Selectively Uniform Concurrency Testing

Huan Zhao
zhaohuan@comp.nus.edu.sg
National University of Singapore
Singapore

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

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

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

Abstract

Buggy behaviors in concurrent programs are notoriously elusive, as they may manifest only in few of exponentially many possible thread interleavings. Randomized concurrency testing techniques probabilistically sample from (instead of enumerating) the vast search space and have been shown to be both an effective as well as a scalable class of algorithms for automated discovery of concurrency bugs. In this work we focus on the key desirable characteristic of black-box randomized concurrency testing algorithms — uniformity of exploration. Unfortunately, prior randomized algorithms acutely fall short on uniformity and, as a result, struggle to expose bugs that only manifest in few, infrequent interleavings. Towards this, we show that, indeed, a sampling strategy for uniformly sampling over the interleaving space, is eminently achievable with minimal additional information for broad classes of programs. Moreover, when applied to a carefully selected subset of program events, this interleaving-uniformity strategy allows for an effective exploration of program behaviors. We present an online randomized concurrency testing algorithm named Selectively Uniform Random Walk (SURW) that builds on these insights. SURW is the first of its class to achieve interleaving-uniformity for a wide class of programs, or an arbitrary subset of events thereof. This property translates to effective behavioral exploration should a subset with desirable characteristics be selected. Extensive evaluation on leading concurrency benchmarks suggests SURW is able to expose more bugs and significantly faster than comparable randomized algorithms. In addition, we show that SURW is able to explore both the space of interleavings and behaviors more uniformly on real-world programs.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; • **Security and privacy** → **Formal methods and theory of security**.

1 Introduction

Concurrency is essential to ensure responsiveness and performance of modern software programs. However, these benefits come at a cost of increased complexity and resulting concurrency-related bugs. These bugs are notoriously difficult to expose and reproduce, yet they can have dire consequences for the safety, security and availability of critical software systems [14, 29, 59]. While many developers turn to stress testing to find concurrency bugs, such methods are often insufficient because their effectiveness is highly dependent on the non-deterministic OS scheduler.

In response, *Controlled Concurrency Testing* (CCT) has emerged as a popular testing paradigm that addresses these challenges and is seeing increasing adoption in industry settings [12, 56]. CCT tools aim to uncover buggy program *behaviors* that may only exhibit under specific orderings of thread executions, called *interleavings*. To explore these interleavings, CCT tools typically serialize the execution by allowing only one thread to execute one atomic event at a time, allowing for greater control during testing. The number of possible interleavings is typically exponentially larger than the number of instructions in the underlying program; thus, a key challenge in designing effective CCT algorithms, lies in developing an effective exploration algorithm for the vast interleaving space.

Systematic testing approaches such as model checkers [18, 27, 43, 63] attempt to exhaustively traverse the space, and often employ techniques such as reduction [22, 57], bounding exploration to a small subset of behaviors, such as those with a small number of preemptive context switches [43] or short delays [18]. Unfortunately, in practice the enumerative

search algorithm may only explore a small proportion of similar interleavings in the allotted time-budget, especially for large software systems.

In contrast, randomized testing [6, 52, 63, 69, 70] approaches *sample* from the space to witness more diverse interleavings. As with their systematic counterparts, randomized testing algorithms also employ reduction techniques [65, 70] or explore only bounded sub-spaces which are considered more likely to harbor bugs [6, 63]. While these algorithms are not able to prove the absence of bugs, some provide probabilistic guarantees for observing buggy behaviors [6, 70]. In practice, these randomized algorithms tend to outperform systematic approaches [34, 61] while remaining scalable to large, real-world software.

In this work, we consider a key desirable objective of randomized CCT algorithms fundamental for effective exploration of the underlying search space — *uniformity*. In absence of prior knowledge about the bug or the program characteristics, the optimal exploration strategy should sample all interleavings with equal probability, i.e., must be *uniform*. Yet existing approaches of choosing each thread or each context switch point with equal probability *do not* yield uniformity, even for relatively simple programs (c.f. Section 2.1). Our first key insight is that interleaving-uniformity is eminently achievable for broad classes of programs. In fact, it can be attained in an efficient online algorithm, with only minimal information about the program under test, namely an accurate estimate of the number of events on each thread.

At the same time, we also observe that uniformity over the *interleaving space* does not necessarily translate to effective exploration of *program behaviors*. Often, many programs behaviors are represented by proportionally more (or fewer) interleavings than others, resulting in the over (or under)-sampling of such behaviors by an otherwise interleaving-uniform algorithm. Towards this, we present our second key insight — uniform sampling on the space of interleavings of an *appropriately selected subset* of program events can in fact serve as a good proxy for uniform exploration of program behaviors.

In this paper, we present a new randomized CCT algorithm, Selectively Uniform Random Walk (SURW), which exploits the aforementioned insights. SURW is the first randomized CCT algorithm that achieves *interleaving-uniformity* for a pre-selected subset of program events. As a result, given the full subset of all events, SURW samples all interleavings uniformly; given a subset of events that are likely key in determining the behavior of the program, it effectively explores program *behaviors*. We compare SURW with other randomized CCT algorithms qualitatively (c.f. Section 3.3), and also provide strong guarantees for finding concurrency bugs under several common concurrency design patterns (c.f. Section 3.4). SURW achieves selective uniformity by making *eager* scheduling decisions with *weighted* random walk, where the weights are determined to be estimates of the number of

events remaining to be scheduled, in each thread. The selected subset of events relevant for achieving selectivity can either be automatically inferred or manually provided as an input by a domain expert, thereby allowing complete control of the sampling granularity and focus.

We evaluate SURW extensively on leading CCT benchmarks [7, 30, 61]. Experiments show that SURW is able to (1) expose more bugs and (2) trigger bugs with fewer schedules than comparable randomized algorithms. We note that SURW achieves these results even with a simple heuristic for identifying promising selected event subsets. These results suggest that SURW has further potential to be combined with more sophisticated subset selection procedures. In our case study, we also find that SURW can explore the space of interleavings and behaviors better than comparable algorithms for real-world programs, both in terms of diversity and evenness.

In summary, we claim the following contributions:

- We propose interleaving-uniformity over selected events as a principled way to sample concurrent behaviors.
- We introduce the first randomized testing algorithm that guarantees interleaving uniformity for broad classes of programs, or for arbitrary parts thereof.
- We evaluated our approach extensively on leading benchmarks and real-world programs to demonstrate its efficacy in bug finding and space exploration.

2 Motivation

In this section, we discuss *selective uniformity* as a key desirable property for randomized CCT algorithms (Section 2.1 and Section 2.2), discuss prior algorithms and how they fare on this property, and also present insights behind how our SURW algorithm is structured.

2.1 Uniformity

The goal of randomized concurrency testing algorithms is to find bugs that arise from multi-threaded program *behaviors*. Without knowing *a priori* which of the many possible program behaviors trigger a bug, these algorithms should aim to sample such behaviors uniformly. Unfortunately, unique behaviors are highly program-dependent, and are impossible to predict or even count for complex programs [46, 64]. However, each unique program behavior *must* correspond to at least one interleaving for sequentially consistent programs. Additionally, unlike program behaviors, program interleavings *can* be sampled uniformly for broad classes of complex programs. A principled randomized CCT algorithm should strive to at least achieve *interleaving uniformity*.

Consider the illustrative program in Figure 1. It consists of two threads, each with 5 *atomic* events on a shared variable x. At each step, either 0 or 1 is appended to the binary representation of x, depending on which thread it is on. This program has $\binom{10}{5} = 252$ possible interleavings and the same

```
1   void thread_A() {        1   void thread_B() {
2       x = x<<1;            2       x = x<<1+1;
3       x = x<<1;            3       x = x<<1+1;
4       x = x<<1;            4       x = x<<1+1;
5       x = x<<1;            5       x = x<<1+1;
6       x = x<<1;            6       x = x<<1+1;
7   }                        7   }
```

Figure 1. Two threads performing atomic bit-shifts on a shared variable x. URW samples each result of x uniformly.

Figure 2. Histogram of the value of x in Figure 1 sampled by URW (left), Random Walk (middle) and PCT-10 (right).

number of behaviors, which in this case is determined by the value of the shared variable x at the end of an execution. Despite the simplicity of this relatively trivial example, existing randomized testing algorithms [6, 70] perform poorly on the uniformity metric, and are heavily biased in practice when sampling program interleavings. We discuss some of these next. At a high level, these algorithms work in an online manner, executing the program one event at a time. At each point during the execution, such algorithms typically pick one of the *enabled* (i.e., unblocked) threads, either deterministically (by means of, say, assigning priorities) or randomly by picking an enabled thread according to some probability distribution.

Random walk. This algorithm is the simplest randomized CCT algorithm and works by scheduling, at each point, each of the enabled threads to execute next, with equal probability. This simplistic uniformity on possible scheduling decisions at each point, however, does *not* translate to uniformity over the space of interleavings! For the program in Figure 1, the distribution on the resulting interleavings generated by Random Walk is highly skewed — the two most likely interleavings[1] are sampled with probability 2^{-5}, whereas a majority of interleavings are sampled with probability only 2^{-9}. We note that this disparity gets exacerbated in larger programs with uneven thread lengths. Figure 2 (middle) shows the empirical distribution of interleavings sampled by Random Walk in this example, with a few interleavings being sampled with much higher frequency than others.

