Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection

Zu-Ming Jiang
Tsinghua University

Jia-Ju Bai
Tsinghua University

Kangjie Lu
University of Minnesota

Shi-Min Hu
Tsinghua University

***Abstract*—Fuzzing is popular for bug detection and vulnerability discovery nowadays. To adopt fuzzing for concurrency problems like data races, several recent concurrency fuzzing approaches consider concurrency information of program execution, and explore thread interleavings by affecting thread scheduling at runtime. However, these approaches are still limited in data-race detection. On the one hand, they fail to consider the execution contexts of thread interleavings, which can miss real data races in specific runtime contexts. On the other hand, they perform random thread-interleaving exploration, which frequently repeats already covered thread interleavings and misses many infrequent thread interleavings.**

In this paper, we develop a novel concurrency fuzzing framework named CONZZER, to effectively explore thread interleavings and detect hard-to-find data races. The core of CONZZER is a context-sensitive and directional concurrency fuzzing approach for thread-interleaving exploration, with two new techniques. First, to ensure context sensitivity, we propose a new concurrency-coverage metric, *concurrent call pair*, to describe thread interleavings with runtime calling contexts. Second, to directionally explore thread interleavings, we propose an adjacency-directed mutation to generate new possible thread interleavings with already covered thread interleavings and then use a breakpoint-control method to attempt to actually cover them at runtime. With these two techniques, this concurrency fuzzing approach can effectively cover infrequent thread interleavings with concrete context information, to help discover hard-to-find data races. We have evaluated CONZZER on 8 user-level applications and 4 kernel-level filesystems, and found 95 real data races. We identify 75 of these data races to be harmful and send them to related developers, and 44 have been confirmed. We also compare CONZZER to existing fuzzing tools, and CONZZER continuously explores more thread interleavings and finds many real data races missed by these tools.

I. INTRODUCTION

Data race is a common class of concurrency problems. It occurs when two concurrently executed threads access a shared variable, and at least one of them writes the variable without proper synchronization operations. If the racy data influences critical data flow or control flow of the program, serious bugs such as memory corruption and permission bypass can occur. Some works [7], [55], [64], [73] have shown that data races have become a main source of runtime problems in both user-level applications and kernel-level programs. As an example, "Dirty COW" [20] launches a privilege-escalation attack against the Linux kernel by exploiting a data race in the memory management subsystem of the Linux kernel. Furthermore, many recent CVE-assigned vulnerabilities (e.g., CVE-2020-1667 [16], CVE-2020-9990 [17] and CVE-2020-11173 [18]) also stem from data races.

As data races can be harmful, many detection approaches have been proposed. Some approaches [22], [25], [26], [42], [77], [78] use static analysis, but they suffer from many false positives, due to lacking exact runtime information and inaccuracy of dataflow analysis. To reduce false positives, some other approaches use dynamic analysis based on lockset analysis [10], [19], [21], [66], [68], happens-before relation [27], [51], [56], [62] or sampling [5], [23]. But these approaches require substantial test cases to cover concurrently-executed code and different thread interleavings at runtime.

To automatically generate effective test cases, many recent approaches [1], [4], [9], [15], [30], [45], [50], [60], [61], [70], [72], [79], [80], [82] use fuzzing to cover infrequently-executed code and thus to discover hard-to-find bugs. Encouraged by the promising results of fuzzing, many developers have used existing fuzzing tools (such as AFL [1] and Syzkaller [70]) with third-party data-race checkers (such as TSan [75] and KCSAN [44]) to detect data races [41], [71]. However, general-purpose fuzzing tools are limited in exploring thread interleavings, due to using *sequential* code coverage as program feedback and neglecting *thread interleavings*.

To improve fuzzing in finding concurrency problems like data races, several recent concurrency fuzzing approaches [8], [37], [53], [76], [81] use concurrent-execution information to perform thread-interleaving exploration. However, these approaches still have two major limitations in testing concurrent programs. First, existing concurrency fuzzing approaches use a *context-insensitive* coverage metric as fuzzing feedback without considering runtime contexts, and thus they can miss many deep-hidden data races that occur only in specific runtime contexts. Second, existing concurrency fuzzing approaches perform *random thread-interleaving exploration* (e.g., injecting random delay or randomly adjusting thread priorities), but recent works [12], [74] reveal that such exploration is often inefficient, namely it frequently repeats already covered thread interleavings and misses many infrequent ones.

To solve these two limitations, we develop a novel concurrency fuzzing framework named CONZZER, which can effectively explore thread interleavings and detect hard-to-find data races. The core of CONZZER is a context-sensitive and directional concurrency fuzzing approach containing two

Network and Distributed Systems Security (NDSS) Symposium 2022
24-28 April 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24296
www.ndss-symposium.org

new techniques. First, this approach uses a new concurrency-coverage metric, *concurrent call pair*, to describe thread interleavings with their runtime calling contexts as fuzzing feedback. Each concurrent call pair represents a pair of two concurrently-executed functions augmented with their calling contexts. Second, by using an *adjacency-directed mutation* and a *breakpoint control method*, this approach can directionally explore thread interleavings as many as possible. Specifically, the adjacency-directed mutation infers new possible thread interleavings according to already covered thread interleavings, so as to provide a promising direction for thread-interleaving exploration. Then, the breakpoint-control method deterministically attempts to actually cover these new possible thread interleavings at runtime, by adaptively affecting thread scheduling with breakpoints. This method can increase the possibility of covering infrequent thread interleavings generated by the mutation. With the two techniques, this fuzzing approach can effectively cover different thread interleavings with concrete context information, helping discover hard-to-find data races.

Based on this concurrency fuzzing approach, we implement a customized race checker in CONZZER, by using dynamic lockset analysis [66], and it can achieve higher accuracy than existing race checkers like TSan [75]. Benefiting from different thread interleavings and related context information identified by the fuzzing approach, this race checker can effectively and precisely detect data races in real-world programs. We have implemented CONZZER with LLVM [54] to automatically test both user-level applications and kernel-level programs. Besides, to be compatible with traditional input-driven fuzzing process, CONZZER can mutate concurrent call pairs and program inputs together.

Overall, we make the following main contributions:

- We first reveal the major limitations of existing concurrency fuzzing. Then, to solve these limitations, we propose a novel context-sensitive and directional concurrency fuzzing approach, with two new techniques. First, this approach uses a new context-sensitive concurrency-coverage metric *concurrent call pair* as program feedback, to describe thread interleavings with runtime calling contexts. Second, this approach performs directional thread-interleaving exploration, to smartly infer and deterministically cover new possible thread interleavings. With the two techniques, this fuzzing approach can help to discover hard-to-find data races.
- Based on this fuzzing approach, we develop a new concurrency fuzzing framework named CONZZER, to effectively explore thread interleavings and detect data races. To our knowledge, CONZZER is the first systematic concurrency fuzzing framework to directionally explore thread interleavings in different runtime contexts for data-race detection.
- We evaluate CONZZER on 8 user-level applications of the latest versions and 4 kernel-level filesystems in Linux 5.4. It in total reports 95 real data races with no false positive. All these data races are actually triggered in real execution. We identify 75 of these data races to be harmful and send them to related developers, and 44 have been confirmed. We also experimentally compare CONZZER to existing fuzzing tools (such as AFL++, Syzkaller, Razzer and KRACE) with existing race checkers (such as TSan and KCSAN), and CONZZER continuously explores more thread interleavings and finds more real data races missed by these tools.

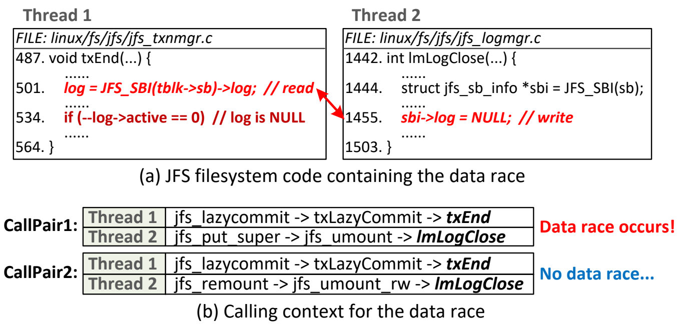
II. BACKGROUND AND MOTIVATION

A. Data Race

The execution of two threads can be interleaved, and thus they can concurrently access the same shared variable. In this case, if at least one thread writes to the variable, its value can be uncertain or even corrupted. To solve this problem, synchronization operations are required to protect the memory accesses to the shared variables in concurrently-executed code. Common synchronization operations are acquiring/releasing locks, setting memory barriers, counting semaphores and so on. Since these synchronization operations reduce program concurrency and decrease program performance, they are supposed to be used in only the code where necessary, e.g., minimizing critical sections. But doing so is difficult for developers in practice, because concurrent situations at runtime can be quite complex and unpredictable. Thus, developers sometimes miss necessary synchronization operations or perform them at incorrect code places, which leads to data races. Data races can cause the program to run abnormally or crash directly. Furthermore, some data races can be even exploited by malicious attackers to cause serious security problems such as privilege escalation [20].

B. Motivating Example

Figure 1 shows a harmful data race found by CONZZER in the *jfs* filesystem in Linux 5.4. Once this data race is triggered, it can cause a null-pointer dereference that crashes the Linux kernel. As shown in Figure 1(a), in the function `txEnd`, the variable `JFS_SBI(tblk->sb)->log` is read and assigned to a local variable `log` on line 501. In the function `lmLogClose`, the variable `sbi->log` is assigned to NULL on line 1455. In our testing, the read operation of `JFS_SBI(tblk->sb)->log` in `txEnd` and the write operation of `sbi->log` in `lmLogClose` are concurrently executed, and the two operations access the same memory area, causing a data race to occur. When the data race is triggered, the read operation can be executed after the write operation, and thus `log` can be NULL, causing a null-pointer dereference when `log->active` is accessed on line 534.


Thread 1

```
FILE: linux/fs/jfs/jfs_txnmgr.c
487. void txEnd(...) {
         ......
501.     log = JFS_SBI(tblk->sb)->log;  // read
         ......
534.     if (--log->active == 0)  // log is NULL
         ......
564. }
```

Thread 2

```
FILE: linux/fs/jfs/jfs_logmgr.c
1442. int lmLogClose(...) {
          ......
1444.     struct jfs_sb_info *sbi = JFS_SBI(sb);
          ......
1455.     sbi->log = NULL;  // write
          ......
1503. }
```

(a) JFS filesystem code containing the data race

CallPair1:	Thread 1	jfs_lazycommit -> txLazyCommit -> ***txEnd***	Data race occurs!
	Thread 2	jfs_put_super -> jfs_umount -> ***lmLogClose***	
CallPair2:	Thread 1	jfs_lazycommit -> txLazyCommit -> ***txEnd***	No data race...
	Thread 2	jfs_remount -> jfs_umount_rw -> ***lmLogClose***	

(b) Calling context for the data race

Fig. 1. A harmful data race in *jfs* filesystem.

This data race was first introduced in Linux 2.6.12 (June 2005) and had existed for about 15 years until CONZZER found it, indicating the difficulty of discovering it. In general, detecting data races is quite challenging, as they often occur in infrequent thread interleavings with specific runtime contexts, which are difficult to trigger, due to non-determinism of thread scheduling. For example, in our testing, the data race in Figure 1 occurs only when the functions `txEnd` and `ImLogClose` are concurrently executed with the calling context *CallPair1* shown in Figure 1(b). When the two functions are concurrently executed with other calling contexts (such as *CallPair2* shown in Figure 1(b)), the data race never occurs. Moreover, the thread interleaving with the calling context *CallPair1* is just infrequently executed in real execution, and thus this data race is hard to trigger at runtime, causing it had existed for a long time, missed by existing race-detection tools though the Linux kernel has been extensively tested. But occurring in an infrequent execution situation does not mean that the data race is less critical. Once knowing the data race, an attacker can transform it into a deterministic vulnerability by intentionally preparing related concurrency workloads [83], [86]. In summary, how to effectively cover infrequent thread interleavings with specific runtime contexts is an important but difficult problem for data-race detection.

C. State of the Art of Concurrency Fuzzing

Nowadays, fuzzing has shown promising results in bug detection and vulnerability discovery for sequential programs. To fuzzing concurrent programs, many developers have used existing fuzzing tools (such as AFL [1], and Syzkaller [70]) with third-party data-race checkers (such as TSan [75] and KCSAN [44]) to detect data races [41], [71]. However, general-purpose fuzzing tools is limited in exploring thread interleavings, due to using *sequential* code coverage as program feedback and neglecting *thread interleavings*.

To improve fuzzing in testing concurrent programs, several recent concurrency fuzzing approaches [8], [37], [53], [76], [81] have been proposed. To perform thread-interleaving exploration, they use several techniques, such as using new concurrency-coverage metrics (e.g., alias instruction pair [81]) as program feedback and randomly adjusting thread priorities [8], [53]. However, these approaches still have two major limitations in testing concurrent programs:

1) Using context-insensitive concurrency-coverage metric. These approaches tune traditional code coverage [8], [37], [53], [76] or use new concurrency-coverage metrics [81] as program feedback. However, they miss the runtime contexts of thread interleavings. On one hand, code coverage only describes sequential-execution situation, and thus it cannot effectively describe thread interleavings though being tuned. On the other hand, as the sole new concurrency-coverage metric of these approaches, the alias instruction pair proposed by KRACE [81] only describes the locations of two concurrently-executed instructions, but neglects their calling contexts, and thus this metric cannot differentiate concurrent-execution situations in different calling contexts. For example in Figure 1, with a context-insensitive concurrency-coverage metric, the pairs of two racy instructions executed in *CallPair1* and CallPair2 are considered identical. But the data race occurs only when the two racy instructions executed in *CallPair1*. Thus, the data race can be missed by concurrency fuzzing in this case.

