Greybox Fuzzing for Concurrency Testing

Dylan Wolff
wolffd@comp.nus.edu.sg
National University of Singapore
Singapore

Zheng Shi
zs357@comp.nus.edu.sg
National University of Singapore
Singapore

Gregory J. Duck
gregory@comp.nus.edu.sg
National University of Singapore
Singapore

Umang Mathur
umathur@comp.nus.edu.sg
National University of Singapore
Singapore

Abhik Roychoudhury
abhik@comp.nus.edu.sg
National University of Singapore
Singapore

Abstract

Uncovering bugs in concurrent programs is a challenging problem owing to the exponentially large search space of thread interleavings. Past approaches towards concurrency testing are either optimistic — relying on random sampling of these interleavings — or pessimistic — relying on systematic exploration of a reduced (bounded) search space. In this work, we suggest a fresh, pragmatic solution neither focused only on formal, systematic testing, nor solely on unguided sampling or stress-testing approaches. We employ a biased random search which guides exploration towards neighborhoods which will likely expose new behavior. As such it is thematically similar to greybox fuzz testing, which has proven to be an effective technique for finding bugs in sequential programs. To identify new behaviors in the domain of interleavings, we prune and navigate the search space using the "reads-from" relation. Our approach is significantly more efficient at finding bugs per schedule exercised than other state-of-the art concurrency testing tools and approaches. Experiments on widely used concurrency datasets also show that our greybox fuzzing inspired approach gives a strict improvement over a randomized baseline scheduling algorithm in practice via a more uniform exploration of the schedule space. We make our concurrency testing infrastructure "Reads-From Fuzzer" (RFF) available for experimentation and usage by the wider community to aid future research.

ACM Reference Format:
Dylan Wolff, Zheng Shi, Gregory J. Duck, Umang Mathur, and Abhik Roychoudhury. 2024. Greybox Fuzzing for Concurrency Testing. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3620665.3640389

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0385-0/24/04.
https://doi.org/10.1145/3620665.3640389

1 Introduction

Despite the increasing importance of parallelism and concurrency in modern software, reasoning about the exponentially large space of interleavings in such programs remains a challenging task today. Consequently, low level issues such as data races [3], and concurrency-related high level issues such as memory corruption, undefined behaviors [11] or crashes often compromise software quality and security. A common approach adopted in practice is to conduct stress testing of the software against exceptional workloads. These approaches can be termed as *optimistic* approaches since they hope that applying varied and large workloads will expose corner cases of the program behavior. Another manifestation of this *optimistic* approach towards concurrency testing is random sampling. Here randomized algorithms sample in the space of interleavings or partial orders with the goal of exploring various neighborhoods in the search space of behaviors evenly.

An alternative category of techniques that have been widely studied in the research community is systematic testing and exploration – the *pessimistic* approach. These approaches insert certain hooks into the program or model the program to control the scheduling and systematically explore the space of interleavings [25]. One common observation to reduce the search space of behaviors is to either employ partial order reduction [1, 21, 38] or heuristics such as context bounding - where the number of context switches is bounded by an arbitrary constant. We view the enumerative approaches such as CHESS [46] or GenMC [38] as *pessimistic* since they attempt to go toward exhaustive exploration, albeit in a reduced search space.

In this paper, we propose a new direction in concurrency testing - which is neither optimistic, nor pessimistic. Our key idea is to conduct a biased, randomized search over a reduced space of functionally equivalent interleaving partitions, based on their "reads-from" information. The reads-from relation induced by an execution, maps each read event to the corresponding write event it observes. This relation serves as a *semantic* abstraction for a given execution of a concurrent program. To effectively explore the reduced space of reads-from partitions, we leverage a pro-active, controlled scheduler which drives executions towards or away from individual partitions. Finally, we also use the reads-from relation as feedback to bias the search towards novel partitions - if a new reads-from pair is encountered, the execution is deemed novel and prioritized in the search. In this way we can use the reads-from relation to effectively pivot our search, and adapt it towards new and semantically different search neighborhoods, by mutating previously encountered schedules. Our approach is, thus, neither *optimistic* nor *pessimistic* - we deem it to be *pragmatic* .

Thematically, our approach can be seen as greybox fuzz testing, adapted to concurrent programs. Greybox fuzzing has been extremely successful for finding bugs and vulnerabilities in sequential programs in practice [5, 12]. Part of this success comes from the adaptive and non-enumerative nature of the search, enabling it to scale to sequential applications such as the PDF readers, multimedia encoders and compression libraries – whose input spaces can be large, encompassing all possible byte combinations of multi-Kilobyte to multi-Gigabyte sized files. Similarly, the concurrency-centric greybox fuzzer we develop for this work, RFF, does not enumerate all possible interleavings or even reads-from partitions, but rather conducts an adaptive, randomized search of the reduced space of partitions.

While greybox fuzzers for testing concurrent programs have been developed by prior works [15, 31, 32, 69], our tool, RFF, advances the state-of-the-art due to fundamental design and implementation choices. RFF fully controls the schedule, as relying on intermittent or loose control of the schedule (e.g. delay injection) [15, 31] could allow the test environment itself to render some interleavings difficult or impossible to reach. RFF uses the semantic, reads-from relation as feedback to bias its search, rather than execution statistics such as function-call pairs [32] or event-pair orderings [69]. Also, unlike other concurrency-aware fuzzers, RFF structures its search in terms of a novel abstraction, again leveraging reads-from information. Aided by our pro-active scheduler, this abstraction effectively reduces the exponentially large space of interleavings.

To conduct this search, RFF must instrument and intercept memory access and synchronization operations. We do so with a combination of binary rewriting via E9patch [19] and a user-space scheduling library which overrides C/C++ threading primitives. Our scheduler implementation *efficiently* serializes program execution, allowing our schedule fuzzer to control the program schedule. With this schedule control, our approach is able to find bugs in significantly fewer schedules than prior work, achieving higher scalability without sacrificing bug detection ability.

We evaluated RFF against state-of-the-art bug-finding tools [38, 67] and algorithms [13, 74] on 49 programs containing concurrency bugs. We find that RFF not only consistently finds bugs in more programs than any other approach –46 programs on average– it also typically finds these bugs in fewer schedules. We also show that RFF explores the scheduling space as partitioned by the reads-from equivalence relation evenly, and that a fuzzing-inspired search leverages this partitioning effectively as compared to a common reinforcement learning framework [53].

Contributions. Concretely our contributions are:

- Conceptually, we bring ideas from greybox fuzzing of sequential programs (such as power schedule or feedback function) into the exploration of concurrent program interleavings.
- We structure the schedules as a collection of "reads-from" constraints to define the fuzzer search space. This structured search space helps in bug finding effectiveness as shown by our experiments. We additionally utilize the "reads-from" relation as supplementary feedback function to guide this search.
- We implement our approach as a fuzzer called Reads-From Fuzzer (RFF). Experiments on well known benchmarks like SCTbench [58] show the bug-finding effectiveness of RFF's biased random search approach vis-a-vis random search (such as Partial order sampling or POS) as well as vis-a-vis systematic testing (such as state of the art model checkers like GenMC).
- We make RFF and its associated instrumentation and execution framework available for usage and research:

https://doi.org/10.6084/m9.figshare.23911299

2 Overview

The central challenge encountered in exposing concurrency bugs is the sensitivity to thread schedules — even for a fixed input, a bug may surface only under certain interleavings. Further, the number of interleavings grow exponentially in the length of executions. Enumerative search over interleavings is thus difficult even even after employing optimizations such as partial order reduction [4, 24].

Randomized concurrency testing, often offering probabilistic guarantees [13, 74], has therefore emerged as a promising paradigm for exposing hard to find interleavings. Existing state-of-the-art randomized techniques though often fall short. Consider, for example, the (stripped down) program sinppet shown in Figure 1, derived from the SCTBench benchmark suite [58]. The `main` thread spawns $\mathtt{n} = 100$ *setter* threads $\mathtt{st}_1, \ldots, \mathtt{st}_\mathtt{n}$ that each set the value of the shared variables `a` and `b` to `1` and `-1` respectively, as well as a single *checker* thread `ct` that asserts that the values of (`a`, `b`) are either $(1, -1)$ or $(0, 0)$. Observe that the assertion in `ct` gets violated only when `a` observes the value written by one of $\mathtt{st}_1, \ldots, \mathtt{st}_\mathtt{n}$, while `b` observes the initial value 0. This happens when `ct` is executed after at least one setter thread

```
1  static int a = 0, b = 0;
2  void setThread() {
3      a = 1;
4      b = -1;
5  }
6  void checkThread() {
7      if(!((a == 0 && b == 0) || (a == 1 && b == -1))){
8          assert(0); // Bug found
9      }
10 }
11 int main() {
12     const int n = 100;
13     std::vector<std::thread> st;
14     for (int i = 0; i < n; ++i) {
15         st.push_back(std::thread(setThread));
16     }
17     std::thread ct(checkThread);
18     return 0;
19 }
```

Figure 1. SCTBench subject `reorder_100` (repurposed)

executes its first step '`a = 1;`', and before any setter thread executes its second step '`b = -1;`'.

Assuming that the read events of `a` and `b` in thread `ct` happen atomically, the total number of executions of this program is $\frac{(3n+2)!}{(n+1)!(3!)^n(2!)}$[1], while, the number of interleavings, that hit the assertion violation is $\sum_{i=1}^{n} \frac{\binom{n}{i}(n+i+1)!(2n-i)!}{(n+1)!2^n}$[2]. A sampling strategy that uniformly samples each execution will hit the violation with a probability of about 2.8×10^{-14} making the expected number of steps to hit the violation prohibitively large. Partial order-aware sampling (POS) [74] offers a similar guarantee. Randomized sampling strategies that rely on *bug depth hypothesis* such as PCT[13] also work poorly here — needing at least one ordering constraint from the first step of some setter thread to the checker thread, and 100 ordering constraints to order the checker thread before the second steps of each setter thread. The depth of this bug therefore is at least 101, which is beyond the tractable boundary of PCT given that the probability guarantee of PCT scales exponentially in the depth. Our experimental evaluation also confirms this analysis — both POS and PCT struggle to hit the bug in a reasonable number of trials.

