C2FUZZ: Coverage-directed Concurrency Fuzzing

```
int a = 6,  *x = &a;

P0() {
   int *r1 = x,  r2 = 0;
   if (r1 != NULL)
      r2 = READ_ONCE(*r1);
}

P1() {
   WRITE_ONCE(x, NULL);
}
```

(a) Data race.

```
P0() {
   file = smp_load_acquire(fdt->fd[fd]);
   if (!file)
      return;

   if (dec_and_test(obj->refcnt))
      kfree(obj);
}

P1() {
   smp_store_release(fdt->fd[fd], file);
   atomic_inc(obj->refcnt);
}
```

(b) Race condition.

Figure 1: Two types of concurrency bugs. DR: I'm not sure we need these examples.

Abstract

C2FUZZ

1. Introduction

DR: Copied from the learning from mistakes paper: The complexity of this design is only polynomial to the number of dynamic memory accesses, which is a huge reduction from the exponential-sized all-interleaving testing scheme. Furthermore, the bug exposing capability of this design is almost as good as exploring all interleavings. It would miss only few bugs in our examination.

We describe our contributions in three folds:

-
- We have found 17 race conditions.

2. Background and Motivation

While concurrency has been a cornerstone of performance benefits in the multicore era, it leaves a risk of malfunction, *i.e.*, concurrency bug, caused by insufficient thought of concurrent events. In this section, we describe two types of concurrency bug, data race and race condition, and introduce prior approaches to spot out concurrency bugs.

2.1. Concurrency Bug

A concurrency bug can be classified into two categories according to how it adversely affects a program.

DR: after the last meeting, I'm not sure we need to explain differences between data race and race conditions. My first thought was to explain the differences and then explain why we choose to observe erroneous outcomes instead of using data race detectors.

Data race. Although the concept of data race is well known, its formal definition depends on a programming language [11, 25] or a program in development [23]. In this paper, we employ the definition from Linux Kernel Memory Model (LKMM) [23]. According to LKMM, a data race occurs when there are two memory accesses such that 1) they access the same location, 2) at least one of them is a store, 3) at least one of them is not annotated with special operations such as `WRITE_ONCE()`, or `smp_load_acquire()`, 4) they occur on different CPUs (or in different threads on the same CPU), and 5) they execute concurrently. Informally, a data race indicates un-annotated concurrent accesses to a shared memory location.

Data races are problematic because they may confuse a compiler. LKMM allows a compiler to assume that there will be no data race during the runtime. Based on the assumption, a compiler has its rights to arbitrarily transform plain accesses (*i.e.*, accesses not annotated with above operations), making the results unpredictable if there is a data race during the runtime. Therefore, LKMM defines the outcome of the program as undefined behavior if a data race occurs. Also, to prevent such undefined behaviors, LKMM requires developers to annotate accesses if they possibly run concurrently [18–20].

Race condition. Race condition is another type of concurrency bug. While a race condition broadly indicates that an outcome differs depending on the timing of concurrent events, we restrict the definition to indicate the correctness of the outcome differs according to the timing of concurrent events. Specifically, if developers do not take consider of all possible interleavings, a program may execute an erroneous interleaving which leads the program into an unintended state. Immediately or after a certain amount of time, the unintended state causes erroneous behaviors of the program such as memory corruption, deadlock, and assertion violation. In order to fix the errorneous interelaving, developers often utilize synchronization primitives such as a lock, or switch the order of instructions in a program [24].

Differences between data race and race condition. Although both race condition and data race are a bug caused by concurrenct events, there are a few differences. First, a data race is regards to a single pair of conflicting accesses while a race condition is about an interleaving. Therefore, methods to detect them are also different. While data race detectors [3, 27, 30, 32, 33] monitor whether two conflicting plain accesses are executed concurrently, race conditions can be observed only through an erroneous behavior caused by a specific interleaving. Second, whereas a data race occurs if such conflicting accesses exists no matter it causes an erroneous behavior or not, a race condition is told by an abnormal outcome of a program. Lastly, they are not a subset or a superset of one another. Although there are race conditions that occur with data races, neither one is the sufficient nor necessary condition for the other. Therefore, we would argue that finding data races and finding race condditions are complementary with each other.

