Precise Detection of Kernel Data Races with Probabilistic Lockset Analysis

Gabriel Ryan, Abhishek Shah, Dongdong She, Suman Jana
Columbia University

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179366

***Abstract*—Finding data races is critical for ensuring security in modern kernel development. However, finding data races in the kernel is challenging because it requires jointly searching over possible combinations of system calls and concurrent execution schedules. Kernel race testing systems typically perform this search by executing groups of fuzzer seeds from a corpus and applying a combination of schedule fuzzing and dynamic race prediction on traces. However, predicting which combinations of seeds can expose races in the kernel is difficult as fuzzer seeds will usually follow different execution paths when executed concurrently due to inter-thread communications and synchronization.**

To address this challenge, we introduce a new analysis for kernel race prediction, Probabilistic Lockset Analysis (PLA) that addresses the challenges posed by race prediction for the kernel. PLA leverages the observation that system calls *almost always* perform certain memory accesses to shared memory to perform their function. PLA uses randomized concurrent trace sampling to identify memory accesses that are performed consistently and estimates the probability of races between them subject to kernel lock synchronization. By prioritizing high probability races, PLA is able to make accurate predictions.

We evaluate PLA against comparable kernel race testing methods and show it finds races at a $3\times$ higher rate over 24 hours. We use PLA to find 183 races in linux kernel v5.18-rc5, including 102 harmful races. PLA is able to find races that have severe security impact in heavily tested core kernel modules, including use-after-free in memory management, OOB write in network cryptography, and leaking kernel heap memory information. Some of these vulnerabilities have been overlooking by existing systems for years: one of the races found by PLA involving an OOB write has been present in the kernel since 2013 (version v3.14-rc1) and has been designated a high severity CVE.

1. Introduction

Data races are a source of serious security vulnerabilities in the OS kernels–many recent data-race-based exploits resulted in privilege escalation [2], denial of service [8], and leaking protected memory [4, 6]. Recent work has demonstrated that even races that appear unexploitable might be deterministically triggered by an attacker [30]. Moreover, even when data races do not immediately result in security vulnerabilities, they cause severe bugs that lead to memory corruption and undefined behavior [1, 31].

Given their security and reliability implications, testing for and identifying data races is critical for modern kernel development. However, testing for data races is challenging both in theory and practice: finding data races is NP-hard [38] because data races only occur under specific concurrent execution schedules, which are exponential in the number of executed instructions. As a result, in practice, many races are not identified until they cause a crash or security vulnerability in released code [27].

In general, there are two widely used approaches to search for races in arbitrary concurrent programs: *schedule exploration* searches by executing many different schedules [18, 37, 51], while *dynamic race prediction* reasons about possible reschedulings of memory accesses subject to synchronization to trigger races based on a single concurrent execution trace [20, 34, 43, 48]. However, these approaches reason exclusively about rescheduling the thread execution order. When testing the kernel, the memory accesses and synchronization operations are determined by which system calls are executed. Identifying a race then requires finding the correct combination of both system calls and execution schedule under which the race occurs.

Kernel Data Race Detection. Kernel race testing systems therefore apply schedule exploration and dynamic race prediction to the kernel by using a two step process: they first select a combination of fuzzer seeds composed of systems calls from a fuzzer corpus, guided by either alias analysis [22, 25] or a coverage metric for concurrent executions [26, 50], and then test the combined seeds with schedule exploration and dynamic race prediction.

However, predicting which memory accesses can race between different combinations of seeds is challenging: alias analysis of shared memory accesses suffers from high false positive rates and does not account for kernel synchronization, while concurrent coverage metrics only provide indirect guidance for selecting seed combinations to test. Moreover, due to inter-thread communications seeds may follow different execution paths and perform different memory accesses when executed concurrently, making any prediction based on a previous execution traces even more error prone. As a result the vast majority of concurrent tests are wasted because races either do not occur or are allowed based on kernel concurrency semantics.

Our Approach. In this paper, we introduce a new approach to predict races between combinations of seeds in a corpus that addresses each of these challenges in kernel race prediction: we only predict races where kernel synchronization

DOI 10.1109/SP46215.2023.00144

is violated and racing is not allowed, and we account for changing execution paths when seeds are executed together, even if we have not observed those particular seeds executing together before. This allows us to predict races accurately and efficiently test a corpus for races, with provable bounds on the false positive rate under mild uniformity assumptions.

Our approach is based on three observations about kernel system call memory access behavior: (i) *Stable memory accesses.* While most memory accesses performed a system call change on each execution, a small subset of memory accesses form a *stable set*, which the system call must make to perform its intended function (e.g., a file read must access the relevant file inode), regardless of which other system calls are executing concurrently. Memory accesses in the stable set will *almost always* occur when the system call is executed (see Section 3.2 for a more precise definition). (ii) Memory accesses in the stable set must be guarded by mutual exclusion synchronization (locks) or allowed to race, since multiple system calls can perform them concurrently. (iii) *Sparse lock interactions.* Kernel concurrency design favors using a small number of common locks for any shared memory, so the number of distinct locksets for accesses to a common address are almost always small, even when the number of accesses is large.

Probabilistic Lockset Analysis. Based on these observations, we introduce Probabilistic Lockset Analysis (PLA): a new analysis for kernel race prediction that identifies memory accesses in the stable set and performs synchronization aware race prediction on them. PLA works by estimating the probability that two seeds can execute racing memory accesses concurrently subject to lock synchronization. It estimates probabilities of memory accesses with regard to other concurrent programs, execution schedules, and variation in the execution context. Therefore, races involving memory accesses that are unlikely to happen concurrently will have low probability, while races involving memory accesses in stable set will have high probability, and these predictions can always be refined to higher precision by taking more samples.

Unlike lockset analysis defined in the dynamic race prediction literature [52], which relies on happens-before relations derived from inter-thread communications to make precise race predictions, PLA is able to make precise race predictions by estimating the probabilities of seeds performing concurrent memory accesses. This allows PLA to make accurate race predictions based on independently collected execution traces sampled from each seed in a corpus, instead of testing each seed combination and execution schedule individually, which would require a potentially exponential number of executions. To scale to large corpuses of fuzzer seeds, PLA's design exploits the intrinsic sparsity of inter-thread communications and locksets in the kernel: on average, less than 1% of memory accesses are performed with high probability, and the vast majority of these accesses only share a small number of distinct locksets (< 100, see Section 5.6). This allows PLA to check each pair of unique locksets on each shared memory address for locking violations, while still scaling linearly in the number of memory accesses processed. In practice, PLA easily scales to analyzing billions of memory accesses for races.

PLA works in three steps: First, PLA executes each seed in the corpus concurrently with other randomly selected seeds and schedules to estimates the probability of the seed performing memory accesses with specific locksets. Next, PLA identifies lockset violations on shared memory accesses by checking for non intersecting locksets. Finally, PLA estimates the joint probability of memory accesses with locking violations occurring concurrently. For each prediction, PLA generates a hypothesized concurrent execution schedule that causes the two memory accesses to race. Each prediction can then be efficiently checked by executing the relevant seeds according to the hypothesized schedule.

Result Summary. We use PLA to find 183 distinct races in linux kernel v5.18-rc5, of which 102 are harmful, and show in a comparative 24 hour evaluation that it finds races at a rate $3\times$ greater than other comparable kernel concurrency testing systems. PLA is effective at identifying hard-to-find races in core kernel modules that have severe security impact: including use-after-free in memory management, OOB write in network cryptography, and leaking kernel heap memory. One of these races found by PLA that causes an OOB write has been present in the kernel since 2013 (version v3.14-rc1) and has been designated a high severity CVE [9].

