Controlled Concurrency Testing via Periodical Scheduling

Cheng Wen
CSSE, Shenzhen University
Shenzhen, China

Mengda He*
SCEDT, Teesside University
Tees Vally, UK

Bohao Wu
CSSE, Shenzhen University
Shenzhen, China

Zhiwu Xu
CSSE, Shenzhen University
Shenzhen, China

Shengchao Qin*
Huawei Hong Kong Research Center
Hong Kong, China

ABSTRACT

Controlled concurrency testing (CCT) techniques have been shown promising for concurrency bug detection. Their key insight is to control the order in which threads get executed, and attempt to explore the space of possible interleavings of a concurrent program to detect bugs. However, various challenges remain in current CCT techniques, rendering them ineffective and ad-hoc. In this paper, we propose a novel CCT technique Period. Unlike previous works, Period models the execution of concurrent programs as periodical execution, and systematically explores the space of possible interleavings, where the exploration is guided by periodical scheduling and influenced by previously tested interleavings. We have evaluated Period on 10 real-world CVEs and 36 widely-used benchmark programs, and our experimental results show that Period demonstrates superiority over other CCT techniques in both effectiveness and runtime overhead. Moreover, we have discovered 5 previously unknown concurrency bugs in real-world programs.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Formal software verification**; • **Security and privacy** → **Formal methods and theory of security**.

KEYWORDS

Concurrency Testing, Concurrency Bugs Detection, Multi-threaded Programs, Systematic Testing, Stateless Model Checking

ACM Reference Format:
Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. 2022. Controlled Concurrency Testing via Periodical Scheduling. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510178

*Corresponding authors: Shengchao Qin and Mengda He.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510178

1 INTRODUCTION

To make the best of the computing power brought with modern multiprocessor hardware, concurrent programming is now prevalent. However, it is difficult to ensure that a concurrent program is bug-free, as unlike sequential programs whose behavioural nondeterminism mainly comes from their input. The behavior of concurrent programs is also subject to how their threads interleave, thus leaving more openings for concurrency bugs [5, 6, 44]. Testing is usually an effective way to ensure software quality; however the same issue of scheduling nondeterminism renders naïve concurrency testing insufficient in practice [39, 43, 81]. Indeed, testing a concurrent program without any scheduling control often covers only a portion of the schedule space, failing to explore the others in which the bugs may reside, even if the program is tested over and over again. Notice that detecting data race is another widely used solution to find concurrency bugs [23, 26, 36, 71], as data races are widely considered as a cause of concurrency bugs. But a data race may not be sufficient or necessary to trigger a concurrency bug. As shown in [15], 90% of data races are benign. And a concurrency bug can happen in a concurrent program that is race-free [42].

Therefore, controlled concurrency testing (CCT) techniques utilizing *controlled scheduling* have been intensively studied [4, 12, 26, 45, 64, 69]. CCT usually inserts "scheduling points" in the target program in front of some *key points/steps* (*i.e.*, instructions that are key to the program's observable behavior, such as the instructions accessing shared memory locations or synchronization primitives) and controls these key points from various threads to execute in different orders. When a bug is triggered during execution, the scheduling decisions (*i.e.*, the order in which instructions from various threads are executed) can be logged and used to deterministically reproduce the buggy execution [53, 54, 77].

A key practical challenge for CCT is *how to achieve controlled scheduling*. Usually, the target program is put in a serialized execution, that is, only one thread is picked to execute at a time; when the execution hits a scheduling point, the scheduler may decide to carry on the current execution or to pick another thread to execute (*i.e.*, a context switch is made). Existing works tend to enforce context switches via preemptions [48], sleeping delay [39, 71], or even dynamic thread priority modifications [11, 18, 42]. However, these scheduling techniques can have pathological interactions with the synchronization operations in the target program. For example, introducing preemptive synchronization (*e.g.*, preemption, lock) may lead to a false deadlock that was not originally exhibited in the target programs [49], and injecting different sleeping delays before any key point may have unpredictable results and significantly

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

slow down the execution speed. Moreover, simply serializing the execution of the target program (*e.g.*, disallowing parallelism) could introduce undesirably high overhead. This motivates us to find a more efficient and effective solution to control scheduling.

Another key challenge is *how to effectively explore the schedule space*. Notice that the size of a program's schedule space grows exponentially with the number of scheduling points in its threads. It is often practically infeasible to exhaustively iterate the schedule space for real-world programs. Schedule bounding techniques [63, 64] are often employed. The idea behind schedule bounding is that many real-world concurrency bugs have a small *bug depth*, that is, the number of context switches needed to expose the bug [22, 46]. Therefore, controlled scheduling techniques could bound the number of their schedule-interfering activities and tremendously shrink the schedule space to (ideally) a space composed of only schedules with context switches no more than a designated number. Existing works explore the schedule space in either a *randomized* or *systematic* way. Randomized testing (*e.g.*, PCT [11] and PPCT [50]) employ a randomizer to generate schedules, but they could only provide probabilistic guarantees of finding bugs. Systematic testing (*e.g.*, IPB [48, 64] and IDB [22, 64]), also known as stateless model checking [27], explore all possible schedules within a limited number of preemptions or delays, but they tend to go through a larger schedule space than needed, requiring more overhead or missing some bugs. Therefore, a more effective way to explore the schedule space is still badly needed.

In this paper, we propose a novel controlled concurrency testing technique, called Period, to achieve controlled scheduling and to effectively explore the space of possible interleavings. In Period, the execution of a concurrent program is modeled as *periodical execution*, wherein context switches can be achieved via period switches naturally. That is, Period uses a series of execution periods to host the execution of the target program and key points assigned to an execution period only get to be executed when the previous period is finished. Periodical execution can be enforced by deadline task scheduling [3], without any preemption or sleeping delay. Parallelization (*i.e.*, concurrent execution) can be readily achieved by allocating key points from different threads into the same period (in our implementation, we put them in the last period).

Period employs a period-bounding technique to explore the possible interleavings. It takes a preset period-number upper bound P and aims to detect bugs with bug-depth less than P. The schedule exploration process works as follows. It starts with the smallest possible period-number 2 (*i.e.*, for bugs with bug-depth 1). For a given period-number p, Period explores schedules systematically in a quasi-lexicographical order on the corresponding thread identifiers. Once all schedules of the current period-number p are explored, it increases the period-number p by 1 to carry on exploring, until either all possible schedules are explored or the period-number p reaches the preset bound P. Meanwhile, we explore the schedule space gradually, targeting at feasible interleavings. For that, we introduce *dynamic key point slice* to represent the key points of each thread that are covered by a dynamic run of the target program. Specifically, we statically generate all possible schedules for a starting dynamic key point slice, wherein only one key point is assumed to be covered for each thread. During the executions of the generated schedules, some new dynamic key point slices would be found, on which the exploration continues. To guide the exploration on the newly found slices, schedule prefixes are constructed based on historical executions. In addition, allowing parallelism enables us to hugely reduce the schedule space to explore and boost the performance.

We have implemented Period and performed a thorough evaluation of Period on 10 real-world CVEs and 36 widely-used benchmark programs. For comparison, we have selected 6 well-known and representative CCT techniques (*i.e.*, IPB [46], IDB [22], DFS [63], PCT [11], Maple [75] and the controlled random scheduling [64]). Our experimental results demonstrate that Period substantially outperforms existing CCT techniques in terms of bug finding ability and runtime overhead. Moreover, we have discovered 5 previously unknown concurrency bugs in real-world programs. Notice that Period focuses mainly on bugs caused by thread interleavings, e.g. user-specified assertion failure (AF), use-after-free (UAF), double-free (DF), null-pointer-dereference (NPD), deadlock (DL), etc.

Our main contributions are summarized as follows:

- We model the execution of concurrent programs as periodical execution, which uses non-preemptive synchronization to achieve controlled scheduling and allows parallelism.
- We propose a novel systematic schedule generator that works for each dynamic key points slice of a concurrent program. The proposed schedule generator allows parallelism and is equipped with a feedback analyzer that uses schedule prefixes to guide further schedule generation, hugely reducing the schedule space needed to explore.
- We have implemented Period and our experimental evaluation confirms the superiority of Period over existing CCT techniques.

2 OVERVIEW

In this section, we give a high level overview of Period through a simple motivating example selected from the CVEs [20].

2.1 Motivating Example

To illustrate our technique, Fig. 1 shows an example simplified from CVE-2016-1972 [21]. Two threads T_0, T_1 concurrently invoke the function `once()`, and are synchronized with the help of three variables (*i.e.*, `lock`, `done`, `waiters`). The variable `lock`, allocated in the main thread (Ln. 20) and released in the child thread (Ln. 15), is used to protect the critical section (Ln. 9-13). The variable `done` will be set to 1 once the critical section is completed (Ln. 11-12). Threads created after a thread finishes the critical section would return directly (Ln. 6-7). The variable `waiters` indicates the number of threads waiting to enter the critical section (Ln. 8). The expected behavior is that only the last thread should release `lock` (Ln. 14-16). Note that all statements except for "`return 0`" in function `once()` either access shared memory locations or contain synchronization primitives, therefore are considered as key points.

This program demonstrates three kinds of concurrency bugs, namely, null-pointer-dereference (NPD), use-after-free (UAF), and double-free (DF), all of which have been detected by Period. In more detail, if the `lock` is set to null in T_0 at Ln. 16 before T_1 uses the `lock` at Ln. 9, a NPD will occur. A UAF can be triggered at Ln. 9 where thread T_0 releases the `lock` at Ln. 15 and thread T_1 uses the `lock` at Ln. 9. A DF can be triggered at Ln. 15 where both threads

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

```
1  static pthread_mutex_t* lock;
2  static long waiters = 0;
3  static int done = 0;
4
5  void *once(void *) {
6      if(done)
7          return 0;
8      ++waiters;
9      pthread_mutex_lock(lock);
10     // do some thing ...
11     if(!done)
12         done = 1;
13     pthread_mutex_unlock(lock);
14     if(!--waiters) {
15         free(lock);
16         lock = NULL;
17     }
18 }
19 void main() {
20     lock = malloc(sizeof(pthread_mutex_t));
21     pthread_t T0, T1;
22     pthread_create(&T0, NULL, once, NULL);
23     pthread_create(&T1, NULL, once, NULL);
24     pthread_join(T1, NULL);
25     pthread_join(T0, NULL);
26 }
```

Figure 1: An example simplified from *CVE-2016-1972*.