2) Performing random thread-interleaving exploration. These approaches inject random delays at memory access points [81] or randomly adjust thread priorities at runtime [8], [53], to cover different thread interleavings. However, recent works [12], [74] reveal that such exploration is often inefficient, namely it frequently repeats already covered thread interleavings and misses many infrequent ones. For example in Figure 1, the functions `txEnd` and `ImLogClose` are executed quickly with different calling contexts at runtime, and thus their executions actually have little chance to interleave with the calling contexts *CallPair1*. For this reason, to detect the data race, it is important to delicately schedule *Thread 1* and *Thread 2* during testing. In the evaluation (Section V-D), we try random delay injection in concurrency fuzzing for 24 hours, but the data race is missed.

Due to lacking concrete runtime contexts and missing many infrequent thread interleavings, existing concurrency fuzzing approaches are still limited in discovering hard-to-find data races. Thus, solving these limitations is important to improving concurrency fuzzing in data-race detection.

III. Context-Sensitive and Directional Fuzzing

To address the limitations of existing concurrency fuzzing, we propose a context-sensitive and directional concurrency fuzzing approach to effectively explore thread interleavings:

1) To ensure context sensitivity, this approach uses a new concurrency-coverage metric, *concurrent call pair*, to describe thread interleavings with their runtime calling contexts as fuzzing feedback.

2) To reduce the inefficiency caused by randomness and cover infrequent thread interleavings, this approach performs directional thread-interleaving exploration, which first uses an *adjacency-directed mutation* to infer new possible thread interleavings with already covered thread interleavings and then uses a *breakpoint-control method* to attempt to actually cover these new ones at runtime.

A. Context-Sensitive Concurrent Call Pair

Inspired by existing function-level concurrency-coverage metrics (e.g., concurrent function pair [19] and concurrent method pair [12]), which are context-insensitive, we propose a new metric *concurrent call pair* that considers the calling contexts of concurrently-executed functions:

$$ConCallPair = [CallCtx_a, CallCtx_b] \tag{1}$$

We describe the calling context of a function using its runtime call stack (*CallCtx*), which contains the information of each function call (*CallInfo*) at the call stack (from caller to callee), including the location of this function call (*CallLoc*) and the location of the called function (*FuncLoc*). Specifically, we describe a calling context as:

$$CallCtx = \{CallInfo_1, CallInfo_2, ..., CallInfo_n\} \tag{2}$$
$$CallInfo = [CallLoc, FuncLoc] \tag{3}$$

Based on the above description, the information about each concurrent call pair can be hashed as a key for storage, and the number of times that each pair is covered can be represented as a hash value. In this way, similar to the code coverage stored in AFL [1], concurrent call pairs can be stored as key-value pairs in a hash table:

KEY	Hash(ConCallPair$_1$)	Hash(ConCallPair$_2$)	......	Hash(ConCallPair$_x$)
VALUE	0 - 255	0 - 255	......	0 - 255

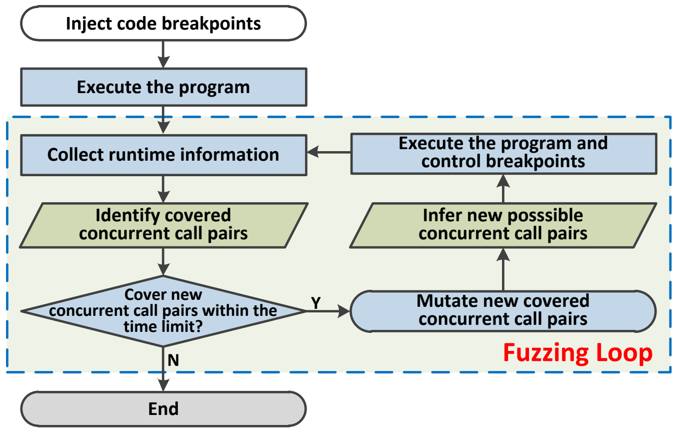

Inject code breakpoints
Execute the program
Collect runtime information
Execute the program and control breakpoints
Identify covered concurrent call pairs
Infer new posssible concurrent call pairs
Cover new concurrent call pairs within the time limit?
Y
N
Mutate new covered concurrent call pairs
Fuzzing Loop
End

Fig. 2. Process of context-sensitive and directional fuzzing.

In fact, concurrently-executed functions and their runtime call stacks are actually decided by program execution, and thus statically determining them is quite difficult. For this reason, concurrent call pairs should be dynamically identified and collected during program execution.

According to our metric, when two functions are concurrently executed in *N* different calling contexts, there will be *N* concurrent call pairs. In this way, *CallPair1* and *CallPair2* in Figure 1(b) can be differentiated. Thus, concurrent call pair is helpful to performing finer-grained concurrency fuzzing and detecting data races in specific runtime contexts.

B. Directional Thread-Interleaving Exploration

We believe that the already covered thread interleavings provide a reasonable direction to infer other possible and infrequent thread interleavings. For example, if the function $Func_A$ is concurrently executed with the function $Func_B$, we can infer that the callees and callers of $Func_A$ are possible to be concurrently executed with $Func_B$. Then, we can deterministically affect thread scheduling to check whether the inferred possible thread interleavings are realistic at runtime.

Based on this basic idea and concurrent call pair, we propose a new directional thread-interleaving exploration method, which contains two parts: an *adjacency-directed mutation* (Section III-B1) to generate new possible thread interleavings with already covered thread interleavings; and a *breakpoint-control method* (Section III-B2) to attempt to actually cover these new possible thread interleavings at runtime.

Figure 2 shows the process of our context-sensitive and directional concurrency fuzzing, which has six basic steps:

(S1) At compile time, it injects a code breakpoint containing time delay at the entry of each function;

(S2) It executes the tested program without enabling any breakpoint, and collects covered concurrent call pairs and thread-execution information;

(S3) After program execution, it adds these covered concurrent call pairs into a global set *pair_set* that records all distinct covered concurrent call pairs;

(S4) It performs the adjacency-directed mutation for these covered concurrent call pairs with thread-execution information, to generate new possible concurrent call pairs;

(S5) It executes the tested program, uses the breakpoint-control method to deterministically and adaptively cover the possible concurrent call pairs, and collects covered concurrent call pairs and thread-execution information;

(S6) After program execution, it identifies new covered concurrent call pairs from these covered concurrent call pairs by comparing with *pair_set*, and iteratively mutates these new ones using thread-execution information again as (S4) does, which constructs a fuzzing loop.

1) Adjacency-Directed Mutation: Our mutation is inspired by an insight that *if two functions are concurrently executed, the functions executed adjacently to them are probably concurrently executed as well.* Based on this insight and concurrent call pair, we propose an inference where new possible thread interleavings can be identified in promising directions, according to the already covered concurrent call pairs:

Inference: if $Func_A$ and $Func_B$ are concurrently executed with the calling contexts $CallCtx_a$ and $CallCtx_b$, and $Func_B$ and $Func_C$ are executed adjacently with the calling context $\mathrm{CallCtx}_b$ and $CallCtx_c$, so $Func_A$ and $Func_C$ are possible to be concurrently executed with the calling contexts $CallCtx_a$ and $CallCtx_c$.

Algorithm 1: Adjacency-Directed Mutation

```
Input: covered_pair_set
Output: possible_pair_set
1  possible_pair_set ← ∅
2  for covered_pair ∈ covered_pair_set do
3      [CallCtx_1, CallCtx_2] ← GetCallCtx(covered_pair)
4      for AdjCallCtx_1 ∈ GetAdjCallCtx(CallCtx_1) do
5          ConCallPair ← GenNewPair(AdjCallCtx_1, CallCtx_2)
6          if ConCallPair ∉ covered_pair_set then
7              AddConCallPair(possible_pair_set, ConCallPair)
8      for AdjCallCtx_2 ∈ GetAdjCallCtx(CallCtx_2) do
9          ConCallPair ← GenNewPair(CallCtx_1, AdjCallCtx_2)
10         if ConCallPair ∉ covered_pair_set then
11             AddConCallPair(possible_pair_set, ConCallPair)
```

Our adjacency-directed mutation is shown in Algorithm 1. The mutation takes the set of covered concurrent call pairs, namely *covered_pair_set* as input, and outputs the set of new possible concurrent call pairs, namely *possible_pair_set*. For each covered concurrent call pair, the mutation gets its two corresponding calling context $CallCtx_1$ and $CallCtx_2$ (lines 2-3). Then, each adjacent calling context $AdjCallCtx_1$ of $CallCtx_1$ is combined with $CallCtx_2$ to generate a new possible concurrent call pair *ConCallPair* (lines 4-5). If *ConCallPair* does not repeat with existing concurrent call pairs, it will be added into *possible_pair_set* (lines 6-7). Similarly, $CallCtx_2$ is also handled in this way (lines 8-11) to generate new possible concurrent call pairs. Note that *AdjCallCtx* is regarded as the adjacent calling context of *CallCtx* if their corresponding functions can be executed adjacently in these calling contexts.

Example. Figure 3 presents an example of our adjacency-directed mutation, which generates four new possible concurrent call pairs and drops two repeated ones. For a given covered concurrent call pair *[*$CallCtx_a$*,* $CallCtx_b$*]*, the mutation

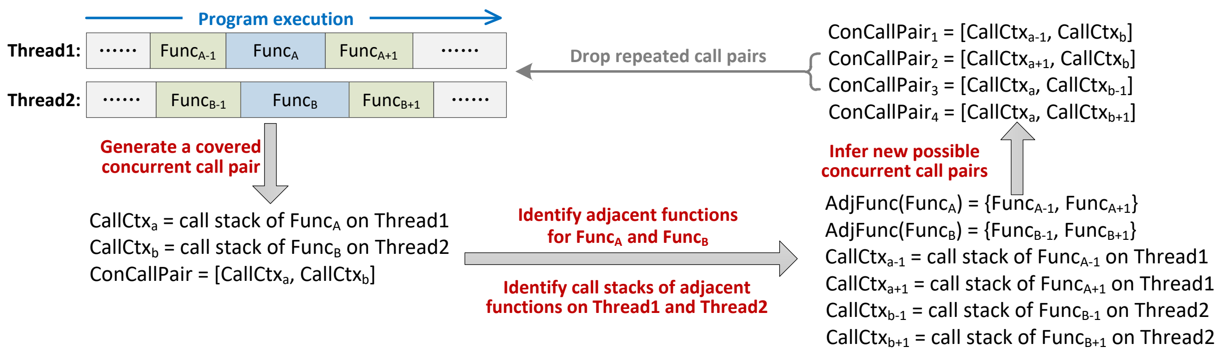

Program execution
Thread1:
Thread2:
FuncA-1
FuncA
FuncA+1
FuncB-1
FuncB
FuncB+1
Generate a covered concurrent call pair
CallCtxa = call stack of FuncA on Thread1
CallCtxb = call stack of FuncB on Thread2
ConCallPair = [CallCtxa, CallCtxb]
Identify adjacent functions for FuncA and FuncB
Identify call stacks of adjacent functions on Thread1 and Thread2
AdjFunc(FuncA) = {FuncA-1, FuncA+1}
AdjFunc(FuncB) = {FuncB-1, FuncB+1}
CallCtxa-1 = call stack of FuncA-1 on Thread1
CallCtxa+1 = call stack of FuncA+1 on Thread1
CallCtxb-1 = call stack of FuncB-1 on Thread2
CallCtxb+1 = call stack of FuncB+1 on Thread2
Infer new possible concurrent call pairs
ConCallPair1 = [CallCtxa-1, CallCtxb]
ConCallPair2 = [CallCtxa+1, CallCtxb]
ConCallPair3 = [CallCtxa, CallCtxb-1]
ConCallPair4 = [CallCtxa, CallCtxb+1]
Drop repeated call pairs

Fig. 3. Example of mutating concurrent call pairs.

first selects their concurrently-executed functions $Func_A$ and $Func_B$, and identifies the runtime information about their threads *Thread1* and *Thread2*. Then, the mutation checks the information about *Thread1* and identifies the two functions $Func_{A-1}$ and $Func_{A+1}$ that are executed adjacently to $Func_A$. the mutation infers that $Func_{A-1}$ and $Func_{A+1}$ would be also likely concurrent with $Func_B$ when they are executed in specific calling contexts. Based on this inference, from the collected information about *Thread1*, the mutation identifies the corresponding call stacks of $Func_{A-1}$ and $Func_{A+1}$, namely $CallCtx_{a-1}$ and $CallCtx_{a+1}$, and then combines each of them with $CallCtx_b$ to generate two new possible concurrent call pairs [$CallCtx_{a-1}$, $CallCtx_b$] and [$CallCtx_{a+1}$, $CallCtx_b$]. Similarly, the mutation also handles $Func_B$ and the runtime information about *Thread2*, to generate another two new possible concurrent call pairs [$CallCtx_a$, $CallCtx_{b-1}$] and [$CallCtx_a$, $CallCtx_{b+1}$]. In total, four new possible concurrent call pairs are generated from the given pair, and two of them are dropped in the comparison with existing covered ones.