In summary, this paper makes the following contributions:

1) We introduce Probabilistic Lockset Analysis (PLA), a new race prediction method for the kernel that leverages probabilistic reasoning to predict races from corpuses of fuzzer seeds. PLA is fast and accurate, easily scaling to billions memory accesses. We provide an open source release of PLA[1].
2) We compare PLA against other kernel race testing systems on a benchmark seed corpus and show it finds more than $3\times$ as many races in a 24 hour period.
3) We use PLA to find 183 races in the kernel, including 102 harmful races with security implications, one of which in the kernel networking cryptography has remained undetected for nearly 10 years and has been designated a high severity CVE.
4) Finally, we derive rigorous error bounds on false postive rates for PLA's probabilistic race predictions, and show empirically PLA's trace sampling is able to predict memory accesses with high accuracy.

2. Background

In this section we first formulate the problem of race prediction on the kernel and discuss its challenges. We then describe current approaches to race prediction used on the kernel and their limitations.

1. www.github.com/gryan11/PLA

```
static int global_handle = 0;

void nf_newtable(struct net *net)
{
  mutex_lock(net->mutex);
  table = nft_lookup(net);
  ...

  table->h = ++global_handle;
  ...

  mutex_unlock(net->mutex);
}
```

(a) Function with racing global variable update.

```
thread 1:
------------------------------
sock_sendmsg(sock=sock1)
  ...
  nf_newtable(net=net1)
    mutex_lock(net1->mutex)
    table1 = nft_lookup(net1)
    table1->h = ++global_handle
```

```
thread 2:
-------------------------
    sock_sendmsg(sock=sock2)
  ...
  nf_newtable(net=net2)
    mutex_lock(net2->mutex)
    table2 = nft_lookup(net2)
    table2->h = ++global_handle
```

(b) Execution schedule with racing memory accesses on `handle`.

Figure 1: Simplified example of race found by PLA in `net/netfilters/`. The race occurs on the global variable `global_handle` shown in 1a, which can be concurrently modified by multiple threads when passed different `net` structs (which each have different per-thread `net->mutex`). 1b shows an execution schedule and the associated pair of unsynchronized memory accesses used to identify the race.

```
r0 = socket$nl_netfilter(0x10, 0x3, 0xc)
sendmsg$NFT_BATCH(r0, &(0x7f0180)=
    {0x0, 0x0, @NFT_MSG_NEWTABLE={0x20, 0x0,
    0xa, 0x801}})
```

Figure 2: Simplified kernel fuzzer seed used to trigger the race shown in Figure 1. The seed opens a socket and then sends a message that executes the `nf_newtable` function.

```
  thread 1:                 thread 2:
  ----------------------    ----------------------
1
2 msg1->data = data1;
3 WRITE_ONCE(msg,msg1);
4 // ----------- happens before ---------------
5                           msg2 = READ_ONCE(msg);
6                           data2 = msg2->data;
```

Figure 3: Lockless message passing pattern commonly used in kernel. Thread 1 and thread 2 can both access the same aliased `data` field, but the pointer exchange from line 3 to line 5 imposes a happens-before relation between the two memory accesses on lines 2 and 6. Therefore, there is no execution schedule where the accesses can race. Alias analysis will generate a false positive race prediction on these accesses, but a dynamic race predictor using happens-before analysis will correctly identify there is no race.

2.1. Problem Definition

We use the standard definition of a data race: two memory accesses to the same address can be scheduled on different threads to happen concurrently, and at least one of the accesses is a write [16]. Figure 1 shows the unsynchronized access pair and schedule for a race found by PLA in `net/netfilters`. The race occurs on a global variable `handle` highlighted in 1a that is guarded by mutex in a `net` struct. The memory access pair and their respective system calls are shown in 1b, along with an execution schedule that will cause the two memory accesses two race. Since the function can be called concurrently with two different `net` structs (and therefore, two different mutexes), the global handle variable can be concurrently updated by two different threads, causing the netfilter table handles to be inconsistent (e.g., two table may receive the same handle value).

Racing Schedules. In order for a race to occur, there must be a execution schedule that performs a pair of accesses to the same memory address concurrently – lack of synchronization between accesses is a necessary but not sufficient condition for a data race. Even when there is no explicit synchronization between two shared memory accesses, inter-thread communications can make data races infeasible. This can cause kernel race prediction approaches that do not explicitly reason about schedules (e.g., by only checking for aliased memory accesses) to make large numbers of false positive race predictions.

For example, the two methods shown in Figure 3 demonstrate a common lockless message passing pattern in the kernel (memory barriers have been omitted for clarity). In lockless message passing, a struct (in this case `msg1`) is first populated with relevant data and then a pointer to the struct sent to another thread via a shared pointer (in this case `msg`). Although thread 1 and thread 2 both access the same aliased data field without synchronization, thread 2 cannot access the `data` field unless thread 1 has already written the struct address to the shared pointer `msg`. This makes any execution schedule that attempts to perform the thread 1 data write and thread 2 data read concurrently *infeasible*.

In contrast, the accesses to shared pointer `msg` can race in Figure 3, but this is expected and allowed during kernel message passing and the `READ_ONCE` and `WRITE_ONCE` macros indicate the two accesses are allowed to race.

Kernel Fuzzer Seeds. In practice, kernel concurrency testing systems typically operate on corpuses of kernel fuzzer seeds, each of which is composed of a sequence of system calls which operate on hardcoded parameter values and return values or pointers passed to previous system calls. Figure 2 shows an example syzkaller fuzzer seed that triggers the race shown in Figure 1. Kernel concurreny testing systems generate corpuses of kernel fuzzer seeds by either running a single threaded fuzzer and maximizing branch

coverage [22, 25], or using concurrency specific coverage metrics [26, 50].

Problem Formulation. Based on the common usage of kernel fuzzer seeds in concurrency testing, we define the whole corpus race testing problem as following: given a corpus of kernel fuzzer seeds, identify data races in the corpus, where each data race comprises (1) two unsynchronized accesses to the same memory address, (2) two (or more) fuzzer seeds that perform the predicted accesses when executed concurrently, and (3) an execution schedule that executes both accesses concurrently.

Challenges. Kernel race testing has two properties that make it extremely challenging:

1) *Exponential search space.* For any given corpus size and bounded execution length, there is an exponential number of possible seed combinations and execution schedules that can potentially expose races. For k seeds executing n instructions, each instruction in the schedule is selected from one of the k seeds, so there are $O(k^n)$ possible schedules. Moreover, for a corpus $\mathbb{P}$, there are $\binom{|\mathbb{P}|}{k}$ possible seed combinations.
2) *Unpredictable execution behavior.* Kernel seeds will follow different execution paths and perform different memory accesses on each execution due to changing background processes and environment, even when executed from a fixed image, so any analysis based on independently observed execution traces will be highly error prone.

2.2. Kernel Race Prediction Approaches

Recent kernel concurrency testing systems use two types of analysis to identify races, however, both approaches miss many kernel races due to the two challenges in kernel race prediction:

1) *Dynamic race prediction* makes predictions based on observed concurrent execution traces. It is precise (no false positives), but cannot efficiently search the exponential space of seed combinations and execution schedules for races.
2) *Alias analysis* efficiently makes predictions between independently observed traces that contain accesses to common memory addresses, but makes overwhelming numbers of false positive predictions due to the unpredictable kernel execution behavior and not checking if aliases are synchronized (e.g., covered by a common lock).

We discuss the tradeoffs made by these approaches here and provide precise definitions in Appendices A and B.

Dynamic Race Prediction. Dynamic race prediction used in kernel testing typically combines *happens-before analysis*, which reasons about ordering dependencies such as the message passing shown in Figure 3 to avoid false positive predictions, with *lockset analysis*, which identifies locking violations such as the non-overlapping mutexes bug shown in Figure 1. When used together, hybrid happens-before lockset analysis can make precise race predictions (no false positives), but can only reason about one concurrent trace at a time, because the happens-before ordering used in the analysis is derived from the observed trace. In practice this means testing systems based on dynamic race prediction will miss many races because they must search directly over the exponential space of seed combinations and execution schedules (See Section 5.2).