2.2. Concurrency Fuzzing

Since fuzzing has proved its effectiveness in finding bugs, several attempts [4, 7, 12, 30] have been made to improve fuzzing techniques specifically in exposing concurrency bugs. These approaches make different choices in various aspects depending on their own purposes; some of these approaches [7, 30] adopt a data race detector to find out data races, while others [4, 12] rely on existing sanitizers to observe erroneous behaviors (*i.e.*, race conditions). Aside from the use of data race detectors, they take different strategies for the sake of two objectives, namely scheduling instructions and capturing interesting behaviors (*i.e.*, coverage) in the concurrency dimension.

Instruction scheduling. Instead of relying on the uncontrollable kernel scheduler, concurrency fuzzing generally consists of a customized scheduler to increase the probability of bugs being exposed. Depending on how a customized scheduler schedules instructions, it is largely categorized into randomized scheduler and scheduling hint-directed scheduler (shortly, hint-directed scheduler).

A randomized scheduler [2, 5, 17, 30] is designed to randomly schedule intstructions in a disciplined manner. It tries to keep only one thread to execute while the thread runs until a certain number of instructions are executed or a certain amount of time has elapsed. A fuzzer and the scheduler share a small number of parameters used for the scheduler to pseudo-randomly determine the number of instructions or the amount of time to execute. In this way, interleavings are diversified across fuzzing runs while a fuzzer can tailor the non-determinism caused by scheduling.

On the other side, a hint-directed scheduler [7, 12] enfoces a specific requirement of interleaving called a scheduling hint. A scheduling hint is generally in the form of an interleaving pattern that possibly causes a concurrency bug. For example, a scheduling hint may consists of execution order indicating a single-variable order violation; *"thread A executes a store operation writing into X before thread B executes a load operation reading from X"*. During fuzzing, a fuzzer keeps generating different scheduling hints, and enforces an interleaving that contains a scheduling hint to observe the interleaving causes an erroneous behavior.

Coverage in the concurrency dimension. To the best of our knowledge, Krace [30] is the first work that asserts the necessity of a coverage metric in the concurrency dimension. While many coverage metrics are proposed with their own pros and cons [29], they only track the sequential aspect of a program without paying attention to communication between threads. Krace states that even after sequential coverage is saturated, there could be more interesting behaviors caused by thread interleavings, and that if there are unexplored thread interleavings in an input, a fuzzer should keep paying attention to the input.

To capture unique behaviors in the concurrency dimension, Krace proposes a coverage metric called alias coverage. Alias coverage tracks inter-thread reads-from relation between a pair of instructions. In other words, alias coverage tracks $I_S \rightarrow I_L$ if a load instruction I_L reads a value written by a store instruction I_S and they are executed in different threads. DR: TODO: MUZZ, and others? Although Razzer [12] and Snowboard [7] adopt a hint-directed scheduler, they do not make use of a concurrency coverage metric. Therefore they are not able to distinguish whether two inputs exhibit different behaviors. Instead, they concentrate on less frequent inter-thread data flow based on an idea that those inter-thread data flow is less likely tested and more likley causes an abnormal behavior.

2.3. Motivation

DR: TODO: all belows are about race conditions. what to say about data races?

In spite of enormous efforts, finding concurrency bugs is still a daunting task. To better find concurreny bugs, we first study when and how concurrency bugs manifest. Afterwards, we identify characteristics of concurrency bugs that have not been tackled in prior approaches, and clarify our goals for solving them at the end of this section.

Manifestation of concurrency bugs. According to an extensive survey of concurrency bugs [24], most race conditions manifest depending on a partial order of a few memory accesses. Specifically, 92% of race conditions the authors studied are guaranteed to manifest if a specific partial order among at most 4 memory accesses is enforced. For all other memory accesses, their execution orders and values written and read by them do not affect bugs's manifestation.

One of the factors that makes it difficult to find concurrency bugs is that, they may be detected only when a program shows an abnormal behavior. Figure 1b is the example of such concurrency bug. In this example, the timing of accesses are not correctly contemplated. When instructions are executed in the order of DR: TODO:..., the program runs differently than the developer intention, causing a use-after-free bug. However there are no plain accesses (*i.e.*, all accesses are annotated), and therefore, there is no data race. By the definition, data race detectors [3, 22, 27, 30, 32, 33] are not applicable to detect this kind of race conditions.

