all the generated *multithreading-relevant seeds* at least once for all the projects.

Less time budget, e.g., 1h, may make the replay phase to miss certain generated seeds triggering concurrency violation conditions. In fact, from Table 6, we see that time-to-exposure for the concurrency-bugs may take 101.5 minutes. Meanwhile, more time budget, e.g., 4h, might be a waste of time for the exercised 12 benchmarks. In fact, in a case study for *gm-cnvt*, when time budget is 4h, despite that N_e^m is nearly doubled, the number of revealed B_m (i.e., N_B^m) is still the same as the results in Table 4, regardless of $\mathbb{P}1$ or $\mathbb{P}2$.

6.5.5 Statistical Evaluation Results

Specific to the nature of multithreaded programs and our evaluation strategy to determine seeds' relevance with multithreading, we decide not to provide some commonly-used statistical results [27].

First, it is *unfair* to track *coverage over time* when comparing MUZZ, MAFL with AFL or MOPT due to the different meanings of "coverage". In fact, owing to coverage-oriented instrumentation (in MUZZ) and threading-context instrumentation (in MUZZ and MAFL), MUZZ and MAFL cover more execution states (corresponding to N_{all}), therefore naturally preserve more seeds. That is also the reason that in §6.2 the values of N_{mt} and $\frac{N_{mt}}{N_{all}}$ are more important than N_{all}.

Second, we *cannot* compare the *multithreading-relevant paths over time* among MUZZ, MAFL, AFL, and MOPT. This reason is simple: we resort to a separate procedure *after fuzzing* to determine whether it covers thread-forking routines. We have to do so since AFL and MOPT do not provide a builtin solution to discovering seeds' relevance with multithreading. Consequently, we cannot plot multithreading-relevant crashing states over time.

Third, despite that the *statistical variance* is important, it is not easy to be calculated comprehensively. During evaluation, to reduce the variance among individuals, we apply an *ensemble strategy* by sharing seeds among the six runs, for each of the specific fuzzers [63]. However, for multithreaded target programs, another variance comes from the *thread scheduling* for different threads (in our experiments, four working threads were specified). MUZZ and MAFL have the schedule-intervention instrumentation to help diversify the effects, while it is absent in AFL and MOPT. In fact, from the case studies in §6.5.2, we envision that the variance may be huge for different machines under different workloads. Due to this, providing *fair* statistical results w.r.t. the variance may still be impractical. Therefore, we tend to exclude variance metrics and only choose those that exhibit the "overall results", i.e., N_{mt}, $\frac{N_{mt}}{N_{all}}$, N_c^m, N_v^m, N_e^m, and N_B^m. Similarly, the case studies or comparisons in §6.2, §6.3, §6.4 are all based on "overall results". During the evaluation, we indeed observed that the results of MUZZ and MAFL are more stable than those of AFL and MOPT.

7 Related Work

7.1 Grey-box Fuzzing Techniques

The most relevant is the fuzzing techniques on concurrency-vulnerability detection. ConAFL [30] is a thread-aware GBF that focuses on user-space multithreaded programs. Much different from MUZZ's goal to reveal both V_m and B_m, ConAFL only detects a subset of concurrency-bug induced vulnerabilities (V_{cb}) that cause *buffer-overflow*, *double-free*, or *use-after-free*. ConAFL also utilizes heavy thread-aware static and dynamic analyses, making it suffer from scalability issues. The other difference is that MUZZ's thread-aware analyses aim to provide runtime feedback to distinguish more execution states in multithreading contexts, to bring more multithreading-relevant seeds; meanwhile, ConAFL relies on the discovery of sensitive concurrency operations to capture pairs that may introduce the aforementioned three kinds of vulnerabilities. Further, since the static and dynamic analyses aim to capture and intervene "sensitive concurrency operation pairs", ConAFL suffers from the scalability issue. In fact, the biggest binary it evaluated was 196K (*bzip2smp*), while MUZZ can handle programs scaling to 19.4M (*im-cnvt*). In the evaluation, we did not evaluate ConAFL — the GitHub version of ConAFL (`https://github.com/Lawliar/ConAFL`) does not work since its static analysis is not publicly available and it is not trivial to implement that technique ourselves; further, we have not obtained the runnable tool after we requested from the authors. RAZZER [24] utilizes a customized hypervisor to control thread-interleaving deterministically to trigger data races in Linux kernel. It is a *kernel fuzzer* that cannot reveal multithreading-relevant bugs in *user-space* programs. As a matter of fact, the proof-of-crashes are essentially *sequences of system calls* that could trigger race conditions, and the fix of the detected vulnerabilities requires patches to the *kernel* code. Consequently, the guidance of fuzzing is also different. RAZZER spots the over-approximated racing segments and tames non-deterministic behavior of the kernel such that it can deterministically trigger a race. While MUZZ's solution is to distinguish more thread-interleaving states to trap the fuzzing to reveal more multithreading-relevant paths. Practically, it is not easy to effectively sequentialize the thread-interleavings to fuzz the user-space programs [64].