Alias Analysis. In contrast, alias analysis does not directly search over seed combinations and schedules, but independently checks for accesses to the same memory address either statically [25] or dynamically [22]. This avoids the scalability issues of dynamic race prediction, but causes extremely high false positive rates. These false positives occur because either the aliases are spurious (two observed accesses appear to access the same memory address but cannot do so concurrently, see Figure 3), or the aliases are synchronized (e.g., mutually locked). Therefore, testing systems using alias analysis will miss many races because they will waste most of their test executions on false positive race predictions (see Section 5.2).

3. Methodology

In this paper, we introduce Probabilistic Lockset Analysis (PLA), a new approach to kernel race prediction for the kernel that incorporates the advantages of both dynamic race prediction and alias analysis while avoiding their shortcomings. Like alias analysis, PLA makes predictions across independently observed traces, allowing it to scale linearly in the number of corpus seeds and memory accesses. However, like dynamic race prediction, PLA makes accurate predictions by taking kernel synchronization and schedule dependencies into account when making predictions.

3.1. PLA Overview

PLA's design is based on three observations about the memory accesses performed by system calls. (1) System calls must make certain memory accesses to shared memory to perform their intended function. These memory accesses form a *stable set* that will be performed with high probability, regardless of any concurrently executing syscalls and how they are scheduled. (2) Memory accesses in the stable set must guarded by mutual exclusion (i.e., locks) or allowed to race, since multiple system calls can perform them concurrently. (3) Locking interactions in the kernel are sparse, so the number of unique locksets for a common kernel memory address will almost always be small (we confirm this empirically in Section 5.6).

PLA leverages these three observations to perform precise race predictions between independently observed traces. Since memory accesses in the stable set are performed with high probability for any concurrent schedule, it can make accurate race predictions between stable set accesses without first executing the seeds together to apply happens-before analysis. Since memory accesses in the stable set must be guarded with mutual exclusion, PLA is able to check synchronization based on commonly held locks. Finally,

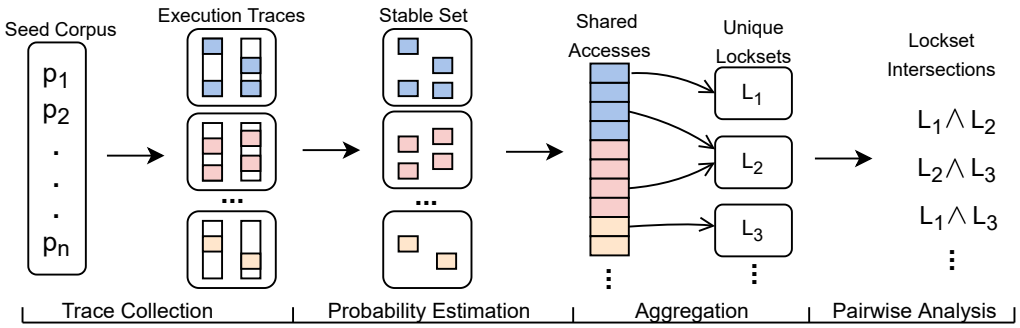


Figure 4: PLA's workflow. PLA collects traces independently from each seed in the corpus to identify its stable set of memory accesses. It then groups all memory accesses first by memory address and then by unique locksets, and performs pairwise intersections on the aggregated locksets. This procedure is linear in both the number of corpus seeds and individual memory accesses in the traces, allowing it to scale to large corpuses and traces.

PLA exploits the sparsity of kernel locking by performing precise pairwise lockset analysis on the distinct locksets associated with each memory address.

PLA Workflow. Figure 4 provides a high level summary of PLA's workflow. PLA first executes each seed in the corpus concurrently with other randomly selected seeds and schedules. It then identifies high probability memory accesses (the stable set) in each set of seed traces and aggregates them based on common memory addresses. Each set of stable memory accesses is then grouped by their locksets, and potentially racing access pairs are identified with pairwise lockset analysis and prioritized based on their joint probability. For each race prediction, PLA generates a hypothesis execution schedule that can be executed to check for feasibility. We formally describe PLA's analysis below.

PLA vs. Lockset Analysis. Lockset analysis can suffer from very high false positive rates, so it is usually applied as a hybrid race predictor with happens-before analysis. However, happens-before analysis must observe a concurrent execution trace between two threads to derive happens-before constraints, so it requires at least $O(n^2)$ executions to test each pair of seeds in a size n corpus (Section 2.2). In contrast, PLA is able to make precise race predictions between two threads *without* observing their communications by representing each memory access with a random variable and estimating the probability that two memory accesses can be performed concurrently. Since each random variable is estimated by independently sampling traces from each input, this only requires $O(n)$ traces for n seeds. Figure 5 illustrates the difference between PLA and hybrid race prediction when run a corpus of fuzzer seeds.

Hybrid Dynamic Race Prediction on Corpus:

```
// Check O(n²) traces of all seed pairs:
for p1 in ℙ do
  for p2 in ℙ do
    for s in schedules over p1, p2 do
      // Check accesses in each individual trace:
      T = trace(p1, p2, s)
      for address m in T do
        for αi, αj in accesses to m in T do
          if locks(αi) ∩ locks(αj) = ∅ and
             not HB(αi, αj, T) then
            predict race(αi, αj)
```

Probabilistic Race Prediction on Corpus:

```
// Sample O(n) traces of each seed:
for p1 in ℙ do
  for sample p2 from ℙ do
    for sample s from schedules over p1, p2 do
      add trace(p1, p2, s) to 𝕋
// Check random variables estimated from sampled traces:
for address m in 𝕋 do
  for Aαi, Aαj in accesses to m in 𝕋 do
    if locks(Aαi) ∩ locks(Aαj) = ∅ and
       Pr[Aαi ∩ Aαj] > β then
      predict race(αi, αj)
```

Figure 5: High level comparison of PLA to hybrid race prediction running on a corpus $\mathbb{P}$ of n seeds. The happens-before check on two accesses $\text{HB}(\alpha_i, \alpha_j, T)$ used in hybrid race prediction requires a trace T, so each pair of seeds must be checked individually, requiring $O(n^2)$ traces to check combinations of 2 threads. In contrast, PLA estimates the probability of races based on random indicator variables for each access A_α, which can be independently estimated for each seed from $O(n)$ sampled traces $\mathbb{T}$. See Section 3.2 for precise definitions of traces, α, and A_α.

3.2. PLA Definitions and Error Bounds

Tuple Notation. We make extensive use of tuples and denote named elements of a tuple with dot notation. For a tuple $x = (a, b)$, we refer to element a as $x.a$.

Fuzzer Seeds and Corpus. We refer to a kernel fuzzer seed as p where each seed is drawn from a corpus $\mathbb{P}$. PLA's current implementation uses two seeds at a time, so to simplify notation we assume PLA is operating on two seeds $\{p_1, p_2\}$ in this section. However, PLA can be used with any number of concurrent threads.

Access Locksets. When performing probabilistic lockset analysis, we operate on instruction, address, lockset tuples called *access-locksets*, denoted α. Each access-lockset is uniquely identified by its executing seed p, instruction pointer ip, memory address m, operation type $op \in \{r, w\}$, and the set of held locks when they executed:

$$\alpha = (p, ip, m, op, lockset)$$

Two seeds, p_1 and p_2, can be executed concurrently according to a schedule s to obtain the set of access-locksets that appear in its execution trace.

$$\text{trace}(p_1, p_2, s) = \{\alpha_1, \alpha_2, ...\}$$

We describe the procedure for constructing access-locksets from traces in Section 3.3.1. For the remainder of the section, we refer to access-locksets simply as accesses or memory accesses.