In worse cases, race conditions hardly manifest with the kernel scheduler. In detail, a race condition may manifest only when one thread stalls for a long time while another thread executes numerous instructions. One could argue that those concurrency bugs are not a threat because it may take too long time, or even impossible to exploit. However, a recent study, ExpRace [16], reveals that an attacker can affect the kernel's scheduler using inter-processor interrupts (IPI), and exploit even such concurrency bugs.

Existing works' limitation. As most race conditions manifest with at most four memory accesses, a fuzzer's interest could be to tests such partial orders as many as possible. In this regard, instruction scheduling methods and coverage metrics proposed by previous studies fall shorts.

DR: TODO: rewrite this paragraph. not understandable Approaches that adopt a randomized scheduler [4, 5, 30] suffer from the inherent randomness. When the randomness comes to concurrency bugs, this could become a significant drawback. Although interleavings are affected by a fuzzer, a fuzzer cannot fully control how interleaving takes place. Thus, a fuzzer may need to indiscriminately execute the same input until its coverage is saturated. Furthermore, it is almost impossible for randomized schedulers to expose concurrency bugs requiring an extreme interleaving such that DR: TODO: explain:non-inclusive race conditions [16] or ones that require a very small race window.

In contrast, approaches with a hint-directed scheduler is able to explore all interelavings including even ones that hardly occur if a proper scheduling hint is given. Therfore, they are able to find out such race conditions. However, existing approaches [7, 12] are not suitable of diversifying interleavings. *i.e.*, they change very small part of interleaving across runs. Even with a single input, they require a large number of execution to test the input enough.

Last but not least, none of existing approaches incorporate a proper coverage for concurrency bugs. Suggested coverages are not suitable as they do not capture partial orders of a small instruction group, and bugs may not be exposed even after the coverages are saturated. Consequently, existing approaches have difficulty in distinguishing a given input will exhibit a more interesting behavior, *i.e.*, race conditions.

Our goal. Our goal is to design a fuzzer to effectively explore partial orders of a small number (*e.g.*, four) of memory accesses. Considering characteristics of concurrency bugs mentioned above, it will increase the probabilty of finding more concurrency bugs if we diversify interleavings to experience such partial orders as many as possible.

To this end, we organize subgoals as follows: 1) We first design a coverage metric to capture such partial orders. By this coverage metric, we do not waste the computing power for uninteresting inputs, and not de-prioritize inputs in which there are more unexplored interleavings. 2) Based on the coverage metric, we design an instruction scheduling mechanism directed towards unexplored coverage. This scheduling mechanism not only makes the investigation of such partial orders faster, but also does not miss concurrency bugs caused by an "extreme" interleaving (*i.e.*, ones that hardly occur with the kernel scheduler). 3) While scheduling instructions, we detect concurrency bugs through abnormal behaviors instead of relying on data race detectors. Although data race detectors are useful development tools, they are inherently limited in detecting race conditions. We thus choose to observe the direct evidence of manifestation of concurrency bugs, an abnormal behavior. Incorporating a data race detector will be discussed in **??**.

3. Exploring Interleaving Space

The essence of a concurrency fuzzing relies on how to explore the interleaving space. In this section, we describe our approach to effectively explore the interleaving space. We first introduce the key idea of our approach (§3.1), and then based on the key idea, we propose a novel coverage concept in the concurrency dimension (§3.2) and an instruction scheduling mechanism (*i.e.*, interleaving mutation) to unveil undisclosed coverage (§3.3).

3.1. Key Idea

Our intuition comes from an observation that even a non-failing interleaving bears hints on which interleavings require further testing. For example, if two consecutive load operations reading from a memory location are followed by a store operation writing to the same location, we can easily deduce that there might be a single-variable atomicity violation. If we run an interleaving generated by rearranging these operations, we can identify whether the atomicity violation actually occurs or not. Our approach to explore the interleaving space realizes this intuition.

Overall steps. Let us assume we obtain a totally ordered instruction sequence after executing concurrent jobs (Figure 2-(a)). In this instruction sequence, our purposes are 1) to track an interesting behavior of the sequence, and 2) to schedule instructions in the sequence for exposing more interesting behaviors.