POS. Partial Order Sampling (POS) [70] employs reduction to reduce the bias induced due to equivalent interleavings on top of Random Walk. Operationally, it assigns a single random priority per thread to increase the chance of concurrent accesses being simultaneously enabled (i.e., *racing*), and, when that happens, POS chooses each racing event with equal probability. In our running example (Figure 1) though, all events race with each other and thus POS chooses a random thread at each step, degrading it to Random Walk.

PCT. The Probabilistic Concurrency Testing (PCT) family of algorithms [6, 44–46] cater to bugs of shallow *depth*, parametrized by a parameter d, and operationally impose
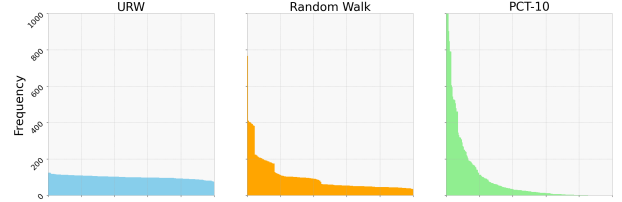
only $d - 1$ intentional context switches. These switch points are sampled from all possible positions in the interleaving with equal probability. For PCT to witness all possible results of x in Figure 1, the depth parameter must be set so that $d \geq 10$ (giving us PCT-10). PCT-10 is guaranteed to witness any result with a probability of $2^{-1} \cdot 10^{-9}$, which is admittedly a poor bound for a program with only 252 possible interleavings. Indeed, we see that the distribution of interleavings imposed by PCT-10 is highly skewed (see Figure 2 (right)). We observed that, even after more than 25000 repeated trials in our experiment, PCT-10 fails to generate 38 unique interleavings.

URW. Let us now discuss the Uniform Random Walk (URW), the first algorithm we propose in this work. URW samples from the interleaving space of the entire program in Figure 1 provably uniformly, and our evaluation confirms this (see Figure 2 (left)). Crucially, we observe that sampling each thread (Random Walk) or each context switch point (PCT) uniformly *does not* yield uniformity in the interleaving space. Instead, the key ingredient for uniformity is an accurate estimate of the *number of events* in each thread that are remaining to be scheduled. URW is then essentially a *weighted* random walk that chooses each thread T_t with probability proportional to the number of remaining events in T_t.

2.2 Selectivity

While interleaving uniformity is desirable for randomized CCT algorithms, admittedly, it *does not* always translate to behavioral uniformity. In Figure 1, program behaviors and interleavings are synonymous; but in general, a single program behavior may often result from multiple distinct interleavings. Further, the number of interleavings that are behaviorally equivalent to each other could be highly variable – some behaviors could correspond to disproportionately more interleavings than others. An algorithm like URW that focuses on interleaving uniformity is likely to *over-sample* these behaviors. Our second key insight is to uniformly sample interleavings *selectively*, only for a subset of events whose interleavings are more evenly distributed across behavioral partitions. We call this subset of events as *interesting* events, since distributions on interleavings of these events have a closer correspondence with distribution on behaviors.

---

[1]The two most likely interleavings are $x = 31$ or 31×2^5. The least likely ones end with 0b10 or 0b01. There are $2 \cdot \binom{8}{4} = 140$ such results.

```
1   void thread_A() {          1   void thread_B() {
2       x = x<<1;               2       y = y<<1+1;
3       x = x<<1;               3       // repeat 1000x
4       x = x<<1;               4       y = y<<1+1;
5       x = x<<1;               5       x = x<<1+1;
6       x = x<<1;               6       x = x<<1+1;
7       y = y<<1;               7       x = x<<1+1;
8       // repeat 1000x         8       x = x<<1+1;
9       y = y<<1;               9       x = x<<1+1+y%8;
10  }                           10  }
```

Figure 3. Two threads performing atomic bit-shifts on dependent shared variables x and y. Behaviors $x = 31 + y\%8$ happen in dominantly many interleavings.

Consider the more involved illustrative program in Figure 3; here again we use the final value of the shared variable x as proxy for the program behavior. Here, x is dependent on y (line 9 of thread_B, or B9 for short), and y is accessed 1000 times on each thread (in blue). The behavioral partitioning of the interleaving space is thus highly skewed. Concretely, to witness any result $x \neq (31 + y\%8)$, event A6 has to be scheduled after event B5. This happens with vanishingly small probabilities even with interleaving uniformity, since A6 has to be delayed after 1000 accesses of y on thread B (B2-B4).

Readers may have observed that accesses to x (in black) are a candidate subset of *interesting* events. Indeed, their interleavings are more evenly partitioned behaviorally and heavily influence the program's behavior. If we have this information in advance, we might try to explore interleavings of only accesses to x, rather than the entire program. This allows us to sample 251 values of x with probability $\frac{1}{252}$, and 8 other values with a combined probability of $\frac{1}{252}$.[2] Despite a slight bias, the resulting distribution is much closer to behavioral uniformity.

We remark that such selectivity must be exercised with caution. Consider, for example, the natural (but naive) extension of URW that invokes the weighted random walk only when enabled events are interesting, and otherwise continues executing the previously executed thread. This strategy indeed recovers interleaving uniformity for x, but it unfortunately comes with considerable loss of exploratory power — now only two of $\approx 2^{1000}$ possible interleavings of y (in blue) can be witnessed, and thus $y\%8$ only takes its value from either 0 or 7. This makes the algorithm *incomplete*, because some program behaviors could never be sampled. Moreover, this precludes interactions between accesses on x and y, which could also be problematic for a general program.

Therefore, we aim to design a *selectively uniform* algorithm featuring the following two key properties; here we use Γ to denote the set of all events of the program, and Δ to be the set of events chosen to be interesting (also note $\Delta \subseteq \Gamma$):

Γ-**Completeness** For any feasible interleaving of the entire program (that include all events Γ), the algorithm samples it with *non-zero* probability;

Δ-**Uniformity** For the chosen subset $\Delta \subseteq \Gamma$ of events, the algorithm samples their interleavings *uniformly*.

In this paper, we devise a algorithm named Selectively Uniform Random Walk (SURW) that satisfies the aforementioned properties. Given a pre-defined subset of interesting events Δ, it prioritizes sampling of their interleavings uniformly (with URW), and schedules other events carefully to respect the ordering constraints from interesting events. We delegate detailed discussions on how to isolate the set of interesting events Δ to Section 3.6. For Figure 3, if Δ contains all accesses to x, SURW can sample each interleaving of x uniformly, without disabling any result of y or any interaction between x and y. This achieves the desirable distribution described earlier.

3 Approach

In this section, we present the design and analysis of URW and SURW in detail, and discuss their guarantees for interleavings (Section 3.3) and bug finding (Section 3.4) in simple settings, and likely behaviors in more complex programs (Section 3.5). We provide guidance on how to identify the set of interesting events in Section 3.6.

3.1 URW

Randomized CCT algorithms typically sample interleavings in a *streaming* fashion: at each step, the algorithm determines an *enabled* thread to execute its next event. A thread is said to be enabled if its next event is executable. We assume that the next event of any enabled thread is visible to the algorithm and is solely determined by the execution history.

We first present our basic algorithm URW (Algorithm 1). URW takes, as input, integers $n_1, n_2 \ldots, n_k$, corresponding to threads $T_1, \ldots, T_k$ of the program-under-test (line 1). URW is guaranteed to uniformly sample those interleavings of the program whose length is $n = \sum_{i=1}^{k} n_i$ and contain, for each i, n_i events from thread T_i, when there is no blocking (e.g., wait-signal) synchronization, i.e., a thread is always enabled until it exits. We postpone the discussion on programs with synchronizations to Section 3.5.

At its core, URW is essentially *weighted* random walk, where the weights are determined by the number of remaining events in each thread. At each step, URW selects a thread T_t until no more thread is enabled (line 2). Each enabled thread T_i is selected with a weight that is proportional to the number of events n_i remaining on that thread (line 3). The weight of thread T_t is initialized using the input provided, and decreased by one whenever an event of thread T_t is selected to be executed next (line 4).

URW's uniformity is a consequence of the choice of the weights. Intuitively, the weight of thread T_i represents the

---

[2]Here, the less likely behaviors are $x = (31+y\%8)$, from a single interleaving of x (A6 before B5) and different interleavings of y. All other interleavings of x give $y\%8 = 0$ and thus corresponds to a single program behavior.