Multithreading-relevant bugs are inherently *deep*. To reveal deep bugs in the target programs, some GBFs facilitate other feedback [7,14,29,44,52,55,56,61]. Angora [7] distinguishes different calling context when calculating deputy instruction transitions to keep more valuable seeds. Driller [44], QSYM [61], and Savior [8] integrate symbolic execution to provide additional coverage information to exercise deeper paths. MUZZ inspires from these techniques in that it provides more feedback for multithreading context with stratified coverage-oriented and thread-context instrumentations, as well as schedule-intervention instrumentation. Other fuzzing techniques utilize the domain knowledge of the target program to generate more effective seeds [39,53,54]. Skyfire [53] and Superion [54] provide customized seed generation and mutation strategies on the programs that feed grammar-based inputs. SGF [39] relies on the specifications of the structured input to improve seed quality. These techniques are orthogonal to MUZZ and can be integrated into *seed mutation* (c.f. Ⓑ in Figure 3).

7.2 Static Concurrency-bug Prediction

Static concurrency-bug (B_m) predictors aim to approximate the runtime behaviors of a concurrent program without actual execution. Several static approaches have been proposed for analyzing Pthread and Java programs [40, 45, 50]. LOCKSMITH [40] uses existential types to correlate locks and data in dynamic heap structures for race detection. Goblint [50] relies on a thread-modular constant propagation and points-to analysis for detecting concurrent bugs by considering conditional locking schemes. [51] scales its detection to large codebases by sacrificing soundness and suppressing false alarms using heuristic filters. FSAM [45,46] proposes a sparse flow-sensitive pointer analysis for C/C++ programs using context-sensitive thread-interleaving analysis. Currently, MUZZ relies on flow- and context-insensitive results of FSAM for thread-aware instrumentations. We are seeking solutions to integrating other bug prediction techniques to further improve MUZZ's effectiveness.

7.3 Dynamic Analysis on Concurrency-bugs

There are a large number of dynamic analyses on concurrency-bugs. They can be divided into two categories: modeling concurrency-bugs and strategies to trigger these bugs.

The techniques in the first category [12,41,42,59] typically monitor the memory and synchronization events [19]. The two fundamentals are *happens-before model* [12] and *lockset model* [41]. Happens-before model reports a race condition when two threads read/write a shared memory arena in a causally unordered way, while at least one of the threads write this arena. Lockset model conservatively considers a potential race if two threads read/write a shared memory arena without locking. Modern detectors such as TSan [42], Helgrind [49] usually apply a hybrid strategy to combine these two models. MUZZ *does not* aim to improve existing concurrency violation models; instead, it relies on these models to detect concurrency-bugs with our fuzzer-generated seeds.

The second category of dynamic analyses focuses on how to trigger concurrency violation conditions. This includes random testings that mimic non-deterministic program executions [4,25,38], regression testings [47,60] that target interleavings from code changes, model checking [13,57,62] and hybrid constraint solving [20–22] approaches that systematically check or execute possible thread schedules, heuristically avoid fruitless executions [10,17,18,66], or utilizing multicore to accelerate bug detection [37]. Our work differs from all the above, as our focus is *not* to test schedules with a given seed file, but to generate seed files that execute multithreading-relevant paths. In particular, our goal of schedule-intervention instrumentation is to diversify the actual schedules to help provide feedback during fuzzing.