Probabilistic Access-Locksets. The memory accesses that are performed by a given seed p will vary depending on the

concurrent seed, execution schedule s, and any changes to the kernel environment (e.g., background processes).

Therefore, we represent the occurrence of α in an execution trace of p with indicator random variable A_α:

$$A_\alpha = \begin{cases} 1 & \text{if } \alpha \in \text{trace}(p_1, p_2, s) \\ 0 & \text{otherwise} \end{cases}$$

We can estimate the likelihood of a seed performing a particular access-lockset (i.e., $A_\alpha = 1$) by executing it concurrently with other seeds and schedules. This can be thought of as drawing independent samples of the random variable A_α, where each execution produces a sample $A_\alpha^{(i)}$. When sampling we assume each random variable is independent of the other variables. This allows us to estimate probabilities efficiently:

$$\mathbf{P}\left[A_\alpha = 1 \mid p_1 = p\right] \approx \frac{1}{N}\sum_{i}^{N} A^{(i)}$$

where N is the number of samples, and $p_1 = p$ denotes that we fix the first seed in $\text{trace}(p_1, p_2, s)$ to p, and p_2 and s are uniformly sampled from a corpus and set of schedules respectively.

Stable Set. We define the *stable set* of a seed p with regard to a stability threshold β as the set of accesses $\mathcal{S}$ where:

$$\mathcal{S} = \left\{\alpha : \mathbf{P}[A_\alpha = 1 | p_1 = p] > \beta\right\}$$

Making predictions on the stable set drastically reduces the cost of PLA's analysis (since only a small proportion of accesses are stable, see evaluation in Section 5.5) and makes it more accurate, since any pair of stable accesses are likely to have a feasible concurrent schedule (see evaluation in Section 5.3).

Probabilistic Races. Given two accesses α_1 and α_2 to a common address, we consider two memory accesses as probabilistically racing with stability threshold β if the following condition is met:

$$\alpha_1.lockset \cap \alpha_2.lockset = \varnothing \ \wedge \ \alpha_1, \alpha_2 \in \mathcal{S} \qquad (1)$$

Witness Schedule. Given two accesses α_1 and α_2 that satisfy Eq. 1 and their respective seeds p_1 and p_2, PLA generates a witness schedule s that will execute the two accesses concurrently with high probability. Since α_1 and α_2 are estimated to be executed with high probability for *any* schedule, PLA generates a schedule in which α_1 and α_2 execute concurrently by ordering instructions from p_1 up to α_1 first, followed by instructions from p_2 up to α_2.

Race Predictions. A full race prediction is composed of two racing accesses, their respective seeds, and the witness schedule to trigger the race:

$$\text{PLA-race-prediction} := (\alpha_1, \alpha_2, p_1, p_2, s) \qquad (2)$$

PLA's predictions can be quickly checked by executing p_1 and p_2 according to the witness schedule. If the schedule is feasible, then the prediction is confirmed as a race and the witness schedule can be used for reproduction and future testing.

Error Bounds. We derive the following error bounds on false positives and false negatives based on a threshold β. The bound on false positives is stated as follows:

Theorem 1. *For a threshold β, relative error bound $0 < \delta < 1$, and two access locksets α_1 and α_2 with non-intersecting locksets and random variables A_{α_1} and A_{α_2} sampled N times such that $\alpha_1, \alpha_2, \beta$ satisfy Eq. 1 and $\mathbf{P}\left[A_{\alpha_1} = 1 \cap A_{\alpha_2} = 1\right] \geq \beta$, then with probability $e^{-\delta^2 N\beta/(2-\delta)}$, the probability of a false positive is bounded by:*

$$\mathbf{P}[A_{\alpha_1} = 0 \cup A_{\alpha_2} = 0] < 1 - \beta(1+\delta)$$

See Appendix C for proof.

The bound on false negatives is similarly constructed:

Theorem 2. *For a threshold β, relative error bound $0 < \delta < 1$, and two access locksets α_1 and α_2 with non-intersecting locksets and random variables A_{α_1} and A_{α_2} sampled N times such that α_1 and α_2 do not satisfy equation 1 and $\mathbf{P}\left[A_{\alpha_1} = 1 \cap A_{\alpha_2} = 1\right] < \beta$, then with probability $e^{-\delta^2 N\beta/2}$, the probability of a false negative is bounded by:*

$$\mathbf{P}[A_{\alpha_1} = 1 \cap A_{\alpha_2} = 1] < \beta(1-\delta)$$

See Appendix D for proof.

In both cases, the probability of an error decreases exponentially with the number of samples collected. This means that probabilistic locksets can arrive at precise estimates of the probability of races with relatively few samples, and we find that in practice only four samples are needed to achieve accurate predictions of access locksets (Section 5.5).

3.3. PLA: Algorithm Design

We perform PLA in three stages: 1) access lockset probability estimation, 2) probabilistic lockset analysis, 3) coverage guided race checking.

Design Optimizations. We apply three optimizations in the design of PLA that allow it to scale to large corpuses: (1) We apply the probabilistic race prediction threshold β to access locksets immediately after sampling each input before further analysis. (2) We perform an initial coarse grained linear lockset analysis pass before applying pairwise lockset analysis. (3) We select race predictions to test that maximize the overall coverage of tested instructions while minimizing the number of required tests. We evaluate the impact of these optimizations in Section 5.4 and show that ablating any one of them prevents PLA from scaling effectively.

3.3.1. Probability Estimation. We use the following procedure to estimate access lockset probabilities for each seed in the corpus. First, we collect a set of concurrent execution traces for each seed p executed with randomly selected

concurrent seed p'. For each p', we concurrently execute and trace p and p' with two schedules, one where p starts first, and on where p' starts first. For each sample we count if an access lockset is present in the trace but do not count the number of occurrences, which would bias the probability estimate towards frequently executed memory accesses. Algorithm 1 describes the sample collection procedure. This sampling procedure is not strictly uniform over the space of possible schedules, but in practice still precisely estimates stable access locksets (see evaluation in Section 5.5).

Algorithm 1 Access Lockset Construction.

Input: p_1 ← Seed 1; p_2 ← Seed 2; s ← Schedule

$sample\_accesses = \{\}$
$held\_locks = \{\}$
for $t \in \text{trace}(p_1, p_2, s)$ **do**
  if $is\_lock\_acquire(t)$ **then**
    $held\_locks = held\_locks \cup \{t.lock\_addr\}$
  if $is\_lock\_release(t)$ **then**
    $held\_locks = held\_locks \backslash t.lock\_addr$
  if $is\_memory\_access(t)$ **then**
    $\alpha = (t.ip, t.m, t.op, held\_locks)$
    $sample\_accesses = sample\_accesses \cup \alpha$
return $sample\_accesses$

For each access lockset α, we estimate the probability $\mathbf{P}\left[A_\alpha = 1 \mid p_1 = p\right]$ based on the execution trace access sets. We then filter the access locksets based on the race prediction threshold β. Algorithm 2 describes the overall procedure for probability estimation.

Algorithm 2 Access Lockset Probability Estimation.

Input: p ← Seed; $\mathbb{P}$ ← Seed Corpus; N ← Seed Sample Count; β ← Race Prediction Threshold

$\mathbb{M}_p = \text{hashmap(default} = \varnothing)$
$access\_counts = \text{hashmap(default} = 0)$
for $i \in \{1..N/2\}$ **do**
  $p' = \text{choose\_random}(\mathbb{P})$
  for $s \in \{p\_first, p'\_first\}$ **do**
    $sample\_accesses = \text{sample}(p, p', s)$ ▷ see Algorithm 1
    for $\alpha \in sample\_accesses$ **do**
      $access\_counts[\alpha]$ += 1
for $\alpha \in access\_counts$ **do**
  if $access\_counts[\alpha]/(N) \geq \beta$ **then**
    $\mathbb{M}_p[\alpha.m] = \mathbb{M}_p[\alpha.m] \cup \{\alpha\}$
return $\mathbb{M}_p$

3.3.2. Whole Corpus PLA. Algorithm 3 describes the overall procedure for PLA. First, probability estimation is performed on the seeds in the test corpus and high probability access locksets are aggregated by memory address in the map $\mathbb{M}$. Then, PLA is applied to the access locksets for each memory address in $\mathbb{M}$.