Algorithm 1: URW

```
1 Input: n₁, ..., nₖ;                    // event counts
2 while E ← getEnabled() ≠ ∅ do
3   │  Tₜ ← random Tᵢ with Pr(Tᵢ) = nᵢ/∑ⱼ∈E nⱼ;
4   │  nₜ ← nₜ − 1;
5   │  execute(nextEvent(Tₜ));
```

Algorithm 2: SURW

```
1  Input: Δ;            // set of interesting events
2  Input: n₁, ..., nₖ;  // interesting event counts
3  T_{iNext} ← random Tᵢ weighted by nᵢ;
4  blocked ← ∅;
5  while E ← getEnabled() ≠ ∅ do
6    │  Tₜ ← pickFrom(E - blocked);
7    │  if nextEvent(Tₜ) ∈ Δ then
8    │    │  if T_{iNext} == Tₜ then
9    │    │    │  nₜ ← nₜ − 1;
10   │    │    │  T_{iNext} ← random Tᵢ weighted by nᵢ;
11   │    │    │  blocked ← ∅;
12   │    │  else
13   │    │    │  blocked.add(Tₜ); continue;
14   │  execute(nextEvent(Tₜ));
```

number of interleavings starting with NextEvent(T_i). Suppose that, at some point of the execution of the algorithm, exactly k' threads $T_{j_1}, T_{j_2}, \ldots, T_{j_{k'}}$ are enabled. Let us use e_{j_i} and n_{j_i} to denote the next event and the number of remaining events in thread T_{j_i}. In this case, the number s_{j_i} of possible extensions of the execution generated thus far that also start with e_{j_i} (for some i) can be calculated to be precisely[3]:

$$s_{j_i} = \binom{\sum_{i=1}^{k'} n_{j_i} - 1}{n_{j_1}, \ldots, n_{j_i} - 1, \ldots, n_{j_{k'}}}$$

Now observe that for any $\alpha \neq \beta$, the fraction $\frac{s_{j_\alpha}}{s_{j_\beta}}$ is precisely $\frac{n_{j_\alpha}}{n_{j_\beta}}$, as desired. Therefore, our choice of weights *perfectly* sketches the count of feasible interleavings, and thus inductively leads to uniformity.

3.2 SURW

In this section, we present SURW (Algorithm 2) which extends URW. We denote the set of all events as Γ. SURW takes as input a pre-defined subset of interesting events $\Delta \subseteq \Gamma$ (line 1), and per-thread count n_i's of interesting events (line 2).

As preluded in Section 2.2, selective uniformity is challenging in a streaming setup. In particular, the algorithm must make scheduling decisions about a mix of interesting events (in Δ) and other events (in $\Gamma - \Delta$) that are simultaneously enabled. Deterministic scheduling (e.g., always picking an event in Δ first) could potentially disallow interleavings over Γ that are otherwise feasible, violating the requirement Γ-Completeness. On the other hand, naive randomized scheduling (e.g., choosing e_1 first with some probability) could deter uniformity over Δ, thus violating Δ-Uniformity. Our algorithm circumvents this — it makes *eager* scheduling decisions for events in Δ, potentially even before they are actually enabled.

More specifically, SURW determines (in advance) a thread T_{iNext} that is *intended* to execute the next interesting event from Δ. Out of the many threads with non-zero (remaining) interesting events, we select this intended thread via weighted random walk as before (line 3). Our algorithm then ensures that henceforth, the first interesting event to be scheduled necessarily comes from thread T_{iNext}. This is achieved by actively *blocking* any other thread which is about to execute an interesting event from Δ (line 12-13). On the

---

[3]Multi-choose function $\binom{n}{n_1, \ldots, n_k} = \frac{n!}{\prod_{i=1}^{k} n_i!}$ (where $n = \sum_{i=1}^{k} n_i$) represents the number of ways to partition n elements into k sets with different sizes.

other hand, threads with non-interesting events are allowed to execute. This is enough to ensure that the resulting distribution on the interleavings of the interesting events (from Δ) is fully determined by URW, thus satisfying Δ-Uniformity.

In the meantime, the relative ordering among uninteresting events (from $\Gamma - \Delta$), as well as their order relative to the next interesting event, is determined by the function pickFrom() (line 6). The function pickFrom() yields a thread from the provided set, and is parameterized by another CCT algorithm. We note that the details of pickFrom() *do not* affect the distribution over the interleavings over Δ. This is because any unintended interesting events will always be blocked. Further, when pickFrom() is selected so that it samples each interleaving over Γ with non-zero probability (when $\Delta = \emptyset$), then SURW is guaranteed to be Γ-Complete. In this work, we satisfy this requirement using a simple implementation of pickFrom() that assigns a random priority to each event independently.

3.3 Comparison with Prior CCT Algorithms

In this section, we compare SURW's guarantee with prior popular randomized CCT algorithms, and present a qualitative comparison with them.

SURW v/s POS. POS [70] achieves its efficacy through on-the-fly dynamic partial order reduction. If one marks conflicting accesses as interesting (Δ), SURW could achieve a similar effect. Further, when the equivalence relation induced by the underlying partial order is too fine-grained (as in, Figure 1), then SURW is likely to outperform POS. This happens because, in this case, the effect of the reduction is not prominent and POS essentially degrades to its baseline, namely, Random Walk, and, as discussed earlier, SURW improves over naive

Random Walk by boosting the probability of sampling otherwise infrequent interleavings.

SURW v/s PCT. PCT-d [6, 45], is intuitively catered to finding bugs determined by ordered d-tuples of potentially buggy events. This family of bugs admits efficient random testing approaches [8, 34], and is pivotal to the effectiveness of PCT. Indeed, when the depth d is small and is known accurately a priori, then PCT-d is likely to be more effective. On the other hand, when the bug depth d is known to be large, or is unknown entirely, uniformity of behaviors or interleavings is a good desirable property of the testing strategy, making SURW a good choice, given that SURW is more versatile and requires little knowledge about bug characteristics. In particular, we observed that PCT is ineffective when ordering constraints, a la PCT [8, 34], alone cannot characterize the class of buggy interleavings (see Figure 4 and Section 4.2). Furthermore, the effectiveness of PCT is poor when the bug in question manifests only when multiple context switches happen in close temporal proximity (see CVE-2016-1972 [13, 63]). SURW naturally avoids these pitfalls by sampling uniformly from the interleaving space and allowing for context switches at each point during the execution.

3.4 Bug Finding Probability

We have presented the guarantee of SURW of hitting a particular interleaving. In practice, bugs often manifest in many possible interleavings. Without prior knowledge of bug characteristics, however, it is hard to extrapolate this guarantee to bug finding. If the concurrent program follows certain threading configurations, as often adopted in real-world systems, SURW is able to give strong probabilistic guarantees of hitting a particular *bug*. In contrast, other algorithms such as PCT could not take advantage of it as much. The strengthened guarantees *do not* require additional knowledge about the program under test, but instead rely on the following property: Δ-Uniformity implies $\Delta_{\mathcal{T}}$-Uniformity, where $\Delta_{\mathcal{T}} \subseteq \Delta$ denotes the set of interesting events that appear on a subset of threads $\mathcal{T} \in \mathcal{P}$. Below we introduce three common threading scenarios that admit strong guarantees.

Irrelevant threads. Consider a program where bug manifestation only depends on $k_0 < k$ threads. For example, monitoring or logging threads access shared resources frequently but do not affect program behaviors. With Δ-Uniformity, SURW also guarantees uniformity for the k_0 relevant threads.

Clusters. Consider a program where threads are organized into c duplicated clusters, each containing m threads. Bug exhibition depends on the interleavings of threads *within* a single cluster. For example, a web server may spawn one independent cluster of threads for each incoming client. To witness the bug, it suffices to sample the intra-cluster schedule for any cluster, which is sampled uniformly and independently by SURW. Therefore, the success probability is

$$\geq 1 - \left(1 - \binom{n/c}{n_1, \ldots, n_m}^{-1}\right)^c$$

where $\{T_1, \ldots, T_m\}$ forms a cluster.

Duplicates. Consider a program with k threads but only two distinct types, namely type A and B. Bug exhibition depends on the interleaving of *any* pair of type-A and type-B threads. This could resemble programs adopting a producer-consumer or reader-writer model. Suppose there are k_a type-A threads and k_b type-B threads, each consisting of n_a and n_b interesting events, respectively. With SURW, the schedule of any such pair is sampled uniformly and independently. Therefore, the bug is triggered with probability

$$\geq 1 - \left(1 - \binom{n_a + n_b}{n_a, n_b}^{-1}\right)^{k_a \cdot k_b}$$

3.5 Blocking Synchronizations

In previous sections, we give probabilistic guarantees of SURW in the absence of blocking synchronizations such as wait-signal pairs. Each such pair imposes a partial order constraint that `signal` happens before `wait`, and program interleavings could be viewed as linearizations of the partial order sets. Uniform sampling or exact counting of all linearizations is known to be intractable [17, 64]. While polynomial-time algorithms exist for special cases [15, 41] or approximations [60, 64], they are not applicable in a lightweight streaming algorithm. However, under common synchronization scenarios, such as those induced by thread creation or critical sections, we could adopt simple strategies to achieve more even exploration.