In contrast, our concurrency testing tool RFF exposes the bug in about 6 iterations in each of the 20 trials we performed. The effectiveness of RFF stems from two important insights. The first is the observation that two concurrent executions that follow the same thread-local control and data flow are both equally capable of exposing the same assertion violation; thus exploring one of them suffices. The *reads-from* function, mapping each read event of a shared memory location to its corresponding writer event, serves as a good proxy for control and data-flow. More importantly, the number of different *classes* induced by reads-from equivalence can be exponentially fewer than interleavings, partial orders [2] or even classes induced by bounded-depth ordering constraints. RFF *covers all executions that are reads-from equivalent in one go, by generating only one of them.* In the program from Figure 1, the assertion violation can be exposed by any execution that satisfies the following *abstract schedule*:

$$\alpha_{\text{violation}} = \{\mathtt{w(a)}@\ell_3 \xrightarrow{\text{rf}} \mathtt{r(a)}@\ell_7, \quad \mathtt{w(b)}@\ell_1 \xrightarrow{\text{rf}} \mathtt{r(b)}@\ell_7\}$$

The constraints above demand that the read of `a` in line 7 (in `checkThread()`) observes the write of `a` in line 3, while the read of `b` observes the the initial write in line 1. While the total number of different reads-from constraints is just 4 (2 choices for `a` and 2 choices for `b`), the chance of generating $\alpha_{\text{violation}}$ when (almost) uniformly sampling this space is large. Combined with our *proactive reads-from scheduler* that carefully attempts to schedule any given abstract schedule, RFF can successfully uncover the bug with very few trials.

Our second insight towards the design of a more effective concurrency testing technique is the introduction of (moderate amount of) *statefulness* in contrast to the otherwise unbiased random search techniques. Statefulness in the exploration can help identify previously explored parts of the schedule search space and can be leveraged to drive exploration away from them. RFF effectively tracks information about the set of schedules observed so far, and utilizes it to generate a new abstract schedule. In the program in Figure 1, for example, RFF generates $\alpha_{\text{violation}}$ by extracting a list of events observed in previous schedules and stitching them, through random mutations, to arrive at the constraints in $\alpha_{\text{violation}}$. RFF also uses information about when a previously explored abstract schedule led to *novel* behavior and uses this as further *feedback* to guide what kinds of abstract schedules to generate next.

We show that these insights can be combined into a *greybox fuzzing* approach. Greybox fuzzers [5, 12] have proven effective in discovering software vulnerabilities in sequential programs. While traditional greybox fuzzing techniques represent a biased random search over the space of program inputs in a sequential program, the support for exposing rare concurrency-centric behaviors in such techniques is either absent or remains rudimentary. Section 3 describes the details of our approach of formulating concurrency testing as a greybox fuzzer.

[1]The numerator counts the number of arrangements of $(3n + 2)$ steps $(n+1$ thread spawning steps of the main thread + $2n$ steps of the setter threads + 1 step of the checker thread), while the denominator offsets overcounting of the arrangements of the spawning events, and the total order within each setter and checker thread (together with their spawn events)

[2]The i^{th} term in the numerator counts the number of interleavings where thread `ct` executes after $n + 1$ (thread spwan) steps + i steps (`a = -1;` of i setter threads). The term $(n+1)!$ offsets overcounting the permutations amongst the spawn events performed by `main` while the term $2^n = 2^i \cdot 2^{n-i}$ offsets the relative order of first steps of the i setter threads w.r.t spawn events, and the relative order within setter threads for the later $n - i$ setter threads

Algorithm 1: Greybox Concurrency Fuzzing

```
   Input: Initial corpus of schedules $S_{\text{init}}$
1  $S \leftarrow S_{\text{init}}, S_{\text{fail}} \leftarrow \emptyset$
2  if $S = \emptyset$ then $S \leftarrow \{\varepsilon\}$    /* Empty Schedule */
3  repeat
4      $(\sigma, \eta_\sigma) \leftarrow$ PickNextAndAssignEnergy($S$)
5      for $i \in \{1, \ldots, \eta_\sigma\}$ do
6          $\sigma_{\text{mut}} \leftarrow$ mutateSchedule($\sigma, S$)
7          if $\sigma_{mut}$ crashes then $S_{\text{fail}} \leftarrow S_{\text{fail}} \cup \{\sigma_{\text{mut}}\}$
8          if isInteresting($\sigma_{mut}, S$) then
9              $S \leftarrow S \cup \{\sigma_{\text{mut}}\}$
10 until timeout
11 return $S_{\text{fail}}$
```

3 Core Approach behind RFF

As our core approach, we adapt a well-known grey-box fuzzing algorithm [12] to the space of interleavings/schedules; Algorithm 1 shows our overall template. The algorithm begins with an initial set of schedules. During each iteration of the fuzzing loop, an existing schedule σ is selected from the working set of schedules (line 4). The algorithm then mutates σ to create a new schedule σ_{mut} (line 6), which will be used to test the program. If the program crashes under σ_{mut}, then σ_{mut} is added to the output set of crashing schedules (line 7). Additionally, if the fuzzer determines that the execution of σ_{mut} includes new behavior (line 8) then that schedule is added to the working set of schedules for future exploration via mutation (line 9). This loop is repeated, mutating each schedule proportionally to its behavioral novelty score (η_σ in line 4) before moving on to the next schedule in the working set.

A few observations about the greybox fuzzing template are in order. First, unlike purely random sampling based approaches such as PCT and partial order sampling [13, 74], the algorithm maintains state *across executions* in the form of the corpus of schedules S. It uses this state to guide the next iteration of schedule generation. Further, instead of storing all schedules, the algorithm maintains only a subset of schedules, filtering out schedules with redundant behavior via the `isInteresting()` function. In order to determine what is interesting, the traditional greybox fuzzing algorithm relies on a notion of (lightweight) code-coverage *feedback* that can be collected as the input schedule is being executed. For more granular feedback, energy assignment (line 4) can further prioritize some schedules. The mutation generation works by first randomly chosing members of the existing corpus and then applying a class of pre-defined operations on them to obtain a schedule in a neighborhood that is expected to (but may not) be close to the existing corpus. The mutation operation is thus responsible for the core randomness in the exploration, while the feedback and the energy assignment together bias the otherwise random search towards under-explored regions in the space. This bias is achieved indirectly by controlling the frequencies of schedules for mutation.

While the overall fuzzing template is straightforward, there are several key challenges in adapting this template to the domain of concurrent-program interleavings: First, the space of schedules or thread interleavings can be prohibitively large, given that subtle bugs often only manifest in long executions. Second, traditional feedback metrics such as code-coverage may not be fine-grained enough to expose subtle interleavings, warranting a fresh look at the feedback. Third, unlike the traditional setting where the fuzzing algorithm generates inputs that can always be executed, a synthetic schedule may not be feasible, leading to many redundant iterations, slowing the overall testing loop. In the following we give the specifics of how we address these challenges.

Reads-from relation and equivalence. The key to the effectiveness of our approach stems from the *semantic* treatment of concurrent program executions — we identify when two executions are semantically equivalent and systematically avoid exploring multiple interleavings from the same equivalence class, an insight also central to partial order reduction based model checking techniques [2, 21, 52]. We choose *reads-from* information to define this equivalence relation over schedules. Let us first define events and schedules. An event is a tuple $e = \langle id, t, op(x)@\ell \rangle$, where id is the unique identifier of e, t is a thread identifier, op is an operation such as 'r' (read) or 'w' (write), x is the object (variable/memory location, etc) that e operates upon and ℓ is the source code location that e corresponds to. A schedule σ is a sequence of events, and we use Events_σ to denote the set of its events. A schedule σ is said to be *feasible* for a program P if P can execute σ. The reads-from function of σ, denoted rf_σ maps each read event e in σ to the corresponding write event $f = \text{rf}_\sigma(e)$ it observes its value from. Two schedules σ_1 and σ_2 are said to be reads-from equivalent, denoted $\sigma_1 \equiv_{\text{rf}} \sigma_2$ if they both observe the same events (i.e., $\text{Events}_{\sigma_1} = \text{Events}_{\sigma_2}$) and the same reads-from function (i.e., $\text{rf}_{\sigma_1} = \text{rf}_{\sigma_2}$). If $\sigma_1 \equiv_{\text{rf}} \sigma_2$, we note that σ_1, σ_2 must observe the same control flow and thus identically expose any given bug.

Abstract events and schedules. To reduce the search space of program interleavings, we represent schedules by $\equiv_{\text{rf}}$-equivalence class, rather than a concrete sequence of events. We do so by virtue of generating and mutating only *abstract schedules* comprising of *abstract events*. An abstract event is a tuple $ea = op(x)@\ell$ containing an operation op, a memory location x and a code location ℓ. We say that an event $e = \langle id, t, op(x)@\ell \rangle$ instatiates abstract event $ea = op'(x')@\ell'$ if $op' = op, x' = x$ and $\ell' = \ell$. An abstract schedule α is a set of positive and negative reads-from constraints: $\alpha = \alpha^+ \uplus \alpha^-$, where $\alpha^+ = \{C_1^+, C_2^+, \ldots C_{n_1}^+\}$ and $\alpha^- = \{C_1^-, C_2^-, \ldots C_{n_2}^-\}$.