The lockset analysis is applied in two stages. First, a single linear pass computes the intersection of all locksets associated with a given memory address. If the intersection in empty, indicating the possibility of a race, a precise pairwise check of each unique lockset associated with the memory address determines which pairs of locksets have null intersections. If a pair of locksets have a null intersection, the set of memory accesses associated with each lockset is checked for possible races.

Algorithm 3 Whole Corpus PLA

Input: $\mathbb{P}$ ← Seed Corpus; N ← Seed Sample Count; β ← Race Prediction Threshold

$\mathbb{M} = \text{hashmap(default} = \varnothing)$
for $p \in \mathbb{P}$ **do**
  $\mathbb{M}_p = \text{probability\_estimation}(p, \mathbb{P}, N, \beta)$ ▷ see Algorithm 2
  for $m \in \mathbb{M}_p$ **do**
    $\mathbb{M}[m] = \mathbb{M}[m] \cup \mathbb{M}_p[m]$

$C_{all} = \{\}$
$R_{all} = \{\}$
for $m \in \mathbb{M}$ **do**
  if $\varnothing \neq \bigcap_{\alpha \in \mathbb{M}[m]} \alpha.lockset$ **then**
    Continue
  $C_m = \left(\bigcup_{\alpha \in \mathbb{M}[m]} \alpha.ip\right) \backslash C_{all}$
  if $C_m == \varnothing$ **then**
    Continue
  $\mathbb{L} = \text{hashmap(default} = \varnothing)$
  for $\alpha \in \mathbb{M}[m]$ **do**
    $\mathbb{L}[\alpha.lockset] = \mathbb{L}[\alpha.lockset] \cup \{\alpha\}$
  for each unique $lockset_1, lockset_2 \in \mathbb{L}$ **do**
    if $lockset_1 \cap lockset_2 = \varnothing$ **then**
      $accs_1, accs_2 = \mathbb{L}[lockset_1], \mathbb{L}[lockset_2]$
      $R_{new1}, C_{new1} = \text{select\_races}(accs_1, accs_2, C_m)$
      $R_{new2}, C_{new2} = \text{select\_races}(accs_2, accs_1, C_m)$
      ▷ see Algorithm 4
      $C_{all} = C_{all} \cup C_{new1} \cup C_{new2}$
      $R_{all} = R_{all} \cup R_{new1} \cup R_{new2}$
      if $C_m \subseteq C_{all}$ **then**
        break
return R_{all}

3.3.3. Race Checking. When checking a predicted race, we hypothesize a witness schedule that schedules the first input seed up to the first memory access in the race, and then preempts and schedules the second selected input to cover all memory accesses predicted to race with the first preempted memory access from the first input.

In order to check for races efficiently, we minimize the number of individual race checks that need to be performed and maximize the number of previously untested instructions covered by each requested race check (e.g., only 2 pairwise checks are necessary to confirm 4 racing memory accesses, even though there are 4 possible pairs). Given the set of all possible race predictions $\mathbb{R}$, we select a subset R on which to run race validation based on the following optimization:

$$R = \arg\max_R |cover(R)| \min |R| : R \subseteq \mathbb{R}$$

Algorithm 4 Race Selection.

Input: $accs_1 \leftarrow$ Memory accesses predicted to race with $accs_2$
$accs_2 \leftarrow$ Memory accesses predicted to race with $accs_1$
$C_m \leftarrow$ Max possible cover for address m
$\beta \leftarrow$ Race Prediction Threshold

$prog\_ips$ = hashmap(default = $\varnothing$)
for $\alpha \in accs_2$ **do**
  $p = \alpha.p$
  $prog\_ips[p] = prog\_ips[p] \cup \{\alpha.ip\}$

$C_{new}, R_{new} = \{\}, \{\}$
for $\alpha \in accs_1$ **do**
  if $\alpha.op == w$ **then**
    $p_1 = \alpha.p$
    $P_2 =$ all unique $\alpha_2.p : \alpha_2 \in accs_2$
    while $True$ **do**
      $p_2 = \arg\max_{p_2 \in P_2} \left(\left| prog\_ips[p_2] \setminus C_{new} \right| \right)$
      $P_2 = P_2 \backslash p_2$
      if $\max \mathbf{P}[\alpha \cap \alpha_2] : \alpha_2.p = p_2$ **then**
        Break
    $C_{upd} = prog\_ips[p_2] \cup \alpha.ip$
    if $|C_{upd} \backslash C_{new}| > 0$ **then**
      $r = (p_1, p_2, \alpha.ip, \alpha.m)$
      $R_{new} = R_{new} \cup \{r\}$
      $C_{new} = C_{new} \cup C_{upd}$
      if $C_{new} == C_{max}$ **then**
        break

return R_{new}, C_{new}

where $cover$ denotes the set of instruction addresses in R. In practice we build R directly during analysis and avoid the cost of enumerating possible predicted race in $\mathbb{R}$.

When two sets of conflicting memory accesses with non-intersecting locksets are identified, we take each write access and select a second input to test that will execute as many conflicting accesses as possible with high probability (where at least one of the predicted race probabilities must exceed β). Algorithm 4 describes this procedure.

4. Implementation

We implement PLA in three main components: tracing and probability estimation, lockset analysis and race prediction, and watchpoint-based race checking.

Tracing. We perform tracing using the kernel event ring buffer and modify the kernel concurrency sanitizer (`kcsan`) [7] to record all memory accesses that it would normally check for races using watchpoints. We additionally record all lock events using the kernel's built in lock tracing. We base our tracing implementation on `kcsan` because it incorporates rules to ignore memory accesses that are marked with allowed-to-race macros such as `READ_ONCE` or `WRITE_ONCE`. Racing is allowed for many kernel memory accesses, so ignoring these accesses greatly reduces overhead and prevents predicting races that are benign [11].

When tracing we use a modified syzkaller [13] executor that incorporates a barrier after initialization to execute multiple seeds concurrently. We perform tracing on two isolated CPUs, where each executor process is pinned to a distinct CPU, and use a QEMU 6.2.0 VM (although any VM system could be used). When collecting a trace, we first refresh the VM to a fixed snapshot.

Probability Estimation. Access lockset probability estimation is performed at the same time as tracing. The traces from each seed are temporally stored in memory and then immediately used to estimate the probabilities of its access locksets. Since traces are much larger than the set of high probability locksets, not writing them to disk greatly reduces overhead. High probability access locksets are then grouped by memory address and gathered from all sampled inputs. This procedure follows a map-reduce paradigm, where tracing and sampling is mapped to each input and results are reduced into a common database of access locksets indexed by memory address.

Analysis and Race Prediction. Analysis and race prediction are performed in two parallel stages. First, the linear lockset analysis pass identifies memory addresses that contain racing addresses. These racing memory addresses are then grouped based on possible coverage (i.e., the set of instruction addresses of the accesses to the memory address). Pairwise lockset analysis and coverage guided race selection is then applied to the access locksets in each group of racing memory addresses. Splitting the analysis into two stages and grouping by coverage allows us to perform each analysis in a fully parallel manner, while still minimizing the number of individual race predictions that need to be checked for full instruction coverage.