Thread creation. A child thread is not enabled until the thread creation event on its parent thread is executed. However, the sampling weight of the parent thread does not take this dependency into account, thus *under-sampling* interleavings where the events on children threads are scheduled early. To mitigate this effect, we schedule each parent thread T_p with a combined weight $n_p + \sum_{i \in C_p} n_i$ rather than n_p, where C_p comprises of all unspawned descendant threads of T_p. Moreover, we need to *re-select* the next intended thread with updated weights after it creates a new thread. In effect, this allows us to recover Δ-Uniformity.

Critical sections. Critical sections (CS) protect shared resources by preventing conflicting accesses. Let us consider when Δ is chosen such that all events are protected by CS for mutual exclusion. In that case, SURW may schedule lock acquisition from another thread before the interesting event on T_{iNext}, thus blocking the intended next thread as it waits for a held lock. Moreover, if CS contains multiple events from Δ, the event counts over-approximate the number of feasible interleavings. A naive solution that treats each CS as an interesting atomic block recovers uniformity on the interleavings

of CS, but violates Γ-Completeness since events inside CS and other code segments can no longer be interleaved. One simple mitigation is to mark only the *entrances* (lock acquisition) of CS as the interesting subset Δ. In this way, the interleavings of CS could be uniformly sampled by SURW, and events protected by the CS can still be interleaved with other events, since they are all considered non-interesting.

3.6 Choosing the Set of Interesting Events

Our presentation of SURW, thus far, has intentionally treated the set of interesting events Δ as a parameter, since a characterization of an ideal choice of Δ is orthogonal to the core working of SURW. Nevertheless, the quality of Δ is expected to be crucial to the performance of SURW. In general, there are multiple ways to identify Δ — one can perform simple, yet automated coarse-grained program analyses, possibly together with some simple guidance, or alternatively a domain expert can help manually supply this set. We next discuss specific heuristics to determine the event set Δ in practice.

A precise characterization of the set Δ is challenging in general, especially without working knowledge about the internals of the program under test. Nevertheless, we present a simple but fully automated strategy and also employ it to obtain a baseline instantiation of SURW (see Section 4). Here, we first *randomly select* a small set of shared resources, and then mark all events that access this set of resources as interesting. This strategy is inspired by prior empirical studies [33] that highlight that most real-world concurrency issues can be attributed to only a few shared variables. Concretely, for our baseline implementation of SURW in Section 4, we perform an initial profiling run, and randomly select a few *heap and global variables* observed in this profiling run, and pick Δ to be the set of all events that access these variables. Despite this crude identification of Δ, SURW demonstrates remarkable efficacy in our evaluation. We expect a more sophisticated automated selection strategy to further improve the performance of SURW.

Expert advice from humans (or even Large Language Models) can help identify a more precise and effective choice for the set Δ. We put this hypothesis to test in our case study of LightFTP (see Section 5). Here, we select the set of interesting events to be those that correspond to all events pertaining to file-system accesses. Our high-level intuition behind this choice is rather straightforward — the key behavior or a file-transfer protocol (FTP) server is to modify the file system, and thus the space of behaviors of the FTP server are likely going to be primarily determined by the state of the file system, which can in turn be accurately captured by accesses to it.

We remark that the of design SURW algorithm carefully takes into consideration the imprecision due to a poor choice of the set Δ. This is why we emphasise SURW's Γ-Completeness property in Section 2.2: when the set Δ is imperfect, the behavioral distribution induced by SURW may be skewed, but SURW is still guaranteed to sample each interleaving with non-zero probability. In other words, the algorithm still functions effectively, albeit not optimally.

4 Evaluation

In this section, we aim to understand the bug-finding ability of SURW in practice on popular concurrency testing datasets [7, 30, 61]. Concretely, we answer the following:

RQ1 Is SURW better at exposing bugs compared to other concurrency testing algorithms?

RQ2 How do the two key components of SURW, uniformity and selectivity, contribute to its effectiveness?

4.1 Experiment Setup

Implementation. We implement SURW and all other baselines using a custom scheduler. This scheduler serializes a program's execution by overriding various functions in the pthread library as in prior work [65]. The artifact and data used in the paper is publicly accessible at:

https://doi.org/10.6084/m9.figshare.27627123

Interesting event subsets. Before running SURW, we conduct a *single* profiling run. For each execution of SURW, we mark all accesses to *randomly selected set of shared variables* as interesting events (see Section 3.6). Their counts are also collected from the profiling run (see Section 7 for more discussions).

Instrumentation. We instrument *all* heap memory operations with a static binary rewriting tool E9Patch [16] to invoke the scheduler before each memory operation. For three target programs, we instrument only a subset of heap-memory accesses to achieve higher execution speed by omitting instructions that access read-only memory addresses or are invoked excessively (>500 times) during execution. These partially instrumented targets are denoted in Table 2.

Concurrency testing benchmarks. To measure the bug finding ability of our approach, we adopt three benchmarks, namely SCTBench [50, 61], ConVul [7, 10] and RaceBench-Data [30, 48]. SCTBench and ConVul have been widely adopted in the concurrency testing literature [63, 65, 70]. SCTBench comprises of 42 buggy concurrent programs from real-world concurrency bugs, including command line tools [67, 68], concurrent data structures [11, 43] and a JavaScript engine for FireFox [24]. We omit two programs from SCTBench that are reported by [65] as incompatible with the instrumentation tool E9Patch [16]. The ConVul benchmark consists of 10 bugs extracted from concurrency-related CVEs from 2009 to 2017 that cause memory corruption in FireFox and Linux kernels. We use the version curated in [63]. RaceBenchData comprises of 15 base programs collected from existing suites [5, 66]. 5 of the base programs consist of more than 10k lines of production-level code, including ray tracing libraries, volume rendering and video encoding tools. These programs

Table 1. Result summary on SCTBench and ConVul: the number of bugs found (max. 39) by different algorithms. All other algorithms only find a subset of bugs exposed by SURW.

# of Bugs	SURW	PCT-3	PCT-10	POS	RW	N-U	N-S
Total	35	33	33	30	21	32	34
Mean	34.90	30.70	30.65	29.05	20.85	29.75	31.75

have had synthetic concurrency bugs injected into them, including deadlocks, atomicity violations and order violations. The injected bugs introduce overheads of up to 400% into the base programs. In all three benchmarks, bugs manifest as assertion violations or crashes.

Baselines. We choose other well-known stateless, randomized CCT algorithms for direct comparison with SURW. These algorithms include naive Random Walk (RW), Partial Order Sampling (POS) [70] and Probabilistic Concurrency Testing (PCT) [6]. Specifically, we choose PCT-3 and PCT-10. PCT-3 is reported to perform best on the SCTBench benchmark by [61]. We also include PCT-10 as a representative case for PCT with a large depth. Additionally, we conduct an ablation study for each of the two key components of SURW: uniformity and selectivity. We implement two ablative variants, namely Non-Uniform (N-U) and Non-Selective (N-S), to investigate the impact of each component in isolation.

4.2 RQ1: Discovery of Bugs

To determine if our approach leads to better bug discovery, we evaluate SURW and other algorithms on SCTBench [50], ConVul [10] and RaceBenchData [48] Benchmarks.

SCTBench and ConVul. On SCTBench and ConVul, we measure the number of schedules to the first exposure of a bug, following prior work [65]. For a single session, we sample up to 10^4 schedules for each scheduling algorithm. We conduct 20 such sessions for each program in these benchmarks. In the case of one particular program, SafeStack, we instead sample 10^6 schedules per trial, as it has been proven elusive in previous works [65, 70]. For concision, we omit 11 trivial programs from our results where all algorithms sample the buggy schedule within 10 executions on average.

We greatly *disadvantage* SURW by choosing the interesting events Δ to be all accesses to a *single* randomly selected shared variable. The variable we selectively explore with SURW is picked with probability proportional to its total access count. Still, we find that even with *no additional information* about which events are interesting, SURW is extremely successful in finding bugs.

We present the detailed breakdown for the number of schedules required to expose each bug in Appendix A. Table 1 summarises the number of bugs found for each algorithm, where "Total" refers to the cumulative number of distinct bugs found across 20 sessions, and "Mean" refers to the average number of bugs found in each session. SURW exposes more bugs (34.90 on average) than other algorithms with

Table 2. The # of unique bugs exposed in RaceBenchData. Each row is a base program that contains 100 bugs in total. Targets with ∗ are selectively instrumented.

Target	SURW	PCT-3	PCT-10	POS	RW
blackscholes	36	15	17	26	20
bodytrack	75	34	35	58	29
canneal	62	13	20	56	34
cholesky*	77	36	46	77	49
dedup	79	22	22	80	45
ferret	80	41	54	84	46
fluidanimate*	70	46	53	68	28
pigz	40	28	36	40	16
raytrace	64	24	24	58	42
raytrace2*	73	18	24	74	50
streamcluster	78	31	43	77	31
volrend	32	17	19	29	12
water_nsquared	28	13	15	27	26
water_spatial	86	29	38	90	50
x264	64	12	15	41	11
Total (max. 1500)	944	379	461	885	489

statistical significance ($p < 10^{-4}$ by Mann-Whitney U Test [35]). More importantly, *no* other algorithm triggers bugs that SURW fails to expose. We perform log-rank test [36] to determine the algorithm that requires the least number of schedules for bug discovery. With significance level $p < 0.05$, SURW claims the best performance on 26/35 targets. On 6 of the other 9 targets, SURW requires only ≤ 10 schedules on average to trigger the bug.