T_0 and T_1 try to release the `lock`. All these bugs are actually hard to trigger, as they each require a specific sequence of operations on variable `done`, `waiter`, and `lock`.

We have tested this example with several existing CCT techniques. Maple [75] can detect NPD but misses UAF and DF, as it heuristically steers thread scheduling to attempt to force a set of predefined interleaving idioms, which does not include the required interleaving idioms of such UAF and DF in the example. PCT [11] relies on a randomizer to explore the schedule space of the example. The probability for finding NPD is high, but the probabilities for finding UAF and DF are very low, which are respectively about 0.15% and 0% in our experiment. IPB [46] and IDB [22] are two representative systematic techniques. Both IPB and IDB can detect NPD and UAF, but miss DF, whose bug depth is 5 (*i.e.*, requiring at least 5 context switches to expose the bug). The reason is that they try to iterate the scheduling decision on each key point, going through a large schedule space. Moreover, their bounds could not faithfully reflect the context switch bounds in that they often require (potentially a lot) more context switches than the depth of the bugs they try to expose. These results demonstrate the limitations of current CCT techniques as shown in §1.

Let us now illustrate our approach with this example and explain how Period detects all three bugs.

2.2 Approach Overview

The workflow of our proposed controlled concurrency testing technique Period is shown in Fig. 2. It comprises three main components: *schedule generator* (①), *periodical executor* (②) and *feedback analyzer* (③). The schedule generator systematically generates schedules for a dynamic key point slice of the target program and feeds them to the periodical executor. The periodical executor controls the thread interleavings of the target program through periodical execution, following the schedules generated by the schedule generator. The periodical executor is also responsible for collecting

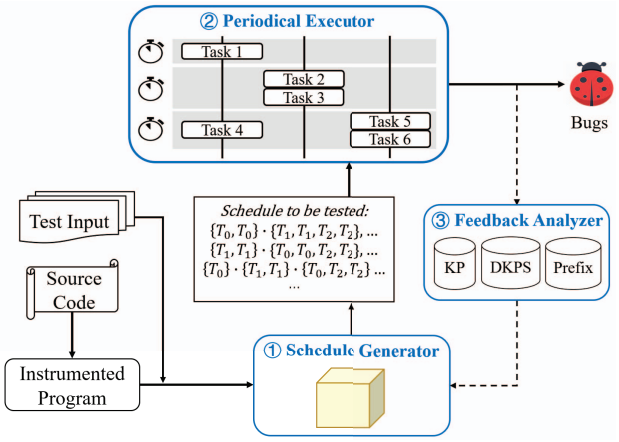

Figure 2: The workflow of Period.

the runtime information, such as the error information and the activated key points. The feedback analyzer makes use of historical execution information to guide the schedule generator to effectively generate legal schedules to cover more untested interleavings.

Considering the motivating example in Fig. 1, let us start with the period-num 2 and the starting dynamic key point slice (DKPS) $s_0 = [T_0{:}[6],\ T_1{:}[6]]$, where the integers denote the line numbers of the statements in Fig. 1 and only the first key point is assumed to be covered. The first step is to generate the schedules with 2 periods for this DKPS. A schedule is a series of execution periods and is represented as $\{\ldots\}{\cdot}\{\ldots\}\ldots{\cdot}\{\ldots\}$, where $\{\ldots\}$ represents a period containing the key points that would be executed in this period. By assigning one thread to a period, we obtain two schedules: $\{T_0{:}6\}{\cdot}\{T_1{:}6\}$ and $\{T_1{:}6\}{\cdot}\{T_0{:}6\}$. Indeed, a scheduler is concerned only with the number of key points from various threads and the order they interleave in. So we can omit key points safely and order the above two schedules lexicographically: $\{T_0\}{\cdot}\{T_1\}$ and $\{T_1\}{\cdot}\{T_0\}$.

In the second step, guided by the above two schedules, we would like to control the thread interleavings via periodical execution. Fig. 3(a) gives the execution guided by the first schedule, where the extra key points of thread T_0 are put in the last period of thread T_0 (in this case, there is only one period, namely period 1, allocated for T_0, hence all key points of T_0 are put in period 1). By analyzing the runtime information, we obtain a new DKPS $s_1 = [T_0{:}[6, 8, 9, 11, 12, 13, 14, 15, 16],\ T_1{:}[6]]$ (new key points are colored gold in Fig. 3(a)). Likewise, we obtain another different DKPS $s_2 = [T_0{:}[6],\ T_1{:}[6, 8, 9, 11, 12, 13, 14, 15, 16]]$ (see Fig. 3(b)) for the second schedule.

If a DKPS obtained by an execution is previously uncovered, we consider the DKPS as an interesting new behavior. A new DKPS indicates there may be some other uncovered feasible schedules of the target program. To explore these schedules, we create a new exploration job to handle it. So two schedule jobs[1] are created for the above newly obtained DKPSs s_1 and s_2, respectively. Moreover, to guide the target program running into the new behavior, we introduce *schedule prefixes*. Intuitively, a prefix is a partial schedule that contains context switches needed to reach a new DKPS. The

[1] Actually, s_1 and s_2 are symmetric, to explore one of them would be sufficient. In the following, we omit the exploration on s_2.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

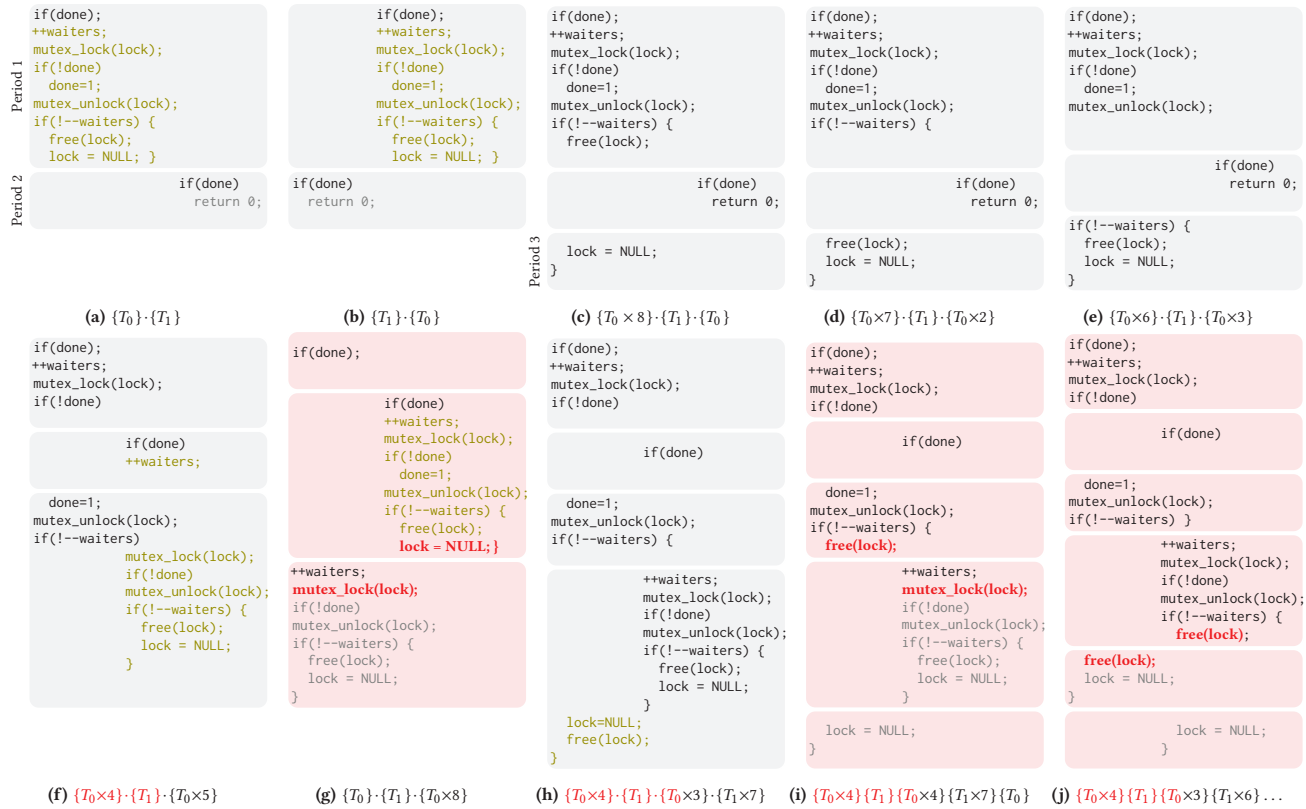

Figure 3: Each sub-figure is an execution of the CVE-2016-1972 program, and their captions are the generated schedule they attempt to follow.

schedule prefixes for the s_1 and s_2 are respectively $[T_0]$ (indicating that the T_0 must be chosen to run first) and $[T_1]$. In general, a schedule's prefix is in the form of $\{\ldots\}\cdots\{\ldots\}\cdot[T_i]$, where the periods in curly brackets will be literally preserved in the exploration job following this prefix. The prefix's last period in square brackets indicates that only key points from T_i can be scheduled to this period while the key points' number can be changed.