2) Breakpoint-Control Method: Because thread scheduling is often non-deterministic during concurrent execution, it is difficult to actually cover specific possible concurrent call pairs (generated by our adjacency-directed mutation) at runtime. To solve this problem, our breakpoint-control method works from two aspects: 1) for each possible concurrent call pair, the method performs deterministic breakpoint control to attempt to cover it; 2) for the whole set of possible concurrent call pairs, the method performs adaptive breakpoint control to attempt to effectively cover them as many as possible.

Deterministic breakpoint control. For a possible concurrent call pair [$CallCtx_p$, $CallCtx_q$], the method aims to cover this pair if the involved functions $Func_P$ and $Func_Q$ can be concurrently executed in related calling contexts, by deterministically controlling the breakpoints. Figure 4 shows the main procedure of deterministic breakpoint control. During program execution, suppose that $Func_P$ is executed and its calling context matches $CallCtx_p$, the method enables the breakpoint of $Func_P$ to perform time delay on the running thread, to wait another function $Func_Q$ to be executed. During time delay, if $Func_Q$ is executed and its calling context matches $CallCtx_q$, indicating the possible concurrent call pair is actually covered at runtime, the breakpoint ends delay and becomes disabled. Otherwise, when the time delay reaches the time limit, indicating the possible concurrent call pair may never be covered in the current running, the breakpoint ends delay and becomes disabled.

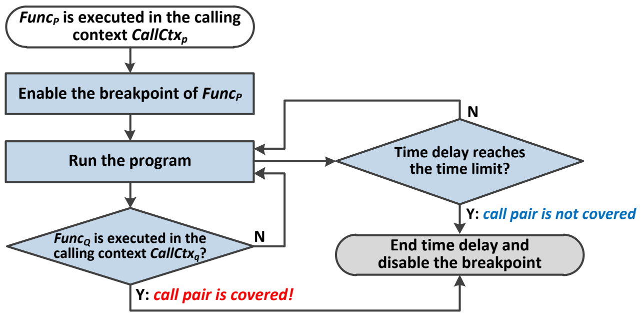

FuncP is executed in the calling context CallCtxp
Enable the breakpoint of FuncP
Run the program
Time delay reaches the time limit?
N
Y: call pair is not covered
End time delay and disable the breakpoint
FuncQ is executed in the calling context CallCtxq?
N
Y: call pair is covered!

Fig. 4. Deterministic breakpoint control.

Indeed, the uncovered case can occur due to two possible reasons. First, the possible concurrent call pair is unrealistic because the involved functions are synchronized. Second, the possible concurrent call pair is indeed realistic but not actually covered in the current running, due to the non-determinism of thread scheduling. For the second reason, the method provides additional chance for the call pair, by running the program several times. The number of running times is decided when performing the adaptive breakpoint control for multiple possible concurrent call pairs, which is introduced as follow.

Adaptive breakpoint control. Our adjacency-directed generates multiple possible concurrent call pairs, and each of them should be properly handled with breakpoint control. If each of these call pairs is handled in each run, the whole running time would be too long; if all of these call pairs are handled in just one run, the possibility of covering one call pair is increased, but too much overhead may be introduced, making the program run abnormally. Thus, it is important to properly select possible concurrent call pairs in each run.

To solve this problem, the method performs adaptive breakpoint control, presented in Algorithm 2. It is analogous to the slow start of TCP congestion control [69] by exponentially increasing the number of possible concurrent call pairs.

The method takes the set of possible concurrent call pairs *possible_pair_set* as input. This method first initializes the number of handling call pairs in each run (*select_num*) to be one (line 1). Then the method enters the main loop which ends when *possible_pair_set* is empty or *select_num* is larger than the number of call pairs in *possible_pair_set* (line 2). In the loop, the method first randomly selects *select_num* call pairs from *possible_pair_set*, and stores them in a set *try_pair_set* (line 3). Then, the method runs the program and perform deterministic breakpoint control for each call pair

Algorithm 2: Adaptive Breakpoint Control

```
Input: $possible\_pair\_set$
1  $select\_num \leftarrow 1$
2  while $possible\_pair\_set \neq \varnothing$ and
     $select\_num \leq Size(possible\_pair\_set)$ do
3      $try\_pair\_set \leftarrow RandSelect(possible\_pair\_set,\ select\_num)$
4      $RunProgramWithBreakpointControl(try\_pair\_set)$
5      $covered\_pair\_set \leftarrow GetCoveredPair()$
6      $common\_pair\_set \leftarrow try\_pair\_set \cap covered\_pair\_set$
7      if $common\_pair\_set \neq \varnothing$ then
8          $DeletPair(possible\_pair\_set, common\_pair\_set)$
9      else
10         $select\_num \leftarrow select\_num \times 2$
```

stored in *try_pair_set*, to attempt to actually cover them (line 4). During program execution, the method collects covered concurrent call pairs in a set *covered_pair_set* (line 5). After program execution, the method compares *try_pair_set* and *covered_pair_set* to get their intersection set *common_pair_set*, which stores the actually covered possible concurrent call pairs in the latest running (line 6). If *common_pair_set* is not empty, the method deletes the call pairs existing in *common_pair_set* from *possible_pair_set* (line 8), indicating that these call pairs are not required to be handled any more. If *common_pair_set* is empty, it indicates that none of the selected call pairs is actually covered, and thus the method doubles *select_num* to provide more chance of covering one call pair in the next running (line 10). Finally, the method checks back the loop condition and prepare to enter the next iteration.

C. Discussion on Alternative Design Choices

When designing our fuzzing approaches, we considered and tried some alternative choices for specific techniques. When selecting the feedback granularity of concurrency fuzzing, we also considered and tried to use concurrent instruction pairs or concurrent basic-block pairs. However, performing instruction-level or basic-block-level dynamic analysis introduces too much runtime overhead in program execution, which greatly degrades the performance of overall fuzzing process. Besides, we observed that enabling context sensitivity on instruction pairs or basic-block pairs causes over-sensitive problems of fuzzing, namely too many test cases are labeled as interesting seeds. To further confirm it, in Section VI-A, we implement an alternative tool named *inst-fuzzer* that uses concurrent instruction pairs as feedback for random thread-interleaving exploration without considering the runtime contexts of instruction pairs, and experimentally compare to it. For these reasons, we finally chose to perform function-level analysis and use concurrent call pairs as program feedback in our concurrency fuzzing approach.

Moreover, when performing adaptive breakpoint control in Algorithm 2, we also considered and tried two possible strategies. First, we tried to enable all possible concurrent call pairs for deterministic breakpoint control in program execution (namely dropping line 3 and replacing *try_pair_set* with *possible_pair_set* on line 4). Second, we tried to linearly increase *select_num* if no expected concurrent pair is covered at runtime (namely using "*select_num*++" on line 10). However, these two possible strategies heavily degrade the performance of fuzzing process. For the first strategy, enabling all possible concurrent call pairs introduces too much runtime overhead of program execution; for the second strategy, linearly increasing the number of possible concurrent call pairs reduces the runtime overhead of program execution, but largely increases the execution number of the tested program, which heavily decreases fuzzing efficiency.

D. Beneficial Effect in Data-Race Detection

Our context-sensitive and directional concurrency fuzzing approach can help to discover hard-to-find data races, from two aspects. On the one hand, with a new context-sensitive concurrency metric, namely concurrent call pair, our approach can effectively explore thread interleavings in different runtime contexts, to help find data races only triggered in specific runtime contexts. On the other hand, with a new adjacency-directed mutation and breakpoint-control method, our approach can efficiently explore infrequent thread interleavings, to help find data races only triggered in infrequent execution situations. To our knowledge, our approach is the first concurrency fuzzing approach to perform context-sensitive and directional thread-interleaving exploration, and it can help to discover many hard-to-find data races missed by existing concurrency fuzzing approaches [8], [37], [53], [76], [81] that use context-insensitive and random thread-interleaving exploration.

IV. CONZZER FRAMEWORK AND IMPLEMENTATION

Based on our context-sensitive and directional concurrency fuzzing approach, we develop a new fuzzing framework named CONZZER, to effectively explore thread interleavings and detect data races. We implement CONZZER with Clang 9.0 [13] and perform analysis on the LLVM bytecode. To be compatible with traditional input-driven fuzzing process, CONZZER is able to mutate concurrent call pairs and program inputs together. Figure 5 shows its architecture that has five parts:

- ***Code analyzer.*** It compiles and instruments the program code, and finally generates an executable tested program.
- ***Runtime analyzer.*** It executes the tested program with the generated program inputs, records runtime information of the program, and performs breakpoint control according to generated possible concurrent call pairs.
- ***Call-pair generator.*** It mutates covered concurrent call pairs to generate new possible concurrent call pairs, according to thread-execution information.
- ***Input generator.*** It exploits input-driven fuzzing process to mutate and generate new inputs, according to code coverage.
- ***Race checker.*** It analyzes the runtime information of thread interleavings covered by our concurrency fuzzing approach to detect data races.

A. Customized Race Checker

The race checker heavily affects race-detection accuracy, so selecting a precise race checker is important to CONZZER. We intended to use a third-party data-race checker like TSan [75], which is used by many existing fuzzing approaches [8], [76]. However, when testing real-world programs, we find that TSan report many false positives (as shown in our comparison experiments in Section VI-A and Table VI), due to neglecting special synchronization primitives such as message queue and condition variable [67]. Thus, to improve race-detection

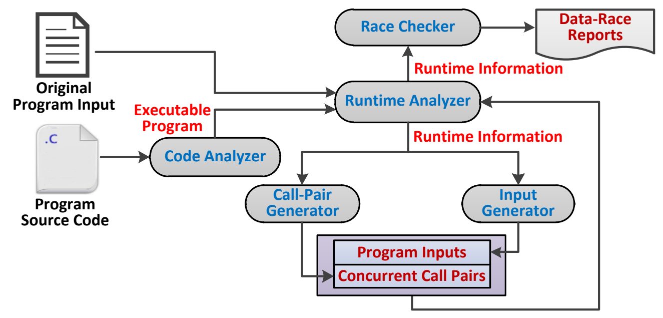


Fig. 5. Overall architecture of CONZZER.

accuracy, we implement a customized race checker based on our concurrency fuzzing approach, instead of using TSan.

Overall, this checker performs two steps: (S1) using dynamic lockset analysis [66] to detect possible data races during fuzzing, and (S2) performing validation of these possible data races to detect real ones after fuzzing.

(S1) During program execution, our checker maintains a lockset for each running thread, and records the runtime information about shared-variables accesses and related context information. Once our checker identifies two functions are concurrently executed at runtime, it dynamically performs lockset analysis for the memory accesses in these two functions to detect possible data races. For each possible data race, our checker records its racy instructions and related concurrency call pairs identified by our concurrency fuzzing approach.

(S2) For each reported possible data race, our checker injects breakpoints at its racy instructions and executes the program again. During program execution, our checker maintains the calling context of each thread, and dynamically controls the breakpoints to attempt to concurrently execute racy instructions accessing to the same variable with the recorded concurrency call pairs. If the attempt succeeds, this possible data race is identified to be real, as it is actually triggered.

The primary advantage of our checker is no false positive, because all the reported data race can be actually triggered in the runtime validation. But it can introduce false negatives for two main reasons. First, the online lockset analysis only detects possible data races in functions that are actually concurrently-executed. If two functions containing real data races are not concurrently-executed in testing, it will miss these data races. To reduce such false negatives, our concurrency fuzzing approach can efficiently cover different thread interleavings. Second, due to non-determinism of thread interleaving, the runtime validation may fail to trigger some real data races during execution. To reduce such false negatives, we suggest to perform the runtime validation multiple times.

Note that besides our customized race checker, other third-party data-race checkers can be also used in CONZZER. For example in Section VI-A, we have used TSan in CONZZER to test user-level applications, without any change of TSan and our fuzzing approach.

B. *Phases of* CONZZER

P1: Code instrumentation. In this phase, the code analyzer identifies and instruments three kinds of places in the program's LLVM bytecode files:

- *Calls to lock-acquiring/-release functions.* They are instrumented to maintain the locksets of memory accesses for each running thread.
- *Function definitions and calls.* The entry and exit of each function and each function call are instrumented, to collect concurrent call pairs and calling-context information.
- *Memory accesses to possible shared variables.* They are instrumented by our customized race checker for runtime monitoring and data-race detection. For identifying these accesses before fuzzing, we implement a static analysis referring to [11], to automatically analyze program code.

Through compiling the instrumented LLVM bytecode files, CONZZER generates the executable tested program for runtime fuzzing in the next phase.

P2: Runtime fuzzing. In this phase, through the instrumented code, CONZZER performs our context-sensitive and directional concurrency fuzzing approach to effectively explore thread interleavings by identifying and covering concurrent call pairs. During fuzzing, CONZZER uses dynamic lockset analysis to detect possible data races. After fuzzing, our customized race checker performs runtime validation of all possible data races, and finally reports real ones. The fuzzing process and data-race detection are automated.

Note that though our concurrency fuzzing explores concurrent call pairs not concurrent instruction pairs, the race detection of CONZZER is performed at instruction level for high accuracy. On one hand, if two instructions are concurrently executed, their caller functions should be concurrently executed. Thus, identifying concurrent call pairs is important to identifying concurrent instruction pairs. On the other hand, our customized race checker performs dynamic lockset analysis of each instruction in the identified concurrent call pairs. Besides, CONZZER uses common lock-acquiring/-release functions (like `pthread_mutex_lock`) described in POSIX thread libraries and Linux official documents. Missing some customized lock-acquiring/-release functions would only cause false positives in dynamic lockset analysis, which are however dropped in runtime validation of our customized race checker.