RaceBenchData. In addition to the SCTBench and Convul benchmarks, we evaluate the algorithms on the RaceBenchData benchmark [48]. For this dataset, we follow the RaceBench [30] authors in using the total number of distinct bugs exposed as our metric for effectiveness. Each base program is embedded with 100 bugs, which we execute for 5×10^4 iterations with each algorithm. For these benchmark programs, we instantiate SURW's interesting event set to be all accesses to a *random memory region* that contains multiple shared variables, whose combined counts are above a threshold.

We present our results in Table 2. Here we see that SURW is able to expose 944 distinct bugs, followed closely by POS at 885 bugs. Of the 15 programs, SURW finds the most unique bugs in 11 of them, followed by POS which finds the most bugs in 6 programs. PCT-3, PCT-10 and Random Walk don't find the most bugs for *any* of the evaluated programs.

We note that each program instance in this benchmark contains multiple bugs, and that this can cause issues for approaches that maintain an event count, such as SURW or PCT. If a shallow bug causes the program execution to terminate early, the observed event counts may no longer be representative for a non-crashing execution. These premature crashes may explain the relatively smaller difference between SURW and POS as compared to the results from our other two benchmarks.

Surprisingly, both variants of PCT perform worse than naive Random Walk on this benchmark. While inaccurate count estimates also affect PCT, we attribute its poor performance predominantly to the characteristics of the injected bugs. Specifically, many of these bugs are of *high depth* (up to 12) and the execution traces are *extremely long* (up to millions of events). The bound of PCT is proportional to $\frac{1}{n^d}$ where n is the trace length and d is the depth.

> SURW *outperforms comparable randomized CCT algorithms by a large margin on three leading benchmarks, even with only randomly selected interesting event sets.*

To further understand the performance of different algorithms, we delve into the series of `reorder` examples from SCTBench [50], where the number in the target name parameterizes the number of threads involved. A simplified version of `reorder_100` is shown in Figure 4. As few as *one* context switch is sufficient to trigger the bug on line 8, by switching to the checker thread immediately after any setter thread executes line 3. That does not imply the bug is straightforward to trigger – the scheduler has to enforce that *no* setter thread executes line 4 before the check on line 7. Indeed, no other algorithm is able to find this bug *even once* in any of the 20×10^4 runs we conducted.

As highlighted in Section 3.3, ordered tuples can be *insufficient* for bug manifestation, as other events (here, line 4 from another setter thread) could disable buggy behaviors for correctly ordered tuples. On the other hand, POS exploits partial order for concurrent reads and writes. However, as setter threads are created first, they tend to execute first as well. It is thus exponentially unlikely for the checker's access on b enabled simultaneously with others'.

In contrast, SURW exposes the bug within 200 schedules on average (see Appendix A). The probability to trigger this bug is indeed $\approx \frac{1}{200}$: after all accesses to variable b are chosen as interesting events Δ (with probability $\frac{1}{2}$), it suffices for SURW to schedule the checker thread first (with probability $\frac{1}{100}$). With thread counts as future information, SURW is able to dictate the first access to variable b even before it is enabled.

4.3 RQ2: Ablation Study

To assess how each component of SURW, namely uniformity and selectivity, contributes to its performance, we conduct an ablation study. We implement two ablative baselines to compare with SURW. The first, Non-Uniform (N-U), uses the same selectivity strategy and algorithm as SURW, but applies a naive random walk to the interesting subset rather than URW (line 9 of Algorithm 2). This setup demonstrates the impact of our uniform sampling algorithm for the interesting subset. The second, Non-Selective (N-S), applies URW to the entire program; i.e. *all* events are considered interesting. This setup shows the impact of selectivity for SURW.

```
1   static int a = 0, b = 0;
2   void setter_thread() {
3       a = 1;
4       b = -1;
5   }
6   void checker_thread() {
7       if (!(a == 0 && b == 0) && !(a == 1 && b == -1)) {
8           assert(0); // Bug triggered
9       }
10  }
11  int main() {
12      int n_set = 99;
13      int n_check = 1;
14      for (int i = 0; i < n_set; i++) {
15          std::thread(setter_thread);
16      }
17      for (int i = 0; i < n_check; i++) {
18          std::thread(checker_thread);
19      }
20      return 0;
21  }
```

Figure 4. Simplified example of `reorder_100` in SCTBench.

We conduct our ablation study on the SCTBench and Convul Benchmarks [10, 50]. A summary is presented in the last two columns of Table 1, with full results shown in the Appendix A. Both ablative versions fail to locate all bugs found by full-fledged SURW.

Effect of uniformity. SURW shows nearly a strict improvement over N-U on the benchmark targets. On the series of `reorder` programs (e.g. Figure 4), N-U performs notably worse than SURW. This is because it suffers from a similar pitfall as POS. More tellingly, uniformity exploits the duplicate and irrelevant threading pattern as discussed in Section 3.4. Consider `reorder_20`, which is a variant of Figure 4 that spawns 10 setter threads and 10 checker threads. SURW samples the interleaving of each checker independently with respect to all the setters, allowing it to consistently crash `reorder_20` in *just 10 schedules*, despite this example having substantially more possible interleavings than other threading configurations with comparable results, such as `reorder_3`. In contrast, N-U requires $> 500\times$ as many schedules to find the bug on average.

Effect of selectivity. Selectivity allows SURW to focus on a smaller subset of interesting events, and here we see it also outperforms our ablative N-S algorithm by a wide margin. In particular, SURW scales much better as the number of events in the program execution increases. Most visibly, SURW could crash programs with large numbers of threads (and thus events) such as `reorder_100` and `twostage_100` within a few hundred schedules on average, improving the performance of N-S by 5-10×. Moreover, N-S also lacks SURW's robustness when facing irrelevant events and locks. For example, SURW successfully triggers the bug in IWSQ by focusing on a single variable with less than 30 accesses. N-S, however, has to navigate through more than 3000 total events, many of which are protected by locks. These locks can greatly

reduce the number of possible interleavings, causing N-S's sampling weights to be misleading. To make matters worse, the number of events unrelated to the bug fluctuates drastically as the control flow of the program changes, further distorting the weights for N-S. SURW, on the other hand, is largely unaffected when the correct variable is selected.

> *Both uniformity and selectivity significantly contribute to* SURW*'s success; on the SCTBench and ConVul benchmarks uniformity appears to have a slightly larger effect.*

5 Case Study: LightFTP

In the case study, we investigate how SURW explores interleavings and behaviors of real-world software. We conduct an in-depth case study with a production-ready web server, LightFTP [31]. LightFTP is a popular implementation of a File Transfer Protocol (FTP) server, that has been widely deployed for software distribution and data exchanges in enterprise settings. Its integrity and reliability is extremely crucial, as it facilitates direct file system operations for clients. The server is multi-threaded in nature to accommodate concurrent client accesses. By design, it spawns a thread (and possibly more worker threads) for each incoming connection. However, it has no built-in scheduler and the order of request processing purely depends on the OS, making it a perfect target for CCT tools.

To expose the multi-threaded behavior of the server, we created 4 concurrent clients which all operate on a single shared directory, each sending a randomly shuffled command sequence.[4] After authenticating itself, each client issues 3 utility commands, 3 MKD to create its own directories and 3 RMD to remove others' directories – all in a random order. Each client then sends one PASV-LIST request to fetch the folder structure before disconnecting.

Here we are interested in both interleavings and behaviors of the target: the interleavings are the temporal ordering of file system accesses; and program behaviors are represented by the final file structure returned by the last LIST command. We investigate how well SURW explores the search space on real-world programs, with the following two primary metrics: (1) [*coverage*] the number of distinct interleavings and behaviors witnessed; and (2) [*uniformity*] how evenly are these interleavings and behaviors sampled.

We include PCT-3, PCT-10 and Random Walk for comparison. POS is excluded since the events do not involve memory accesses. We also *disadvantage* SURW relative to others by granting naive selectivity for all algorithms, i.e., all algorithms only need to schedule events relevant to the interleavings and the behaviors. Unlike the benchmarks used in Section 4, program behaviors are fully determined by the

---

[4]To facilitate automated testing, we modify the server to shut down after all clients disconnect, instead of waiting for termination on keyboard inputs.
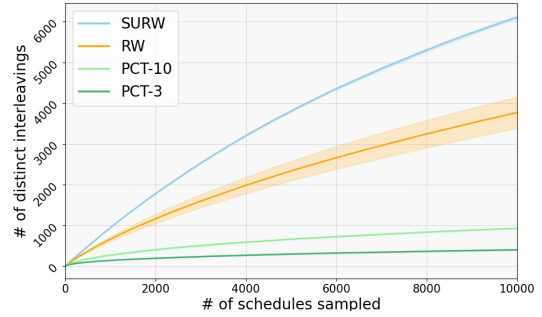
(a) Interleaving coverage
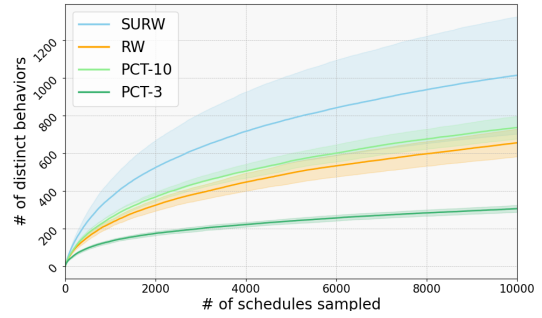
(b) Behavioral coverage

Figure 5. Interleaving (a) and behavioral (b) coverage for LightFTP: # of distinct interleavings/behaviors vs # of schedules sampled. SURW achieves the highest coverage in both.

relative ordering of MKD and RMD pairs. This setup further favors PCT since it specializes in ordering patterns. We record the results for 10k iterations in each trial, and repeat for 20 trials. The main results are shown in Figure 5 and Table 3.