After exploring the only 2 schedules for s_0 the exploration job for s_0 is concluded. However, we still need to explore the job for s_1 with prefix $[T_0]$. We omit its less interesting 2-period schedules and jump to the 3-period phase, where there are 8 schedules to explore, namely $\{T_0{\times}8\}\cdot\{T_1\}\cdot\{T_0\}$, ..., and $\{T_0\}\cdot\{T_1\}\cdot\{T_0{\times}8\}$.The executions of the first three schedules are shown in Fig. 3(c), Fig. 3(d) and Fig. 3(e), respectively.

On the fifth 3-period schedule for s_1, we find a new DKPS s_3 (see Fig. 3(f)), which contains 7 key points in T_0 and 8 key points in T_1. A new exploration job will be created for s_3 associated with the prefix $\{T_0{\times}4\}\cdot[T_1]$ needed to lead us to s_3. Meanwhile, we continue with the job for s_1 and eventually when we get to schedule $\{T_0\}\cdot\{T_1\}\cdot\{T_0{\times}8\}$, the NPD bug will be triggered (see Fig. 3(g)).

Now we focus on the exploration job for s_3 with the prefix $\{T_0{\times}4\}\cdot[T_1]$. We skip the inconsequential schedules and have a close look at the schedule $\{T_0{\times}4\}\cdot\{T_1\}\cdot\{T_0{\times}3\}\cdot\{T_1{\times}7\}$ (Fig. 3(h)), where a new DKPS s_4 is discovered and its prefix is $\{T_0{\times}4\}\cdot\{T_1\}\cdot[T_0]$. Again, we skip some schedules and consider the exploration on the newly-found DKPS s_4. As shown in Fig. 3(i) one of its 5-period schedules $\{T_0{\times}4\}\cdot\{T_1\}\cdot\{T_0{\times}4\}\cdot\{T_1{\times}7\}\cdot\{T_0\}$ triggers the UAF bug. Next, we continue the exploration on s_4 with period-number 6. By using the schedule $\{T_0\times4\}\cdot\{T_1\}\cdot\{T_0{\times}3\}\cdot\{T_1\times6\}\cdot\{T_0{\times}2\}\cdot\{T_1\}$, we are able to trigger the DF bug, as shown in Fig. 3(j). These findings illustrate the effectiveness of Period.

Finally, we stop the exploration when all the obtained DKPSs are explored within the preset period-number bound. In this example, the period-number bound is set to be 6 (or any larger number) so that the NPD, UAF, and DF can be found.

3 METHODOLOGY

We have illustrated how Period works with an example. We shall now present some essential technical details.

3.1 The Top-level Algorithm

Given a concurrent program *Prog* with n threads[2], we assume the input of the target program is given, so the only nondeterminism in execution would be caused by thread interleaving. Different interleavings may lead the execution to divergent paths, that is, different parts of the program get executed/activated. Only the interleavings of the key points are interesting, so we abstract the

[2]We assume the number of active threads n does not change in different executions for simplicity, though our model allows changing number of threads.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

active parts of the program as a dynamic key point slice (DKPS or slice for short), which is represented as a list with n elements, wherein the ith element is a list consisting of every encountered key points in thread T_i's execution in the chronological order. Since a DKPS naturally reflects that there must be some feasible schedules to activate it, Period aims to systematically detect all slices and test each of them with all possible schedules.

Period models the execution of concurrent programs as periodical execution, wherein context switches can be achieved via period switches naturally (see §3.3 for more detail). Motivated by the study [11, 44] that shows many real-world concurrency bugs have shallow depths, Period employs schedule bounding techniques as well and requires the period numbers of the generated schedules to be within a preset bound P, targeting at all potential bugs with depths less than P.

The top-level algorithm of Period is illustrated in Alg. 1, which takes an instrumented program *Prog* with n threads and a preset period bound P as input, and returns a set of bugs and their corresponding schedules. Note that the target program is instrumented so that it follows our schedules and we can collect the runtime information. We create "jobs" for slices to be explored in *Prog*, with each job represented by a pair consisting of a DKPS and a schedule prefix. The schedule prefix is used to lead the execution of the target program to the DKPS as possible. The initial job is denoted as a pair consisting of the smallest approximation assuming there is only one active key point in each thread and an empty schedule prefix (Ln. 2). Indeed, this smallest approximation is used to allow different possible orders for various threads to start running. Each schedule job is performed with the period-number from 2 (*i.e.*, 1 context switch targeting at the bug depth 1) to the preset period bound P (Ln. 4-17). Specifically, for each job *job* and each period-number p, we fisrt generate all schedules (Ln. 6) by invoking our *schedule generator* (§3.2). Note that a scheduler is concerned only with the number of key points from various threads and the order they interleave. We test the target program guided by each generated schedule (Ln. 11) via our *periodical executor* (§3.3), and log bugs captured by our executor (Ln. 12) . Our executor also captures the dynamic key point slice of the execution (Ln. 11). Therefore, the *feedback analyzer* (§3.4) can decide whether a new job should be created to explore the newly found slice or not (Ln. 14). Notice that we may be unable to generate schedules for some job (Ln. 7). For instance, if p were greater than a job's total number of key points, then some periods would be left empty and wasted. In this case, we say the job is finished, that is, the exploration of the corresponding slice is done, and we remove it from *Jobs* (Ln. 8).

Algorithm 1: Period Systematic Concurrency Testing

```
   Input  : an instrumented program Prog, number of worker threads n,
            and a bound P for the maximum periods
   Output : a set Log recording bugs and their corresponding schedules
1  Log ← ∅
2  Jobs ← {([1] × n, ϵ)}
3  p = 2                          // the period-number, starting from 2
4  while p ≤ P do
5      foreach job in Jobs do
           /* schedule generator generates all schedules for 'job'
              with the current period-number p                    */
6          Schedules ← SchedGen(job.dkps, p, job.prefix)
7          if Schedules = ∅ then          // the current job is done
8              Jobs ← Jobs \ {job}        // remove it from the Jobs
9              continue
10         foreach s ∈ Schedules do
11             dkps, Errors ← Run(Prog, s)   // periodical executor
               /* log the bugs and the current schedule          */
12             Log ← Log ∪ {(s, e) | e ∈ Errors}
               /* feedback analyzer                              */
13             prefix = GetPrefix(s)
14             Jobs ← Update(Jobs, dkps, prefix);
15     if Jobs = ∅ then           // all feasible schedules explored
16         break
17     p ← p + 1      // to explore schedules with one more period
```

3.2 Schedule Generator

Our schedule generator focuses on scheduling key points of a slice. To see how our periodical schedule generator is designed, we first introduce the serialized scheduler which lays the foundation of our method, and then present the key optimization that allows concurrency in our schedules and that greatly reduces the number of schedules needed to expose bugs. Without loss of generality, we name the threads in a DKPS as $T_0, T_1, \ldots T_{n-1}$, where $n = |DKPS|$.

3.2.1 Serialized Scheduler. Period models program execution in a series of execution periods, that is, a schedule, and represents it as $\{\ldots\}\cdot\{\ldots\}\ldots\cdot\{\ldots\}$, where $\{\ldots\}$ denotes an execution period and is a multi-set that contains thread identifiers. Note that key points are omitted, as our scheduler is concerned only with the number of key points from various threads and the order they interleave. For example, the schedule with three execution periods $\{T_0, T_0\} \cdot \{T_1\} \cdot \{T_0\}$ indicates that thread T_0 can take two key steps in the first execution period; then it is T_1's turn to take one key step in the second execution period, before T_0 takes another key step in the last execution period. We use $\{T_x \times n\}$ for n occurrences of T_x appearing in a period.

For any DKPS, we can create serialized schedules by imposing the following rules:

- *Rule 1.* Each period only hosts key points from the same thread.
- *Rule 2.* Key points in two adjacent periods should belong to different threads.
- *Rule 3.* No execution period is left empty.
- *Rule 4.* Thread T_i appears exactly $|DKPS[i]|$ times in a schedule.

These rules define the space of serialized periodical schedules for the slice. Consider a schedule job on a slice DKPS with a period-number p. To iteratively generate schedules within the space of DKPS with p, we first introduce *schedule patterns*: skeletons of schedules and represented as $[T_{id_0}]\cdot[T_{id_1}]\ldots\cdot[T_{id_p}]$, where $[T_{id_i}]$ denotes that only T_{id_i} can be scheduled into the corresponding period. All the possible patterns would be generated in a lexicographical order on the thread identifiers. Then for each possible pattern, we generate all schedules in order by allocating the key points to their corresponding periods. In this way, we can systematically explore all schedules on DKPS with period-number p in a quasi-lexicographical order. In the following, we illustrate our exploration of 4-period schedules for the slice *DKPS* that has 3 key points in T_0, 2 key points in T_1 and 1 key point in T_2.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

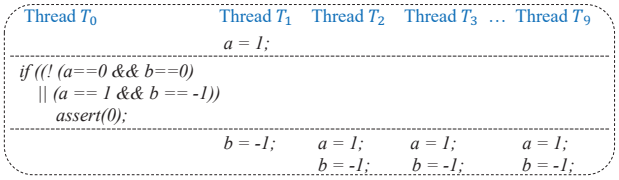

Figure 4: A buggy interleaving of program *reorder_10_bad*.

Firstly, let us consider the first possible pattern, we take T_0 for the first period. Due to Rule 2, we could not use T_0 again for the second period. By lexicographical order, we take T_1 for the second period and then T_0 again for the third period. For the last period, we notice T_1 is the smallest candidate. But taking T_1 would make T_2 left unscheduled, violating Rule 4. So we have to take T_2 for the last period, yielding the first pattern $[T_0]\cdot[T_1]\cdot[T_0]\cdot[T_2]$. By replacing some period with a (next) larger one and reinitializing the periods after it, we can construct the other patterns in order. For example, the next pattern after the first one would be $[T_0]\cdot[T_1]\cdot[T_2]\cdot[T_0]$.