As to tracking interesting behaviors, we follow the survey mentioned in §2.3 describing that most race conditions deterministically manifest depending on a specific partial order of at most four memory accesses. According to this survey, such partial orders are "interesting" behaviors as they have a strong correlation to manifestation of concurrency bugs. Therefore, our purpose in tracking interesting behaviors in the concurrency dimension turns into identifying how such partial orders are established in the given sequence. To this end, we enumerate small groups of a few (*e.g.*, four) memory accesses brought in the given instruction sequence (Figure 2-(b)). As the execution order of memory accesses in each group are already determined, each group explains a part of the interleaving, therefore, we call each group a segment of interleaving, or shortly a segment. During fuzzing, we keep tracking segments as the signal of new interelavings of concurrent jobs.

After collecting segments of concurrent jobs, we need to run the concurrent jobs with different interleaving to observe different segments. Instead of blindly scheduling instructions, we deduce what partial orders need to be explored in advance. In other words, we hypothesize imaginary segments derived from collected segments for further testing (Figure 2-(c)). For example, by rearranging instructions of `Segment 1`, we derive an imaginary segment called `Segment 1'` that are not yet observed. These imaginary segments will be used as scheduling hints; our instruction scheduling mechanism will enforce imaginary segments during further fuzzing runs. It is worth noting that segments may be identical since the execution order of not-conflicting instructions does not affect the outcome. In this example, `Segment 1'` and `Segment 1"` are identical and we consider they are redundant.

As a last step, we generate a hypothetical interleaving containing these imaginary segments and run the interleaving to observe an outcome (Figure 2-(d)). Among all imaginary segments, some of them can be enforced together, while some cannot. For example, `Segment 1'` and `Segment 2'` may be enforced together as they do not make a conflict on the execution order of instructions. We call two imaginary segments are harmonious if they are able to be enforced together. As all imaginary segments are not harmonious to all others, we need to run different interleavings multiple times. For each fuzzing run, we repeat generating a hypothetical interleaving by gathering harmonious segments until we consume all imaginary segments.

Interleaving in a graph form. Our interleaving exploration

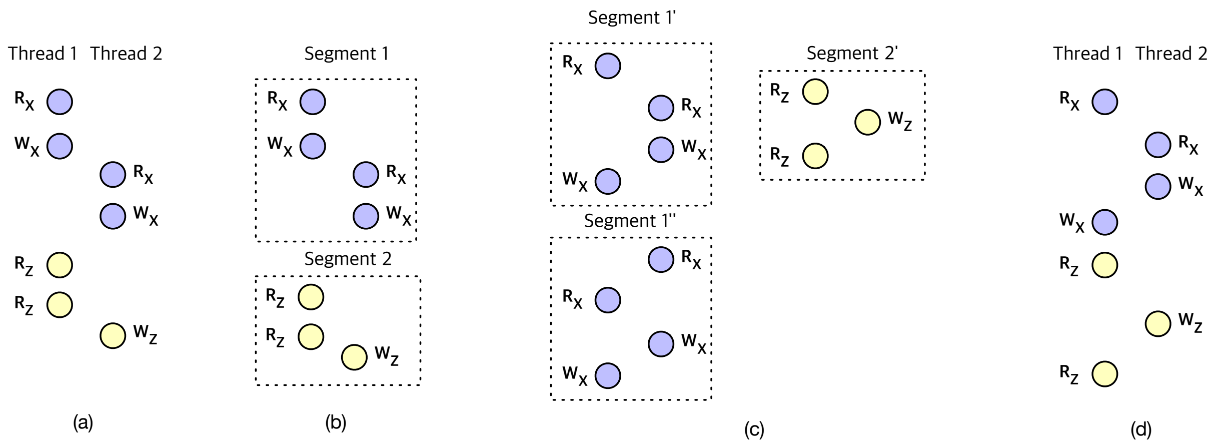

Figure 2: Key idea to explore the interleaving space. Each circle represents a memory access intruction. Annotations R_x and W_x mean that the instruction reads from or writes to a memory location x respectively.

Figure 3: interleaving graph

mechanism requires a few operations on an interleaving such as 1) rearranging instructions in a segment, 2) gathering harmonious segments, and 3) generating a whole interleaving carrying multiple segments. We notice that if we consider an interleaving as a graph of partial orders, all above operations become simple graph operations.

From a given interleaving, we draw a graph consisting of vertices representing memory access operations and edges representing the execution order of the operations that may change the outcome if reversed. DR: explain more intuition: With this graph, the required operations are simplified as follows: 1) rearranging instructions is done by changing the direction of edges, 2) we can determine segments are harmonious if there is no loop in a graph, and 3) a topological sort on a grpah generates a whole interleaving.