Each positive constraint C_i^+ is of the form $C_i^+ = ea_i \xrightarrow{\mathsf{rf}} ea'_i$ and each negative constraint is of the form $C_j^- = ea_j \overset{\mathsf{rf}}{\nrightarrow} ea'_j$ for some read ea_i, ea_j and write abstract events ea'_i, ea'_j (over the same memory location). We say that a (concrete) schedule σ is an *instantiation of* an abstract schedule α if it satisfies the constraints of α, i.e.,

1. for every positive constraint $C_i^+ = ea_i \xrightarrow{\mathsf{rf}} ea'_i \in \alpha$, there are events e_i and e'_i in σ that instatiate ea_i and ea'_i respectively and satisfy $\mathsf{rf}_\sigma(e'_i) = e_i$, and
2. for every negative constraint $C_i^- = ea_i \overset{\mathsf{rf}}{\nrightarrow} ea'_i \in \alpha$, there are no events e_i and e'_i in σ that instatiate ea_i and ea'_i respectively and satisfy $\mathsf{rf}_\sigma(e'_i) = e_i$

If $\sigma_1 \equiv_{\mathsf{rf}} \sigma_2$, then either both or none of them are an instantiation of any given abstract schedule α.

Schedule $\sigma = e^{(1)}_{\mathtt{main}} \dots e^{(\mathsf{n}+1)}_{\mathtt{main}} e^{\mathsf{a}}_{\mathtt{st}_1} e^{\mathsf{b}}_{\mathtt{st}_1} \dots e^{\mathsf{a}}_{\mathtt{st}_\mathsf{n}} e^{\mathsf{b}}_{\mathtt{st}_\mathsf{n}} e^{\mathsf{a}}_{\mathtt{ct}} e^{\mathsf{b}}_{\mathtt{ct}}$, for example, is a sequence of $3\mathsf{n}+2$ events for the program in Figure 1. Here $e^{(i)}_{\mathtt{main}} = \langle i, \mathtt{main}, \mathtt{spawn}(\mathtt{st}_i)@\ell_{15}\rangle$ represents the event that spawns thread $\mathtt{st}_i$, while $e^{\mathsf{a}}_{\mathtt{st}_i}$ and $e^{\mathsf{b}}_{\mathtt{st}_i}$ represent the write to a and b respectively by thread $\mathtt{st}_i$. Events $e^{\mathsf{a}}_{\mathtt{ct}}$ and $e^{\mathsf{b}}_{\mathtt{ct}}$ are read events on a and b in thread ct. This concrete schedule σ spawns and executes all setter threads before executing the checker thread. The reads-from function maps $\mathsf{rf}_\sigma(e^{\mathsf{a}}_{\mathtt{ct}}) = e^{\mathsf{a}}_{\mathtt{st}_\mathsf{n}}$ and $\mathsf{rf}_\sigma(e^{\mathsf{b}}_{\mathtt{ct}}) = e^{\mathsf{b}}_{\mathtt{st}_\mathsf{n}}$. Now consider the alternate schedule

$$\sigma' = e^{(1)}_{\mathtt{main}} \dots e^{(\mathsf{n}+1)}_{\mathtt{main}} e^{\mathsf{a}}_{\mathtt{st}_\mathsf{n}} e^{\mathsf{b}}_{\mathtt{st}_\mathsf{n}} e^{\mathsf{a}}_{\mathtt{ct}} e^{\mathsf{b}}_{\mathtt{ct}} e^{\mathsf{a}}_{\mathtt{st}_1} e^{\mathsf{b}}_{\mathtt{st}_1} \dots e^{\mathsf{a}}_{\mathtt{st}_{\mathsf{n}-1}} e^{\mathsf{b}}_{\mathtt{st}_{\mathsf{n}-1}}$$

that performs all events of the setter thread $\mathtt{st}_\mathsf{n}$ and the checker thread before events of other setter threads. The reads-from function here also maps $\mathsf{rf}_{\sigma'}(e^{\mathsf{a}}_{\mathtt{ct}}) = e^{\mathsf{a}}_{\mathtt{st}_\mathsf{n}}$ and $\mathsf{rf}_{\sigma'}(e^{\mathsf{b}}_{\mathtt{ct}}) = e^{\mathsf{b}}_{\mathtt{st}_\mathsf{n}}$, thus $\sigma \equiv_{\mathsf{rf}} \sigma'$. Thus both *concrete* schedules are instantiations of same the abstract schedule

$$\alpha_{\mathsf{miss}} = \{\mathtt{w(a)}@\ell_3 \xrightarrow{\mathsf{rf}} \mathtt{r(a)}@\ell_7, \quad \mathtt{w(b)}@\ell_4 \xrightarrow{\mathsf{rf}} \mathtt{r(b)}@\ell_7\}$$

Any instantiation of α_{miss} will not find the assertion violation. In contrast, *all* instantiations of $\alpha_{\text{violation}}$ (outlined in Section 2) will witness the bug. Overall, there are only 25 abstract schedules (5 options each for the reads-from constraint on r(a) and r(b)) despite the exponentially larger number of concrete schedules.

Mutating abstract schedules. The choice of abstract schedules as the search space also naturally allows us to mutate them in a *structured* manner instead of resorting to event-level mutations. By changing the higher-level abstract schedule constraints, each mutation will definitively result in different reads-from behavior if that schedule can be feasibly executed. This notion bears semblance to the concept of structured-input fuzzing [10, 26, 54]. The `mutateSchedule()` function in our approach is implemented in two steps. First we randomly choose one of the four mutation operators:

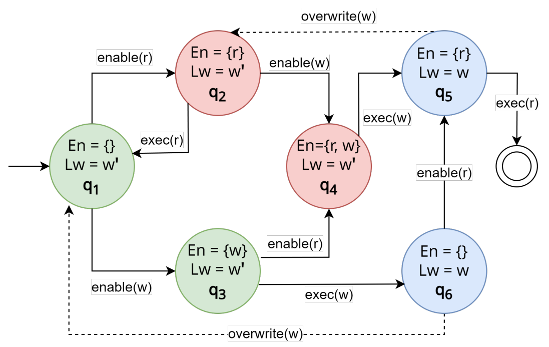
$$\begin{aligned}
&\textsc{insert}(\alpha, C) = \alpha \cup \{C\}\\
&\textsc{swap}(\alpha, C_1, C_2) = (\alpha \setminus \{C_1\}) \cup \{C_2\}\\
&\textsc{delete}(\alpha, C) = \alpha \setminus \{C\}\\
&\textsc{negate}(\alpha, C) = \textsc{swap}(\alpha, C, \neg C)
\end{aligned}$$

Here, for constraint $C = \mathtt{w} \xrightarrow{\mathsf{rf}} \mathtt{r}$ (resp. $C = \mathtt{w} \overset{\mathsf{rf}}{\nrightarrow} \mathtt{r}$), we use $\neg C$ to denote the negated constraint $\mathtt{w} \overset{\mathsf{rf}}{\nrightarrow} \mathtt{r}$ (resp. $C = \mathtt{w} \xrightarrow{\mathsf{rf}} \mathtt{r}$). Then we randomly pick potentially conflicting events from E, the set of all events observed, to form constraints $C_1 \dots C_n$ which are needed by these mutation operators.

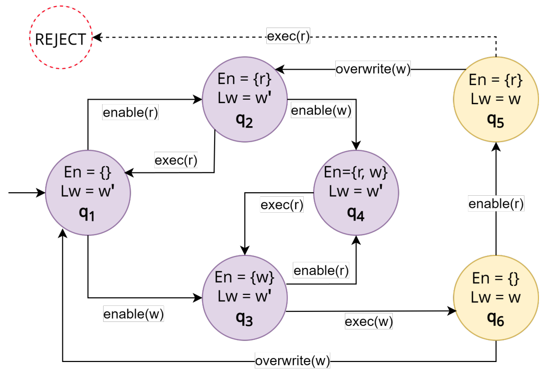
Reads-from feedback. In addition to control-flow feedback used by greybox input-fuzzers, we employ reads-from information as concurrency-aware feedback. We remark that such feedback is strictly more granular than control flow information. Our implementation of $\mathtt{isInteresting}(\sigma_{mut}, S)$ from Algorithm 1 returns true if either (a) there is a reads-from pair (e_1, e_2) in σ_{mut} such that no other the abstract schedules in S instantiates it, or, (b) if the schedule results in a crash (mirroring the behavior of input-level greybox fuzzers [5]). Additionally, we utilize a cut-off exponential power-schedule [12] to further bias exploration towards rarely observed reads-from constraints, discussed in Section 4.2.

Proactive Scheduling of Reads-from Constraints. Our algorithm RFF utilizes feedback and mutations to arrive at abstract schedules. However, given such an abstract schedule, it is not trivial to find a *concrete* execution which satisfies those scheduling constraints. Indeed, a generated abstract schedule may not even be feasible in the program at all. We design a *proactive reads-from scheduler* to drive the program to meet the constraints of the abstract scheduler. The scheduling algorithm works by carefully determining which events to execute next when multiple *enabled* events are possible.

To push the execution towards satisfying abstract scheduling constraints, we use a greedy scheduling algorithm to delay or immediately execute events involved in these constraints. Such a priority change ensures that the event will *always* (or never) be chosen next over other enabled events that are not in the abstract schedule. For example, to satisfy $\mathtt{w} \xrightarrow{\mathsf{rf}} \mathtt{r}$, we can boost the priority of a write event $e_{\mathtt{w}}$ that instantiates the (abstract) event w, and then subsequently a read event $e_{\mathtt{r}}$ that instantiates r, to satisfy the reads-from constraint. However, simply boosting and lowering the priorities when the relevant events are *both* enabled is often inadequate to ensure that a desired abstract schedule constraint is met. Relevant events are often widely separated in execution traces and thus likely will not be simultaneously enabled without further intervention. Therefore, we maintain a succinct state machine for each constraint in the abstract schedule to determine how priorities of relevant events should be proactively adjusted. A schematic description of the state machines for positive ($\mathtt{w} \xrightarrow{\mathsf{rf}} \mathtt{r}$) and negative



(a) Scheduling a positive constraint $w \xrightarrow{rf} r$. Blue states prioritize r and deprioritize any write other than w. Red states prioritize w and deprioritize r. Green states represent no prioritization.



(b) Scheduling a negative constraint $w \overset{rf}{\nrightarrow} r$ Purple states prioritize r and deprioritize w. Yellow states prioritize all writes apart from w and deprioritize r.