Once the patterns are decided, we can then allocate the key points to their corresponding periods. First of all, by Rule 3, we put one key point into each period. We only need to arrange the remaining key points, which might have different ways. Taking $[T_0]\cdot[T_1]\cdot[T_0]\cdot[T_2]$ for example, the remaining T_1 can only be put in the second period, while the T_0 could be put in either the first or the third period, yielding two different schedules $\{T_0\}\cdot\{T_1\times 2\}\cdot\{T_0\times 2\}\cdot\{T_2\}$ and $\{T_0\times 2\}\cdot\{T_1\times 2\}\cdot\{T_0\}\cdot\{T_2\}$. Likewise, we can generate all schedules in order for a given pattern by exploring all possible ways to arrange the key points.

3.2.2 The Parallel Scheduler. Although it systematically explores the schedule space of a slice, our serialized scheduler could be costly for slices with many threads. In the worst case scenario, to expose a d-depth bug in an n-thread slice DKPS, we will have to explore the serialized schedule space with $d+n-1$ periods. We can also estimate the size of the space as $(n\times k)^{d+n-1}$, where k is the maximum number of key points in a thread in DKPS. Apparently, it could be too huge if n is big. For instance, the program in Fig. 4 takes 2 context switches between T_0 and any another thread to trigger the assertion error. But our serialized scheduler has to generate schedules with 11 periods due to Rule 1 and is unable to trigger this error within 10,000 schedules (see Table 2).

In our definitive scheduler (which is termed as "the parallel scheduler"), we address this problem elegantly by allowing some threads to be executed in parallel. Specifically, at a time we pick a set of threads under surveillance. We only serialize these "*chosen*" threads so we can check how they interact. All the other threads are neglected for now and left to run freely in the last period. For instance, assuming T_0 and T_1 are chosen, a 3-period schedule $\{T_1\}\cdot\{T_0\}\cdot\{T_1, T_2, \ldots, T_{10}\}$ is able to trigger the assertion error in Fig. 4. So to allow parallelism, we loosen Rule 1:

- *Rule 1'.* Each period (apart from the last one) only hosts key points from the same thread.

Note that we do not impose any controls over the threads scheduled in the last period, but for consistency, we write them in the lexicographical order.

It is possible that some bugs may be in the neglected threads. This is not an issue as we systematically (following the lexicographical order) choose threads and would eventually reveal bugs caused by any thread combination. Particularly, when producing the p-period schedules, we choose all 2 to $min(p, n)$-thread combinations. For every thread combination, using only the chosen threads and their key points, we first generate serialized schedules as the mid-product, and then for every such serialized schedule we add the neglected key points in its last period to produce a complete schedule.

Allowing parallelism puts every period in use to create meaningful context switches. With the parallel scheduler for any n-thread program/slice ($n \geq 2$), no matter how big n is, we can always start with period-number 2 (instead of n), and can trigger a d-depth bug with no more than $d+1$ periods (instead of $d+n-1$ in the worst case).

This process can be further guided by a schedule *prefix* (see §3.4). A prefix is in the form $\{T_{id_0}\times n_0\}\ldots\cdot\{T_{id_i}\times n_i\}\cdot[T_{id_{i+1}}]\ldots\cdot[T_{id_j}]$. We say a schedule satisfies such a prefix if its first i periods are exactly the same as the prefix's first i periods and its next j periods have the same pattern as that is given in the prefix. Our schedule generator function $\texttt{SchedGen}(DKPS, p, pfx)$ generates all p-period schedules for a slice DKPS satisfying the prefix pfx.

3.3 Periodical Executor

After schedule generation, we would like to enforce the schedules in the executions of the target program. Specially, we use a series of execution periods to host the executions, called *periodical execution*, and impose the following rules: (*i*) key points assigned to a period only get to be executed when the previous period is completed; (*ii*) each period has a lifetime, which should be long enough to cover the key points hosted in any period assuming they are executable; and (*iii*) a period is completed if the lifetime is over.

We implement the periodical executor based on Linux's deadline task scheduling [3], which is originally designed for real-time systems that need tasks to be done *periodically*. For our purpose, we adapt deadline task scheduling for our use by putting all threads under the deadline tasking scheduling with the same period length and start time; so the execution periods for all threads are always synchronized and can be used to fulfill our period based controlled execution. We also instrument the target program with a scheduling point in front of every key points.

Fig. 5 gives the periodical execution for the schedule $\{T_1\}\cdot\{T_0\}\cdot\{T_1, T_2, \ldots, T_{10}\}$, where `sched_yield()` is triggered at the scheduling point to hang the current thread's execution until the next period. Let $\texttt{Run}(P, s)$ denote the periodical execution of the target program P guided by a periodical schedule s.

Our periodical executor is also responsible for collecting information from the execution: (*i*) the error information $Errors$, from which we can know if a bug is triggered, and (*ii*) the activated slice DKPS, which is fed to our feedback analyzer (§3.4) to decide whether a new schedule job should be created. In other words, our periodical executor returns a pair $(DKPS, Errors)$. Note that, the slice newly activated by a schedule may not be the same to the one that generates the schedule. This is due to the fact that different interleavings may cause conditional key points to take different branches. For instance, the schedules shown in Fig. 3(e) and Fig. 3(f)

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

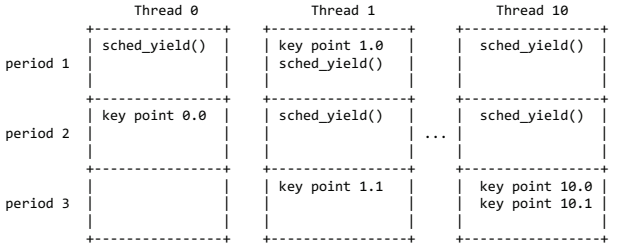

Figure 5: The execution of $\{T_1\} \cdot \{T_0\} \cdot \{T_1, T_2, \ldots, T_{10}\}$

are generated from the same DKPS, but they cause "`if(done)`" in thread T_1 to take different branches, yielding two different DKPSs.

3.4 Feedback Analyzer

As explained in §3.3, the newly activated slice may be different from the original one. Our feedback analyzer can handle this situation. For that, we introduce the *supported* relation between DKPSs: $DKPS_1$ is *supported* by $DKPS_2$ if the number of key points for each thread in $DKPS_1$ is not larger than the corresponding one in $DKPS_2$. From the periodical executor's point of view, $DKPS_1$ is supported by $DKPS_2$, indicates that the schedules generated by $DKPS_2$ provide enough "space" of periods to hold their key points from $DKPS_1$ for each thread. Considering the motivating example, the s_4 is supported by the s_3, just skipping unneeded key points or periods.

When a previously uncovered slice DKPS is discovered, our feedback analyzer first checks if it is supported by the original one. If not, we will consider to create a new job for it. To guide the exploration on DKPS, we make full use of the history execution information. We compare the current schedule with its immediate previous schedule[3] and locate the first different key point; let us say it is a key point from T_i. We can assume the difference here leads us to the new slice, so we construct a schedule prefix by keeping everything before it literally, and then adding a pattern period $[T_i]$. For instance, the schedule $\{T_0 \times 4\}\cdot\{T_1\}\cdot\{T_0 \times 5\}$ in the motivating example (Fig. 3(f)) gives us a new slice. To calculate its prefix, we compare it with its immediate previous schedule, $\{T_0 \times 5\}\cdot\{T_1\}\cdot\{T_0 \times 4\}$, and locate the first different key point, which is the fifth key point T_1 of the schedule in Fig. 3(f), where its previous schedule has T_0. So we keep the $\{T_0 \times 4\}$ before this key point and connect it with a $[T_1]$ to get the prefix: $\{T_0 \times 4\}\cdot[T_1]$. Note that there may already exist a job on this slice DKPS. If so, we update the prefix of the existing job as the common prefix of the newly constructed one and the original one. Otherwise, a new job consisting of DKPS and the newly constructed prefix is created and put into the job list $Jobs$. The procedure is denoted as `Update`$(Jobs, DKPS, prefix)$.

Intuitively, one could explore the schedule space on the slice containing all the key points of the target program instead. However, this space could be too huge and suffers from lots of useless schedules. Considering the motivating example again, any schedule satisfying that at least 5 key points from thread T_0 have been executed prior to the execution of the first key point of thread T_1 would make thread T_1 return directly. There are too many such schedules but only one is sufficient. Moreover, during our exploration, the number of generated schedules grows rapidly, as the size of the slice increases. Thanks to the schedule prefix again, it helps us significantly reducing the schedule space via avoiding some duplicated schedules. As in the motivating example, when exploring s_3 we use the prefix $\{T_0 \times 4\}\cdot[T_1]$ which will guide the execution to s_3. Without prefix, schedules like $\{T_0 \times 5\}\ldots$ would be allowed leading the execution back to s_1 instead of s_3 (Fig.3(c)-3(e)). Finally, concerning the completeness, since our scheduler systematically explores all schedules on all interesting DKPSs with period-number from 2 to P, we argue that most of the possible interleavings of various threads bounded by P context switches will be touched by Period. As illustrated in Fig. 3, all the branches of three conditional statements can be covered and each conditional statement can be shuffled in any possible position if period-number p is enough.

[3]If it is the first schedule in our exploration, we say its immediate previous schedule is an empty schedule $\{\}$.

4 EVALUATION

We have built a prototype for Period upon the LLVM framework [37], SVF [62] and *SCHED_DEADLINE* [9]. In particular, the periodical executor and the feedback analyzer rely on instrumentation based on the LLVM framework. We have implemented a static analysis component, which statically identified key points of the given concurrent program, on top of SVF [62]. The underlying implementation of the periodical execution uses the existing CPU scheduler *SCHED_DEADLINE* available in the Linux kernel, as the implementation vehicle.