When checking pairwise lockset intersections, we set maximum unique locksets threshold, and sample a subset of the access locksets used in analysis when the number of unique locksets exceeds the threshold. In evaluation we set the unique locksets threshold to 1000, which we found takes approximately 2.3 seconds to process. We found that memory addresses with more than 1000 unique locksets in their memory accesses are extremely rare, with only 14 observed out of thousands of racing memory addresses seen in our evaluation (Section 5.6).

Race Validation. We confirm predicted races by executing the generated witness schedule and obtain stack traces for the race using preset watchpoints and the same modified syzkaller executor and CPU configuration used in tracing. Race predictions selected for validation are provided in the form (p_1, p_2, w_ip, w_addr), where w_ip and w_addr are the watchpoint instruction address and memory address, and p_1 is expected to execute the watchpoint with high probability.

5. Evaluation

We address the following research questions in our evaluation:

1) **Security Testing Performance:** Is PLA effective at finding kernel data races that are harmful for kernel security?
2) **Comparison with other Approaches:** How does probabilistic lockset analysis compare to the race prediction methods used by recent kernel concurrency fuzzers?

3) **Probabilistic Analysis and Accuracy:** How accurate are PLA's race predictions, and how does PLA's probabilistic analysis compare to standard lockset analysis when run on traces of a seed corpus?
4) **Design Choices:** How do each of the optimizations in PLA's algorithm design contribute to its performance?
5) **Parameter Choices:** How do the settings for β and sample rate effect PLA's performance?
6) **Scalability:** How well does PLA scale to large numbers of memory accesses and lock events?

Evaluation Setting. All experiments are performed on an Ubuntu 22.04 server with Ryzen Threadripper 2970WX CPU and 128Gb of memory.

5.1. Security Testing Performance

Experimental Setup. We test Linux Kernel v5.18-rc5 and run PLA on a corpus of 129 thousand syzkaller seeds sourced from [22].

Table 1: Summary of races found by PLA categorized by kernel subsystem. We count data races in terms of unique pairs of racing instructions as well as unique number of variables. We classify a race as harmful based on [50]. We provide a full listing of races in Table 5 in Appendix E.

subsystem	instruction pairs	variables	harmful variables
arch/x86	1	1	0
drivers/base	4	1	1
drivers/char	2	1	0
drivers/input	1	1	1
drivers/misc	1	1	1
drivers/net	3	1	1
drivers/pci	4	1	1
drivers/scsi	6	1	0
drivers/tty	21	8	5
fs	2	1	1
kernel	13	5	4
kernel/cgroup	2	1	1
kernel/events	1	1	1
kernel/time	4	1	0
mm	33	7	3
net/core	3	1	1
net/ipv4	8	3	3
net/llc	2	1	0
net/netfilter	2	1	1
net/unix	2	1	1
net/xfrm	50	4	4
security/keys	10	4	2
sound/core	8	5	3
Total	183	52	35

Results. Table 1 summarizes the results with full details in Table 5 in Appendix E. PLA found 52 unique racing variables and 183 unique racing pairs of instructions. As prior work has counted data races based on either racing variables or racing pairs, we provide both metrics. We use the number of racing variables based on Krace [50] as well as the number of unique racing pairs of instructions based on Conzzer [26]. For a concrete example, race ID 48 from Table 5 involves a single variable with races detected across 22 unique pairs of memory accesses, so the number of racing variables is 1 and the number of unique racing pairs of instructions is 22.

We classify the data races as harmful or benign based on approach by Xu et al. [50]. Specifically, we declare a race as benign if (i) reads and writes to a racing variable involve different bits or (ii) involve kernel functions where race conditions are acceptable (e.g., random or logging subsystems). In total, we found 35 harmful racing variables and 102 harmful racing instruction pairs. Out of the 35 variables with races, 4 cause memory corruption, 1 leads to information leakage, 1 causes multiple initializations on a data structure, and 29 cause undefined behavior (but with no confirmed immediate security implications). We disclosed the harmful races to Kernel developers and so far 56 races over 9 variables have been patched and one CVE with high (7.0) severity (CVE-2022-3028) has been allocated based on our reports [9].

5.1.1. Case Studies. PLA finds data races in heavily-tested core kernel subsystems. We detail two data races with security implications below.

Out-of-bounds write in net/xfrm. Figure 6 shows how a data race in networking cryptography algorithm management can cause an out-of-bounds memory write vulnerability. First, at (1), thread A allocates a buffer based on the authentication algorithms list length, which is set to the number of available algorithms in the list. Next, at (2), a concurrent thread B executes the `xfrm_probe_algs` function, which updates the availability of algorithms in the list. However, the buffer size is not increased, so when thread A continues executing at (3), it writes past the bounds of the undersized buffer as it populates the buffer with the available authentication algorithms. This results in an out-of-bounds write vulnerability.

The authentication algorithms list buffer is sent over a socket and therefore can be used as an information leak primitive for kernel heap memory when it is instead oversized during the race (i.e., a concurrent thread decreases the number of available authentication algorithms). This vulnerability has been present in the Linux Kernel since 2013 (v3.14-rc1). We reported this vulnerability and it has been patched and allocated a high severity CVE [9].

Use after free in mm. Figure 7 shows how a data race in the kernel list of shared memory pages can cause a use after free vulnerability. First, at (1), thread A inserts a newly added memory page to the main list of shared memory pages. However, inserting the new page to the list and setting its flags is not atomic. This allows a concurrent thread B to free the newly added memory page at (2). When thread A continues executing at (3) and sets the flags of the page, which was already freed, a use-after-free vulnerability will occur. We have reported this vulnerability and it has been patched.

5.2. Comparison with other Approaches

We evaluate PLA against other recent systems that target data race detection in the kernel based on their ability

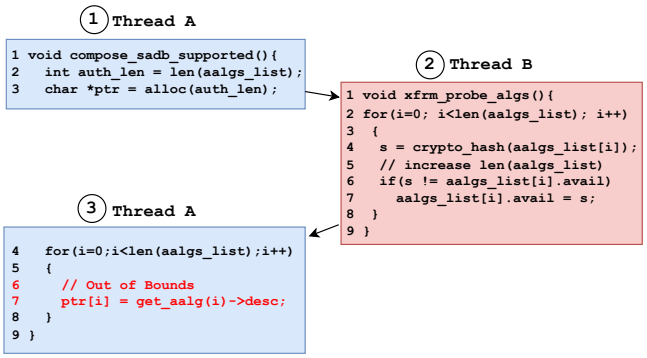


Figure 6: **A harmful data race in the net/xfrm kernel subsystem involving the `aalg_list[i].available` variable (ID 48 in Table 5). The numbers (1), (2), (3) indicate the order of events in the data race that leads to an out-of-bounds write vulnerability.**

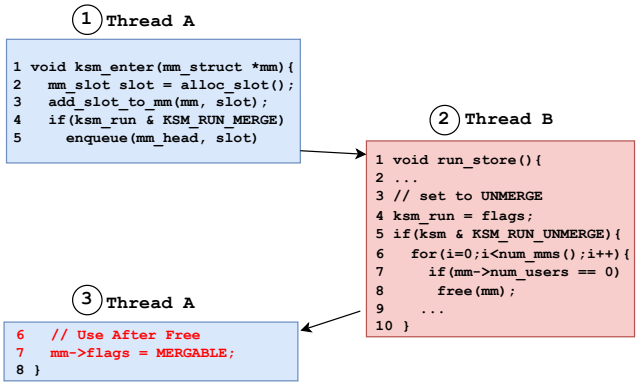


Figure 7: **A harmful data race in the mm kernel subsystem (ID 14 in Table 5). This data race leads to a use-after-free vulnerability.**

efficiently find kernel races with a 24 hour time budget.