Figure 2. State-Transition for scheduling reads-from constraints. En and Lw represent the set of enabled events and last write event on the same location as w. Solid arrows indicate actions that facilitate the constraint. Dashed arrows indicate actions that inhibit the constraint. Label enable(op) indicates that op is ready to be executed. Label exec(op) indicates that the action executes op. Label overwrite(w) indicates executing another write w′ on same variable as w.

($w \overset{rf}{\nrightarrow} r$) reads-from constraints are depicted in Figure 2a and Figure 2b respectively.

Looking first at Figure 2a, we see that, once r has been enabled but not yet executed (states q_2 and q_4 in red), we lower the priority for the thread of the read. This means the read *will not be executed* until the corresponding write has been observed (state q_5) or all other threads are blocked. If the scheduler is forced to execute r in this state, it reverts these priority changes (state q_1). When r has been enabled, the scheduler also increases the priority for the corresponding write w while r is being delayed (states q_2 and q_4 in red). Once the relevant write w has executed (states q_5 and q_6 in blue), the scheduler lowers the priority for all w′ which access the same memory location to avoid overwriting the desired write. At this point, the scheduler also prioritizes the relevant read r to satisfy the constraint as soon as r becomes enabled. Positive constraints are implicitly existentially quantified — we consider such a constraint satisfied after it has been executed at least once on any two threads; if so the scheduler removes it from the abstract schedule for the remainder of the execution.

In Figure 2b, for a negated constraint, our scheduler gives the read r higher priority and deprioritizes the write w as long as another write w′ was the last observed (states q_1, q_2, q_3, q_4, purple). This will greedily execute the relevant read whenever the reads-from can be avoided while prolonging this window for as long as possible. Once w has been observed (states q_5 and q_6, yellow), then the scheduler deprioritizes r to avoid violating the negated constraint. It also prioritizes any other write to that location, w′, in an effort to overwrite w and bring the system back into a state where r can be safely executed. Negative constraints are universally quantified — we only consider these constraints satisfied if the entire execution does not contain the corresponding reads-from pair. Finally, the REJECT state can occur if a negated constraint is unavoidably violated, e.g., if only a single thread is actually runnable.

Our discussion of the pro-active scheduling algorithm so far describes how to (de)prioritize events relevant to the abstract schedule, but an additional mechanism is still needed to make scheduling decisions in the absence of relevant events or in the case of multiple conflicting constraints. Here we employ randomization, à la partial order sampling (POS) [74]. The POS algorithm, first assigns each event a random score if it does not already have one. It then picks the event with the highest score to execute next, resetting that event's score along with the scores of any racing events. POS has shown promising empirical results [74] [73] and, like PCT, provides a bound on the probability a particular interleaving will be exercised. We modify the POS algorithm with our greedy scheduler, such that we adopt POS *only* when our greedy scheduler cannot make a definitive scheduling decision. Our proactive reads-from scheduler, gracefully degrades to baseline POS in the absence of abstract schedule constraints or with multiple competing constraints.

Alternative Design Choices. In initial stages of developing RFF, we investigated a fuzz-testing like (i.e., partially stateful) technique that treats concrete schedules as first class inputs and generates them using a random search guided by reads-from "coverage". Unfortunately, we ruled this approach out because it was less effective than even random testing (POS) in our early evaluations. When the schedule is concrete, large fraction of the mutations of an observed schedule are infeasible. Further, the space overhead of storing concrete schedules (or a reasonable subset of them) also becomes a performance bottleneck. This is one of the key challenges that our proposed notion of abstract schedules aims to address — abstract schedules only partially describe an execution and thus some feasible instantiation of them can usually be discovered using our proactive scheduler. Second, the space of concrete schedules (or even Mazurkiewicz partial orders [74]) is much larger than the space of abstract schedules, allowing us to sample the effective search space without redundancy.

4 RFF Architecture and Implementation

An overview of system architecture of RFF is shown in Figure 3. Here, the basic system architecture consists of two main components: (1) an *instrumented Program Under Test* (PUT) in combination with a deterministic user-mode scheduler (libsched.so), and (2) a *schedule fuzzer* (RFF), based on a modified version of AFL [5]. Unlike traditional fuzzers over the input space, RFF fuzzes the PUT over the *schedule space* for a given fixed input. To do so, the RFF fuzzer maintains a corpus of "interesting" abstract schedules $\{\alpha_1, \alpha_2, \ldots\}$. For each iteration of the fuzzing loop, new abstract schedule α_{mut} is generated by mutating one (or more) existing schedules from the corpus. The mutant abstract schedule consists of a set of (possibly negated) reads-from relations over the events observed from the corpus, which may or may not be feasible. Next, the instrumented PUT is executed using a deterministic user-mode scheduler (libsched.so). The scheduler attempts to bias the actual thread interleaving into satisfying the constraints specified by α_{mut}, based on the proactive reads-from scheduler (see Figures 2a and 2b). After the instrumented program executes, a concrete schedule (σ_{mut}) representing the actual thread interleaving will be observed. The trace of this concrete schedule is analyzed for interesting behavior. If the concrete schedule σ_{mut} is deemed interesting according to the feedback metrics discussed in Section 3, its corresponding *abstract* schedule α_{mut} is saved to the schedule corpus. Note that our design assumes that the thread-interleaving can be sufficiently controlled. Thus it is necessary to manage thread interleavings at the event level, including individual memory and thread primitives. The (default) kernel-mode scheduler is coarse-grained and non-deterministic, so is not suitable. Instead we implement our own fine-grained (user-mode) scheduler.

4.1 Schedule Control

In order to control the thread interleaving, we take some inspiration from prior works on *Deterministic Multi-Threading* (DMT), such as Kendo [48] and CoreDet [9]. Here, the basic idea is to explicitly *serialize* the thread interleaving using instrumentation, as illustrated below:

```
void on_event(event_t id) {
  mutex_lock(&GLOBAL);
  thread_t *T = schedule(id);
  if (T != self()) {
    cond_signal(&T->wake);
    cond_wait(&self()->wake, &GLOBAL);
  }
  record(self(), id);
  mutex_unlock(&GLOBAL);
}
```

```
  on_event(id);  // Instrumentation
  x = *y   or   *x = y   or (thread primitive)
```

Here each event (i.e., memory operation, concurrency primitive) of interest is instrumented with a call to a on_event() which uses a global mutex (GLOBAL) to serialize all threads. Furthermore, each thread T is associated with a thread-local condition variable (T->wake) to control when T can run. When invoked, the on_event() function determines which thread to run next via the schedule() routine. The schedule routine implements the scheduler proper, as discussed below. If a different thread is selected, it will be signalled (woken) and the current thread will be put to sleep. Although serialization removes some of the performance benefits of parallelism (per instance), it is also possible to run multiple instances of the fuzzer in order to achieve full resource utilization.

The scheduling algorithm is implemented as follows:

1. First, the thread must be *runnable* (i.e., not blocked).
2. Secondly, over the set of runnable threads, the Figure 2a and 2b algorithms are used to bias thread selection towards the abstract schedule. If executing e on thread T violates the abstract schedule, then a different thread should be selected if possible.
3. Finally, more than one thread may still be selectable. For this we revert to *partial order sampling* (POS) over a predetermined random seed.

If no thread is runnable then a *deadlock* has been detected. Otherwise, schedule() will return a valid runnable thread with a bias towards the abstract schedule.

Deterministic Multi-Threading. Reproducibility is a key challenge in finding and especially *fixing* concurrency related bugs, as the default scheduler for all mainstream operating systems is non-deterministic. Unfortunately, Deterministic Multi-Threading (DMT) at the granularity of memory operations is known to incur high overheads [39]. In the context of our fork-based fuzz testing driver, however, the overheads of the fork() system call tends to dominate over instrumentation costs [8, 23, 30]. To determine the set of runnable threads, we borrow from earlier DMT works by intercepting operations that can potentially block (mutex_lock, cond_wait, read, poll, etc.). Our runtime also

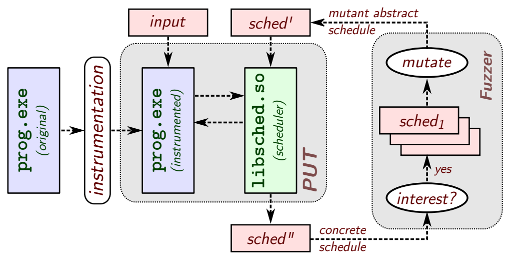


Figure 3. Overall system architecture of RFF. The two main components are (1) the instrumented *Program Under Test* (PUT), and (2) the *schedule fuzzer*. The workflow takes in an uninstrumented program (prog.exe) and a test *input*, and outputs a set of observed racey schedules (if any). The PUT is executed with mutant schedules, with observed schedules as feedback.

hooks pthread functions to track thread creation and destruction. To ensure that events observed by our tool are deterministic across executions, we disable address space layout randomization (ASLR) during a fuzz campaign.

Instrumentation. Our implementation uses low overhead static binary rewriting with E9Patch [19, 23] in order to instrument the program more efficiently than dynamic instrumentation used in prior work [72]. While not a primary focus of this work, RFF can test concurrent programs without access to their source code.

Memory Model. Our implementation assumes sequential consistency, following the recommendations of the SCTBench authors [58], as this benchmark makes up a large component of our evaluation. We look forward to future work which can apply principles from RFF to expose bugs arising from weak memory behaviours [40].

4.2 Schedule Fuzzer

The second main component is the schedule fuzzer itself (RFF), which is implemented as a modification of AFL [5]. AFL is an industry-strength fuzzing tool which has found many critical vulnerabilities in prominent open source software projects. As mentioned in Section 4.1, we use AFL's fork-based testing driver to efficiently execute the program without expensive `execve` calls. Additionally, we utilize AFL's shared memory region for efficient inter-process communication between the instrumented program and the test execution driver. We adapt AFL to the domain of schedules by changing its fuzzing loop to reflect Algorithm 1.

Power Schedule.