C. *Compatibility with Input-Driven Fuzzing*

CONZZER is able to collaborate with traditional input-driven fuzzing, because concurrent call pairs explored by our concurrency fuzzing approach are orthogonal to program inputs from files and system calls. Specifically, for CONZZER, a test case consists of two parts, namely covered concurrent call pairs and program inputs. In seed identification, CONZZER considers a test case as an interesting seed when new concurrent call pairs or new branches are covered at runtime. In seed selection, CONZZER gives more priority to the seeds that cover more new concurrent call pairs, to preferentially explore new thread interleavings. In seed mutation, CONZZER mutates and generates concurrent call pairs and program inputs together. At runtime, CONZZER collects and tries to cover concurrent call pairs via concurrency fuzzing, and collects code coverage via input-driven fuzzing. We have implemented input-driven fuzzing process in CONZZER, by referring to AFL [1] and Syzkaller [70]. We believe that recent input-driven fuzzing approaches (such as AFLFast [4] and Angora [9]) can also collaborate with CONZZER to improve fuzzing performance.

V. Evaluation

A. Experimental Setup

We evaluate CONZZER on 8 user-level C applications of the latest versions as of our evaluation and 4 kernel-level filesystems in Linux 5.4. We select these 12 programs, as they are open-source and widely used, and can run with multiple threads. The information of these programs is listed in Table I (their source-code lines are counted using CLOC [14]). We run the evaluation on a regular PC with an eight-core Intel i7-3770@3.40G processor and 16GB physical memory.

TABLE I. Basic information of the tested programs.

Program	Description	Version	LOC
sqlite	SQL database engine	v3.30.1	416K
memcached	Memory-objected caching systems	v1.6.5	21K
x264	Video stream encoder	v0.157.x	72K
ffmpeg	Solution for media processing	n4.3-dev	1.1M
aget	Download accelerator	v0.4.1	833
axel	Download accelerator	v2.17.6	3K
pigz	Data compression program	v2.4	6K
xz	Data compression program	v5.2.4	24K
btrfs	Linux BTRFS filesystem	Linux 5.4	98K
jfs	Linux JFS filesystem	Linux 5.4	18K
xfs	Linux XFS filesystem	Linux 5.4	94K
reiserfs	Linux ReiserFS filesystem	Linux 5.4	21K

B. Runtime Testing

To show the effectiveness of CONZZER with fixed inputs, we disable input-driven fuzzing and run common workloads in testing. For the 8 user-level applications, we run their official multi-thread test suites; for the 4 kernel-level filesystems, we run a well-known filesystem benchmark *iozone* v3.429 [36]. Following the recommendations of [46], we use CONZZER to fuzz each tested program five times, and set the time limit of each fuzzing as 24 hours. We count the found data races by the locations of racy instructions.

Table II shows the results. The columns *"Cover"*, *"Gen"* and *"Real"* respectively show the numbers of covered concurrent call pairs, possible concurrent call pairs generated by our adjacency-directed mutation and possible concurrent call pairs actually covered at runtime. The columns *"Possible"*, *"Final"* and *"Harmful"* respectively show the numbers of possible data races reported by our customized race checker, final data races reported by CONZZER and harmful data races manually identified by us. From the results, we make some observations:

Thread-interleaving coverage. CONZZER generates many new concurrent call pairs that are actually covered at runtime. Specifically, these new concurrent call pairs account for 11% of all generated concurrent call pairs, indicating that our adjacency-directed mutation is useful to inferring new realistic concurrent call pairs. Through these new concurrent call pairs, CONZZER actually covers many infrequent thread interleavings during execution.

Found data races. CONZZER finally reports 95 data races (the detailed reports are shown in Appendix I), from 872 possible ones reported by our customized race checker. All these 95 data races are real and actually triggered at runtime, and thus CONZZER has no false positive in data-race detection. We manually check them, and identify 75 harmful ones whose racy variables can cause security problems like DoS, data corruption

TABLE II. Testing results.

Program	Concurrent call pair			Data race		
	Cover	*Gen*	*Real*	*Possible*	*Final*	*Harmful*
sqlite	66.0M	480.7K	40.4K	43	6	3
memcached	18.0K	9.6K	1.6K	53	12	12
x264	187.6K	67.2K	5.7K	234	4	3
ffmpeg	714.5K	149.1K	10.7K	139	10	10
aget	31	38	11	14	3	3
axel	572	468	62	9	1	1
pigz	956	1354	85	15	0	0
xz	2762	3139	377	6	0	0
btrfs	13.1M	362.6K	48.8K	115	34	27
jfs	515.0K	163.2K	26.1K	105	12	10
xfs	6.7M	330.4K	44.7K	133	8	5
reiserfs	630.5K	148.6K	13.6K	6	5	1
Total	87.9M	1.7M	192.1K	872	95	75

and undefined behaviors (the details are shown in Section V-C). For the 20 benign data races, they are deliberately introduced for concurrency execution according to code annotations, or their racy variables are just used for debugging and logging. We report the 75 harmful data races to related developers; 44 of them (15 in applications and 29 in filesystems) have been confirmed. We are still waiting for the response of the remaining ones. Among the 44 confirmed races, 19 of them have been fixed by related developers; the developers have not found proper ways to fix the remaining 25 races, indicating the difficulty of fixing data races in practice.

Process of reporting harmful data races. After we reported the 75 harmful data races, only 10 data races were immediately confirmed by related developers, and 34 data races were finally confirmed through our further explanation and discussion with the developers. According to the above experience, we find that developers often have insufficient understanding about complex and infrequent concurrent situations, and thus data races are hard to avoid during software development. Moreover, many data races are difficult to fix, as synchronizing memory accesses without performance degradation requires careful modification and substantial tests. Thus, it often takes much time to fix a concurrency issue like data race [55].

Discussion of benign data races. We randomly select ten benign data races found by CONZZER to discuss with related developers. They explain that the related racy variables have low impact on core functionalities, and thus race conditions are allowed at runtime. Moreover, adding locks to protect these variables may also degrade program performance. Thus, the developers tend not to handle or fix these benign data races.

Data-race features. By reviewing the 95 data races found by CONZZER, we have three interesting features:

1) 77 data races occur on the accesses to data structure fields stored in heap memory, and only 18 data races occur on the accesses to global variables. Indeed, compared to global variables, identifying whether two variables stored in heap memory are identical is more difficult, because their alias relationships may be implicit from their names. The harmful data race shown in Figure 1 is such an example, and the alias relationship of the racy variables `JFS_SBI(tblk->sb)->log` and `sbi->log` is implicit from their names. By checking the racy variables of all found data races, we find that 16 data races involve such implicit alias relationships.

2) 53 data races involve function-pointer calls in their call stacks. Without exact runtime information, it is often hard to

TABLE III. SECURITY IMPACT OF FOUND DATA RACES.

Program	*Harmful / Benign*	*DoS*	*Data Corruption*	*Undefined Behavior*
sqlite	3 / 3	0	3	0
memcached	12 / 0	0	0	12
x264	3 / 1	1	1	1
ffmpeg	10 / 0	0	10	0
aget	3 / 0	0	2	1
axel	1 / 0	0	1	0
pigz	0 / 0	0	0	0
xz	0 / 0	0	0	0
btrfs	27 / 7	12	4	11
jfs	10 / 2	5	5	0
xfs	5 / 3	2	0	3
reiserfs	1 / 4	0	1	0
Total	75 / 20	20	27	28

correctly identify the targets of function-pointer calls. Most existing static-analysis techniques either discard such cases [2], [22], [32], [34] or identify many incorrect functions [35], [48], [57], and thus they are very likely to miss such data races.

3) 5 data races occur in program initialization or finalization processes. These processes are often unexpected to be executed concurrently with the program's main process, so developers may neglect the synchronization in these processes. But in fact, data races in these processes are often dangerous, because such data races may lead to crash and memory corruption when racy data is incorrectly initialized or finalized. The 3 harmful data races that directly cause null-pointer dereferences and data corruption are such examples.

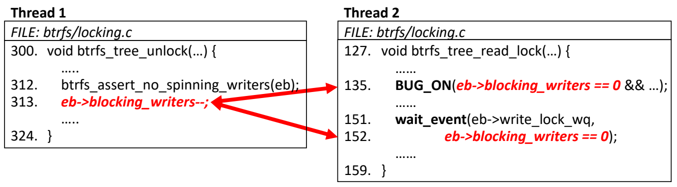
False positives and negatives. CONZZER reports no false positive, as its reported data races can be actually triggered. But CONZZER may miss some real data races for three reasons. First, CONZZER requires proper workloads and program inputs to cover concurrency code. Second, for given workloads and program inputs, CONZZER still cannot cover all possible thread interleavings, due to the non-determinism of thread scheduling. Finally, our customized race checker only detects data races in the functions that are concurrently executed during fuzzing, and it may fail to trigger some real data races in runtime validation, due to the non-determinism of thread scheduling.

C. Security Impact of Found Data Races

We manually review the 95 data races found by CONZZER to estimate their security impact. As the 20 benign data races do not cause security problems, we focus on analyzing the 75 harmful data races. The results are shown in Table III.

First, 20 harmful data races can cause denial of service (DoS). Specifically, 2 races can cause program crashes; 6 races can cause unexpected terminations; and 12 races can cause program hangs by overwriting variables for waiting operations. We use several examples to demonstrate the security impacts.

Examples of DoS. Figure 1 shows a crashing case. The data race causes a null-pointer dereference and thus can crash the *jfs* filesystem. Figure 6 presents two harmful data races that can cause unexpected termination and program hangs in the *btrfs* filesystem. In the function `btrfs_tree_unlock`, the variable `eb->blocking_writters` is decremented on line 313. In the function `btrfs_tree_read_lock`, the variable `eb->blocking_writters` is read on line 135 and line 152, to be compared with zero in the conditions of unexpected



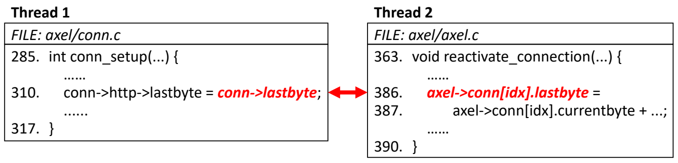
Thread 1

FILE: *btrfs/locking.c*
```
300.  void btrfs_tree_unlock(...) {
          .....
312.      btrfs_assert_no_spinning_writers(eb);
313.      eb->blocking_writers--;
          .....
324.  }
```

Thread 2

FILE: *btrfs/locking.c*
```
127.  void btrfs_tree_read_lock(...) {
          ......
135.      BUG_ON(eb->blocking_writers == 0 && ...);
          ......
151.      wait_event(eb->write_lock_wq,
152.              eb->blocking_writers == 0);
          ......
159.  }
```

Fig. 6. A harmful data race found in *btrfs*.



Thread 1

FILE: *axel/conn.c*
```
285.  int conn_setup(...) {
          ......
310.      conn->http->lastbyte = conn->lastbyte;
          ......
317.  }
```

Thread 2

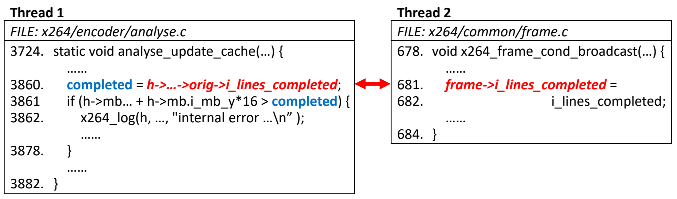
FILE: *axel/axel.c*
```
363.  void reactivate_connection(...) {
          ......
386.      axel->conn[idx].lastbyte =
387.          axel->conn[idx].currentbyte + ...;
          ......
390.  }
```

Fig. 7. A harmful data race found in *axel*.

termination (`BUG_ON`) and waiting operation (`wait_event`), respectively. If `eb->blocking_writters` is decremented to zero on line 313 in `btrfs_tree_unlock`, and then it is read on line 135 in `btrfs_tree_read_lock`, `BUG_ON` can be triggered to terminate the filesystem execution. If `eb->blocking_writters` becomes zero, and this variable is always decremented on line 313 in `btrfs_tree_unlock` before being read on line 151 in `btrfs_tree_read_lock`, `wait_event` will always cause the related filesystem thread to sleep. The two data races have been fixed by the developers.

Second, 27 harmful data races can cause data corruption and manipulation, because they change critical variables about data processing. Depending on how these critical variables are used, various security issues can occur.

Examples of data corruption and manipulation. Some data races can manipulate the content of downloaded files (for aget and axel), which can be exploited by attackers to inject malicious code persistently; some data races can interfere with the encoding process for media processors (for x264 and ffmpeg), which can be exploited by attackers to manipulate the video content displayed to users, leading to attacks such as clickjacking and ”what you see is not what you get”. Figure 7 presents such a data race in *axel*. The variable `conn->lastbyte` in the function `conn_setup` and the variable `axel->conn[idx].lastbyte` in the function `reactivate_connection` can be identical, when the two functions are concurrently executed in two threads. In this case, a data race occurs when the read to `conn->lastbyte` and the write to `axel->conn[idx].lastbyte` are concurrently performed. Once this data race is triggered, `conn->http->lastbyte` in `conn_setup` possibly gets an incorrect value from `conn->lastbyte`, which can corrupt the downloaded file. By exploiting this race, the attacker can control the value of `conn->lastbyte` to modify the content of the downloaded file and inject malicious code. This data race has been fixed by the developers.