Figure 5a and 5b shows the number of distinct interleavings[5] and behaviors witnessed as the sample size increases, respectively. The solid line depicts the mean and the shaded region represents the standard deviation.

SURW consistently covers around 50% more interleavings than Random Walk and 6× more than PCT-10 on average, with a negligible variance. For program behaviors, SURW witnesses the most unique states of around 1000 in expectation. However, it inflicts a high variance due to randomly shuffled client commands. This suggests an unfortunate sensitivity of interleaving sampling when exploring ordering patterns. Notably, despite exploring far fewer *interleavings* than Random Walk, PCT-10 manages to explore more distinct behaviors on average (though still fewer than SURW). This indicates its efficacy in discovering different ordering patterns, but also underscores why it suffers when the bug manifestation requires more fine-grained dependencies between events.

Table 3 reports the Shannon Entropy of both the interleavings and behaviors observed. A larger entropy implies to a

---

[5]We only record the interleaving for two randomly selected clients, since the full interleaving space is so large ($> 10^{14}$) that no algorithm is likely to sample the same interleaving more than once within our testing budget.

Table 3. Shannon Entropy of the distribution of interleavings (top) and program behaviors (bottom) on LightFTP. A larger entropy indicates a more even distribution.

Entropy	SURW	PCT-3	PCT-10	RW
Interleavings	8.57 ± 0.01	3.81 ± 0.05	4.80 ± 0.10	7.17 ± 0.34
Behaviors	5.69 ± 0.64	3.76 ± 0.06	4.84 ± 0.10	4.66 ± 0.17

more even distribution. SURW displays the highest entropy in both categories. This suggests that (1) SURW's guarantee of interleaving uniformity translates well to real-world programs, achieving a higher coverage and evenness; and (2) in practice, this increased uniformity in the interleaving space leads to increased uniformity in program behaviors observed.

6 Related Work

Randomized concurrency testing. Randomized concurrency testing algorithms opportunistically generate feasible interleavings, often in a streaming fashion. SURW belongs to the class of *stateless* algorithms, where each schedule is independently sampled. These algorithms are lightweight, portable, and often provide a probabilitic guarantee of hitting certain classes of bugs under some assumptions. Their design and implementation typically imposes minimal overhead to the program execution. Notably, PCT [6] samples from preemption-bounded interleavings to boost its success probability [8, 34], and has been extended to a multi-core [44] or message-passing [45] setup. RAPOS [51], POS [70] and taPCT [46] incorporates partial order information when making online decisions. We compare with the two most prominent stateless CCT algorithms, namely PCT and POS, in Section 4 and 5.

In contrast to stateless randomized CCT algorithms, *stateful* algorithmic frameworks, such as greybox-fuzzing [65] or reinforcement learning [42], conduct an adaptive, biased random search. For example, RFF [65] enforces partial ordering constraints guided by past executions (i.e. adaptive bias), and instantiates the constraints with a stateless algorithm (i.e. random). While these approaches can be extremely efficient per schedule executed, they tend to be more heavyweight due to the burden of additional information tracking during and between executions. RFF maintains in-memory state machines for each constraint at runtime and QL [42] must maintain a set of all visited states in all previous executions. In contrast, stateless algorithms only track a few integers of per-thread event counts or priorities. Concretely, we benchmarked both RFF and our implementation of SURW on the program in Figure 1, and each scheduling decision of RFF is 15× slower than that of SURW on average (305 ns versus 20 ns). We also remark that these approaches are orthogonal to their stateless counterparts, as stateless algorithms can be incorporated into adaptive search frameworks. RFF, for example, explicitly incorporates the stateless POS algorithm as a subroutine.

Systematic concurrency testing & model checking. To explore the intractably large search space, systematic approaches often focus exclusively on a subspace, prioritize certain execution patterns or employ reduction techniques. CHESS [18, 43] focus on the preemption- or delay-bounded space. Maple [69] employs shared memory access patterns whereas Period [63] adopts thread switching patterns to partition the interleaving space. Model checking tools [9, 27] assume a similar enumerative stance and often involve reduction techniques [1, 2, 4, 20]. Given sufficiently long time, these techniques may formally prove the absence of bugs. In contrast, SURW is highly lightweight and instead provides probabilistic guarantees of hitting any buggy interleaving. Therefore it is more scalable on large programs, and gives incrementally stronger confidence to the system under test as the sample size / time budget increases.

Dynamic analyses. Dynamic concurrency bug detectors including sanitizers like TSan [54], aim to detect concurrency issues from execution traces. Predictive analyses [25, 28, 37–39, 53, 55, 58] enhance their bug finding capability by exploring reorderings of the observed interleaving under a predefined notion of equivalence [19], without rerunning the program. This allows effective detection of potential data races [23, 25, 37, 47, 49], deadlocks [62], atomicity violations [40] and other properties [3, 21]. We note that these techniques are intrinsically incomplete, as traces out of the observed equivalence class are never accounted for. Moreover, traditional concurrency bugs such as data races are often insufficient for bug manifestation [7, 32]. We believe dynamic analyses and SURW are complementary to each other, as they crave for a diverse and representative sample of interleavings and in return identify interesting events for SURW to target.

7 Limitations

Accurate estimation of event counts. SURW leverages per-thread event counts to achieve selective uniformity in the interleaving space. However, we acknowledge that the problem of determining event counts is undecidable in general.

We note that the accuracy of the estimated counts can affect the effectiveness of our approach. Concretely, if a thread T_t executes more events than its count estimate, the interleavings where events on T_t happen early in the execution tend to be under-sampled by SURW. We expect the algorithm to function reasonably well only when the error in estimation is not systematically biased towards certain threads, that is, the *relative ratio* of the estimated counts of different threads is not too far from the actual ratios.

In our evaluation (Section 4), we obtain these counts using a *single profiling run*. Such an estimation can be inaccurate

when there is a large variance across the set of all runs, and more exhaustive but heavyweight methods, such as Knuth's monte carlo estimation [26], can instead be employed to further improve the accuracy. For programs whose control flow depends on the schedule, such as those involving spin locks, the number of events may vary drastically. If these events are of interest, their counts could be arbitrarily far-off from the estimate, and as a result, scheduling decisions can be misled by distorted weights, leading to degraded performance of SURW (see Non-Selective in Section 4.3). In practice, SURW appears to be effective even with our simple event count estimation, thanks to its selectivity. In particular, we note that event counts are highly variable in the RaceBenchData [30, 48] suite, where many variables are accessed only if specific program paths are exercised. Nevertheless, in our empirical evaluation, we observe that the core advantages of SURW ensure that it outperforms other stateless algorithms.

Varied inputs, threads and other non-determinism. Our experiments and case study are conducted on deterministic programs with a *fixed* input, following prior works on concurrency testing [6, 63, 65]. The setup mimics a typical use case of CCT techniques, where a given test case is executed repeatedly to uncover bugs under different interleavings. Consequently, the reported results may not extrapolate beyond this class of programs. For example, SURW is not directly applicable if the number of threads varies, or if the execution depends on other source of non-determinism such as environmental interactions.

8 Discussion

In this work, we construct a concurrency testing algorithm from first principles. We endorse interleaving-uniformity on appropriately selected events as a principled means to achieve effective behavioral exploration. With this insight, we develop a lightweight, yet highly effective algorithm, which we demonstrate on a broad suite of concurrency benchmarks. Looking forwards, we believe this approach can be the basis for future CCT algorithms. We see great potential in combining stateless algorithms such as SURW with their stateful counterparts or dynamic analysis techniques, leading to further innovation in the research area.

9 Acknowledgments

References

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

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

[3] Zhendong Ang and Umang Mathur. 2024. Predictive Monitoring against Pattern Regular Languages. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 2191–2225.

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

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. 72–81.

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

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

[8] Dmitry Chistikov, Rupak Majumdar, and Filip Niksic. 2016. Hitting families of schedules for asynchronous programs. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*. Springer, 157–176.

[9] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. 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*. Springer, 168–176.

[10] ConVul CVE dataset. 2019. https://github.com/mryancai/ConVul. Accessed 20-04-2024.