5.2.1. Evaluated Approaches. We evaluate against three classes of approaches: Coverage guided concurrency fuzzers with happens-before/lockset dynamic race predictors, alias-analysis-guided race fuzzers, and standard fuzzers with watchpoints.

1.) *Concurrency fuzzers.* Concurrency fuzzers combine a concurrency coverage guided fuzzing with a hybrid happens-before/lockset dynamic race predictor. Krace [50] and Conzzer [26] are two recent kernel concurrency fuzzers.

Krace is open sourced [10], but the release does not contain any documentation on usage. We attempted to run krace but encountered errors with missing data files that had been previously reported in issue #2 on the github repository [3]. We emailed the Krace authors to report the issue but did not receive a response. Conzzer has a binary-only release available from [5]. We attempted to run Conzzer but encountered several errors that were not addressed in the provided documentation and could not be debugged without access to source code. We emailed the Conzzer authors to report the issue but did not receive a response.

Since we were unable to run either Krace or Conzzer, we emulate a concurrency fuzzer based on Krace's alias coverage, which we refer to as *Alias Fuzzer*. We base Alias fuzzer on the descriptions of Krace's runtimes in [50] and make optimistic assumptions about its performance (i.e., if a race can be detected for given set of seeds, the fuzzer's race predictor will always identify it without errors).

2.) *Targeted Race Fuzzers.* Targeted race fuzzers select seeds and schedules designed to trigger specific candidate races predicted by alias analysis on a seed corpus. We consider two targeted race fuzzers, Razzer [25] and Snowboard [22]. Razzer identifies candidate races through static alias analysis, while Snowboard identifies candidate races dynamically by comparing memory accesses between traces, and then performs additional concurrency fuzzing. We evaluate Snowboard because it is more recent (SOSP 2022), incorporates both concurrency fuzzing and targeted race checking, and supports current 5.x linux kernels (Razzer only supports 4.x linux kernels).

3.) *Fuzzing with Watchpoints.* We additionally evaluate against Syzkaller [13], a standard kernel fuzzer that performs multithreaded fuzzing, using the kernel concurrency sanitizer (`kcsan`) [7], a watchpoint-based data race detector that is deployed for continuous linux kernel testing [12].

5.2.2. Experiment Design. Concurrency testing systems perform two distinct tasks: input generation and concurrency testing on those inputs. In this evaluation we measure concurrency testing performance and control for input generation by running all evaluated systems on a fixed benchmark corpus of 10,000 fuzzer seeds. We run each evaluated system five times for 24hr on the benchmark corpus, and configure each system to fully utilize the server cpu and memory.

For reported races on all evaluated systems races, we filter to ensure the races occur in the executing seed processes (`kcsan` will sometimes detect races in background processes) and are not allowed by the linux kernel memory model (Snowboard's race detector can report races that are actually allowed in the linux kernel). For PLA and Snowboard, we include the time for tracing and analysis of the corpus in the results. When evaluating Syzkaller, we initialize it to use the benchmark corpus and disable new seed generation/mutation so that it focuses exclusively on concurrently executing the seeds in the benchmark.

5.2.3. Results and Discussion. Figure 8 shows race finding results for the 24hr run on the 10k seed benchmark. On average, PLA finds 164 races on the benchmark, Syzkaller finds 43 races, Snowboard finds 21 races, and Alias Fuzzer finds 15 races.

PLA's ability to efficiently and accurately search over the entire corpus to predict races is critical to its good performance on this benchmark. Because it can effectively prioritize high probability races, it finds many races quickly (over 100 in less than an hour after completing its analysis) and is able to able to quickly check predictions with a single execution without resorting to schedule fuzzing.

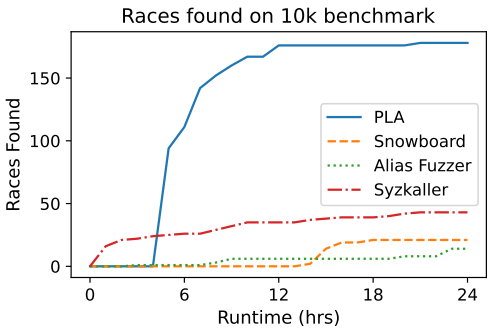


Figure 8: Evaluation of races found over five 24hr runs on benchmark of 10k minimized seeds. On average, PLA finds 164 races in total, Snowboard 21, Alias Fuzzer 15, and Syzkaller with Kcsan finds 43.

Snowboard performs analysis on the corpus to identify potential memory communications (PMCs), but unlike PLA does not have any way to estimate if a communication is feasible or potentially racy. As a result it must test many more PMCs for each race found. Snowboard also performs additional concurrency fuzzing based on each PMC, which allows it to reach new states and potentially find additional races, but reduces its throughput when testing. We also tried running Snowboard's fuzzing stage for a total of 24 hours after it completed its analysis, but in that time it only found two additional races.

The simulated Alias Fuzzer also only finds 15 races on average in the benchmark, in spite of the optimistic assumptions we used in its simulation. This result illustrates the intrinsic hardness of searching a corpus of seed inputs for races using concurrency fuzzing and dynamic race prediction. In total the simulated fuzzer fuzzed 31,900 three seed combinations (each of which exposed new alias coverage, requiring the two minute race prediction check) for a total of 95,700 input pairs searched. However, the total space of possible input pairs for a 10,000 seed corpus is roughly $10^8/2$, more than four orders of magnitude larger. At the rate of the simulated Alias Fuzzer, which we believe to be an optimistic estimate for running concurrency fuzzing and race prediction based on the description in [50], so fully fuzzing and running race prediction on all input pairs in the corpus would take over a year!

Syzkaller with `kcsan` achieves the next best performance on the benchmark after PLA, although it it has performed poorly in prior evaluations on finding races in filesystems [26] and finding specific races associated with CVEs [25]. We hypothesize that Syzkaller's good performance on this benchmark is due to initialization with a corpus of high quality seeds. Unlike other systems in the benchmark, which test 2 or 3 concurrent inputs at a time, Syzkaller runs 8 fuzzing processes on each vm and checks for races between any of them with `kcsan`.

Table 2: Comparison of PLA with standard lockset analysis (Lockset) for accuracy predicting which observed memory accesses are racing, analysis runtime, and number of tested predictions per race found (Tests/Race) on benchmarks of 10 to 50 seeds. Because accuracy is evaluated per-access but race predictions are made on pairs of accesses, lockset analysis's much lower accuracy leads to millions of erroneous predictions. Each race found on the 50 seed benchmarks with lockset analysis requires approximately 6 days of checking predictions in our evaluation setting, compared to roughly 10 seconds for PLA.

Metric	Approach	# Seeds in Benchmark 10	20	30	40	50
Accuracy	PLA	0.997	0.992	0.990	0.989	0.989
	Lockset	0.711	0.595	0.561	0.542	0.512
Runtime(s)	PLA	0.7	2.3	4.3	6.8	10.0
	Lockset	28.9	98.9	185.6	321.6	481.8
Tests/Race	PLA	1.5	2.9	3.0	3.9	4.2
	Lockset	8.1e+04	2.7e+05	5.1e+05	8.1e+05	1.2e+06

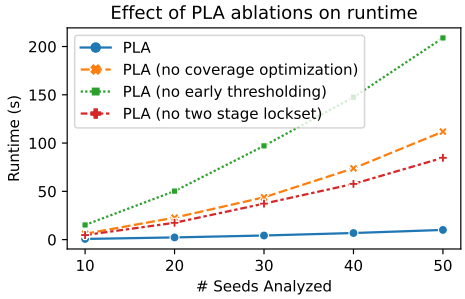


Figure 9: Impact of ablations on analysis runtime averaged over 5 randomly sampled benchmarks. On benchmarks of 50 seeds, ablations increase PLA's runtime between $8.5\times$ and $21\times$ and cause the analysis to scale superlinearly in the number of seeds.