8 Conclusion

This paper presented MUZZ, a novel technique that empowers thread-aware seed generation to GBFs for fuzzing multithreaded programs. Our approach performs three novel instrumentations that can distinguish execution states introduced by thread-interleavings. Based on the feedback provided by these instrumentations, MUZZ optimizes the dynamic strategies to stress different kinds of multithreading context. Experiments on 12 real-world programs demonstrate that MUZZ outperforms other grey-box fuzzers such as AFL and MOPT in generating valuable seeds, detecting concurrency-vulnerabilities, as well as revealing concurrency-bugs.

Acknowledgement

This research was supported (in part) by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), National Satellite of Excellence in Trustworthy Software System (Award No. NRF2018NCR-NSOE003-0001), and NRF Investigatorship (Award No. NRFI06-2020-0022) administered by the National Cybersecurity R&D Directorate. The research of Dr Xue is supported by CAS Pioneer Hundred Talents Program.

References

[1] L. O. Andersen. Program analysis and specialization for the c programming language. Technical report, DIKU, University of Copenhagen, 1994.

[2] S. Blackshear, N. Gorogiannis, P. W. O'earn, and I. Sergey. Racerd: Compositional static race detection. *OOPSLA*, 2:144:1–144:28, Oct. 2018.

[3] M. Böhme, V. T. Pham, and A. Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *CCS '16*, pages 1032–1043. ACM, 2016.

[4] Y. Cai and W. K. Chan. Magicfuzzer: Scalable deadlock detection for large-scale applications. In *ICSE '12*, pages 606–616. IEEE, 2012.

[5] Y. Cai, B. Zhu, R. Meng, H. Yun, L. He, P. Su, and B. Liang. Detecting concurrency memory corruption vulnerabilities. In *ESEC/FSE '19*, pages 706–717, 2019.

[6] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *CCS '18*, pages 2095–2108. ACM, 2018.

[7] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *SP '18*, pages 711–725, 2018.

[8] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu. SAVIOR: towards bug-driven hybrid testing. In *SP '20*, 2020.

[9] I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of System Architecture*, 57(3):294–313, Mar. 2011.

[10] M. Christakis, A. Gotovos, and K. Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *ICST 2013*, pages 154–163, March 2013.

[11] P. Di and Y. Sui. Accelerating dynamic data race detection using static thread interference analysis. In *PMAM '16*, pages 30–39. ACM, 2016.

[12] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI '09*, pages 121–133. ACM, 2009.

[13] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05*, pages 110–121. ACM, 2005.

[14] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. Collafl: Path sensitive fuzzing. In *SP '18*, pages 1–12. IEEE, 2018.

[15] Google Inc. OSS-Fuzz, 2018.

[16] Google Inc. Clusterfuzz, 2019.

[17] S. Guo, M. Kusano, and C. Wang. Conc-iSE: Incremental symbolic execution of concurrent software. In *ASE '16*, pages 531–542. ACM, 2016.

[18] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *ESEC/FSE '15*, pages 854–865, 2015.

[19] S. Hong and M. Kim. A survey of race bug detection techniques for multithreaded programmes. *STVR*, 25(3):191–217, May 2015.

[20] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI '15*, pages 165–174. ACM, 2015.

[21] J. Huang. UFO: Predictive concurrency use-after-free detection. In *ICSE '18*, pages 609–619. ACM, 2018.

[22] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *PLDI '14*, pages 337–348, 2014.

[23] IEEE and The Open Group. POSIX.1-2017, 2001.

[24] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. RAZZER: Finding kernel race bugs through fuzzing. In *SP '19*, volume 00, pages 279–293, 2019.

[25] P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09*, pages 110–120, 2009.

[26] M. Kerrisk. *The Linux Programming Interface*. No Starch, 2010.

[27] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *CCS '18*, pages 2123–2138. ACM, 2018.

[28] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–. IEEE Computer Society, 2004.

[29] Y. Li, B. Chen, M. Chandramohan, S. Lin, Y. Liu, and A. Tiu. Steelix: Program-state based binary fuzzing. In *ESEC/FSE '17*, pages 627–637. ACM, 2017.