The `PickNextAndAssignEnergy()` function in Algorithm 1 additionally also computes an energy value η_α for abstract schedule α to dictate how many times α will be used as a base to create additional schedules via `mutateSchedule()`. As a result, it allows for a more fine-grained prioritization of particular schedules as compared to the binary feedback of `isInteresting()`. This function is shown below:

$$p(\alpha) = \begin{cases} 0 & \text{if } f(\alpha) > \mu \\ min(\frac{\gamma(\alpha)}{\beta} * 2^{s(\alpha)},\ M) & \text{otherwise} \end{cases} \quad \mu = \frac{\sum_{\alpha \in S^+} f(\alpha)}{|S^+|}$$

For example, even if both schedule α_1 and α_2 observe new interesting behavior, we can prioritize the neighborhood around α_1 over that surrounding α_2. In the preceding, $s(\alpha)$ is the number of times the abstract schedule α has been chosen since it was last skipped, $\gamma(\alpha)$ is the performance score for the schedule, β is a hyperparameter, and $f(\alpha)$ is the frequency that the reads-from combination exercised by α has been observed. M is the maximum iterations per fuzzing stage. S^+ is the working set of schedules deemed interesting by the fuzzer. According to this energy distribution, schedules which produce combinations of reads-froms which are more common than average are given zero energy (i.e. skipped entirely). Under-explored reads-from combinations are given exponentially increasing energy until they are skipped. In doing so, we explore *under-explored* regions of the scheduling space rapidly until they *become* over-explored.

5 Evaluation

In this section, we discuss how we evaluate our concurrency testing approach. RFF is inherently a randomized testing approach and its efficacy can be best determined by the diversity of the behaviors it explores. Specially, its ability to cover the search space as quickly and evenly as possible. To quantify this exploration we use RFF to find bugs in real-world applications and examples from existing concurrency-bug benchmarks; we also examine the distribution of schedules explored over a given partitioning of the schedule space.

Research Questions. The main goal of our evaluation is to answer the following three research questions (RQ).

1. Is our tool more effective at finding bugs than other state-of-the-art concurrency testing techniques?
2. How does our focus on the abstract schedule space contribute to the effectiveness of our approach?
3. How evenly distributed is our search in the space of reads-froms?
4. Is an an alternative approach leveraging reads-from information as effective as our fuzzing-inspired biased random search?

5.1 Experimental Setup

Benchmarks. To evaluate the effectiveness of our approach, we utilize two widely used benchmarks of (pthread) concurrent programs: SCTBench [58] and ConVul [14]. We choose these benchmarks for their availability and breadth, covering a wide variety of bug-types, workloads, and applications. Additionally, we selected our evaluation benchmarks based on the standards established in the concurrency testing community, to allow fair comparison with prior work. The programs in SCTBench are derived from existing concurrency-bug benchmarking suites and real-world programs containing concurrency bugs. SCTBench encompasses several smaller benchmarks used in prior work such as file downloading and compression tools [71], implementations of work stealing queues [46], small multi-threaded algorithms from [17], JavaScript engines for modern web browsers [29], parallel numeric computing applications [68], and other sample programs [70]. Some programs, such as the `RADBench` programs or `CB/pbzip2` consist of several thousand lines of production C/C++ code. We use the version of SCTBench from [67], omitting two programs due to incompatibilities with our instrumentation framework. The ConVul benchmark contains 10 programs with *critical* memory and pointer related concurrency vulnerabilities in real-world programs [14]. Together these benchmarks provide a large, diverse set of programs for evaluation.

Bugs. Of the 49 programs evaluated in these benchmarks, most (34) bugs are manifested as assertion violations. Four programs contain deadlocks. Lastly, 13 programs contain memory safety issues related to concurrency. Our evaluation criteria is the number of schedules needed to find the first instance of a bug in each program. We choose this as our criteria to avoid issues with bugs masking other failures that might occur later in that same execution. For bug detection, we use only `RFF`'s built-in deadlock-detector and a simple crash oracle (e.g. assertion failure, segmentation fault etc.), following the advice of [58].[3]

Baselines. To determine whether our approach is competitive with the state-of-the-art in controlled concurrency testing and model checking, we compare with PERIOD [67], PCT [13] and GenMC [37, 38]. PERIOD is a systematic testing tool that explores schedules in parallel using Linux deadline-based task scheduling and GenMC is a stateless model checker that enumerates all possible behaviors of a program to find bugs. PCT is a well known randomized concurrency testing algorithm with probabilistic bounds for finding bugs based on the bug's *depth*. We implement PCT ($depth = 3$)[4] within our framework to provide a fair comparison in terms of number of instrumented events and execution overhead. We also conduct an ablation study of `RFF` without its key components: abstract schedule constraints and reads-from feedback. This includes a comparison with our own implementation of Partial Order Sampling [74] as a baseline. In addition, we implement a custom Q-Learning [53] algorithm within our concurrency testing framework to evaluate an alternative approach for leveraging reads-from information for concurrency testing

Unfortunately the implementations for state-of-the-art concurrency-aware fuzzers, MUZZ[15] and CONZZER[32], are not publicly available. We reached out to the authors of these works during the early development of `RFF`, but were not able to obtain access to either implementation. Other available concurrency fuzzers [27, 31, 69] are specific to the Linux kernel and thus incompatible with the user-space benchmark programs in our evaluation.

We attempted to implement MUZZ's approach to interleaving exploration, namely (1) changing OS thread priorities on creation and (2) modifying AFL's instrumentation to give per-thread edge coverage on each execution. For (1) we set each thread priority on entry to a random value with the `sched_setscheduler` system call. For (2) we include the thread ID in the hash for each edge recorded by the coverage instrumentation of AFL. However, we found that even on simple benchmark programs, this implementation was not able to trigger bugs in practice. On the example program from Figure 1, it was not able to find the bug after millions of executions on only the three-thread version of this program. CONZZER has a far more complex implementation and we are not confident of re-implementing and replicating CONZZER system.

Experimental Environment. We conduct all experiments on a Intel(R) Xeon(R) Gold 6258R CPU @ 2.70GHz running Ubuntu 20.04 in a Docker container (kernel 5.4.0-155-generic). We run each benchmark program for 5 minutes as this is sufficient for at least one tool to discover a bug on 47/49 benchmark programs. This departs from prior concurrency testing evaluations in favor of recommendations found in literature for input-space exploration in greybox fuzzing [34]. Using a timeout rather than a bounded number of schedules allows us to compare each approach while taking into account the overhead introduced by instrumentation, schedule serialization and more sophisticated analyses. We repeat each run 20 times for each tool to account for variance in the results, excepting GenMC, which is deterministic. We did observe some variance in the output of PERIOD, and we incorporate this variance across runs into our analysis. We attribute the variance to a timeout for each period inherent in their deadline-based scheduling implementation, which may result in missed orderings if it expires. Despite this, results from our evaluation of PERIOD agree strongly reported results on these benchmarks in [67]. We make our experimental data and infrastructure available via our artifact (Appendix A):

https://doi.org/10.6084/m9.figshare.23911299

5.2 RQ1: Comparison with the State-of-the-Art

We applied PERIOD, Partial Order Sampling (POS), GenMC and `RFF` on SCTBench [58] and ConVul benchmarks [14]. Our primary metric for assessing the effectiveness of our tool is the number of schedules to the first bug found in each program. We include a table with the mean number schedules-to-bug for each tool and program in Appendix B.

[3]Many of the programs in SCTBench contain multiple data-races which would otherwise mask more challenging bugs if considered.
[4]A depth of 3 provided the best results for PCT in [58]

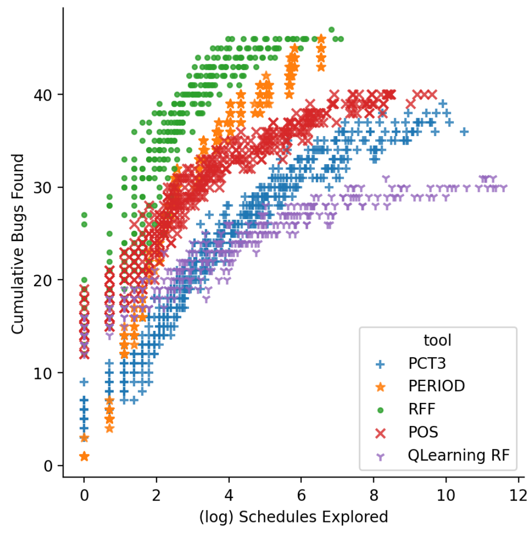


Figure 4. Total Bugs Discovered After Log(# Schedules) Across All Trials (higher is better)[5]

Unfortunately, GenMC does not run on 36 of the evaluated programs. In most cases, this seems to be a limitation of the GenMC implementation, which supports only a subset of valid LLVM IR programs. On the programs which GenMC is able to check, it explores fewer schedules than RFF's average schedules-to-bug on 5 programs. However, RFF explores fewer schedules on average relative to GenMC on 7 programs. Overall, RFF appears to be competitive with state-of-the-art model checkers in terms of bugs found per schedule, while also being more broadly applicable to real-world programs.

PCT performs well, finding approximately 37 bugs on average. However, RFF finds bugs in the most programs on average ($\mu = 46.1$), followed closely by PERIOD ($\mu = 44.6$). Despite the narrow gap, this difference in the number of bugs found over 20 trials is statistically significant by the Mann-Whitney U-test (p-value < 0.001). RFF consistently finds bugs in 46/49 programs and finds bug in the 47th program in some fuzz campaigns (hence $\mu = 46.1$). To visualize the bugs found per schedule executed for each tool, we plot the cumulative bugs found as the (log) number of total schedules increases across all programs in Figure 4. This graph shows the number of bugs found by each tool after a certain number of schedules, rather than only counting the number of bugs found at the end of our timeout. For each of our 20 trials, we track the schedule number M at which the Nth bug was computed on each benchmark. This figure plots a point at (M, N) for each bug found, in ascending order.[5] Here we see that RFF consistently finds more bugs per schedule than either PERIOD or POS at *all* schedule counts measured in our evaluation. In other words, apart from finding more bugs than prior work, RFF generally finds those bugs in fewer schedules. In fact, our tool finds bugs in significantly fewer schedules than PERIOD on 30/49 programs, whereas PERIOD finds bugs in fewer schedules for only 9/49 programs.[6] We attribute the success of our tool over PERIOD to our abstract schedule structure. PERIOD explores all possible *orderings* of synchronization and shared-access events below a given depth bound. Many different orderings of these events correspond to the same "reads-from" relations.