In the rest of this section, we define the graph form of an interleaving, and then provide details of the interleaving mutation.

3.2. Interleaving Graph

In order to represent a partial order of memory accesses, we propose a form of directed acyclic graph (DAG) called interleaving graph. An interleaving graph describes an interleaving of concurrent jobs, and represents partial orders between instructions. In an interleaving graph, vertices indicate memory access operations, and edges represent partial orders between these operations. If there is a path from a vertex from another vertex, then the execution order of two operations is established.

We categorize edges into two types, called immutable edges and mutable edges. A immutable edge corresponds to a program order in which instructions appear on a thread [1, 23]. As a program order is defined between instructions executed by the same thread, all immutable edges connect instructions from the same thread. On the other hand, a mutable edge is responsible to connect conflicting instructions that 1) are executed by different threads, 2) access the same memory location, and 3) at least one of them is write. Therefore, the execution order of instructions connected by a mutable edge may directly affect the behavior of a program.

Figure 3 shows an example of an interleaving graph. DR: TODO:

DR: TODO: imprecise description In terms of interleaving, immutable edges and mutable edges have different properties. Let us suppose we have two interleaving graphs derived from different interleavings of the same concurrent jobs. As immutable edges represent program orders, direction of immutable edges does not differ in the two interleaving graphs. Whereas, a direction of mutable edges may be different in the two interleaving graphs. It is worth noting that an edge may appear in only one interleaving graph regardless of its type. This is because the control flow may be changed according to the change of the data flow.

Segment of interleaving graph as coverage. Although all execution order of conflicting instructions (*i.e.*, mutable edges) in an interleaving graph may affect an outcome, we concentrate on a small number of instructions because a few instructions are enough to cause most “interesting” behaviors, *i.e.*, race conditions.

DR: divide? We thus divide the interleaving graph into a set of subgraphs called interleaving segment graphs, short for segment graphs. While the interleaving graph represents a whole interleaving, an segment graph is a subgraph of the given interleaving graph that contains two mutable edges and at most four vertices. In addition, vertices in a segment graph are connected by at least one mutable edge.

We capture segment graphs as coverage of a given interleaving of concurrent jobs. If a new segment graph is not found while continuing to run the concurrent jobs with different interleavings, the concurrent jobs unlikely expose a race condition. In other words, if we collect all segment graphs of concurrent jobs, we can lower the priority of the concurrent jobs. It is worth noting that we select four as the maximum number of vertices in a segment graph because not only it is enough for most race conditions, but also DR: ...

In order to identify segment graphs from execution, we need to know the total order of memory accesses. Otherwise, we may miss the execution order between instructions so cannot faithfully capture segment graphs. Therefore, our scheduling mechanism serializes execution of concurrent jobs during fuzzing. Details are described later in §4.

3.3. Interleaving Mutation

Our interleaving mutation is designed in a different way from previous studies. Instead of randomly selecting scheduling points [5, 30] or changing the execution order of a few instructions [7, 12], we draw a whole hypothetical interleaving directed towards uncaptured segment graphs. To this end, our interleaving mutation consists of three steps: 1) identifying uncaptured segment graphs, 2) selecting *harmonious* segment graphs, and 3) generating scheduling points to test the selected segment graphs.

Identifying uncaptured segment graphs. Given segment graphs derived from an interleaving, identifying uncaptured segment graphs is the first step of interleaving mutation for generating another interleaving. For each segment graph, we change directions of mutable edges to infer uncaptured segment graphs. As each segment graph contains two mutable edges, at most four uncaptured segment graphs are derived for one segment graph.

DR: TODO:...

Selecting interleaving segments to direct. Given all uncaptured segment graphs, it is unlikely that all segment graphs can be tested at once. Our approach is to find out a subset of segment graphs that are *harmonious*. Segment graphs are harmonious if they do not form a cycle in an imaginary graph.

It may require heavy computation to identify the largest subset that are all harmonious to each other. Instead of finding the optimal solution, we choose to use a greedy algorithm. Especially, given uncaptured segment graphs extracted from an interleaving graph, our interleaving mutation starts by selecting a random segment graph. And then it iteratively selects a segment graph while confirming that the selected segment graph is harmonious. Determining a given segment graph is harmonious is conducted by checking a loop in an accumulated interleaving graph.