Thread 1

FILE: *x264/encoder/analyse.c*
```
3724.  static void analyse_update_cache(...) {
           ......
3860.      completed = h->...->orig->i_lines_completed;
3861       if (h->mb... + h->mb.i_mb_y*16 > completed) {
3862.          x264_log(h, ..., "internal error ...\n" );
               ......
3878.      }
           ......
3882.  }
```

Thread 2

FILE: *x264/common/frame.c*
```
678.  void x264_frame_cond_broadcast(...) {
          ......
681.      frame->i_lines_completed =
682.                           i_lines_completed;
          ......
684.  }
```

Fig. 8. A harmful data race found in *x264*.

TABLE IV. RESULTS OF SENSITIVITY ANALYSIS.

Program	Normal Running		Random delay		Insensitive		CONZZER	
	Pair	*Race*	*Pair*	*Race*	*Pair*	*Race*	*Pair*	*Race*
sqlite	59.2M	6	64.4M	6	53.9M	6	66.0M	6
memcached	13.3K	10	14.4K	10	15.4K	10	18.0K	12
x264	161.6K	3	162.3K	3	177.6K	3	187.6K	4
ffmpeg	338.7K	1	355.2K	1	483.4K	3	714.5K	10
aget	31	3	31	3	31	3	31	3
axel	557	1	559	1	567	1	572	1
pigz	932	0	954	0	812	0	956	0
xz	2651	0	2651	0	2600	0	2762	0
btrfs	9.1M	30	9.8M	30	5.5M	27	13.1M	34
jfs	421.2K	8	445.4K	8	485.0K	10	515.0K	12
xfs	6.1M	8	6.5M	8	4.6M	6	6.7M	8
reiserfs	505.8K	5	513.5K	5	406.1K	1	630.5K	5
Total	75.8M	75	82.1M	75	65.6M	70	87.9M	95

Finally, 28 harmful data races are less obvious in security impact, but at least can cause undefined behaviors because they can manipulate racy variables that decide critical control flows.

Example of undefined behavior. Figure 8 shows such a data race in *x264*. In the functions `analyse_update_cache` and `x264_frame_cond_broadcast`, two variables containing the field `i_lines_completed` are identical in our testing. Once the data race is triggered, the variable `complete` in `analyse_update_cache` can be assigned with an incorrect value, and the condition on line 3861 in `analyse_update_cache` can be satisfied, causing *x264* to report an internal error that cause program malfunction. This data race has been fixed by the developers.

D. Sensitivity Analysis

The core of CONZZER is our context-sensitive and directional concurrency fuzzing approach. To validate the value of this approach, we substitute it with three testing methods in data-race detection:

- *Normal running:* just repeatedly running the tested programs with their official multi-thread test suites.
- *Randomly running:* repeatedly running the tested programs with their official multi-thread test suites, by injecting random delay (0-32ms) in each executed function.
- *Insensitive:* substituting our context-sensitive concurrency-coverage metric *concurrent call pair* with a context-insensitive metric *concurrent function pair* [19] without considering calling context, to implement a context-insensitive concurrency fuzzing method, and then using it to fuzzing the tested programs with their official multi-thread test suites. This method is used to validate the value of considering context sensitivity in concurrency fuzzing by using concurrent call pair as program feedback.

We apply these three substitution methods with our customized race checker to testing the 12 programs in Table I. We use each substitution method to test each program five times, and set the time limit of each fuzzing as 24 hours, as recommended in [46]. Table IV shows the results.

CONZZER outperforms the three substitution methods in covering concurrent call pairs. On average for each program, it respectively covers 24%, 19% and 30% more concurrent call pairs than the normal, random-delay, and context-insensitive fuzzing methods. These additional concurrent call pairs are not covered by the three substitution methods, because they all infrequently occur at runtime.

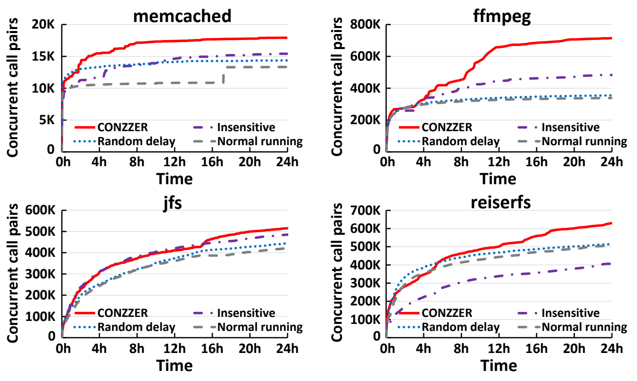


Fig. 9. Growth of covered concurrent call pairs.

Due to the additional concurrent call pairs, CONZZER finds 20 data races missed by the normal-running and random-delay methods, and finds 25 data races missed by the context-insensitive fuzzing method. These additional races are hard to find, as they occur only in infrequent thread interleavings missed by the three substitution methods. Moreover, all of these additional races are harmful. The data race causing null-pointer dereference in Figure 1 is such an example, which had existed for about 15 years, indicating the difficulty of finding it. The results prove that our context-sensitive and directional concurrency fuzzing is effective in covering infrequent thread interleavings and discovering hard-to-find data races.

We also observe that the context-insensitive fuzzing method misses 9 data races found by the normal-running and random-delay methods. Indeed, as a concurrent function pair can be covered in different runtime calling contexts, the context-insensitive fuzzing method performs delay in all these cases, which largely slows down each test. Thus, within the same testing time, it misses some thread interleavings covered by the normal-running and random-delay methods. But the context-insensitive fuzzing method also finds 4 data races missed by the normal-running and random-delay methods, as it benefits from our adjacency-directed mutation and breakpoint-control method to cover some infrequent thread interleavings missed by the two methods, even though each test is slowed down.

By analyzing the growth of covered concurrent call pairs along with the testing time, we find that the normal-running, random-delay and context-insensitive fuzzing methods cover new concurrent call pairs effectively but only in earlier tests, and hardly cover new ones in the later tests. By contrast, CONZZER continuously covers new concurrent call pairs in the later tests, thanks to our concurrency fuzzing approach. We select four tested programs *memcached*, *ffmpeg*, *jfs* and *reiserfs* as examples, and show their results in Figure 9. The completed results are shown in Figure 12 in Appendix II.

VI. COMPARISON TO EXISTING FUZZING TOOLS

A. AFL++, Syzkaller and Instruction-Level Fuzzer

To show the advantages of CONZZER over existing fuzzing tools on concurrency fuzzing and race detection, we compare CONZZER to two state-of-the-art and open-source fuzzing tools, AFL++ [24] and Syzkaller [70]. As AFL [1] is outdated and considered obsolete, we use AFL++ here. We disable the CPU affinity of AFL++ to avoid that multiple threads of the tested programs are bound to single CPU. To understand

TABLE V. COMPARISON OF CONCURRENT CALL PAIRS.

Program	AFL++	Syzkaller	inst-fuzzer	CONZZER
x264	151.9K	-	171.3K	200.6K
ffmpeg	636.6K	-	667.3K	772.8K
pigz	535	-	577	977
xz	726	-	1460	3786
lbzip2	1062	-	893	1371
pbzip2	629	-	1004	1223
libwebp	307	-	323	327
ImageMagick	6542	-	6672	7683
btrfs	-	12.7M	9.5M	15.4M
jfs	-	471.8K	318.2K	536.9K
xfs	-	2.6M	5.1M	7.1M
reiserfs	-	187.8K	252.2K	679.1K

the advantages of novel techniques over alternative choices in CONZZER, we implement a tool named *inst-fuzzer* that uses concurrent instruction pairs as feedback for random thread-interleaving exploration, without considering the runtime contexts of instruction pairs. This tool is similar to KRACE [81], which uses alias instruction pairs (identical to concurrent instruction pairs) as feedback and injects random delays.

We evaluate CONZZER and inst-fuzzer on both user-level applications and kernel-level programs. As AFL++ can only test user-level applications and Syzkaller can only test kernel-level programs, we use them to test different programs in the comparison. Among the 12 tested programs in Table I, the inputs of *sqlite*, *memcached*, *aget* and *axel* are not simple files, so generating their inputs needs to consider many syntactic and semantic requirements, which is a separate research problem that is not considered by this paper. For example, sqlite receives well-formatted and semantically-correct SQL statements as inputs. Generating such inputs is quite difficult for existing AFL-like fuzzing and requires much effort to modify its code. Thus, we do not test the four programs when comparing to AFL++. Instead, we use AFL++ to test 4 additional user-level applications (*lbzip2*, *pbzip2*, *libwebp* and *ImageMagick*) and 4 original ones (*ffmpeg*, *x264*, *pigz* and *xz*); we also use Syzkaller to test 4 kernel-level filesystems in Table I. Following the recommendations of [46], we run each fuzzing tool five times for each program, and set the time limit of each fuzzing as 24 hours. For fair comparison, we enable input-driven fuzzing in CONZZER to test these 12 programs.

Thread-interleaving coverage. To specifically compare the ability of covering thread interleavings, we disable our customized race checker in CONZZER and just collect the covered concurrent call pairs of the three frameworks. Table V shows the comparison results. On average for each program, CONZZER covers 88%, 118% and 58% more concurrent call pairs than AFL++, Syzkaller and inst-fuzzer, respectively. In the experiment, CONZZER has a lower fuzzing throughput (139 execs/min) than AFL++ (262 execs/min), due to the instrumented code and time delays in breakpoint control. Even so, CONZZER still covers many infrequent thread interleavings missed by AFL++ and Syzkaller, indicating its effectiveness in thread-interleaving exploration for fuzzing. Though inst-fuzzer uses concurrent instruction pairs as concurrency-coverage metric, it misses many infrequent thread interleavings covered by CONZZER due to context-sensitivity loss. Moreover, collecting concurrent instruction pairs requires instruction-level monitoring, which introduces more runtime overhead than CONZZER.

Coverage growth. We measure the growth of covered concurrent call pairs along with testing time. Figure 10 shows the results for 8 of the tested programs, and the completed results are shown in Figure 13 in Appendix II. These tools quickly cover new concurrent call pairs in earlier tests, but compared to AFL++, Syzkaller and inst-fuzzer, CONZZER continuously covers more new concurrent call pairs in the later tests.

TABLE VI. COMPARISON OF FOUND DATA RACES.

Program	AFL++ +TSan	Syzkaller +KCSAN	inst-fuzzer +Checker	CONZZER +Checker	CONZZER +TSan
x264	3/167	-	4/4	4/4	4/113
ffmpeg	4/16	-	7/7	10/10	10/22
pigz	0/0	-	0/0	0/0	0/0
xz	0/0	-	0/0	0/0	0/0
libzip2	0/0	-	0/0	0/0	0/0
pbzip2	0/0	-	0/0	0/0	0/0
lbwebp	0/0	-	0/0	0/0	0/0
ImageMagick	9/174	-	8/8	15/15	15/282
btrfs	-	0/0	22/22	34/34	-
jfs	-	1/1	8/8	12/12	-
xfs	-	0/0	2/2	8/8	-
reiserfs	-	0/0	5/5	5/5	-

Found data races. To compare the ability of finding data races, we use two mature data-race checkers TSan [75] and KCSAN [44] with AFL++ and Syzkaller, respectively, and enable our customized race checker with CONZZER and inst-fuzzer. Table VI shows the detection results. CONZZER finds all data races found by AFL++ with TSan, Syzkaller with KCSAN and inst-fuzzer, and additionally finds 13, 58, and 32 real data races missed by the three tools, respectively, because CONZZER covers significantly more concurrent call pairs which infrequently occur during execution.

Using TSan in CONNZER. As CONZZER can conveniently support third-party race checkers, we use TSan in CONZZER to substitute our customized race checker for testing user-level applications (CONZZER +TSan), without any change of TSan and our fuzzing approach. By comparing the results of CONZZER and AFL++ with TSan in Table VI, we observe that with CONZZER, TSan additionally finds 13 real data races missed by using AFL++, because CONZZER covers the related infrequent thread interleavings. In this case, TSan finds all real data races found by CONZZER in the tested applications, but it reports more false positives.

Precision of race detection. Figure 11(a) presents that CONZZER produces less false positives than AFL++ with TSan in our experiment. Indeed, TSan uses a hybrid of happens-before-relation inference and lockset analysis, to drop false positives of pure lockset analysis and false negatives of pure happens-before-relation inference. But TSan still reports some false positives caused by neglecting special synchronization primitives such as message queue and condition variable [67]. By contrast, our customized race checker performs runtime validation of possible data races with breakpoint control, which can eliminate all false positives. In addition, as TSan also considers happens-before relation in race detection, it requires to cover different thread interleavings to reduce false negatives, which is similar to our customized race checker. This explains why AFL++ with TSan misses 13 data races found by CONZZER but CONZZER with TSan finds these races.