We have conducted thorough experiments to evaluate Period with a set of widely-used benchmarks, and compared it with various existing techniques. With these experiments, we aim to answer the following research questions:

RQ1. How capable is our proposed parallel scheduler in reducing the schedule space, compared to the serialized scheduler?

RQ2. How capable is Period in terms of finding concurrency bugs, compared to other techniques?

RQ3. What runtime overhead is incurred by Period?

4.1 Evaluation Setup

4.1.1 Benchmark Programs. To evaluate Period, we make use of a set of widely-used benchmarks and real-world CVEs, written in C/C++ for the Linux/Pthreads platform, including 36 programs from SCTBench [1] and all 10 programs from the CVE benchmark [2]. The CVE benchmark contains 10 programs that have various concurrency bugs, and each program corresponds to a real-world CVE. The SCTBench collects 52 concurrency bugs from previous parallel workloads [10, 70] and concurrency testing/verification works [19, 48, 73, 74]. Note that we exclude 16 programs in SCTBench as 5 of them fail to compile on the LLVM platform and the bugs in the other 11 programs can be exposed 100% of the time.

4.1.2 Baselines. We used existing implementations of compared baselines when available. Based on the category of CCT techniques, we select 3 systematic CCT techniques: IPB [46], IDB [22], and DFS [64], and 3 non-systematic CCT techniques: PCT [11], Maple [75] and a controlled random scheduler (Random) that randomly chooses a thread to execute at a time. For comparison, we

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

Table 1: Descriptive statistics and detection results on CVE benchmark

				Systematic Testing												Non-systematic Testing								Con. Bug Detector		Data Race Detector		
				Period / Serial			IPB			IDB			DFS			Native		PCT		Random		Maple							
Bug ID	Programs	Bug Type	Bug Depth	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	runs to 1st bug	buggy runs	schedules to bug (1st)	buggy schedules	schedules to bug (1st)	buggy schedules	founds?	schedules	ConVul	UFO / UFO_{NPD}	FastTrack	Helgrind	TSAN
CVE-2009-3547	Linux-2.6.32-rc6	NPD	1	2	6	3	3	36	5	5	33	4	4	10	1	249	3	5	3333	8	2506	✓	30	✓	✓	✓	✓	✓
CVE-2011-2183	Linux-2.6.39-3	NPD	2	3	906	130	8	98	11	6	85	9	5	31	3	681	10	5	394	8	3745	✓	60	✗	✗	✗	✗	✗
CVE-2013-1792	Linux-2.8.3	NPD	2	13	179	6	15	321	18	22	260	14	15	88	5	✗	0	61	124	8	741	✓	165	✓	✗	✗	✗	✗
CVE-2015-7550	Linux-4.3.4	NPD	2	3	14	6	8	73	11	6	64	9	5	22	3	✗	0	3	394	1	3745	✓	160	✓	✓	✗	✗	✗
CVE-2016-1972	Firefox-45.0	NPD	2	3		20	16		881	11		472	228		90	✗	0	3	731	55	430	✓		✗	✗	✗	✗(△)	✗(△)
		UAF	4	159	573	11	91	L	918	66	6176	663	229	L	337	✗	0	134	15	1	1539	✗	144	✓	✗	✗	✗(△)	✗(△)
		DF	5	447		1	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✗		✗	✗	✗	✗(△)	✗(△)
CVE-2016-1973	Firefox-45.0	NPD	2	5	31	2	✗		0	✗		0	✗		0	1415	3	✗	0	✗	0	✓		✗	✗	✗	✗	✗
		UAF	3	17		5	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	157	✓	✗	✗	✗	✗
CVE-2016-7911	Linux-4.6.6	NPD	2	3	19	8	8	204	66	6	170	54	5	58	21	799	15	5	511	5	3733	✓	143	✓	✗	✗(△)	✗(△)	✗(△)
CVE-2016-9806	Linux-4.6.3	DF	2	6	42	4	9	226	84	7	193	65	6	71	28	✗	0	3	1135	1	2353	✓	36	✓	-	✗(△)	✗	✗(△)
CVE-2017-15265	Linux-4.9.13	UAF	2	11	96	1	✗	88	0	✗	83	0	✗	31	0	✗	0	✗	0	✗	0	✓	73	✓	✓	✗	✗	✓
CVE-2017-6346	Linux-4.13.8	NPD	2	5		60	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✓		✗	✗	✗(△)	✗(△)	✗(△)
		UAF	3	47	182	6	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	118	✓	✗	✗(△)	✗(△)	✗(△)
		DF	2	46		14	✗		0	✗		0	✗		0	✗	0	✗	0	20	1625	✓		✓	-	✗(△)	✗(△)	✗(△)
Total bugs found (Buggy Programs)				15 (10)			8 (7)			8 (7)			8 (7)			4 (3)		8 (7)		9 (7)		11 (7)		10 (9)	3 (3)	1 (1)	1 (1)	2 (2)

* All the sub-thread numbers of programs are 2. NPD, UAF, and DF are short for null-pointer-deference, use-after-free , and double-free, respectively. 'L' denotes our schedule limit 10,000 is reached. '✗' denotes that no bug was found. "-" denotes an inapplicable case. △ denotes that a race detector reports a race on the related variables of the concurrency bug.

also include the version of Period equipped with our serialized scheduler (we call this version of Period as Serial) and the native execution (Native) wherein schedules are uncontrolled.

4.1.3 Configuration Parameters. In our experiments, the schedule bounds for all the CCT techniques are set to check bugs with bug-depth no more than 5 on the CVE Benchmark (resp. 3 on SCTBench), as the maximum bug-depth for known bugs in the CVE-benchmark (resp. SCTbench) is 5 (resp. 3). Each invocation of a CCT technique has a budget of exploring up to 10,000 schedules. For the other non-CCT techniques, we adopted their default configurations. For each compared technique, we invoke tests run for each program 10 times and collect their results. All our experiments have been performed on a workstation with an Intel(R) Xeon(R) Silver 4214 processor, installed with Ubuntu 18.04, GCC 7.5, LLVM 10.0.

4.2 Improvement of Parallel Scheduler (RQ1)

The descriptive statistics and detection results on the CVE benchmark [2] and SCTBench [1] are shown in Table 1 and Table 2, respectively. Each column denotes the experimental results of a technique. The *schedules to bug (1st)*, *schedules*, and *buggy schedules* denote the number of schedules that were explored up to and including the detection of a bug for the first time, the total number of schedules explored by a technique, and the number of explored schedules that exhibited the bug. These figures can demonstrate how capable and how quickly each technique finds the bugs on these benchmark programs.

As shown in Table 1, the results for Period (which uses the parallel scheduler) and Serial on the CVE benchmark are exactly the same. This is because, as discussed in §3.2.2, the schedule space sizes for Serial and Period to explore on programs with n threads and a d-depth bug are respectively over-approximated as $(n{\times}k)^{n+d-1}$ and $(n{\times}k)^{d+1}$, and all the thread numbers of programs in the CVE benchmark are 2 so that both space sizes are exactly the same. While their results on SCTBench are different and highlighted in blue in Table 2. All the differences are due to the thread numbers are larger than 2. In fact, when testing programs with a large thread number, Serial could require a larger schedule space than needed to detect bugs, resulting in missing some bugs, even the ones with depth 2. While, thanks to parallelization, Period can always trigger a d-depth bug with no more than $d+1$ periods. Table 2 shows that Period reports 8 more bugs than Serial, with an improvement about 26.67%.

Moreover, allowing parallelism greatly improves the schedule space. For example, Serial generates 30, 384 and 5040 schedules for the 3-threads, 4-threads, and 5-threads versions of *CS.reorder_bad* (*CS.reorder_3_bad*, *CS.reorder_4_bad* and *CS.reorder_5_bad*), respectively. When the thread number grows to 10 (*CS.reorder_10_bad*), Serial requires a particularly large number of schedules, which quickly exceeds the budget limit. While Period respectively generates 27, 100, and 225 schedules for the 3-threads, 4-threads, and 5-threads versions and still performs well (2350 schedules, still lots of remaining budgets) on the 10-threads version. This indicates that, as the thread number increases, the improvement of the parallel scheduler would be more substantial. In addition, allowing parallelism also enables Period to detect bugs more quickly. As shown in Table 2, Period always requires fewer schedules to detect the same bug for the first time, compared with Serial.

> Our parallel scheduler significantly improves serialized one in terms of the schedule space, enabling us to detect more bugs.