DR: TODO: describe an algorithm to check a loop. Its time complexity is $O(V)$.

Generating scheduling points. After selecting harmonious segment graphs, generating scheduling points can be easily done by conducting a topological sort [13]. Since an imaginary interleaving graph is acyclic, a topological sort always returns a sequence of vertices (*i.e.*, instructions) that does not violate a program order. It is well known that the time complexity of a topological sort is $O(V+E)$. Considering that the graph is sparse, E is a small value so the time complexity can be asymptotically considered as $O(V)$. In this sequence, scheduling points are just instructions that the preemption should happen; *i.e.*, the next instruction is executed by a different thread.

DR: TODO: what if scheduling points are missing

Figure 4: Overall architecture of C2Fuzz

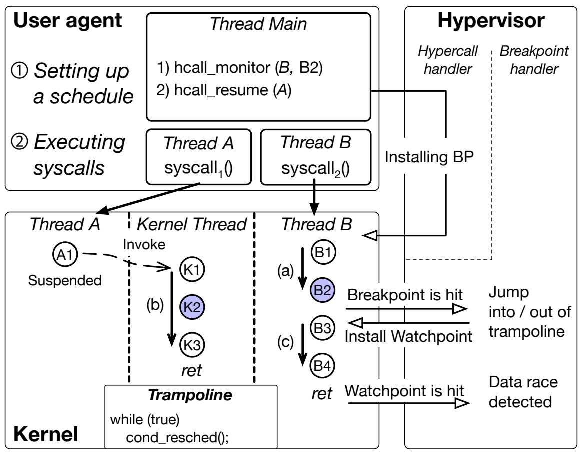

Figure 5: The workflow of the execution engine. IMPORTANT: this is copied from AITIA. Need to redraw

4. Design of C2Fuzz

C2Fuzz is a coverage-directed concurrency fuzzer adopting approaches described in §3.

4.1. Overall Architecture

4.2. Kernel Instrumentation

C2Fuzz's scheduling mechanism requires to track memory accesses of concurrent jobs. To this end, we instrument a kernel under test. Our instrumentation is similar with KCOV [21] which tracks executed basic blocks.

4.3. Userspace Fuzzer

A userspace fuzzer implmenets the interleaving mutation described in §3.

Storing interleaving coverage. In order to use interleaving graph as interleaving coverage, we need a method to store them, and compare them to a new interleaving graph. We choose to use a hash value of

the FNV-1 [6, 8], non-cryptographic hash function,

4.4. Execution Engine

We introduce an execution engine in order to enforce an interelaving provided by a fuzzer. Specifically, the execution engine is to enforce an interleaving of threads controlled by a fuzzer while all other kernel threads work ordinarily.

Figure 5 shows the overall workflow of our execution engine. The execution engine and a fuzzer communicates through hypercall interfaces.

Suspending a thread. A few prior approaches [5, 7, 12] suspend a vCPU instead of a guest thread. While suspending a vCPU grants the ability to control an interleaving, it is not suitable for our purpose because suspending a vCPU may unexpectedly suspend another vCPU. An example we observed is TLB shootdown [?]. TLB shootdown sends inter-process interrupts (IPIs) to all cores, and wait until all cores to execute the TLB shootdown handler.

Crash ID	Kernel Version (Commit)	Subsystem	Crash Type	Crash Summary
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				

TABLE 1

Therfore, instead of adopting the prior approach, our hypervisor is designed to suspends and resumes a guest thread.

4.5. Implementation

We implement C2FUZZ

5. Evaluation

To verify the effectiveness of our approach, we evaluate C2FUZZ to answer the following questions

- Is C2FUZZ able to find real-world concurrency bugs (§5.1)
- Is C2FUZZ able to capture concurrency coverage faster than previous approaches (§5.2)
- Is XXX: suitable to capture interesting behaviors (§5.3).

Experimental setup. All of our evaluations were performed on an Intel(R) Xeon(R) CPU E5-4655 v4 @ 2.50GHz (30MB cache) with 512GB of RAM.

5.1. Finding Real-world Concurrency Bugs

In order to prove the practicality of C2FUZZ, we ran C2FUZZ on latest versions of the Linux kernel ranging from XXX: to XXX: . We ran C2FUZZ for approximately two months.

Table 1 summarizes crashes found by C2FUZZ. During our experiment, 17 crashes are newly found.