[11] Lucas Cordeiro and Bernd Fischer. 2011. Verifying multi-threaded software using SMT-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering*. 331–339.

[12] Microsoft Coyote. [n. d.]. GitHub - microsoft/coyote: Coyote is a library and tool for testing concurrent Csharp code and deterministically reproducing bugs. https://github.com/microsoft/coyote. Accessed 20-04-2024.

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

[14] CVE-2016-5195. 2016. Available from NIST, CVE-ID CVE-2016-5195. https://nvd.nist.gov/vuln/detail/cve-2016-5195

[15] Karel De Loof, Hans De Meyer, and Bernard De Baets. 2006. Exploiting the lattice of ideals representation of a poset. *Fundamenta Informaticae* 71, 2-3 (2006), 309–321.

[16] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. 2020. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*. 151–163.

[17] Martin Dyer and Catherine Greenhill. 2000. The complexity of counting graph homomorphisms. *Random Structures & Algorithms* 17, 3-4 (2000), 260–289.

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

[19] Azadeh Farzan and Umang Mathur. 2024. Coarser Equivalences for Causal Concurrency. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 911–941.

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

[21] Chujun Geng, Spyros Blanas, Michael D. Bond, and Yang Wang. 2024. IsoPredict: Dynamic Predictive Analysis for Detecting Unserializable Behaviors in Weakly Isolated Data Store Applications. *Proc. ACM Program. Lang.* 8, PLDI, Article 161 (jun 2024), 25 pages. https://doi.org/10.1145/3656391

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

[23] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

[24] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. 2011. {RADBench}: A Concurrency Bug Benchmark Suite. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar 11)*.

[25] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. *ACM SIGPLAN Notices* 52, 6 (2017), 157–170.

[26] Donald E Knuth. 1975. Estimating the efficiency of backtrack programs. *Mathematics of computation* 29, 129 (1975), 122–136.

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

[28] Rucha Kulkarni, Umang Mathur, and Andreas Pavlogiannis. 2021. Dynamic Data-Race Detection Through the Fine-Grained Lens. In *32nd International Conference on Concurrency Theory (CONCUR 2021)* *(Leibniz International Proceedings in Informatics (LIPIcs), Vol. 203)*, Serge Haddad and Daniele Varacca (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:23. https://doi.org/10.4230/LIPIcs.CONCUR.2021.16

[29] Nancy G Leveson and Clark S Turner. 1993. An investigation of the Therac-25 accidents. *Computer* 26, 7 (1993), 18–41.

[30] Jiashuo Liang, Ming Yuan, Zhanzhao Ding, Siqi Ma, Xinhui Han, and Chao Zhang. 2023. RaceBench: A Triggerable and Observable Concurrency Bug Benchmark. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*. 415–428.

[31] LightFTP. 2015. GitHub - hfiref0x/LightFTP: A small x86-32/x64 FTP Server. https://github.com/hfiref0x/LightFTP. Accessed 01-05-2024.

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

[33] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. 329–339.

[34] Rupak Majumdar and Filip Niksic. 2017. Why is random testing effective for partition tolerance bugs? *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–24.

[35] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60. https://doi.org/10.1214/aoms/1177730491

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

[37] Umang Mathur, Dileep Kini, and Mahesh Viswanathan. 2018. 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 (2018), 1–29.

[38] Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2020. The Complexity of Dynamic Data Race Prediction. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science* (Saarbrücken, Germany) *(LICS '20)*. Association for Computing Machinery, New York, NY, USA, 713–727. https://doi.org/10.1145/3373718.3394783

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

[40] Umang Mathur and Mahesh Viswanathan. 2020. 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*. 183–199.

[41] Rolf H Möhring. 1989. Computationally tractable classes of ordered sets. L. Rival, ed. Algorithms and Order.

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

[43] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 267–280.

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

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

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

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

[48] RaceBenchData. 2023. RaceBenchData: a synthetic concurrency bug dataset. https://github.com/rb130/RaceBenchData. Accessed 20-04-2024.

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

[50] SCTBench. 2016. SCTBench: a set of C/C++ pthread benchmarks for evaluating concurrency testing techniques. https://github.com/mc-imperial/sctbench. Accessed 2024-04-01.

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

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

[53] Traian Florin Șerbănuță, Feng Chen, and Grigore Roșu. 2013. Maximal causal models for sequentially consistent systems. In *Runtime Verification: Third International Conference, RV 2012, Istanbul, Turkey, September 25-28, 2012, Revised Selected Papers 3*. Springer, 136–150.

[54] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, New York, USA) *(WBIA '09)*. Association for Computing Machinery, New York, NY, USA, 62–71. https://doi.org/10.1145/1791194.1791203

[55] Zheng Shi, Umang Mathur, and Andreas Pavlogiannis. 2024. Optimistic Prediction of Synchronization-Reversal Data Races. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 134, 13 pages. https://doi.org/10.1145/3597503.3639099

[56] Shuttle. 2021. GitHub - awslabs/shuttle: a library for testing concurrent Rust code. https://github.com/awslabs/shuttle. Accessed 20-04-2024.

[57] A.Prasad Sistla. 2004. Employing symmetry reductions in model checking. *Computer Languages, Systems & Structures* 30, 3 (2004), 99–137. https://doi.org/10.1016/j.cl.2004.02.002 Analysis and Verification.

[58] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. 2012. Sound Predictive Race Detection in Polynomial Time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 387–400. https://doi.org/10.1145/2103656.2103702

[59] SweynTooth. 2020. SweynTooth: Bluetooth Vulnerabilities Expose Many Devices to Attacks – Securityweek.com. https://www.securityweek.com/sweyntooth-bluetooth-vulnerabilities-expose-many-devices-attacks/.

[60] Topi Talvitie, Kustaa Kangas, Teppo Niinimäki, and Mikko Koivisto. 2018. Counting linear extensions in practice: MCMC versus exponential Monte Carlo. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[61] Paul Thomson, Alastair F Donaldson, and Adam Betts. 2016. Concurrency testing using controlled schedulers: An empirical study. *ACM Transactions on Parallel Computing (TOPC)* 2, 4 (2016), 1–37.

[62] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI, Article 177 (jun 2023), 26 pages. https://doi.org/10.1145/3591291

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

[64] PETER Winkler and G Brifhtwell. 1991. Counting linear extensions. *Order* 8, e (1991), 225–242.

[65] Dylan Wolff, Zheng Shi, Gregory J Duck, Umang Mathur, and Abhik Roychoudhury. 2024. Greybox Fuzzing for Concurrency Testing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 482–498.

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

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

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

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

[70] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 2018. 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*. Springer, 317–335.

A Mean Number of Schedules to 1st Bug

Table 4. Full results on SCTBench and ConVul: the number of schedules to 1st bug exposure (mean ± standard deviation). Results are indicative of the time to exposure (TTE). A smaller number indicates the algorithm is able to expose bugs faster.