Figure 11(a) also presents that CONZZER has less false negatives than Syzkaller with KCSAN. KCSAN is a watchpoint-based checker that observes concurrent memory accesses. To increase the possibility of finding data races, KCSAN performs

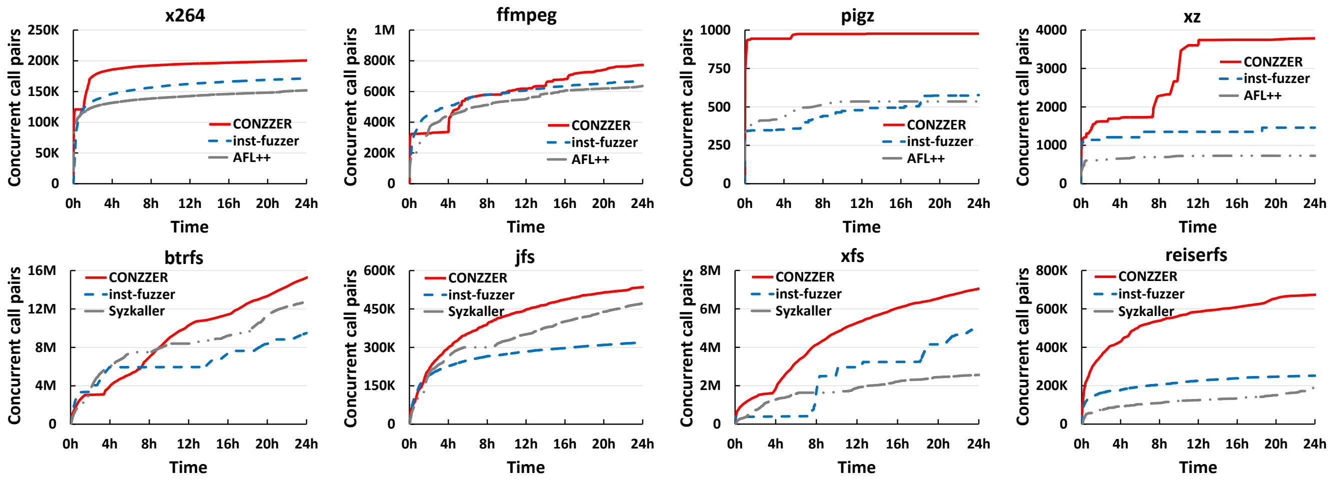


Fig. 10. Growth of covered concurrent call pairs in comparison.

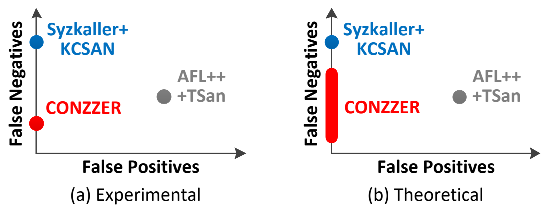


Fig. 11. Precision of data-race detection in comparison.

random delay in the watchpoint of each monitored instruction, but catching two concurrently-executed instructions that have race conditions is still difficult in this way. By contrast, our customized checker extends the detection scope to all the memory accesses in concurrently-executed functions, which can find many data races missed by KCSAN. Moreover, our concurrency fuzzing approach covers more thread interleavings than Syzkaller, which can help find more data races.

Indeed, TSan just reports possible data races using lockset analysis and happens-before relation, without validating these data races at runtime; while our customized race checker performs runtime validation of possible data races, and thus it may miss some real data races when they are not triggered in runtime validation, due to the non-determinism of thread scheduling. As a result, in some cases, TSan is able to find real data races missed by our customized checker. For this reason, and considering that our concurrency fuzzing approach can explore more thread interleavings than AFL++, it is difficult to determine whether CONZZER have less or more false negatives than AFL++ with TSan in theory, as shown in Figure 11(b).

B. Concurrency Fuzzing Approaches

Razzer [37]. It first uses static analysis to identify possible racy instructions, and then injects code breakpoints at possible racy instructions to try to trigger related race bugs at runtime. CONZZER has three significant differences from Razzer:

1) The static analysis of Razzer is not accurate enough and fails to identify real racy instructions involving complex cases (such as alias relationships and function-pointer calls), causing its dynamic analysis to miss many real data races. By contrast, CONZZER uses a context-sensitive and directional concurrency fuzzing approach to effectively explore infrequent thread interleavings, helping discover hard-to-find data races.

2) Razzer finds only memory bugs and warnings caused by race conditions, and it does not use any race checker. However, many harmful data races never cause memory bugs. The harmful data race shown in Figure 7 is such an example, and it can cause malfunction not memory bug. By contrast, CONZZER focuses on detecting data races, without checking whether they can cause memory bugs or not.

3) Razzer can test only OS kernels with a modified virtual machine. By contrast, CONZZER can test both kernel-level programs and user-level applications on a physical machine, which is more convenient to deploy.

We successfully run Razzer with its source code [63], to test 4 kernel-level filesystems in Table I. Each filesystem is tested for 24 hours. The static analysis of Razzer identifies 1503 pairs of possible racy instructions in the four filesystems, but its dynamic analysis finds no race bug in the four filesystems with these pairs. By contrast, CONZZER finds 59 data races (including 2 data races triggering null-pointer dereferences) in these four filesystems.

KRACE [81]. It exploits a new concurrency-coverage metric, alias instruction pair, to describe thread interleavings. This metric is identical to concurrent call pair that used by inst-fuzzer in Section VI-A. It also injects random delays to cover different thread interleavings. To cover more concurrency code, it uses an evolution algorithm to mutate and generate multi-threaded syscall sequences as inputs for concurrency fuzzing. We find a KRACE repository in the Github [47]. Unfortunately, after much manual effort, we were still not able to run it due to various runtime errors and the lack of its documentation. We also compare CONZZER to KRACE in methodology:

1) The alias instruction pair used by KRACE is a *context-insensitive* metric, namely it only describes the locations of two concurrently-executed instructions without considering their execution contexts. By contrast, CONZZER uses a new context-sensitive metric, concurrent call pair, which considers both the locations and calling contexts of two concurrently-executed functions. Thus, CONZZER is more effective in finding data races that occur only in specific runtime contexts.

2) The random delay injection used by KRACE is inefficient in thread-interleaving exploration, namely it frequently repeats already covered thread interleavings and misses many

infrequent thread interleavings. By contrast, CONZZER uses a directional fuzzing approach to infer and cover new possible thread interleavings, which can more efficiently cover infrequent thread interleavings. The results in Table IV show that this fuzzing approach indeed covers many infrequent thread interleavings missed by random delay injection, which can help discover hard-to-find data races.

According to the KRACE paper, three filesystems (*btrfs*, *ext4* and *vfs*) in Linux 5.4-rc5 are tested, and the found data races are listed in the paper. Accordingly, we select *btrfs* and *ext4* in this kernel version and use CONZZER to test them. We also intended to test *vfs* with CONZZER, but at present CONZZER can only test filesystems running as loadable kernel modules and *vfs* cannot run in this mode. The KRACE paper presents that it finds 15 real data races in the two filesystems (11 in *btrfs* and 4 in *ext4*). In our evaluation, CONZZER finds 39 real data races in the two filesystems (34 in *btrfs* and 5 in *ext4*). By comparing to the race reports in the KRACE paper, we find that: (1) CONZZER and KRACE both finds 4 identical data races; (2) CONZZER finds 35 data races missed by KRACE, because our context-sensitive and directional concurrency fuzzing approach actually covers the related infrequent thread interleavings in specific runtime contexts; (3) KRACE finds 11 data races missed by CONZZER, as its evolution algorithm optimizes syscall-sequence generation and covers more concurrency code than Syzkaller-like input-driven fuzzing used by CONZZER. We believe that by using KRACE's evolution algorithm to generate syscall sequences, CONZZER can also find these missed data races.

MUZZ [8]. It uses thread-aware instrumentation to collect the changes of code coverage caused by thread interleavings, in order to improve seed selection of concurrency fuzzing. Moreover, it randomly adjusts thread priorities of the tested program at runtime, to try to cover different thread interleavings during fuzzing. Because MUZZ is not open-source, we focus on making a methodological comparison. CONZZER has two significant differences from MUZZ:

1) Although being enhanced with the changes caused by thread interleavings, code coverage still mainly describes sequential-execution situations, and thus it is limited in describing concurrent-execution situations and guiding concurrency fuzzing. To thoroughly replace code coverage, CONZZER exploits a new concurrency-coverage metric, concurrent call pair, which can effectively describe thread interleavings with different calling contexts.

2) Similar to random delay injection, randomly adjusting thread priorities is also inefficient in covering infrequent thread interleavings. To solve this problem, CONZZER uses a directional fuzzing approach to perform more effective and efficient thread-interleaving exploration.

VII. DISCUSSION

Feasibility of found data races. To affect thread scheduling of the program, CONZZER injects and controls breakpoints containing time delays during fuzzing. These time delays just slightly increase the running time of related threads, and have no effect on synchronization and concurrency logic. Moreover, our customized race checker guarantees that all found races are triggerable with specific inputs and thread interleavings. Besides, the program is actually tested on a physical machine instead of a tuned virtual machine. For these reasons, the data races found by CONZZER are all realistic and feasible.

Identifying harmful data races. In the current evaluation, we manually check the found data races to estimate their harmfulness. In fact, some existing approaches [43], [59], [85] can automatically classify data races by analyzing the runtime information about concurrent execution or statically analyzing source code. Thus, these approaches can help CONZZER to reduce the manual work of identifying harmful data races.

Supporting other concurrency checkers. CONZZER is able to conveniently support other concurrency checkers (such as atomicity-violation checkers and other data-race checkers) to detect hard-to-find concurrency problems, by covering infrequent thread interleavings. For example in Section VI-A, we have used TSan in CONZZER to test user-level applications, without any change of TSan and our concurrency fuzzing approach. With CONZZER, TSan finds many data races missed by using AFL++ in our experiments.

Limitations and future works. First, in the current evaluation, CONZZER still misses much error handling code and thus cannot find related data races. We plan to introduce fault injection [3], [40] in CONZZER to strengthen data-race detection in error handling code. Second, CONZZER does not specially handle atomic instructions, which are mostly used in low-level libraries and kernel core parts. We plan to statically identify all atomic instructions in LLVM bytecode and neglect these instructions in dynamic lockset analysis. Third, as shown in Section VI-A, CONZZER has a lower fuzzing throughput than AFL++, which limits testing efficiency. Thus, we plan to improve CONZZER fuzzing performance, by optimizing code instrumentation to reduce runtime overhead [33], [87] and using parallel systems to speed up fuzzing [49], [52]. Finally, besides data races, we plan to extend CONZZER to detect other concurrency problems, such as atomicity violations.

VIII. RELATED WORK

A. Coverage-Guided Fuzzing

Coverage-guided fuzzing evolutionally mutates and generates program inputs to increase testing coverage, guided by the code-coverage metric. Compared to traditional mutation testing [38], [39], coverage-guided fuzzing can more efficiently generate useful program inputs.

Code-coverage-guide fuzzing. AFL [1] and Syzkaller [70] are two well-known scalable coverage-guided fuzzing frameworks that integrate many practical mutation strategies and engineering techniques. AFL tests user-level applications through program inputs, while Syzkaller tests kernel-level programs through system calls. These two frameworks have found thousands of bugs and vulnerabilities in common applications and OS kernels. Based on AFL and Syzkaller, many approaches [4], [15], [30], [45], [50], [60], [61], [72], [79], [80], [82] have been proposed to further improve fuzzing effectiveness for code coverage and bug detection.

Concurrency fuzzing. To fill the gaps between coverage-guided fuzzing and concurrent program testing, several recent concurrency fuzzing approaches [8], [37], [53], [76], [81] have been proposed. As for fuzzing feedback, except KRACE [81] using a concurrency-coverage metric, the other approaches tune traditional code coverage. In fact, these approaches are still limited in finding concurrency issues like data races, due to lacking effective concurrency-coverage metric and using random thread-interleaving exploration. To solve this problem, CONZZER uses a new context-sensitive concurrency-coverage metric and a new directional concurrency fuzzing approach, to discover hard-to-find data races that occur only in specific runtime contexts and infrequent thread interleavings.

B. Data-Race Detection

Static analysis. Many approaches [22], [42], [77], [78] are based on flow-sensitive static lockset analysis. RacerX [22] uses inter-procedural and context-sensitive analysis to maintain and check locksets of memory accesses in each code path. To speed up static analysis, it creates a summary cache for each analyzed function to store the information about inside memory accesses and related locksets. Some approaches [25], [65] are based on flow-insensitive type-based analysis. Flanagan et al. [25] propose a type-based approach that uses formal type annotations to capture synchronization patterns from program code, and detects data races based on these patterns.

Static analysis approaches can achieve high coverage of data-race detection without running the tested program. But they suffer from overwhelming false positives, without exact runtime information about concurrent memory accesses.

Dynamic analysis. Some approaches [10], [19], [21], [66] are based on lockset analysis. They maintain a lockset for each running thread, and compute the intersection between the locksets of shared-variable accesses to detect data races. But due to neglecting concurrent contexts of memory accesses and non-lock synchronization primitives, these approaches often have many false positives. To reduce false positives, some approaches [27], [51], [56], [62], [84] perform happens-before-relation inference. They track memory accesses and synchronization events to infer happens-before relation between these events. But they have many false negatives and introduce much runtime overhead caused by tracking memory accesses and inferring happens-before relation. Some approaches [5], [23] are based on sampling, and they monitor memory accesses at intervals to reduce runtime overhead. But they may miss many real races when the sampling frequency is low and have much runtime overhead when the sampling frequency is high.

Dynamic analysis approaches need to cover different thread interleavings for race detection. For this purpose, they often perform random delays to cover more thread interleavings, but this method frequently repeats already covered thread interleavings and miss many infrequent ones [12], [74]. To solve this problem, CONZZER uses a new context-sensitive and directional concurrency fuzzing approach to efficiently explore thread interleavings, helping discover hard-to-find data races.