5.2. Coverage Growth in the Concurrency Dimension

As the coverage growth is the important performance metric, we compare C2FUZZ with prior works.

5.3. Suitability of Coverage Metric

6. Related work

For decades, immense efforts have been made to effectively find out bugs in various softwares through fuzzing. We

TABLE 2

describe prior efforts in two categories, kernel fuzzing and concurrency fuzzing.

Kernel fuzzing. Since a kernel is a security basis of most systems, eliminating bugs in a kernel is a paramount task to provide reliable services. Therefore, many fuzzing approaches have been proposed to find out bugs as the first step towards eliminating them.

IMF [10]
Syzkaller [9],
Moonshine [26]
HFL [14]
Healer [28] is
Janus [31]
Hydra [15]
Although we adopt Syzkaller as its sequential mutation,

Concurrency fuzzing.
Razzer [12]
MUZZ [4]
Krace [30]
Snowboard [7]

7. Conclusion

References

[1] J. Alglave, L. Maranget, P. E. McKenney, A. Parri, and A. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the linux kernel. In *Proceedings of the 23rd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, Mar. 2018.

[2] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2010.

[3] Y. Cai, J. Zhang, L. Cao, and J. Liu. A deployable sampling strategy for data race detection. In *Proceedings of the 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Seattle, WA, Nov. 2016.

[4] H. Chen, S. Guo, Y. Xue, Y. Sui, C. Zhang, Y. Li, H. Wang, and Y. Liu. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual, Aug. 2020.

[5] P. Fonseca, R. Rodrigues, and B. B. Brandenburg. {SKI}: Exposing kernel concurrency bugs through systematic schedule exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[6] G. Fowler, L. C. Noll, and K.-P. Vo. FNV Hash, 2022. http://isthe.com/chongo/tech/comp/fnv/index.html.

[7] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[8] Google. The Go Programming Language, 2022. https://pkg.go.dev/hash/fnv.

[9] Google. Syzkaller - kernel fuzzer, 2022. https://github.com/google/syzkaller.

[10] H. Han and S. K. Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Dallas, Texas, Nov. 2017.

[11] International Organization for Standardization. ISO/IEC 9899:2018, 2017. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2310.pdf.

[12] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[13] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[14] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee. Hfl: Hybrid fuzzing on the linux kernel. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[15] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Huntsville, Ontario, Canada, Oct. 2019.

[16] Y. Lee, C. Min, and B. Lee. {ExpRace}: Exploiting kernel races through raising interrupts. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual, Aug. 2021.

[17] C. Lidbury and A. F. Donaldson. Sparse record and replay with controlled scheduling. In *Proceedings of the 2019 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Phoenix, AZ, June 2019.

[18] Linux. ipv6: annotate accesses to fn->fn_sernum, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=aafc2e3285c2d7a79b7ee15221c19fbeca7b1509.

[19] Linux. ipv6: fix data-race in fib6_info_hw_flags_set / fib6_purge_rt, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d95d6320ba7a51d61c097ffc3bcafcf70283414e.

[20] Linux. af_packet: fix data-race in packet_setsockopt / packet_setsockopt, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=e42e70ad6ae2ae511a6143d2e8da929366e58bd9.

[21] Linux. kcov: code coverage for fuzzing, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/dev-tools/kcov.rst.

[22] Linux. The kernel concurrency sanitizer (kcsan), 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/dev-tools/kcsan.rst.

[23] Linux. Explanation of the Linux-Kernel Memory Consistency Model, 2022. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/memory-model/Documentation/explanation.txt.

[24] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.

[25] Oracle. The java language specification, 2022. https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.5.

[26] S. Pailoor, A. Aday, and S. Jana. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[27] K. Serebryany and T. Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, 2009.

[28] H. Sun, Y. Shen, C. Wang, J. Liu, Y. Jiang, T. Chen, and A. Cui. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*, Virtual, Oct. 2021.

[29] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Beijing, China, Sept. 2019.

[30] M. Xu, S. Kashyap, H. Zhao, and T. Kim. Krace: Data race fuzzing for kernel file systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[31] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim. Fuzzing file systems via two-dimensional input space exploration. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[32] T. Zhang, D. Lee, and C. Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[33] T. Zhang, C. Jung, and D. Lee. Prorace: Practical data race detection for production use. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, Apr. 2017.