Target	SURW	PCT-3	PCT-10	POS	Random Walk	Non Uniform	Non Selective
CS/twostage	**8 ± 4**	**13 ± 14**	**9 ± 10**	15 ± 12	464 ± 581	**8 ± 5**	9 ± 8
CS/twostage_20	**6 ± 3**	159 ± 151	101 ± 92	156 ± 137	–	172 ± 164	24 ± 28
CS/twostage_50	**20 ± 17**	1676 ± 1715	692 ± 392	1637 ± 1385	–	1834 ± 1894	93 ± 58
CS/twostage_100	**454 ± 444**	7466 ± 831*	5726 ± 2591*	6674 ± 2877*	–	–	2992 ± 2589*
CS/reorder_3	**7 ± 7**	185 ± 199	148 ± 191	86 ± 70	–	26 ± 20	24 ± 23
CS/reorder_4	**7 ± 6**	554 ± 643	362 ± 213	533 ± 651	–	53 ± 45	22 ± 27
CS/reorder_5	**10 ± 9**	647 ± 517	1094 ± 1216	2169 ± 2182	–	152 ± 118	27 ± 28
CS/reorder_10	**17 ± 11**	3225 ± 2426*	4462 ± 3266*	–	–	4358 ± 3358*	74 ± 65
CS/reorder_20	**6 ± 4**	3005 ± 2680	3297 ± 2877*	–	–	3783 ± 3345*	**10 ± 9**
CS/reorder_50	**13 ± 12**	3304 ± 1721*	–	–	–	–	75 ± 62
CS/reorder_100	**194 ± 214**	–	–	–	–	–	3785 ± 3062*
CS/stack	5 ± 3	3 ± 2	3 ± 1	**2 ± 1**	176 ± 136	5 ± 4	986 ± 1163*
CS/deadlock01	**2 ± 0**	13 ± 12	7 ± 6	**4 ± 3**	3 ± 3	4 ± 3	3 ± 1
CS/token_ring	**8 ± 6**	**8 ± 6**	**8 ± 8**	9 ± 11	13 ± 13	10 ± 7	9 ± 6
CS/lazy01	**2 ± 0**	5 ± 3	5 ± 3	5 ± 3	21 ± 21	6 ± 3	4 ± 2
CS/bluetooth_driver	70 ± 55	112 ± 73	83 ± 86	**36 ± 29**	215 ± 225	70 ± 55	1494 ± 1474*
CS/account	6 ± 5	4 ± 2	4 ± 3	**4 ± 3**	18 ± 24	7 ± 4	**3 ± 1**
CS/wronglock	7 ± 7	51 ± 50	20 ± 20	10 ± 14	36 ± 42	**4 ± 3**	**4 ± 1**
CS/wronglock_3	9 ± 9	58 ± 46	14 ± 10	11 ± 14	36 ± 42	**5 ± 4**	**5 ± 2**
CB/stringbuffer-jdk1.4	**8 ± 7**	281 ± 261	26 ± 24	23 ± 19	2243 ± 2324*	**8 ± 6**	834 ± 608
Chess/IWSQ	**6 ± 5**	48 ± 36	13 ± 14	14 ± 11	–	40 ± 42	428 ± 742
Chess/IWSQWithState	**6 ± 5**	907 ± 740	204 ± 220	**5 ± 4**	26 ± 28	**7 ± 5**	1367 ± 1581*
Chess/SWSQ	7 ± 6	842 ± 679	199 ± 216	**6 ± 5**	–	15 ± 11	5 ± 3*
Chess/WSQ	**6 ± 4**	48 ± 49	16 ± 14	12 ± 7	–	33 ± 33	13 ± 10
Inspect/bbuf	–	–	–	–	–	–	–
Inspect/boundedBuffer	9 ± 7	20 ± 18	**6 ± 4**	**5 ± 4**	7 ± 7	8 ± 6	7 ± 5
Inspect/qsort_mt	3048 ± 2669*	3004 ± 2787*	4714 ± 2931*	3689 ± 2997*	3442 ± 2806*	2392 ± 1718	**297 ± 209**
RADBench/bug4	**23 ± 24**	1973 ± 2668*	1044 ± 1440	240 ± 185	1688 ± 1590*	**48 ± 51**	438 ± 609
RADBench/bug5	–	–	–	–	–	–	–
RADBench/bug6	**4 ± 3**	42 ± 39	21 ± 21	18 ± 10	48 ± 46	6 ± 4	5 ± 4
SafeStack[†]	**368921 ± 329371***	–	–	–	–	–	–
ConVul/CVE-2013-1792	**15 ± 13**	95 ± 83	50 ± 61	39 ± 33	364 ± 289	**24 ± 25**	141 ± 228
ConVul/CVE-2016-1972	**11 ± 8**	4902 ± 2391*	2712 ± 2704*	34 ± 34	299 ± 256	**12 ± 9**	34 ± 28
ConVul/CVE-2016-1973	**5 ± 3**	10 ± 8	**6 ± 3**	**5 ± 4**	308 ± 333	**5 ± 4**	17 ± 23
ConVul/CVE-2016-7911	8 ± 9	20 ± 18	15 ± 15	11 ± 9	**3 ± 2**	16 ± 12	**6 ± 8**
ConVul/CVE-2016-9806	**3 ± 2**	7 ± 6	**4 ± 3**	7 ± 5	2209 ± 2065	6 ± 5	**4 ± 2**
ConVul/CVE-2017-15265	–	–	–	–	–	–	–
ConVul/CVE-2017-6346	15 ± 10	24 ± 18	20 ± 19	10 ± 9	**3 ± 4**	9 ± 7	**2 ± 0**

Each cell shows the # of schedules (mean ± standard deviation) to the 1st bug across 20 sessions

Results include 1 trial run schedule required for SURW, PCT-3, PCT-10, Non-Uniform and Non-Selective

† SafeStack is run for 10^6 iterations for each algorithm (up from 10^4)

– indicates the bug is not triggered across all sessions

∗ indicates the bug is not triggered in at least 1 session

bold indicates the best result on the target with statistic significance (by log-rank test [36], $p < 0.05$)

B Artifact Appendix

B.1 Abstract

The SURW artifact mainly includes a dynamic library that (1) wraps `pthread` functions to serialize the execution of concurrent programs, and (2) controls thread scheduling with SURW as well as other stateless scheduling algorithms evaluated in the paper. In addition, there is a binary instrumentation tool that invokes the dynamic library at every instruction involving memory operations. These functionalities are implemented using Zig, C/C++ and Python.

This artifact also contains the containerized setup for targets used in the evaluation and case study section, and all raw data and scripts generate all figures and tables presented in the paper. Convenience scripts are provided for reproducing the results for the following subject: (1) An experiment comparing SURW to other algorithms and ablative versions on two widely used benchmark suites in terms of schedules-to-first-bug; (2) An experiment comparing SURW to other algorithms on another benchmark in terms of the total number of bugs found; and (3) A case study comparing SURW to other algorithms on a real-world protocol server in terms of coverage and uniformity.

The recommended software dependencies are recent version of Ubuntu with Docker, Python3 and bash. The experiment requires disabled address space layout randomization (ASLR). A modern 8 core machine should be able to finish the experiments within 3 days.

B.2 Artifact check-list (meta-information)

- **Algorithm:** Algorithm 2
- **Program:** SCTBench [50] and ConVul [10] curated by [63], RaceBenchData [48], LightFTP [31]
- **Transformations:** binary rewriting tool E9Patch [16]
- **Run-time environment:** Linux (Docker)
- **Metrics:** Schedules to 1st bug, interleaving and behavioral coverage, Shannon entropy
- **Output:** Json files, LaTeX files, graphs
- **Experiments:** Bash and Python scripts
- **How much disk space required (approximately)?:** <30 GB for all Docker images combined; <8 GB for the largest image alone if the each image is built separately.
- **How much time is needed to prepare workflow (approximately)?:** 1-3 hours to build Docker images
- **How much time is needed to complete experiments (approximately)?:** Around 3 days on a typical 8 core CPU
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** Yes, via https://doi.org/10.6084/m9.figshare.27627123

B.3 Description

B.3.1 How to access. Via the DOI link above.

B.3.2 Hardware dependencies. An x86 CPU and at least 8 GB of RAM. More cores will make the experiments faster with parallelism, but are not necessary.

B.3.3 Software dependencies. A recent version of Ubuntu with admin privileges. Bash, Docker, plus Python3 and a few package dependencies.

B.3.4 Data sets. All raw data used in the paper is included in the artifact under the `stats` directory.

B.4 Installation

Note that installation steps and scripts assume a recent version of Ubuntu (e.g., 20.04). These steps can be adapted to other Linux distros with some modifications of the bash scripts and commands below.

1. Install Docker and Python3:

```
sudo snap install docker
sudo groupadd docker
sudo usermod -aG docker $USER
sudo apt install -y python3
```

2. Clone E9Patch [16]:

```
git clone git@github.com:GJDuck/e9patch.git
cd e9patch; git checkout b4e7175; cd ..
```

3. Install Python dependencies:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
```

4. Run `./build_all.sh` to build all Docker containers.

5. Run the following command to disable address space layout randomization (ASLR):

```
echo 0 | \
sudo tee /proc/sys/kernel/randomize_va_space
```

Note: your system will be more vulnerable to exploits while ASLR is disabled. ASLR will be reinstated on reboot.

B.5 Experiment workflow

The experiments can be run with convenience scripts on the host OS, which invoke multiple Docker containers in isolation and save the collected results to the `stats` directory. When started, each Docker container runs the target up to a fixed number of iterations. The compiled version of the dynamic library and target programs are constructed when building the Docker images.

B.6 Evaluation and expected results

All experiments should produce comparable, if not identical, results as reported in the paper.

SCTBench and ConVul. The experiment can be rerun with

```
python3 scripts/eval/run_period.py
```

Table 1 and Table 4 (in Appendix A) can be generated from saved (or new) results using

```
python3 scripts/analyze/eval_period.py
```

RaceBenchData. The experiment can be rerun with

```
python3 scripts/eval/run_racebench.py
```
Table 2 can be generated from saved (or new) results using
```
python3 scripts/analyze/eval_racebench.py
```

LightFTP. The case study can be rerun with
```
python3 scripts/eval/run_lftp.py
```
Table 3 and two sub-figures of Figure 5 can be generated from saved (or new) results using
```
python3 scripts/analyze/plot_lftp_a.py
python3 scripts/analyze/plot_lftp_b.py
```

B.7   Experiment customization

The dynamic library supports a wide range of customization options. The artifact contains configuration files for many stateless CCT algorithms, some of which are not discussed in the paper. For example, it supports different implementations of `pickFrom()` in Algorithm 2. Additionally, it supports selecting different events as interesting for SURW, such as accesses to specified memory addresses, locks or `sched_yield()` calls. Interested readers may refer to the README.md file in the artifact for more detailed explanations. However, we note that the artifact is a research prototype that is still under development, and that some customization options may not be stable.

B.8   Notes

Please run all convenience scripts and commands from the documentation at the base directory of the artifact. The experiments are run with fixed random seeds, which in principle should produce consistent results. However, it is possible to have slight variations in the results due to different system environments or workloads.

B.9   Methodology

Submission, reviewing and badging methodology:
- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html