C. Thread-Interleaving Exploration

To find more concurrency problems during runtime testing, some approaches [6], [28], [29], [31], [58] use randomized thread scheduling to explore thread interleavings. PCT [6] is an effective priority-based and randomized scheduler that can increase the probability of finding concurrency bugs when the program is repeatedly executed. When a thread is created, PCT assigns it a random scheduling priority, and dynamically changes this priority at some randomly chosen steps.

Similar to random delay injection, the efficiency of thread-interleaving exploration for these approaches is also limited, as they frequently repeat already covered thread interleavings and miss many infrequent ones [12], [74]. To solve this problem, CONZZER uses a new context-sensitive and directional concurrency fuzzing approach to deterministically infer and cover new possible thread interleavings, according to already covered thread interleavings.

IX. CONCLUSION

In this paper, we design CONZZER, a novel and practical concurrency fuzzing framework for data-race detection. It uses a new context-sensitive and directional concurrency fuzzing approach to perform thread-interleaving exploration, with a new concurrency-coverage metric, concurrent call pair, as program feedback. This approach can effectively cover infrequent thread interleavings with concrete context information, to help discover hard-to-find data races. We have evaluated CONZZER on 8 user-level applications and 4 kernel-level filesystems, and found 95 real data races. CONZZER is available at *https://oslab.cs.tsinghua.edu.cn/CONZZER/*.

ACKNOWLEDGMENT

This work was mainly supported by the National Natural Science Foundation of China under Project 62002195. Kangjie Lu was supported in part by the NSF awards CNS-1931208 and CNS-2045478. Jia-Ju Bai is the corresponding author.

REFERENCES

[1] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[2] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 255–268, 2019.

[3] Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu. Testing error handling code in device drivers using characteristic fault injection. In *Proceedings of the 2016 USENIX Annual Technical Conference*, pages 635–647, 2016.

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. *IEEE Transactions on Software Engineering (TSE)*, 45(5):489–506, 2019.

[5] Michael D Bond, Katherine E Coons, and Kathryn S McKinley. PACER: proportional detection of data races. In *Proceedings of the 31st International Conference on Programming Language Design and Implementation (PLDI)*, pages 255–268, 2010.

[6] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010.

[7] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys)*, pages 1–5, 2011.

[8] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. MUZZ: thread-aware greybox fuzzing for effective bug hunting in multithreaded programs. In *Proceedings of the 29th USENIX Security Symposium*, pages 2325–2342, 2020.

[9] Peng Chen and Hao Chen. Angora: efficient fuzzing by principled search. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, pages 711–725, 2018.

[10] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 366–376, 2019.

[11] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the 23rd International Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269, 2002.

[12] Ankit Choudhary, Shan Lu, and Michael Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 266–277, 2017.

[13] Clang compiler. https://clang.llvm.org/.

[14] CLOC: count lines of code. https://cloc.sourceforge.net.

[15] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. DIFUZE: interface aware fuzzing for kernel drivers. In *Proceedings of the 24th International Conference on Computer and Communications Security (CCS)*, pages 2123–2138, 2017.

[16] CVE-2020-1667. https://nvd.nist.gov/vuln/detail/CVE-2020-1667.

[17] CVE-2020-9990. https://nvd.nist.gov/vuln/detail/CVE-2020-9990.

[18] CVE-2020-11173. https://nvd.nist.gov/vuln/detail/CVE-2020-11173.

[19] Dongdong Deng, Wei Zhang, and Shan Lu. Efficient concurrency-bug detection across inputs. In *Proceedings of the 2013 International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 785–802, 2013.

[20] Dirty COW: a privilege escalation vulnerability in the Linux kernel. https://dirtycow.ninja/.

[21] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI)*, pages 245–255, 2007.

[22] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[23] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 151–162, 2010.

[24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, August 2020.

[25] Cormac Flanagan and Stephen N Freund. Type-based race detection for Java. In *Proceedings of the 21st International Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000.

[26] Cormac Flanagan and Stephen N Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 International Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 90–96, 2001.

[27] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133, 2009.

[28] Pedro Fonseca, Cheng Li, and Rodrigo Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*, pages 215–228, 2011.

[29] Pedro Fonseca, Rodrigo Rodrigues, and Björn B Brandenburg. SKI: exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 415–431, 2014.

[30] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: path sensitive fuzzing. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, pages 679–696, 2018.

[31] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the 28th International Symposium on Operating Systems Principles (SOSP)*, pages 66–83, 2021.

[32] Haryadi S. Gunawi, Cindy Rubio-González, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Ben Liblit. EIO: error handling is occasionally correct. In *Proceedings of the 6th International Conference on File and Storage Technologies (FAST)*, pages 207–222, 2008.

[33] Shantanu Gupta, Florin Sultan, Srihari Cadambi, Franjo Ivancic, and Martin Rotteler. Using hardware transactional memory for data race detection. In *Proceedings of the 2009 International Symposium on Parallel and Distributed Processing*, pages 1–11, 2009.

[34] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A system and language for building system-specific, static analyses. In *Proceedings of the 23rd International Conference on Programming Language Design and Implementation (PLDI)*, pages 69–82, 2002.

[35] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the 22nd International Conference on Programming Language Design and Implementation (PLDI)*, pages 254–263, 2001.

[36] IOzone filesystem benchmark. http://iozone.org.

[37] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: finding kernel race bugs through fuzzing. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, pages 754–768, 2019.

[38] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology (IST)*, 51(10):1379–1393, 2009.

[39] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering (TSE)*, 37(5):649–678, 2010.

[40] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using context-sensitive software fault injection. In *Proceedings of the 29th USENIX Security Symposium*, pages 2595–2612, 2020.

[41] Viktor Johansson and Alexander Vallén. Random testing with sanitizers to detect concurrency bugs in embedded avionics software, 2018.

[42] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 2009 International Symposium on Foundations of Software Engineering (FSE)*, pages 13–22, 2009.

[43] Baris Can Cengiz Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with Portend. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 185–198, 2012.

[44] KCSAN: concurrency sanitizer for the Linux kernel. https://github.com/google/ktsan/wiki/KCSAN.

[45] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th International Symposium on Operating Systems Principles (SOSP)*, pages 147–161, 2019.

[46] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 International Conference on Computer and Communications Security (CCS)*, pages 2123–2138, 2018.

[47] KRACE repository. https://github.com/sslab-gatech/krace.

[48] Chris Lattner, Andrew Lenharth, and Vikram S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, 2007.

[49] Wang Hao Lee, Murali Srirangam Ramanujam, and SPT Krishnan. On designing an efficient distributed black-box fuzzing system for mobile devices. In *Proceedings of the 10th International Symposium*

on Information, Computer and Communications Security (ASIACCS), pages 31–42, 2015.

[50] Caroline Lemieux and Koushik Sen. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*, pages 475–485, 2018.

[51] Guangpu Li, Shan Lu, Madanlal Musuvathi, Suman Nath, and Rohan Padhye. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In *Proceedings of the 27th International Symposium on Operating Systems Principles (SOSP)*, pages 162–180, 2019.

[52] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. PAFL: extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 International Symposium on the Foundations of Software Engineering (FSE)*, pages 809–814, 2018.

[53] Changming Liu, Deqing Zou, Peng Luo, Bin B Zhu, and Hai Jin. A heuristic framework to detect concurrency vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pages 529–541, 2018.

[54] LLVM compiler infrastructure. https://llvm.org/.

[55] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, 2008.

[56] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for Android applications. In *Proceedings of the 35th International Conference on Programming Language Design and Implementation (PLDI)*, pages 316–325, 2014.

[57] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for C programs with function pointers. *Automated Software Engineering*, 11:7–26, 2004.

[58] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 267–280, 2008.

[59] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI)*, pages 22–31, 2007.

[60] Shankara Pailoor, Andrew Aday, and Suman Jana. MoonShine: optimizing OS fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium*, pages 729–743, 2018.

[61] Van-Thuan Pham, Marcel Böhme, Andrew Edward Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering (TSE)*, 2019.

[62] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the 9th International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 179–190, 2003.

[63] Razzer repository. https://github.com/compsec-snu/razzer.

[64] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: taming device drivers. In *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, pages 275–288, 2009.

[65] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the 10th International Symposium on Principles and Practice of Parallel programming (PPoPP)*, pages 83–94, 2005.

[66] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[67] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, pages 62–71, 2009.

[68] Ohad Shacham, Mooly Sagiv, and Assaf Schuster. Scaling model checking of dataraces using dynamic information. In *Proceedings of the 10th International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 107–118, 2005.

[69] Slow start of TCP congestion control. https://www.keycdn.com/support/tcp-slow-start.

[70] Syzkaller: a coverage-guided kernel fuzzer. https://github.com/google/syzkaller.

[71] Data races found by Syzkaller+KCSAN in the Linux kernel. https://syzkaller.appspot.com/upstream?manager=ci2-upstream-kcsan-gce.

[72] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Security Symposium*, pages 291–307, 2018.

[73] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[74] Valerio Terragni and Mauro Pezzè. Effectiveness and challenges in generating concurrent tests for thread-safe classes. In *Proceedings of the 33rd International Conference on Automated Software Engineering (ASE)*, pages 64–75, 2018.

[75] ThreadSanitizer: a data race detector for C/C++. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[76] Nischai Vinesh, Sanjay Rawat, Herbert Bos, Cristiano Giuffrida, and M Sethumadhavan. ConFuzz: a concurrency fuzzer. In *Proceedings of the 1st International Conference on Sustainable Technologies for Computational Intelligence*, pages 667–691, 2019.

[77] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 391–402, 2016.

[78] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the 2007 International Symposium on Foundations of Software Engineering (FSE)*, pages 205–214, 2007.

[79] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. Typestate-guided fuzzer for discovering use-after-free vulnerabilities. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, pages 1–12, 2020.

[80] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019.

[81] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRACE: data race fuzzing for kernel file systems. In *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, pages 1643–1660, 2020.

[82] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, pages 818–834, 2019.

[83] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2012.

[84] Kunpeng Yu, Chenxu Wang, Yan Cai, Xiapu Luo, and Zijiang Yang. Detecting concurrency vulnerabilities based on partial orders of memory and thread events. In *Proceedings of the 29th International Symposium on the Foundations of Software Engineering (FSE)*, pages 280–291, 2021.

[85] Lu Zhang and Chao Wang. RClassify: classifying race conditions in web applications via deterministic replay. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 278–288, 2017.

[86] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. OWL: understanding and detecting concurrency attacks. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN)*, pages 219–230, 2018.

[87] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: hardware-assisted lockset-based race detection. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132, 2007.