[5]For example, using only two trials and three programs, we might find bugs after $[3, 5, 7]$ schedules and $[3, 6, 9]$ schedules respectively. We then plot a point for the cumulative number of bugs found within each trial. In this case, a point at $(3, 1)$, $(5, 2)$ and $(7, 3)$ for the first trial and $(3, 1)$, $(6, 2)$ and $(9, 3)$ for the second trial.

*RFF finds **more bugs** in **fewer schedules** than state-of-the-art concurrency testing tools.*

5.3 RQ2: Contribution of the Abstract Schedule

We compare RFF with Partial Order Sampling POS because our scheduling algorithm leverages POS on the space of *abstract schedules* while the original POS operates on *concrete schedules*. We show the results of this comparison on the SCTBench [58] and ConVul benchmarks in Figure 4. We can see from the average number of bugs found by each tool that the abstract schedule structure improves the bug-finding ability of our tool significantly. By searching the space of reads-froms, we are able to find approximately six more bugs on average! The cumulative plot in Figure 4 similarly shows a strong effect from the schedule structure. Here we can see that our tool with abstract schedule constraints (green) finds slightly more bugs relative to POS (red) after exploring a small number of schedules. This initial gap widens as the number of schedules explored increases. In other words, POS is comparable in efficacy to our more structured search for finding relatively easy bugs (found in small numbers of schedules). For harder bugs, however, POS fails to find them as quickly or at all.

We observed that POS alone fails on several programs with larger number of threads, such as `reorder_100`, `reorder_50` and `twostage_50`, etc. As discussed in Section 2, the probability for POS to trigger a particular interleaving in these examples is exponentially low in the number of threads. RFF overcomes this problem by defining events in terms of their effects (reads-from) rather than which thread executed them. Looking more closely at each approach, we find that a structured random search finds the bug in significantly fewer schedules on 16/49 programs (via the log-rank test). In contrast POS does not find the bug in significantly fewer schedules on *any* of the benchmark programs. This implies that our tool gracefully degrades to POS in the absence of abstract schedule constraints, and the schedule structure improves POS.

[6]Significance determined by the log-rank test [41] ($p < 0.05$)

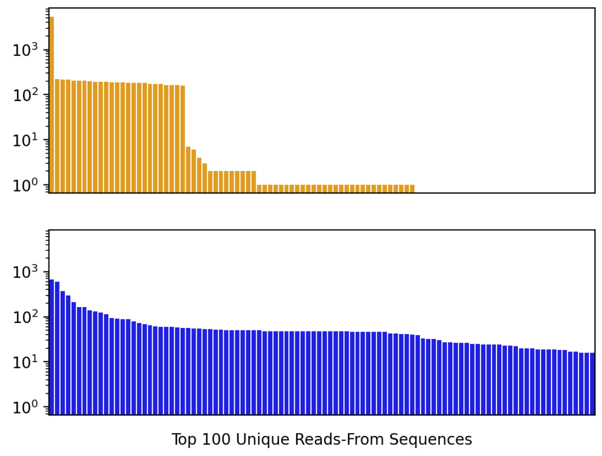


Figure 5. Log-scale frequency of reads-from sequences observed by POS (top) and RFF (bottom) in 10000 executions of the SafeStack program

> *Searching over* ***the reduced space of abstract reads-from schedules is crucial*** *for* RFF*'s bug-finding effectiveness.* RFF *provides a* ***strict improvement over Partial Order Sampling in practice.***

5.4 RQ3: Exploration of the Schedule Space

To assess RFF's ability to evenly explore the space of *meaningfully different* interleavings, we examine the number of times each unique *combination* of "reads-from" relation is observed by RFF during a fuzz campaign. We ran RFF both with and without grey-box feedback. Figure 5 shows the (log) frequency each combination of reads-from in the SafeStack program was executed after 10000 schedules. We choose this program because it is the most complex in either benchmark suite evaluated, both in terms of bug depth and difficulty. If we expect a bug to be equally likely to occur in any of the reads-from combinations, then a strategy which explores each individual combination evenly will be more successful. Here we can see that without grey-box feedback (top), there is a heavily skewed distribution. A single reads-from sequence comprises more than 50% of all executions! Including grey-box feedback (bottom) results in a more even exploration of the reads-from space. We attribute this success to our prioritization of rarely observed reads-from sequences via a suitable power-schedule, which was described in Section 4.2.

Figure 5 (top) shows that random exploration alone results in a tendency to execute some reads-from sequences exponentially more than others. By incorporating feedback via our power schedule, we can bias the search towards under-explored sequences.

> ***Greybox feedback promotes an even exploration of the schedule space and can lead to significant improvements on certain programs.***

5.5 RQ4: Reads-from Testing via Q-Learning

To further evaluate the effectiveness of our fuzzing-inspired approach, we compare it with a different framework leveraging the same reads-from information: Q-learning [53]. Q-Learning is a well-understood reinforcement learning technique which maximizes the reward over several actions which can be taken from each of a many successive states. To do so, a *Q-function* maps each state-action pair to a score: $Q : S \times A \rightarrow \mathbb{R}$. This score, $Q(s_t, a_j)$ represents the probability with which the action a_i should be taken from the state s_t. After sampling over these probabilities to determine the next action, the algorithm updates the Q-score for the state-action pair via a *reward function*. This technique can be applied to the problem of concurrency testing by considering each scheduling decision to be an action.

Our goal is to investigate Q-Learning as an *alternative framework* to leverage the reads-from relation, which is critical in uncovering both bugs and new control flows in concurrent programs. Thus, *in contrast* to prior applications of Q-Learning to concurrency testing [45], we use the reads-from relation to distinguish states in our Q-Learning implementation. More specifically, we represent the state of a partial execution of the first t events in a concrete schedule $\sigma_{1\dots t}$ as a running commutative hash of the N reads-from event pairs observed so far in that execution:

$$h((e_{w_1}, e_{r_1}),\ h((e_{w_1}, e_{r_1}),\ \dots\ , (e_{w_N}, e_{r_N})\ )))$$

for each pair (e_{w_i}, e_{r_i}) such that $\mathrm{rf}(e_{r_i}) = e_{w_i}$. We utilize a constant negative reward function for observed state-action pairs, as in [45]. Looking at Figure 4, we can see that our instantiation of "Reads-From Q-Learning" (Q-Learning RF) is not as effective in utilizing the reads-from relation to find bugs on our two datasets as RFF. In total it finds only about 30.2 bugs on average relative to RFF's 44 bugs. Additionally, RFF finds bugs in significantly fewer schedules on 30 of the 49 programs evaluated. However, we do note that the Q-Learning RF approach consistently finds the bug on the first trial in more instances than any other tool (13 programs). We attribute this one-shot success to the reinforcement learning taking place on partial traces. In comparison, RFF considers its greybox feedback only after a complete execution.

> ***Greybox schedule fuzzing provides an effective framework to leverage the reads-from relation, as compared to our instantiation of a reads-from based Q-Learning approach.***

6 Related Work

Concurrency-Aware Fuzzers. Concurrency-aware fuzz testing tools attempt to explore interleavings in addition to searching the input space of user space programs [15], kernels [27, 31] and file systems [69]. AutoInter-fuzzing [35] employs static analysis to generate concurrent memory access pairs, executing two operations from each pair in different orders to trigger concurrent bugs. Some works also combined fuzzing with symbolic execution to conduct concolic testing on concurrent programs [20, 56, 61]. MUZZ [15] emphasizes extremely low-overhead, exploring the interleaving space in an *uncontrolled* manner by randomly changing the OS thread priorities *only* at thread creation time. It also leverages threading events as feedback to find and favor inputs which trigger concurrent executions. Conzzer [32] exercises *intermittent* control at runtime, using concurrent-function-call feedback to determine if it should trigger a delay in an adjacently executed function in a future execution. It only conducts a fine-grained, controlled analysis on demand for race detection between two functions on two threads. This means that it cannot detect races involving more than two threads or more than two functions, and that the fine grained analysis will not run for issues not involving data-races (or not detected as data races by their incomplete race detector). The design of RFF is backed by the insight that control flow and data flow can be altered only when the reads-from information changes; thus a fuzzer that directly mutates this semantic information and uses it as feedback can avoid redundant exploration. RFF also fully controls the schedule throughout each execution, rather than an uncontrolled or intermittently controlled exploration as is typical for concurrency-aware fuzzers. This control over schedule is very fine grained (e.g. individual loads/stores), allowing RFF to find very intricately interleaved schedules. RFF also uses a proactive scheduler which attempts to coerce abstract (reads-from) constraints, rather than rely entirely on randomization.

Randomized Concurrency Testing. Many approaches employ a randomized thread scheduler to expose bugs, foregoing exhaustive enumeration for performance, and often probabilistic guarantees. Randomized approaches recognize that while the interleaving space is vast, many interleavings exhibit similar concurrent behaviors. A prominent strategy to design such schedulers is to identify a reduced search space, often backed by a hypothesis about the existence of bug patterns. Randomized schedulers categorize interleavings into distinct classes and sample one execution per class. PCT[13, 47] exemplifies this, grouping interleavings with the same preemption behaviors and randomly selects executions across diverse preemption behaviors. PPCT [47] the multi-core counterpart of PCT, accelerates fuzzing campaigns without compromising exploration efficiency. Partial Order Sampling (POS) [74] improves on its predecessor RAPOS [59] by uniformly sampling executions across different partial orders. Tools such as Coyote [18, 45] employ reinforcement learning to improve the search space diversity. Randomized testing approaches focusing on likely error prone partial orders have also been proposed [60]. Our approach mitigates the explosion of number of partial orders faced by such approaches, by directly working on abstract schedules representing reads-from pairs. Recent extensions of randomized methodologies encompass weak memory programs [22, 40] and distributed systems [49, 50, 73], building on the foundation of PCT and POS.