4.3 Bug-finding Ability Evaluation (RQ2)

In Table 1, Period has successfully identified all 10 programs from the CVE benchmark as buggy ones, while other CCT techniques (*i.e.*, IPB, IDB, DFS, PCT, Random, Maple) have identified only 7 buggy ones. Notice that the native execution (*i.e.*, no control on schedules), could identify only 3 buggy programs. In terms of bugs,

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

Table 2: Descriptive statistics and detection results on SctBench

Programs	Sub Threads	Bug Type	Bug Depth	Systematic Testing: Period			Serial			IPB			IDB			DFS			Non-systematic Testing: Native		PCT		Random		Maple	
				schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	schedules to bug (1st)	schedules	buggy schedules	runs to 1st bug	buggy runs	schedules to bug (1st)	buggy schedules	schedules to bug (1st)	buggy schedules	founds?	schedules
CS.account_bad	3	AF	1	2	65	26	4	136	56	3	70	13	4	43	6	3	28	4	2903	1	5	2396	8	1177	✓	80
CS.bluetooth_driver_bad	2	AF	2	9	205	9	9	205	9	7	92	9	7	92	9	36	177	10	✗	0	700	85	8	648	✗	57
CS.carter01_bad	2	DL	2	5	42	11	5	42	11	11	396	38	9	250	18	8	1708	49	555	1	3	608	1	4750	✓	6
CS.circular_buffer_bad	2	AF	2	17	871	207	17	871	207	26	806	363	42	623	219	20	3991	2043	✗	0	11	842	1	9110	✗	58
CS.deadlock01_bad	2	DL	2	3	14	6	3	14	6	11	81	8	8	65	6	10	46	3	4353	2	38	174	1	3745	✗	89
CS.lazy01_bad	3	AF	1	3	39	12	4	60	20	1	208	13	1	87	62	1	118	81	2	6631	1	5128	2	6092	✓	1
CS.queue_bad	2	AF	2	25	998	119	25	998	119	101	L	7275	106	8310	3768	43	L	6405	7993	1	6	984	1	L	✓	64
CS.reorder_10_bad	10	AF	2	27	2350	89	✗	L	0	✗	L	0	✗	7406	0	✗	L	0	✗	0	85	9	✗	0	✗	56
CS.reorder_20_bad	20	AF	2	39	L	2870	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	891	18	✗	0	✗	56
CS.reorder_3_bad	3	AF	2	6	27	6	10	30	10	50	1192	25	33	205	6	126	2494	23	✗	0	192	54	39	237	✗	56
CS.reorder_4_bad	4	AF	2	9	100	12	37	384	90	393	L	31	262	518	7	6409	L	4	✗	0	164	40	68	86	✗	56
CS.reorder_5_bad	5	AF	2	12	225	20	283	5040	816	3587	L	3	✗	996	0	✗	L	0	✗	0	355	28	68	23	✗	56
CS.stack_bad	2	AF	2	3	918	144	3	918	144	25	2429	318	23	1595	273	22	L	512	✗	0	15	6	3	6189	✓	2
CS.token_ring_bad	4	AF	1	2	232	58	7	2424	401	8	503	114	15	113	13	8	280	57	✗	0	11	6	9	1293	✓	45
CS.twostage_100_bad	100	AF	2	690	L	141	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	13453	11	✗	0	✗	56
CS.twostage_bad	2	AF	2	4	33	3	4	33	3	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✓	8
CS.wronglock_3_bad	4	AF	2	6	172	42	26	1320	265	277	L	1227	16	1568	197	3233	L	94	7212	1	7	313	1	3197	✓	19
CS.wronglock_bad	8	AF	2	10	1464	210	✗	L	0	✗	L	0	32	L	710	✗	L	0	✗	0	44	307	1	3286	✓	19
CB.aget-bug	3	AF	2	9	3279	679	12	L	1900	1	4359	2903	1	292	194	1	2847	1814	2	3341	7	3202	4	4853	✓	1
CB.stringbuffer	2	AF	2	12	163	16	12	163	16	13	38	2	13	38	2	8	30	2	✗	0	3555	5	23	673	✓	40
CB.pbzip2	4	BOF	2	41		14	578		27	✗		0	✗		0	✗		0	✗	0	✗	0	✗	0	✗	42
		NPD	2	52	1626	41	82	L	427	16	L	36	4	L	1733	12	L	573	✗	0	62	136	1	2545	✓	
		UAF	1	53		24	83		498	2		4902	3		3053	2		1267	✗	0	7	2193	5	1594	✗	
Chess.WSQ	3	AF	3	105	434	15	574	5175	117	4502	L	306	845	L	270	✗	L	0	✗	0	2	1118	392	10	✓	49
Chess.IWSQ	3	AF	3	108	711	17	836	L	138	✗	L	0	3554	L	192	✗	1503	0	✗	0	5	1173	443	24	✗	10
Chess.SWSQ	3	BOF	3	1628	L	11	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	60
		AF	3	1630		13	✗		0	284		1	222		1	✗		0	✗	0	68	257	17	1078	✗	
Chess.IWSQWS	3	BOF	3	1661	L	11	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	70
		AF	3	1663		13	✗		0	284		1	222		1	✗		0	✗	0	68	261	3	1918	✗	
Inspect.qsort_mt	3	AF	2	27	8643	135	30	L	96	33	L	365	20	3882	158	✗	L	0	✗	0	44	194	72	100	✗	115
Inspect.boundedbuffer	4	AF	2	20	L	1514	39	L	1382	✗	L	0	608	L	103	✗	L	0	15	294	27	109	8	2808	✗	158
Misc.safestack	3	AF	-	✗	6519	0	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	59
Splash2.barnes	2	AF	1	2	L	1186	2	L	1186	3	L	1006	3	L	741	2	L	2202	7	1251	2	5013	2	4893	✓	1
Splash2.fft	2	AF	1	2	L	3963	2	L	3963	3	L	9221	3	L	9221	2	L	7210	1	6862	2	5047	2	6241	✓	2
Splash2.lu	2	AF	1	2	6129	2848	2	6129	2848	3	L	6900	3	L	6900	2	L	5560	12	2177	2	5724	2	9714	✓	4
RADBench.bug2	2	AF	3	1985	L	9	1985	L	9	✗	L	0	✗	L	0	✗	L	0	✗	0	1813	10	✗	0	✗	264
RADBench.bug3	2	AF	2	42	L	3478	42	L	3478	✗	L	0	✗	L	0	✗	L	0	✗	0	1	489	✗	0	✗	227
RADBench.bug4	2	AF	3	259	L	6	259	L	6	✗	L	0	✗	L	0	✗	L	0	4	2013	✗	0	1275	13	✓	1
RADBench.bug5	2	DL	2	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	L	0	✗	0	✗	0	✗	0	✗	224
RADBench.bug6	2	DL	2	24	2950	340	24	2950	340	30	8327	112	27	4039	70	✗	L	0	236	7	484	34	3	855	✓	14
Total bugs found (Buggy Programs)				38 (34)			30 (28)			25 (24)			27 (26)			19 (18)			13 (13)		33 (32)		29 (28)		18 (18)	

* AF, BOF, and DL are short for assertion failure, buffer-overflow, and deadlock, respectively. The marker 'L' and '✗' respectively denote our schedule limit 10,000 is reached (except program *CS.twostage_100_bad* using 100,000 as schedule limit) and that no bug was found.

Period and Serial identified 15 bugs in the CVE benchmark, performing the best. In the program of CVE-2017-6346, Period reports a null-pointer-dereference (NPD) bug, a use-after-free (UAF) bug and a double-free (DF) bug. While all other systematic CCT techniques fail to report these three bugs, and only two non-systematic CCT techniques (*i.e.*, Random and Maple) can identify NPD or DF. In particular, the DF bug in the program of CVE-2016-1972 can be found only by Period, which is also an undocumented bug. Moreover, two concurrency vulnerabilities detectors (*i.e.*, UFO [30] and ConVul [15]) and three data race detectors (*i.e.*, FastTrack [24], Helgrind [35] and TSAN [59]) are considered, whose results are included in Table 1 as well. However, most of their performances (except for ConVul) are worse than CCT techniques: ConVul and UFO identify 9 and 3 buggy programs, respectively, and the three data race detectors can identify at most 2 bugs. The above results

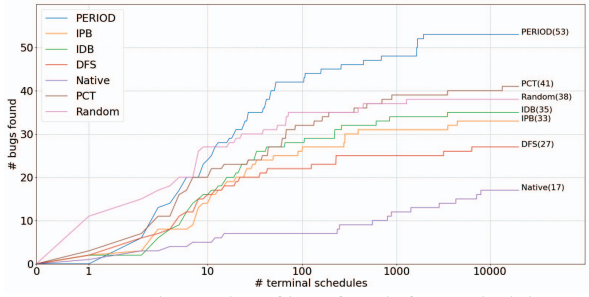

Figure 6: The number of bugs found after x schedule.

demonstrate the effectiveness of our systematic schedule exploration in terms of execution periods for both Period and Serial.

On SCTBench, as shown in Table 2, Period identified 38 bugs and 30 buggy programs in total, performing the best. In particular, Period performs the best on about half (20 out of 40) bugs in terms

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

of *schedules to bug (1st)*, and requires fewer *schedules* than other systematic testing techniques on about 83.33% (30 out of 36) programs, due to our efficient exploration of schedule space. Moreover, our Serial identified 30 bugs, performing better than the other CCT techniques (except for PCT). Due to the larger thread numbers, Serial still missed 10 bugs, 7 of which are also missed by IPB and IDB. However, about 80% of these missed bugs can be identified by our parallel version Period. The last two lucky bugs missed by all the techniques are *Misc.SafeStack* and *RADBench.bug5*. We had a manual inspection on these two bugs. *Misc.SafeStack* is an implementation of a lock-free stack [67], which requires at least 5 context switches. While the schedule bound for the CCT techniques is set to 3. In addition, as shown in a prior evaluation [64], *Misc.SafeStack* is very difficult to detect, since all techniques in [64] fail to detect it. *RADBench.bug5* was unable to reproduce in our experiments, despite that we followed the instructions in the document.

To further compare the CCT techniques, we present a cumulative plot in Fig. 6 with a schedule limit of 10,000, where each line represents a technique and is labeled by the name of the technique and the number of bugs found by the technique, and a point (x,y) represents that y concurrency bugs are exposed by the technique using x schedules. On the whole, Period has a larger growth trend, indicating that Period requires fewer schedules to find the same number of bugs or can find more bugs in the same number of schedules. For example, more than 40 bugs could be exposed by Period using lower than 100 schedules, while the others would require 1,000 to 10,000 schedules.

The above results signify that Period is more effective than the other techniques. There are two main factors that contribute to the effectiveness. Firstly, as demonstrated by the performances of our Serial and Period, the proposed gradual exploration in terms of execution periods guided by schedule prefixes is effective in detecting concurrency bugs. Secondly, as shown in §4.2, allowing parallelism can significantly improve our serialized scheduler.