Appendix I

Detailed Reports of the Data Races Found by CONZZER

ID	Program	Racy instruction pair	Impact	Status
1	sqlite	sqlite3.c, line 61097 ↔ sqlite3.c, line 61110	B	-
2	sqlite	sqlite3.c, line 60429 ↔ sqlite3.c, line 61048	H	R
3	sqlite	sqlite3.c, line 60429 ↔ sqlite3.c, line 61048	H	R
4	sqlite	sqlite3.c, line 60357 ↔ sqlite3.c, line 61162	B	-
5	sqlite	sqlite3.c, line 61048 ↔ sqlite3.c, line 60269	H	R
6	sqlite	sqlite3.c, line 60352 ↔ sqlite3.c, line 61110	B	-
7	memcached	items.c, line 1401 ↔ memcached.c, line 7535	H	C
8	memcached	items.c, line 1182 ↔ memcached.c, line 7535	H	C
9	memcached	memcached.c, line 6334 ↔ memcached.c, line 7535	H	C
10	memcached	items.c, line 488 ↔ memcached.c, line 7535	H	C
11	memcached	items.c, line 1202 ↔ memcached.c, line 7535	H	C
12	memcached	crawler.c, line 161 ↔ memcached.c, line 7535	H	C
13	memcached	items.c, line 663 ↔ memcached.c, line 7535	H	C
14	memcached	items.c, line 1526 ↔ memcached.c, line 7535	H	C
15	memcached	items.c, line 1497 ↔ memcached.c, line 7535	H	C
16	memcached	items.c, line 1475 ↔ memcached.c, line 7535	H	C
17	memcached	items.c, line 1534 ↔ memcached.c, line 7535	H	C
18	memcached	crawler.c, line 151 ↔ memcached.c, line 7535	H	C
19	x264	threadpool.c, line 161 ↔ threadpool.c, line 55	B	-
20	x264	analyse.c, line 3860 ↔ frame.c, line 681	H	F
21	x264	analyse.c, line 363 ↔ frame.c, line 681	H	F
22	x264	encoder.c, line 3353 ↔ lookahead.c, line 92	H	F
23	ffmpeg	mpeg4video.h, line 224 ↔ mpeg4video.h, line 274	H	R
24	ffmpeg	mpeg4video.h, line 223 ↔ mpeg4video.h, line 274	H	R
25	ffmpeg	mpeg4video.h, line 224 ↔ mpegvideo.c, line 1918	H	R
26	ffmpeg	mpeg4video.h, line 223 ↔ mpegvideo.c, line 1918	H	R
27	ffmpeg	mpeg4video.h, line 224 ↔ mpegvideo.c, line 1919	H	R
28	ffmpeg	mpeg4video.h, line 223 ↔ mpegvideo.c, line 1919	H	R
29	ffmpeg	mpeg4video.h, line 224 ↔ mpegvideo.c, line 1932	H	R
30	ffmpeg	mpeg4video.h, line 223 ↔ mpegvideo.c, line 1932	H	R
31	ffmpeg	mpeg4video.h, line 224 ↔ mpegvideo.c, line 1933	H	R
32	ffmpeg	mpeg4video.h, line 223 ↔ mpegvideo.c, line 1933	H	R
33	aget	Download.c, line 120 ↔ Download.c, line 118	H	R
34	aget	Download.c, line 120 ↔ Download.c, line 106	H	R
35	aget	Misc.c, line 184 ↔ Misc.c, line 192	H	R
36	axel	conn.c, line 310 ↔ axel.c, line 386	H	F
37	btrfs	locking.c, line 136 ↔ locking.c, line 308	B	-
38	btrfs	locking.c, line 135 ↔ locking.c, line 313	H	F
39	btrfs	locking.c, line 152 ↔ locking.c, line 313	H	F
40	btrfs	locking.c, line 285 ↔ locking.c, line 313	H	F
41	btrfs	locking.c, line 285 ↔ locking.c, line 118	H	F
42	btrfs	transaction.c, line 490 ↔ delayed-ref.c, line 112	H	C
43	btrfs	transaction.c, line 2095 ↔ transaction.c, line 720	B	-
44	btrfs	block-rsv.c, line 195 ↔ delayed-ref.c, line 112	H	C
45	btrfs	block-rsv.c, line 51 ↔ block-rsv.c, line 195	H	C
46	btrfs	extent_io.c, line 3481 ↔ inode.c, line 6935	H	F
47	btrfs	extent_io.c, line 4194 ↔ extent_io.c, line 4194	H	C
48	btrfs	block-rsv.c, line 391 ↔ delalloc-space.c, line 277	H	R
49	btrfs	block-group.c, line 404 ↔ free-space-cache.c, line 1643	B	-
50	btrfs	block-group.c, line 404 ↔ free-space-cache.c, line 1629	B	-
51	btrfs	transaction.h, line 151 ↔ disk-io.c, line 1388	H	R
52	btrfs	transaction.h, line 152 ↔ disk-io.c, line 1390	H	C
53	btrfs	block-group.c, line 754 ↔ block-group.h, line 246	H	F
54	btrfs	block-group.c, line 736 ↔ block-group.h, line 246	H	F
55	btrfs	block-rsv.c, line 391 ↔ block-rsv.c, line 23	H	R
56	btrfs	transaction.c, line 512 ↔ delayed-ref.c, line 112	H	C
57	btrfs	tree-log.c, line 3125 ↔ transaction.h, line 151	H	R
58	btrfs	inode.c, line 6935 ↔ inode.c, line 6935	H	F
59	btrfs	block-rsv.c, line 195 ↔ block-rsv.c, line 228	H	C
60	btrfs	extent_io.c, line 4194 ↔ extent_io.c, line 4085	H	C
61	btrfs	locking.c, line 282 ↔ locking.c, line 308	B	-
62	btrfs	block-group.c, line 650 ↔ block-group.h, line 246	H	C
63	btrfs	block-rsv.c, line 51 ↔ transaction.c, line 512	H	C
64	btrfs	locking.c, line 152 ↔ locking.c, line 118	H	F
65	btrfs	locking.c, line 287 ↔ locking.c, line 313	H	F
66	btrfs	locking.c, line 293 ↔ locking.c, line 282	B	-
67	btrfs	transaction.c, line 2259 ↔ transaction.c, line 720	B	-
68	btrfs	disk-io.c, line 1388 ↔ file.c, line 1997	H	R
69	btrfs	ctree.c, line 132 ↔ ctree.c, line 1131	H	R
70	btrfs	block-rsv.c, line 391 ↔ block-rsv.c, line 226	H	R
71	jfs	jfs_logmgr.c, line 2001 ↔ jfs_logmgr.c, line 2193	B	-
72	jfs	jfs_logmgr.c, line 2001 ↔ jfs_logmgr.c, line 2207	B	-
73	jfs	jfs_txnmgr.c, line 932 ↔ jfs_logmgr.c, line 979	H	R
74	jfs	jfs_dmap.c, line 437 ↔ jfs_logmgr.c, line 979	H	R
75	jfs	jfs_xtree.c, line 242 ↔ jfs_xtree.c, line 242	H	R
76	jfs	jfs_xtree.c, line 357 ↔ jfs_xtree.c, line 357	H	R
77	jfs	jfs_dmap.c, line 2421 ↔ super.c, line 140	H	C
78	jfs	super.c, line 130 ↔ jfs_dmap.c, line 2421	H	C
79	jfs	jfs_txnmgr.c, line 489 ↔ jfs_txnmgr.c, line 332	H	F
80	jfs	jfs_metapage.c, line 651 ↔ jfs_metapage.c, line 651	H	R
81	jfs	jfs_imap.c, line 2734 ↔ jfs_imap.c, line 2592	H	R
82	jfs	jfs_txnmgr.c, line 501 ↔ jfs_logmgr.c, line 1455	H	F
83	xfs	xfs_log_cil.c, line 444 ↔ xfs_log_cil.c, line 917	H	F
84	xfs	xfs_trans_ail.c, line 733 ↔ xfs_inode_item.c, line 371	H	C
85	xfs	xfs_file.c, line 156 ↔ xfs_inode.c, line 3892	B	-
86	xfs	xfs_inode_item.c, line 146 ↔ xfs_inode_item.h, line 30	H	F
87	xfs	xfs_super.c, line 1033 ↔ xfs_inode.h, line 175	B	-
88	xfs	xfs_inode_item.c, line 531 ↔ xfs_inode.h, line 134	B	-
89	xfs	xfs_trans_inode.c, line 141 ↔ xfs_inode_item.h, line 30	H	C
90	xfs	xfs_inode_item.c, line 424 ↔ xfs_inode_item.h, line 30	H	F
91	reiserfs	lock.c, line 26 ↔ lock.c, line 64	B	-
92	reiserfs	lock.c, line 28 ↔ lock.c, line 26	B	-
93	reiserfs	lock.c, line 79 ↔ lock.c, line 26	B	-
94	reiserfs	lock.c, line 26 ↔ lock.c, line 47	B	-
95	reiserfs	inode.c, line 2523 ↔ inode.c, line 2943	H	R

The column "Racy instruction pair" shows the source file name and code line number of the racy instructions.
The column "Impact" shows whether the data race is harmful. "H" means that the data race is identified to be harmful; "B" means that the data race is identified to be benign.
The column "Status" shows the current status of the data race. "R" means that the data race has been reported but has not been replied; "C" means that the data race has been confirmed. "F" means that the data race has been confirmed and fixed by related developers. Note that we only report harmful data races.

Appendix II

Figure 12 shows the completed sensitivity results about the growth of covered concurrent call pairs for all the 12 tested program in Section V-D. CONZZER covers more concurrent call pairs than other alternative methods, which proves the effectiveness of CONZZER in thread-interleaving exploration.

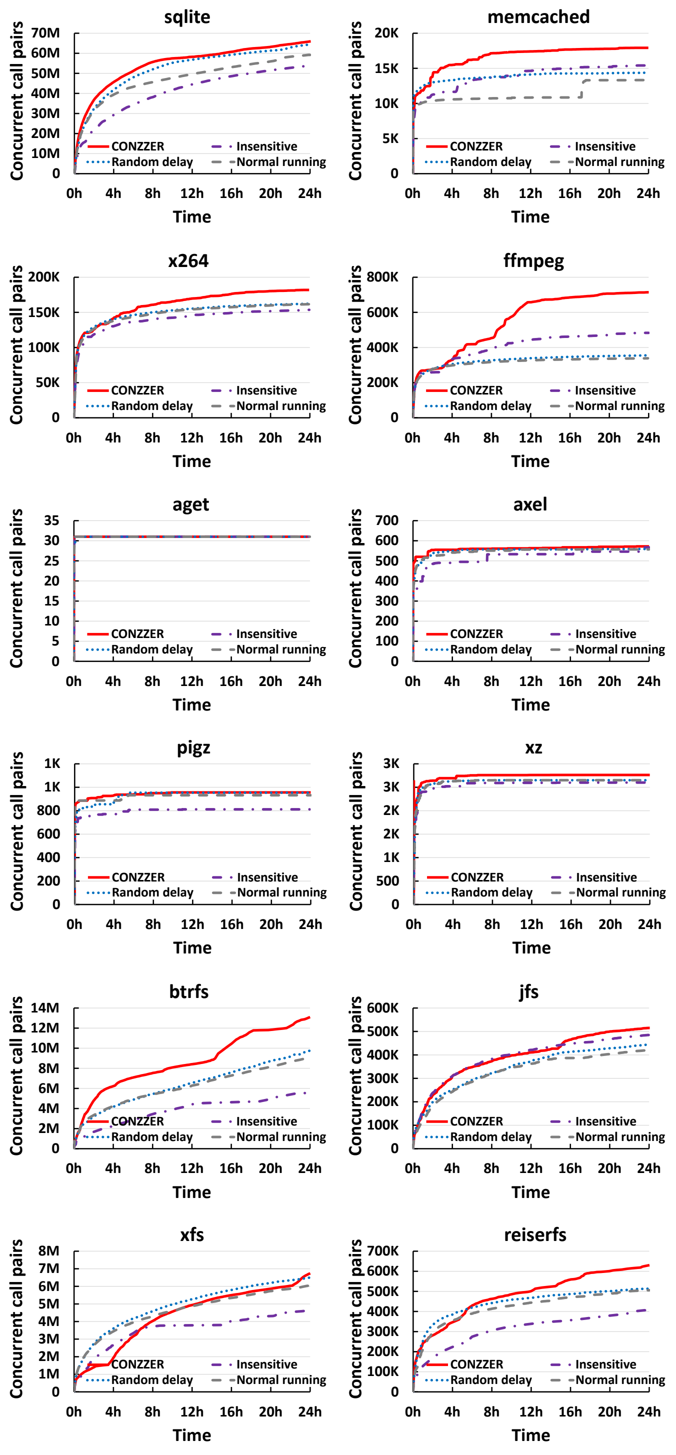


Fig. 12. Complete results about the growth of concurrent call pairs in sensitivity analysis.

Figure 13 shows the completed comparison results about the growth of covered concurrent call pairs for all the 12 tested program in Section VI-A. CONZZER covers more concurrent call pairs than other fuzzing tools, which proves the effectiveness of CONZZER in thread-interleaving exploration.

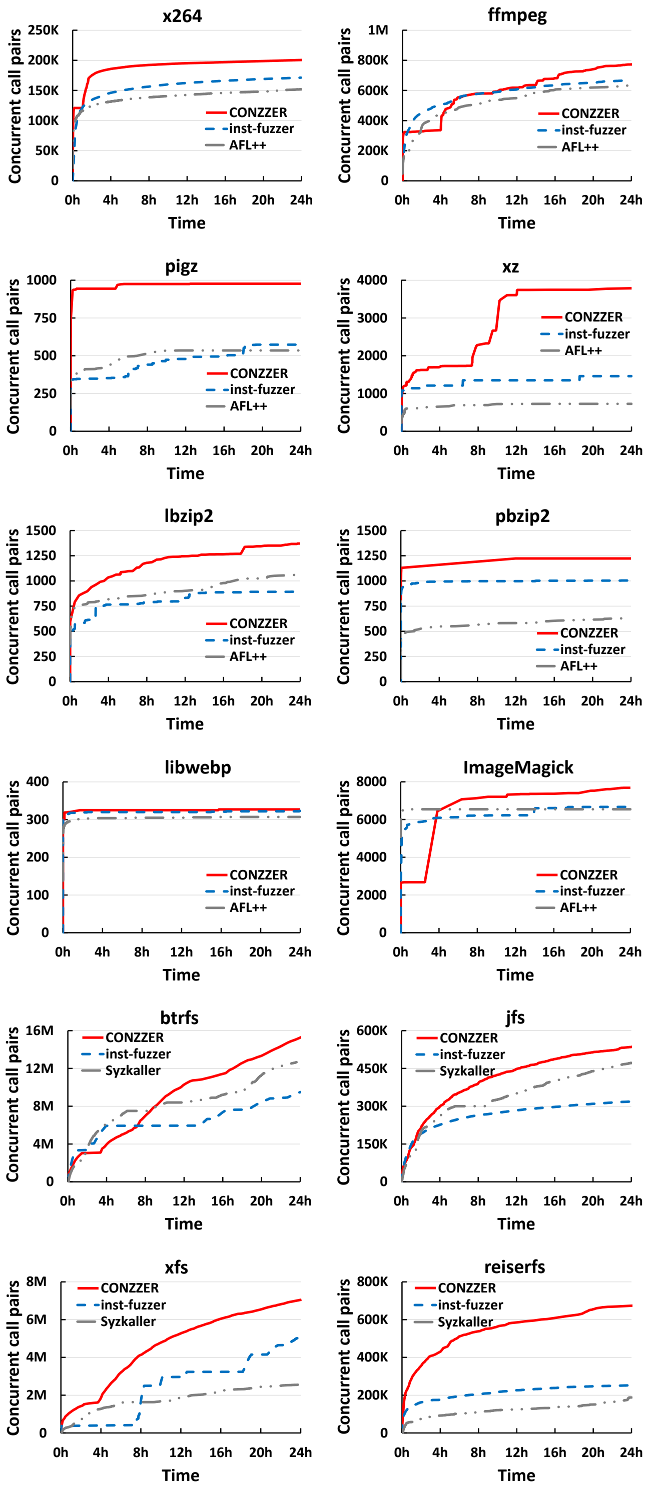


Fig. 13. Complete results about the growth of concurrent call pairs in comparison experiment.