Systematic Concurrency Testing. Many systematic concurrency testing methods focus on shared memory access patterns, or program contexts. Approaches such as [46, 55] explore preemption or context-bounded program executions. Maple [72] predefines a set of shared memory access patterns and dynamically explores unvisited yet feasible patterns during run-time. PERIOD [67] treats schedule representations as a sequence of code pieces with thread-sensitive information. We have presented experimental comparison of RFF with PERIOD (Fig. 4); RFF finds bugs in fewer schedules as compared to PERIOD.

Model Checking. In the realm of concurrency testing, model checking assumes a deterministic stance and meticulously searches all possible program executions. GenMC [37, 38] is a prominent tool applied to diverse memory models within this category. CBMC [16] encodes executions into SAT/SMT formulas, harnessing solvers to assess satisfiability. The scalability of model checking is hampered by the exponential growth of the state space with program size. Dynamic partial-order reduction (DPOR) help reduce the set of program executions by categorizing executions into equivalent classes and considering one interleaving from each equivalence class [1, 21]. DPOR techniques have further been improved to be *optimal* with respect to coarser equivalence classes [2, 4, 7, 36]. Our experiments show favorable comparison of RFF w.r.t. state-of-the-art (stateless) model checker GenMC [37, 38]. This is because of the large number of program executions any enumerative search approach needs to consider. RFF uses a biased random search, and is not enumerative. Thus RFF will not necessarily exhaustively explore the search space. However, by using a random search augmented with reads-from information, RFF explores the space of interleavings relatively evenly in practice with respect to witnessing different reads-from relations.

Dynamic Analyses. Tools such as ThreadSanitizer [62, 63] aim to detect traditional concurrency bugs such as data races, and deadlocks by analyzing executions of concurrent software. While their detection ability may be limited to the observed interleaving, predictive methods for data race detection [33, 42, 43, 51, 57], deadlocks [66] and atomicity violations [44] have been developed. Such techniques generalize the execution observed during testing to a larger class of *correct reorderings* [64] and look for bugs in these, thereby enhancing the coverage of dynamic analysis without having to rerun the underlying program. Predictive analyses have also been developed for more properties beyond races, deadlocks and atomicity violations [6, 28]. We remark that despite the enhanced coverage, runtime predictive analyses do not consider executions that fall outside of the causal purview of the observed execution. In the setting of full system testing, we believe predictive testing can be used in conjunction with other concurrency techniques such as RFF to achieve faster convergence.

7 Threats To Validity

We mitigate selection bias in our choice of tools and benchmarks, a threat to *internal validity* by evaluating the reported state-of-the-art tools on the most widely used benchmarks in this domain. There is a risk that these results do not generalize beyond our evaluated dataset, a threat to *external validity*. We believe that our evaluation of 49 programs –real-world and hand-crafted– on two well-known benchmarks minimizes this risk. Our approach is intended to effectively search the space of interleavings for bugs in concurrent programs. We reduce the threat of *construct validity* by directly measuring the number of bugs found in the benchmarks evaluated. Finally, we conduct an ablation study against POS and a comparison with another algorithmic framework in Q-Learning to confirm the impact of key aspects of our approach and avoid threats to *conclusion validity*.

8 Discussion

In this paper, we have proposed a new approach which effectively searches the interleaving space of concurrent programs for bugs. To formulate this approach, we took inspiration from the success of biased random search approaches such as greybox fuzzing in sequential programs. By leveraging the reads-from relation, our approach both reduces the search space and guides the search towards under-explored regions of the search space (of interleavings). We believe that biased-random search presents a *pragmatic* middle ground between purely randomized search algorithms and heavyweight systematic testing via model checkers.

Acknowledgements

We would like to sincerely thank all the anonymous reviewers and our shepherd for their valuable feedback that greatly helped us to improve this paper. This research is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme (Fuzz Testing <NRF-NCR25-Fuzz-0001>). Any opinions, findings and conclusions, or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, and Cyber Security Agency of Singapore.

A Artifact Appendix

A.1 Abstract

The RFF tool includes (1) a dynamic library that wraps pthread functions to serialize a concurrent programs execution (2) a scheduler within that library that implements the RFF scheduling algorithm as well as other scheduling algorithms such as PCT (3) binary instrumentation to inject pre-emptions into a program at individual memory operations and (4) a modified version of AFL which executes a program in a loop while making structured mutations to its schedule. The relevant code for these components is mostly, but not exclusively, a mix of C, C++ and Python. This artifact also contains the setup in Docker for nearly 50 benchmark programs used in the paper evaluation, as well as some additional curated examples not discussed in the paper. We additionally directly include all raw data used to generate figures and tables in the paper.

The core results of this paper can be reproduced with two main experiments. The first experiment compares RFF to various alternative approaches on two widely used benchmark suites in terms of schedules-to-first-bug. This experiment generates the data for Figure 4 and in Appendix B. The latter experiment counts the frequency of various reads-from sequences in a run of RFF and Partial Order Sampling (POS), generating the data for Figure 5. The steps for each experiment have been encapsulated into corresponding convenience bash scripts. By reading the convenience scripts for these experiments and benchmarks, it should be possible to run RFF with different parameters or on different programs.

A.2 Recommended Dependencies

The recommended software dependencies are a native (not WSL), recent version of Ubuntu as the host OS with Docker, Python3, bash, and GNU Parallel [65]. Additional Python dependencies can be installed via pip and a requirements.txt file. Before each experiment, make sure all Python dependencies are installed using e.g. an activated virtual environment and that ASLR is disabled on the host system.

The experiments in this paper use the E9Patch [19, 23] binary instrumentation framework, which requires a machine with the x86_64 architecture. While no other hardware is strictly necessary, experiments can be run with up to 50 cores – fewer cores will mean slower reproduction times for the experiments. A 16 core machine should be able to run the largest experiment in about 24 hours.

A.3 Access and Details

Our artifact has been archived at:

https://doi.org/10.6084/m9.figshare.23911299

We have included a complete artifact appendix PDF with detailed documentation and reproduction steps in the latest version of the artifact. Please also see the README.md file for information regarding all configuration options for RFF.

B Mean Number of Schedules to 1st Bug

Benchmark/program	PCT3	PERIOD	RFF	POS	QLearning RF	GenMC
CB/aget-bug2	**1 ± 0**	9 ± 0	**1 ± 0**	**1 ± 0**	**1 ± 0**	Error
CB/pbzip2-0.9.4	-	**45 ± 6***	**2 ± 0***	-	-	Error
CB/stringbuffer-jdk1.4	195 ± 174	27 ± 37	**15 ± 18**	**18 ± 23**	1405 ± 1592	Error
CS/account	9 ± 7	10 ± 0	**1 ± 0**	**1 ± 0**	6 ± 8	5
CS/bluetooth_driver	161 ± 162	**9 ± 0**	45 ± 35	72 ± 79	155 ± 154	4
CS/carter01	5 ± 4	4 ± 1*†	2 ± 1	2 ± 1	**1 ± 0**	4†
CS/circular_buffer	5 ± 4	3 ± 0	**2 ± 1**	**2 ± 1**	**2 ± 1**	8
CS/deadlock01	20 ± 20	3 ± 0†	5 ± 4	4 ± 3	**1 ± 0**	3†
CS/lazy01	10 ± 6	7 ± 2	**6 ± 6**	**5 ± 4**	**12 ± 15**	5
CS/queue	12 ± 14	4 ± 1	**1 ± 0**	**1 ± 0**	**1 ± 0**	22
CS/reorder_10	2356 ± 2302	27 ± 0	**6 ± 4**	-	-	Error
CS/reorder_100	7447 ± 0*	297 ± 0	**6 ± 4**	-	-	Error
CS/reorder_20	2128 ± 2284	39 ± 0	**6 ± 4**	-	-	Error
CS/reorder_3	241 ± 336	**6 ± 0**	**7 ± 5**	223 ± 166	45843 ± 32338*	Error
CS/reorder_4	395 ± 320	9 ± 0	**6 ± 5**	1464 ± 1829	-	Error
CS/reorder_5	1126 ± 1045	12 ± 0	**6 ± 4**	4377 ± 4208*	-	Error
CS/reorder_50	12346 ± 6682*	129 ± 0	**6 ± 4**	-	-	Error
CS/stack	2 ± 2	8 ± 0	2 ± 1	2 ± 2	**1 ± 0**	20
CS/token_ring	8 ± 6	**2 ± 0**	5 ± 5	7 ± 5	12 ± 12	14
CS/twostage	**9 ± 9**	**4 ± 0**	8 ± 7	15 ± 16	336 ± 501	3
CS/twostage_100	3888 ± 3473*	690 ± 0	**56 ± 71**	-	-	Error
CS/twostage_20	188 ± 168	76 ± 0	**22 ± 19**	185 ± 215	-	Error
CS/twostage_50	849 ± 870	286 ± 0	**35 ± 27**	1984 ± 1238*	-	Error
CS/wronglock	88 ± 98	4 ± 2	**1 ± 0**	**1 ± 0**	37 ± 32	3
CS/wronglock_3	40 ± 36	5 ± 1	**1 ± 0**	**1 ± 0**	37 ± 32	Error
Chess/InterlockedWorkStealQueue	24 ± 19*	57 ± 0	**1 ± 0**	**1 ± 0**	-	Error
Chess/InterlockedWorkStealQueueWithState	16 ± 0*	224 ± 80	**7 ± 6**	**9 ± 9**	16 ± 14	Error
Chess/StateWorkStealQueue	12 ± 0*	249 ± 101	**1 ± 0**	**1 ± 0**	-	Error
Chess/WorkStealQueue	**12 ± 14**	57 ± 0	**10 ± 8**	**10 ± 9**	-	Error
ConVul-CVE-Benchmarks/CVE-2009-3547	6 ± 5	2 ± 0	**1 ± 0**	**1 ± 0**	**1 ± 0**	Error
ConVul-CVE-Benchmarks/CVE-2011-2183	9 ± 9	3 ± 0	2 ± 2	2 ± 1	**1 ± 0**	Error
ConVul-CVE-Benchmarks/CVE-2013-1792	87 ± 65	**13 ± 0**	**23 ± 43**	50 ± 62	388 ± 361	1
ConVul-CVE-Benchmarks/CVE-2015-7550	8 ± 7	3 ± 0	6 ± 5	7 ± 7	**1 ± 0**	Error
ConVul-CVE-Benchmarks/CVE-2016-1972	-	**3 ± 0***	39 ± 29	86 ± 78	74 ± 39*	Error
ConVul-CVE-Benchmarks/CVE-2016-1973	8 ± 5	6 ± 0	**3 ± 3**	7 ± 6	5947 ± 6063	Error
ConVul-CVE-Benchmarks/CVE-2016-7911	16 ± 13	3 ± 0	13 ± 10	12 ± 11	**1 ± 0**	Error
ConVul-CVE-Benchmarks/CVE-2016-9806	**4 ± 3**	**6 ± 0**	11 ± 8	14 ± 10	554 ± 577	Error
ConVul-CVE-Benchmarks/CVE-2017-15265	-	**11 ± 0**	36 ± 39	-	-	Error
ConVul-CVE-Benchmarks/CVE-2017-6346	15 ± 11	5 ± 0	5 ± 4	13 ± 14	**1 ± 0**	Error
Inspect_benchmarks/boundedBuffer	15 ± 16	**8 ± 7***	**8 ± 7**	**6 ± 5**	14 ± 13	Error
Inspect_benchmarks/ctrace-test	**1 ± 0**	3 ± 0	**1 ± 0**	**1 ± 0**	**1 ± 0**	1
Inspect_benchmarks/qsort_mt	3838 ± 4458	**27 ± 0**	322 ± 344	646 ± 753	-	Error
SafeStack	-	-	-	-	-	Error
Splash2/barnes	-	**2 ± 0**	**3 ± 3**	**2 ± 2**	**2 ± 1**	Error
Splash2/fft	**1 ± 0**	2 ± 0	**1 ± 0**	**1 ± 0**	**1 ± 0**	Error
Splash2/lu	-	2 ± 1	**1 ± 0**	**1 ± 0**	47 ± 38	Error
RADBench/bug4	15599 ± 9907*	-	**163 ± 151**	**216 ± 209**	-	Error
RADBench/bug5	-	-	-	-	-	Error
RADBench/bug6	61 ± 49	24 ± 0†	4 ± 3	11 ± 8	**1 ± 0**	Error