Case Studies. To demonstrate the reasons behind Period's superiority, we present two case studies. The first case is the program in *CVE-2016-1972*, which suffers from the NPD, UAF and DF bugs. With the help of feedback analyzer, Period only generates 573 schedules and successfully finds all the bugs. A brief introduction for how Period find the UAF and DF bugs have been given in §2, which demonstrates our effective exploring strategies. However, both IPB and DFS generated more than 10, 000 schedules but failed to find the DF bug. The program *CS.reorder_10_bad* from SCTBench contains an assertion error. As illustrated in Fig. 4, two context switches between T_0 and any other thread can trigger the error. But most of the CCT techniques, except for our technique Period and PCT, were unable to detect it. Similar to Serial, with no supports of parallelization, existing systematic techniques have to explore a much larger schedule space than needed, causing schedule budget running out quickly, especially for programs with a large number of threads. On the other hand, although PCT can expose this error, the probability to trigger it is extremely low (about 9/10000 = 0.09% in our experiment).

> Period significantly outperforms existing CCT techniques in terms of concurrency bug finding ability.

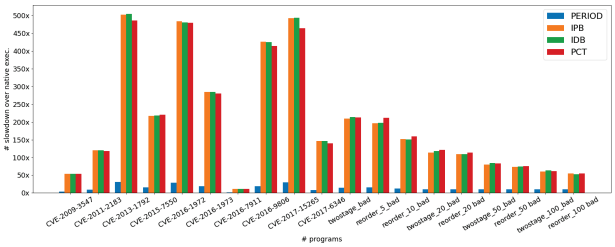

Figure 7: Average execution speed of CCT techniques relative to native execution. Lower is better.

4.4 Overhead Evaluation (RQ3)

To evaluate the runtime overhead required for achieving controlled scheduling, we use programs from the SCTbench with various numbers of threads and all the programs from the CVE benchmark. Fig. 7 shows the average execution speed during testing achieved by Period, IPB, IDB and PCT relative to native execution, where the number inside the program name represents the number of threads. On all benchmark programs, the runtime overhead incurred by Period is lower than other techniques (*i.e.*, IPB, IDB, and PCT). In detail, Period requires only 2 to 30 times of execution slowdown over native execution, while the others require 10 to 500 times. As the number of threads increases, the effect of execution slowdown can be weakened for all techniques. The reasons for the low overhead of Period could be: (*i*) the periodical execution achieves control scheduling with non-preemptive synchronizations, thus avoiding false deadlocks and starvation; and (*ii*) Period allows parallelism in periodical execution, instead of serializing execution, which can boost the performance.

> Period incurs some noticeable runtime overhead, which is significantly lower than that of IPB, IDB, and PCT.

4.5 Discussion

Additional Experiments. The above experiments show that Period is effective and efficient in finding concurrency bugs. Note that it is possible that a technique "gets lucky" and finds a bug quickly due to the search order. For that, we consider the *worst-case bug-finding ability* in terms of the total number of non-buggy schedules within the bound, that is, the difference between *schedules* and *buggy schedules*. The result shows that Period performs the best in 33 out of all 46 programs. Since Period allows parallelism in periodical execution, which could be *accelerated by multi-cores*, we also evaluate the runtime speedup with different CPU cores. For a program with 10, 20, 50, and 100 threads, Period respectively provides 8×, 11×, 17×, and 34× speedups on the 8-cores configuration, compared to the single-core one. In addition, we have tested 20 open-source programs, with source lines of code ranging from 325 to 233,431 lines. Period successfully *identified 5 previously unknown concurrency bugs* (*e.g.*, a UAF in *lrzip*, buffer-overflow in *pbzip2*, invalid address dereference in *ctrace*). These concurrency bugs were not previously reported and we have informed the maintainers. All extra experimental results are available on our website[4].

Threats to Validity. We selected a variant of existing benchmarks and real-world programs to show the capabilities of Period, and

[4]Period's website: https://sites.google.com/view/period-cct/

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

compared it against other CCT techniques. However, our evaluation dataset may still include a certain sample bias. Further studies on more real-world programs can help to better evaluate Period. Period uses static analysis to identify key points. In practice, if some key points are missed, it could result in some bugs being missed. Thus the static analysis for identifying key points should be an over-approximation. Our static analysis is currently built on top of SVF [62], and it works well for all the benchmark programs in our evaluation. A more powerful static analysis may help to improve Period further. Moreover, this work assumes that the inputs to the program are predetermined, for example, by an existing test suite, and do not vary between runs, as it is typical in other work in the literature on CCT. Adopting some test case generation techniques (*e.g.*, fuzzing and symbolic execution) might help mitigate this threat. Finally, this work doesn't handle weak memory models (WMMs) [7] and probabilistic programming models [56]. We are seeking solutions to further improve Period.

5 RELATED WORK

There is a wide range of research proposed in the literature on testing and analysis of concurrent programs. Here we briefly describe the related work and compared it to Period.

Concurrency Testing. Controlled concurrency testing has been the subject of extensive research, given the elusive nature of concurrency bugs. Chess [22, 48] showed the effectiveness of iterative preemption-bounding (IPB), and later iterative delay-bounding (IDB), for finding bugs in multi-threaded software. PCT [11] and its parallelized variation PPCT [50] set thread priorities that are used by the scheduler of the underlying operating system or runtime to schedule the threads exactly as required by the testing algorithm, showcasing the power of randomized scheduling [4, 17, 69]. Maple [75] employs a coverage-driven approach, based on a generic set of interleaving idioms, for testing multi-threaded programs. RPro [12] is a radius-aware probabilistic testing for triggering deadlocks, where it selects priority changing points within the radius of the targeted deadlocks but not among all events. TSVD [39] dynamically identifies potential thread-safety violations and injects delays to drive the program towards unsafe behaviors. QL [45] improves CCT by leveraging classical Q-learning algorithm to explore the space of possible interleavings. Several techniques are also developed for distributed systems, including dBug [60], MoDist [72], Samc [38], Spider [52] and Morpheus [79]. The presence of such a large number of techniques clearly indicates the importance of CCT. Our proposed Period achieve controlled scheduling by a novel periodical executions which is non-preemptive and allows parallelism, and it systematically explore the schedule space of each DKPS with the guidance of schedule prefixes.

Dynamic partial-order reduction (DPOR) [25, 47] computes persistent sets during testing to identify equivalent interleavings so some interleavings can be skipped if their equivalents are already tested. Some recent research has achieved considerable improvements over DPOR [16, 34, 58, 80]. It would be interesting for future work to combine DPOR techniques into Period.

Several other techniques leverage fuzzing [40, 41] to find test inputs exposing bugs in concurrent programs [18, 36, 42, 65, 71]. Challenges caused by test inputs are orthogonal to this work as we focus on finding buggy interleavings.

Static Analysis Approaches. Static analysis aims to approximate concurrent programs' behaviors without actually executing them [8, 13, 55, 61, 66]. For example, LOCKSMITH [55] uses existential types to correlate locks and data in dynamic heap structures for race detection. Goblint [66] relies on a thread-modular constant propagation and points-to analysis for detecting concurrency bugs by considering conditional locking schemes. DCUAF [8] statically detects concurrency use-after-free bugs in Linux device drivers through a summary-based lockset analysis. FSAM [61, 62] proposes a sparse flow-sensitive pointer analysis for C/C++ programs using context-sensitive thread-interleaving analysis. Canary [13] conducts interference-aware value-flow analysis for checking inter-thread value-flow bugs, achieving both good precision and scalability for millions of lines of code. The static approaches may produce false positives. In Period, the static analysis component is currently built on top of SVF, but can be replaced by a more powerful static analysis if available.

Dynamic Analysis Approaches. There is a line of work using dynamic analysis to find concurrency bugs [14, 28, 51, 76, 76, 78]. The two fundamentals are happens-before model [24] and lockset model [57]. The happens-before model reports a race condition when two threads read/write a shared memory arena in a causally unordered way, while at least one of the threads writes into this arena. The lockset model conservatively considers a potential race if two threads read/write a shared memory arena without locking. FT [24] and Helgrind [35] are two well-known happens-before based race detectors. Modern detectors such as TSan [59] apply a hybrid strategy to combine both the happens-before model [24] and the lockset model [57]. Predictive analysis [14, 15, 30, 31, 68] collects traces consisting of different types of events and then predicts bugs offline based on the dependencies of events or known bug patterns. UFO [30] applies an extended maximal causality model [29, 32, 33] to predict concurrency UAFs based on a single execution trace, even though the UAFs may not happen in the observed execution. ConVul [15] predicts concurrency vulnerabilities by judging whether two events are exchangeable based on the happens-before model. ConVulPOE [76] enhances ConVul by introducing partial-order reduction. Period's focus is not on improving dynamic detection of concurrency violation; instead, it can employ these techniques as bug detectors to work with the periodical scheduling.

6 CONCLUSION

In this paper, we present a novel controlled concurrency testing technique Period. Period models the execution of concurrent programs as periodical execution, and systematically explores the space of possible interleavings, guided by periodical scheduling and the history execution information. Our evaluation has demonstrated Period shows superiority over state-of-the-art CCT techniques in terms of both bug-finding effectiveness and runtime overhead.

ACKNOWLEDGEMENTS

This work was supported in part by the National Natural Science Foundation of China (Nos. 61972260, 61772347, 61836005), the Guangdong Basic and Applied Basic Research Foundation (No. 2019A1515011577) and the Stable Support Programs of Shenzhen City (No. 20200810150421002).

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] 2016. SCTBench: a set of C/C++ pthread benchmarks for evaluating concurrency testing techniques. Retrieved August 20, 2021. https://github.com/mc-imperial/sctbench [online].

[2] 2019. CVE Benchmark. Retrieved August 20, 2021. https://github.com/mryancai/ConVul [online].

[3] 2021. Deadline Task Scheduling. The Linux kernel user's and administrator's guide. https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html [online].

[4] Mahmoud Abdelrasoul. 2017. Promoting secondary orders of event pairs in randomized scheduling using a randomized stride. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 741–752.