[30] C. Liu, D. Zou, P. Luo, B. B. Zhu, and H. Jin. A heuristic framework to detect concurrency vulnerabilities. In *ACSAC '18*, pages 529–541. ACM, 2018.

[31] LLVM. libFuzzer, 2015.

[32] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS '08*, pages 329–339. ACM, 2008.

[33] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *USENIX Security '19*, pages 1949–1966. USENIX Association, 2019.

[34] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140:1–29, 2018.

[35] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[36] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, Dec. 1990.

[37] S. Nagarakatte, S. Burckhardt, M. M. K. Martin, and M. Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI '12*, pages 543–554. ACM, 2012.

[38] C. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *ESEC/FSE '08*, pages 135–145, 2008.

[39] V. Pham, M. Böhme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *CoRR*, abs/1811.09447:1–16, 2018.

[40] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. *ACM SIGPLAN Notices*, 41(6):320–331, 2006.

[41] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997.

[42] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *WBIA '09*, pages 62–71. ACM, 2009.

[43] M. O. Shudrak and V. Zolotarev. Improving fuzzing using software complexity metrics. *CoRR*, abs/1807.01838:1–16, 2018.

[44] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS '16*. The Internet Society, 2016.

[45] Y. Sui, P. Di, and J. Xue. Sparse flow-sensitive pointer analysis for multithreaded programs. In *CGO '16*, pages 160–170. ACM, 2016.

[46] Y. Sui and J. Xue. SVF: Interprocedural static value-flow analysis in LLVM. In *CC '16*, pages 265–266. ACM, 2016.

[47] V. Terragni, S. Cheung, and C. Zhang. RECONTEST: effective regression testing of concurrent programs. In *ICSE '15*, pages 246–256, 2015.

[48] The MITRE Corporation. Download CVE List, 1999.

[49] Valgrind. Helgrind: a thread error detector, 2000.

[50] V. Vojdani and V. Vene. Goblint: Path-sensitive data race analysis. *Annales Univ. Sci. Budapest., Sect. Comp.*, pages 1–12, 2009.

[51] J. W. Voung, R. Jhala, and S. Lerner. Relay: static race detection on millions of lines of code. In *ESEC/FSE '07*, pages 205–214. ACM, 2007.

[52] H. Wang, X. Xie, Y. Li, C. Wen, Y. Liu, S. Qin, H. Chen, and Y. Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *ICSE '20*, 2020.

[53] J. Wang, B. Chen, L. Wei, and Y. Liu. Skyfire: Data-driven seed generation for fuzzing. In *SP '17*, pages 579–594, May 2017.

[54] J. Wang, B. Chen, L. Wei, and Y. Liu. Superion: grammar-aware greybox fuzzing. In *ICSE '19*, pages 724–735. IEEE / ACM, 2019.

[55] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *NDSS' 20*, 2020.

[56] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu. Memlock: Memory usage guided fuzzing. In *ICSE '20*, 2020.

[57] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded c programs. Technical report, Technical Report UUCS-08-004, University of Utah, 2008.

[58] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Distributed dynamic partial order reduction based verification of threaded software. In *SPIN '07*, pages 58–75. Springer-Verlag, 2007.

[59] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *OOPSLA '12*, pages 485–502, 2012.

[60] T. Yu, Z. Huang, and C. Wang. Contesa: Directed test suite augmentation for concurrent software. *IEEE Transactions on Software Engineering*, 2018.

[61] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Security '18*, pages 745–761. USENIX Association, 2018.

[62] A. Zaks and R. Joshi. Verifying multi-threaded C programs with SPIN. In *SPIN '08*, pages 325–342, 2008.

[63] M. Zalewski. Technical "whitepaper" for afl-fuzz, 2014.

[64] M. Zalewski. What is the drawback of fuzzing a multi-threaded binary?, 2015.

[65] M. Zalewski. "fidgetyafl" implemented in 2.31b, 2016.

[66] W. Zhang, J. Lim, R. Olichandran, J. Scherpelz, G. Jin, S. Lu, and T. Reps. ConSeq: detecting concurrency bugs through sequential errors. In *ASPLOS '11*, pages 251–264. ACM, 2011.