Each cell shows the (mean ± standard deviation) number of schedules to first bug across 20 trials
* indicates the tool did not find the bug in at least one trial
† indicates the tool does not explicitly detect deadlocks
- indicates the tool did not find the bug in any trials
bold indicates the statistically significant (by the log-rank test [41], $p < 0.05$) best result on that program
Error indicates we were unable to successfully run the tool on this program

References

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. *ACM SIGPLAN Notices*, 49(1):373–384, 2014.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Långste, Tuan Phong Ngo, and Konstantinos Sagonas. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[3] Sarita Adve. Data races are evil with no exceptions: Technical perspective. *Commun. ACM*, 53(11):84, nov 2010.

[4] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless model checking under a reads-value-from equivalence. In *International Conference on Computer Aided Verification*, pages 341–366. Springer, 2021.

[5] American fuzzy lop (AFL). https://github.com/google/AFL.

[6] Zhendong Ang and Umang Mathur. Predictive monitoring against pattern regular languages. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024.

[7] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–248. Springer, 2018.

[8] Jinsheng Ba, Gregory J. Duck, and Abhik Roychoudhury. Efficient greybox fuzzing to detect memory errors. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2023.

[9] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, page 53–64, New York, NY, USA, 2010. Association for Computing Machinery.

[10] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, 2019.

[11] Hans-J. Boehm. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, page 9–14, New York, NY, USA, 2012. Association for Computing Machinery.

[12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

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

[14] Yan Cai, Biyun Zhu, Ruijie Meng, Hao Yun, Liang He, Purui Su, and Bin Liang. 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*, pages 706–717, 2019.

[15] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. {MUZZ}: Thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2325–2342, 2020.

[16] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004. Proceedings 10*, pages 168–176. Springer, 2004.

[17] Lucas Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 331–340, 2011.

[18] Pantazis Deligiannis, Aditya Senthilnathan, Fahad Nayyar, Chris Lovett, and Akash Lal. Industrial-strength controlled concurrency testing for c# programs with coyote. In *Tools and Algorithms for the Construction and Analysis of Systems: 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22–27, 2023, Proceedings, Part II*, page 433–452, Berlin, Heidelberg, 2023. Springer-Verlag.

[19] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *ACM SIGPLAN Sympsoium on Programming Language Design and Implementation (PLDI)*, 2020.

[20] Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. Con2colic testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 37–47, 2013.

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

[22] Mingyu Gao, Soham Chakraborty, and Burcu Kulahcioglu Ozkan. Probabilistic concurrency testing for weak memory programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 603–616, 2023.

[23] Xiang Gao, Gregory J. Duck, and Abhik Roychoudhury. Scalable fuzzing of program binaries with e9afl. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, page 1247–1251. IEEE Press, 2022.

[24] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, 1996.

[25] Patrice Godefroid. Software model checking: The verisoft approach. *Formal Methods in System Design*, 26:77–101, 2005.

[26] Patrice Godefroid, Bo-Yuan Huang, and Marina Polishchuk. Intelligent rest api data fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 725–736, 2020.

[27] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 66–83, New York, NY, USA, 2021. Association for Computing Machinery.

[28] Jeff Huang, Qingzhou Luo, and Grigore Rosu. Gpredict: Generic predictive concurrency analysis. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 847–857. IEEE Computer Society, 2015.

[29] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. {RADBench}: A concurrency bug benchmark suite. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*, 2011.

[30] Yuseok Jeon, Wookhyun Han, Nathan Burow, and Mathias Payer. Fuzzan: Efficient sanitizer metadata design for fuzzing. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX Association, 2020.

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

[32] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.

[33] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 157–170, New York, NY, USA, 2017. Association for Computing Machinery.

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

[35] Youngjoo Ko, Bin Zhu, and Jong Kim. Fuzzing with automatically controlled interleavings to detect concurrency bugs. *Journal of Systems and Software*, 191:111379, 2022.

[36] Michalis Kokologiannakis, Iason Marmanis, Vladimir Gladstein, and Viktor Vafeiadis. Truly stateless, optimal dynamic partial order reduction. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–28, 2022.

[37] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 96–110, 2019.

[38] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In *International Conference on Computer Aided Verification*, pages 427–440. Springer, 2021.

[39] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, page 327–336. Association for Computing Machinery, 2011.

[40] Weiyu Luo and Brian Demsky. C11tester: a race detector for c/c++ atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 630–646, 2021.

[41] Nathan Mantel et al. Evaluation of survival data and two new rank order statistics arising in its consideration. *Cancer Chemother Rep*, 50(3):163–170, 1966.

[42] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. What happens-after the first race? enhancing the predictive power of happens-before based dynamic race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–29, 2018.

[43] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. Optimal prediction of synchronization-preserving races. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[44] Umang Mathur and Mahesh Viswanathan. Atomicity checking in linear time using vector clocks. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 183–199, New York, NY, USA, 2020. Association for Computing Machinery.

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

[46] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 267–280, USA, 2008. USENIX Association.

[47] Santosh Nagarakatte, Sebastian Burckhardt, Milo MK Martin, and Madanlal Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 543–554, 2012.

[48] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 97–108, New York, NY, USA, 2009. Association for Computing Machinery.

[49] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, 2018.

[50] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[51] Andreas Pavlogiannis. Fast, sound, and effectively complete dynamic race prediction. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–29, 2019.

[52] Doron Peled. All from one, one for all: on model checking using representatives. In *5th International Conference on Computer Aided Verification (CAV)*, 1993.

[53] Jing Peng and Ronald J Williams. Incremental multi-step q-learning. In *Machine Learning Proceedings 1994*, pages 226–232. Elsevier, 1994.

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

[55] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

[56] Niloofar Razavi, Franjo Ivančić, Vineet Kahlon, and Aarti Gupta. Concurrent test generation using concolic multi-trace analysis. In *Asian Symposium on Programming Languages and Systems*, pages 239–255. Springer, 2012.

[57] Jake Roemer, Kaan Genç, and Michael D Bond. Smarttrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 747–762, 2020.

[58] Sctbench: a set of c/c++ pthread benchmarks for evaluating concurrency testing techniques. https://github.com/mc-imperial/sctbench, 2016. Accessed: 2023-07-01.

[59] Koushik Sen. Effective random testing of concurrent programs. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 323–332, 2007.

[60] Koushik Sen. Race directed random testing of concurrent programs. In *29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008.

[61] Koushik Sen and Gul Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa verification conference*, pages 166–182. Springer, 2006.

[62] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[63] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with llvm compiler. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 110–114, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[64] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 387–400, New York, NY, USA, 2012. Association for Computing Machinery.

[65] Ole Tange et al. Gnu parallel-the command-line power tool. *The USENIX Magazine*, 36(1):42–47, 2011.

[66] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. Sound dynamic deadlock prediction in linear time. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[67] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. Controlled concurrency testing via periodical scheduling. In *Proceedings of the 44th International Conference on Software Engineering*, pages 474–486, 2022.

[68] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[69] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1643–1660. IEEE, 2020.

[70] Yu Yang, Xiaofang Chen, and Ganesh Gopalakrishnan. Inspect: A runtime model checker for multithreaded c programs. Technical report, Technical Report UUCS-08-004, University of Utah, 2008.

[71] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. *ACM SIGARCH Computer Architecture News*, 37(3):325–336, 2009.

[72] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 485–502, 2012.

[73] Xinhao Yuan and Junfeng Yang. Effective concurrency testing for distributed systems. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1141–1156, 2020.

[74] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, pages 317–335. Springer, 2018.