[5] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, and Hans Hansson. 2017. Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications* 8, 1 (2017), 1–15.

[6] Sara Abbaspour Asadollah, Daniel Sundmark, Sigrid Eldh, Hans Hansson, and Wasif Afzal. 2017. 10 Years of research on debugging concurrent and multicore software: a systematic mapping study. *Software quality journal* 25, 1 (2017), 49–82.

[7] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. 2010. On the verification problem for weak memory models. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 7–18.

[8] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. 2019. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *28th USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX, 255–268.

[9] Alessio Balsini. 2014. SCHED DEADLINE. In *Workshop on Real-Time Scheduling in the Linux Kernel*.

[10] Christian Bienia. 2011. *Benchmarking modern multiprocessors*. Princeton University.

[11] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. *ACM SIGARCH Computer Architecture News* 38, 1 (2010), 167–178.

[12] Yan Cai and Zijiang Yang. 2016. Radius aware probabilistic testing of deadlocks with guarantees. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 356–367.

[13] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. Canary: practical static detection of inter-thread value-flow bugs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1126–1140.

[14] Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. 2021. Sound and Efficient Concurrency Bug Prediction. In *Proceedings of the 2021 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.

[15] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 2019. Detecting concurrency memory corruption vulnerabilities. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Paris, France, 706–717.

[16] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-centric dynamic partial order reduction. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

[17] Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. 2018. Testing multithreaded programs via thread speed control. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 15–25.

[18] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. MUZZ: Thread-aware Grey-box Fuzzing for Effective Bug Hunting in Multithreaded Programs. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX, Virtual, 2325–2342.

[19] Lucas Cordeiro and Bernd Fischer. 2011. Verifying multi-threaded software using SMT-based context-bounded model checking. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 331–340.

[20] The MITRE Corporation. 1999. Common Vulnerabilities and Exposures.

[21] CVE-2016-1972. 2016. Available from MITRE. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-1972.

[22] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. 2011. Delay-bounded scheduling. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. ACM, 411–422.

[23] Haining Feng, Liangze Yin, Wenfeng Lin, Xudong Zhao, and Wei Dong. 2020. Rchecker: A CBMC-based Data Race Detector for Interrupt-driven Programs. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, Macau, China, 465–471.

[24] Cormac Flanagan and Stephen N Freund. 2009. FastTrack: efficient and precise dynamic race detection. *ACM Sigplan Notices* 44, 6 (2009), 121–133.

[25] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. *ACM Sigplan Notices* 40, 1 (2005), 110–121.

[26] Pedro Fonseca, Rodrigo Rodrigues, and Björn B Brandenburg. 2014. SKI: Exposing Kernel Concurrency Bugs through Systematic Schedule Exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 415–431.

[27] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 174–186.

[28] Shin Hong and Moonzoo Kim. 2015. A survey of race bug detection techniques for multithreaded programmes. *Software Testing, Verification and Reliability* 25, 3 (2015), 191–217.

[29] Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. *ACM SIGPLAN Notices* 50, 6 (2015), 165–174.

[30] Jeff Huang. 2018. UFO: predictive concurrency use-after-free detection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, Gothenburg, Sweden, 609–619.

[31] Jeff Huang, Qingzhou Luo, and Grigore Rosu. 2015. GPredict: Generic predictive concurrency analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 847–857.

[32] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*. 337–348.

[33] Shiyou Huang and Jeff Huang. 2016. Maximal causality reduction for TSO and PSO. *ACM SIGPLAN Notices* 51, 10 (2016), 447–461.

[34] Shiyou Huang and Jeff Huang. 2017. Speeding up maximal causality reduction with static dependency analysis. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[35] Ali Jannesari, Kaibin Bao, Victor Pankratius, and Walter F Tichy. 2009. Helgrind+: An efficient dynamic race detector. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, Chengdu, China, 1–13.

[36] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, Hyatt Regency, San Francisco, CA, 754–768.

[37] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[38] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F Lukman, and Haryadi S Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 399–414.

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

[40] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 6.

[41] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. 2018. Fuzzing: State of the art. *IEEE Transactions on Reliability* 67, 3 (2018), 1199–1218.

[42] Changming Liu, Deqing Zou, Peng Luo, Bin B Zhu, and Hai Jin. 2018. A heuristic framework to detect concurrency vulnerabilities. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, San Juan, PR, USA, 529–541.

[43] Ziheng Liu, Shuofei Zhu, Boqin Qin, Hao Chen, and Linhai Song. 2021. Automatically detecting and fixing concurrency bugs in go software systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 616–629.

[44] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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*. 329–339.

[45] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. 2020. Learning-based controlled concurrency testing. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–31.

[46] Madanlal Musuvathi and Shaz Qadeer. 2007. Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices* 42, 6 (2007), 446–455.

[47] Madanlal Musuvathi and Shaz Qadeer. 2007. *Partial-order reduction for context-bounded state exploration.* Technical Report. Citeseer.

[48] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs.. In *OSDI*, Vol. 8. USENIX, USA, 267–280.

[49] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. 2007. *Chess: A systematic testing tool for concurrent software.* Technical Report. Technical Report MSR-TR-2007-149, Microsoft Research.

[50] Santosh Nagarakatte, Sebastian Burckhardt, Milo MK Martin, and Madanlal Musuvathi. 2012. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Beijing, China,

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.

543–554.
[51] Jihyun Park, Byoungju Choi, and Seungyeun Jang. 2020. Dynamic analysis method for concurrency bugs in multi-process/multi-thread environments. *International Journal of Parallel Programming* 48 (2020), 1032–1060.
[52] João Carlos Pereira, Nuno Machado, and Jorge Sousa Pinto. 2020. Testing for Race Conditions in Distributed Systems via SMT Solving. In *International Conference on Tests and Proofs*. Springer, 122–140.
[53] Ernest Pobee and Wing Kwong Chan. 2019. Aggreplay: Efficient record and replay of multi-threaded programs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Paris, France, 567–577.
[54] Ernest Pobee, Xiupei Mei, and Wing Kwong Chan. 2019. Efficient transaction-based deterministic replay for multi-threaded programs. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, San Diego, California, USA, 760–771.
[55] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. 2011. LOCKSMITH: Practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 1 (2011), 1–55.
[56] András Prékopa. 2003. Probabilistic programming. *Handbooks in operations research and management science* 10 (2003), 267–351.
[57] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.
[58] Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. 2020. Symbolic partial-order execution for testing multi-threaded programs. In *International Conference on Computer Aided Verification*. Springer, 376–400.
[59] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. ACM, New York, NY, USA, 62–71.
[60] Jiří Šimša, Randy Bryant, and Garth Gibson. 2011. dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In *International SPIN Workshop on Model Checking of Software*. Springer, 188–193.
[61] Yulei Sui, Peng Di, and Jingling Xue. 2016. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. 160–170.
[62] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th international conference on compiler construction*. 265–266.
[63] Paul Thomson, Alastair F Donaldson, and Adam Betts. 2014. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 15–28.
[64] Paul Thomson, Alastair F Donaldson, and Adam Betts. 2016. Concurrency testing using controlled schedulers: An empirical study. *ACM Transactions on Parallel Computing (TOPC)* 2, 4 (2016), 1–37.
[65] Nischai Vinesh and M Sethumadhavan. 2020. Confuzz—a concurrency fuzzer. In *First International Conference on Sustainable Technologies for Computational Intelligence*. Springer, Jaipur, Rajasthan, India, 667–691.
[66] Vesal Vojdani and Varmo Vene. 2009. Goblint: Path-sensitive data race analysis. In *Annales Univ. Sci. Budapest., Sect. Comp*, Vol. 30. Citeseer, 141–155.
[67] Dmitry Vyukov. 2010. Bug with a context switch bound 5. In *Microsoft CHESS Forum*.
[68] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*. Springer, 256–272.
[69] Zan Wang, Dongdi Zhang, Shuang Liu, Jun Sun, and Yingquan Zhao. 2019. Adaptive randomized scheduling for concurrency bug detection. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, Nansha, Guangzhou, China, 124–133.
[70] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news* 23, 2 (1995), 24–36.
[71] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, Virtual, 1643–1660.
[72] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent model checking of unmodified distributed systems. (2009).
[73] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. 2008. *Inspect: A runtime model checker for multithreaded C programs*. Technical Report. Citeseer.
[74] Jie Yu and Satish Narayanasamy. 2009. A case for an interleaving constrained shared-memory multi-processor. *ACM SIGARCH Computer Architecture News* 37, 3 (2009), 325–336.
[75] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. 2012. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. ACM, Tucson, Arizona, USA, 485–502.
[76] Kunpeng Yu, Chenxu Wang, Yan Cai, Xiapu Luo, and Zijiang Yang. 2021. Detecting Concurrency Vulnerabilities Based on Partial Orders of Memory and Thread Events. In *Proceedings of the 2021 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM.
[77] Tingting Yu, Tarannum S Zaman, and Chao Wang. 2017. DESCRY: reproducing system-level concurrency failures. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn, Germany, 694–704.
[78] Ming Yuan, Yeseop Lee, Chao Zhang, Yun Li, Yan Cai, and Bodong Zhao. 2021. RAProducer: efficiently diagnose and reproduce data race bugs for binaries via trace analysis. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 593–606.
[79] Xinhao Yuan and Junfeng Yang. 2020. Effective Concurrency Testing for Distributed Systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1141–1156.
[80] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. Partial order aware concurrency sampling. In *International Conference on Computer Aided Verification*. Springer, 317–335.
[81] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. 2018. Owl: Understanding and detecting concurrency attacks. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, Luxembourg City, Luxembourg, 219–230.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO PORTO. Downloaded on November 28,2022 at 13:48:35 UTC from IEEE Xplore. Restrictions apply.