5.3. Probabilistic Analysis and Accuracy

We evaluate PLA's accuracy in predicting which observed memory accesses in the traces are racing and compare it to standard lockset analysis. We evaluate on five randomly sampled benchmarks of 50 seeds, and evaluate the scaling of each tested approach on subsets of 10 through 50 seeds from each benchmark set. We use relatively small benchmarks for this study (compared to 10k seed benchmark used in Section 5.2) because the extremely high error rates of standard lockset analysis make testing it on even small benchmarks prohibitively time consuming.

PLA vs. Lockset Analysis. Table 2 shows a comparison of PLA with standard lockset analysis with averaged results for analysis accuracy, analysis runtime, and test executions required to find each observed race in the benchmark. The results in Table 2 demonstrate how critical PLA's probabilistic reasoning is to achieving performance at a scale: when all observed memory accesses are included in the analysis, a significant proportion appear as spurious aliases that access the same memory address in some traces with low probability, but cannot race when executed concurrently with

Table 3: Impact of sample count (N) on accuracy and runtime. For each N, the highest f1 accuracy achieved by varying β is shown. Collecting more than 4 samples greatly increases sample collection time with marginal accuracy improvements, therefore we use $N=4$ in all experiments.

Samples Collected (N):	2	4	8	16
F1 Score for Best β:	0.72	0.87	0.87	0.93
Sample Time/Input:	15s	30s	60s	120s

one another. This causes the analysis runtime to increase drastically and severely reduces the accuracy of the analysis. Since even a small number of seeds perform millions of distinct memory accesses, this results in over 1.2 million incorrect race predictions on average for each race found with standard lockset analysis on the 50 seed benchmarks, compared to 4.2 for PLA.

PLA Accuracy on 10k Seed Benchmark. We also evaluate PLA's accuracy on the 10k seed benchmark used for the systems comparison evaluation in Section 5.2 and find that it runs in 34 minutes, identifies racing instructions with 89.9% accuracy, and requires 12.1 tests on average for each race observed in the benchmark.

5.4. Design Choices

We evaluate three of the design optimizations in PLA with ablations: early probability thresholding, two stage linear and pairwise lockset analysis, and coverage optimization in race checking. Figure 9 shows the average analysis runtime of PLA with each of the ablations on the 5 benchmark sets (the ablations do not effect the accuracy of PLA's race predictions, only runtime). While removing early thresholding has the largest impact on runtime ($21\times$ slower than PLA on 50 seeds on average), ablating coverage optimization or two stage linear and pairwise lockset analysis also incurs a significant performance penalty ($11\times$ and $8.5\times$ slower on average, respectively). Moreover, each PLA ablation scales superlinearly while PLA's runtime scaling is linear, so all of PLA's design optimizations are critical to achieving scalable runtimes on large real world corpuses of thousands of seeds.

5.5. Impact of Parameter Choices

Parameter Choices. PLA's performance is governed by two parameters: β, the threshold at which access locksets are included in the analysis, and N, the number of samples collected for each input. We evaluated PLA's accuracy in identifying stable access-locksets while varying the N and β parameters on the seed benchmarks used in Section 5.2. We tested sample counts of $N=2, 4, 8, 16$ and varied β from 0.0 to 1.0 in increments of 0.1 for $N=8$ and $N=16$, and increments of 0.5 and 0.25 for $N=2$ and $N=4$, respectively.

Table 3 shows the f1 accuracy for best-performing β setting and sample collection time for each tested N. We found that increasing the samples collected beyond $N=4$ only achieves marginal accuracy improvements while significantly increasing sample collection time, therefore we use $N=4$ and the associated best $\beta=0.5$ setting for all experiments. See Appendix F for detailed results.

Table 4: Input sizes and runtimes for PLA on 10k inputs.

Stage	Inputs	Runtime
Sampling	10 billion trace events	3.5 hr
Memory Mapping	380 million access locksets	19 min
Linear Lockset Analysis	380 million access locksets	12 min
Pairwise Lockset Analysis	3.4 million access locksets	135 sec
Race Prediction Checking	Per 140 predictions	60 sec

5.6. Scaling

Benchmark Corpus Runtime. We evaluate PLA's ability to scale to large number's of memory accesses based on the corpus of 10k inputs used in Section 5.2. As described in Section 4, PLA works in 3 states: tracing and sampling, race prediction analysis, and race checking. Table 4 shows a breakdown of the runtimes and input sizes for each stages in PLA's pipeline. PLA spends most of its time collecting traces, which is slow due to the large size of traces. Subsequent stages (memory mapping, linear lockset analysis), are much faster because they operate on fewer inputs.

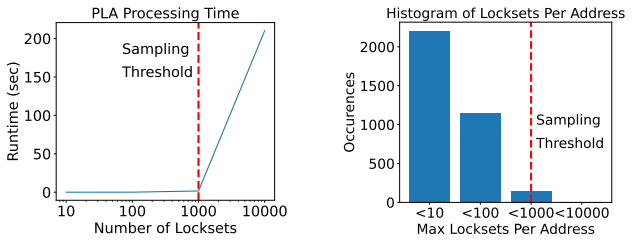
The numbers in Table 4 illustrate that two design optimizations in PLA (Section 3.3) are absolutely critical to its performance: 1.) Applying probability thresholding during initial trace collection reduces the number of events that must handled by the subsequent, more expensive, stages of the analysis by a factor of over 100. 2.) Applying coarse grained linear lockset analysis before running the more precise but expensive pairwise lockset analysis reduces the access locksets that must be processed by pairwise lockset analysis by another factor of 100. Without these two optimizations, running PLA on the same corpus would take at least six days instead of four hours.

Pairwise Lockset Analysis Scaling. Since pairwise lockset analysis has a quadratic term for the number of unique locksets on a single address, we also investigate the runtime of PLA relative to locksets and the distribution of unique locksets in the test corpus. For 1000 unique locksets, pairwise lockset analysis takes 2.3 seconds, but over 200 seconds for 10000 unique locksets, as shown in Figure 10a. Therefore, when the access locksets for a single address have more than 1000 unique locksets, we perform pairwise lockset analysis on a sample of the locksets (Section 4.

We found that only a very small number of memory addresses with lock violations have more than 1000 locksets. Figure 10b summarizes these results. We found that out of 3511 memory addresses predicted to be involved in races, only 14 had more than 1000 unique locksets. As has been noted in prior work [32], harmful data races usually involve rarely accessed memory, and all of the harmful races we found involved infrequently used memory addresses.

6. Related Work

Dynamic Race Prediction Dynamic race prediction identifies possible data races based on concurrent program ex-



(a) PLA processing time with respect to number of locksets.

(b) Distribution of max number of locksets per address.

Figure 10: Lockset runtimes and statistics. Sparsity in lock interactions in the kernel means that only a few distinct locksets are used for the vast majority of shared memory addresses as shown in 10b.

ecution traces. Happens-before methods reason about partial orders on traces based on Lamport's happens-before relation on interthread communications to predict races soundly [20, 24, 29, 35]. Extensive work has focused on developing weakened partial orders that soundly predict more races from a trace [28, 33, 34, 39, 41, 46], or using SMT reductions, which are sound and complete with regard to the observed trace but limited in scalability [23, 42, 45, 48, 49]. Lockset analysis is a form of dynamic race prediction that performs an intersection over all held locks for each memory access to a given address, and alerts if the intersection is null, but suffers from high false positive rates [16, 43]. Therefore, many race predictors such as RaceTrack and Goldilocks combine happens-before and lockset analysis to make precise lockset-based race predictions [17, 19, 40, 47, 52]. These methods are also limited to operating on one concurrent trace a time in order to infer happens-before ordering constraints, which limits their scalability. In contrast, PLA uses lockset analysis with probabilistic predictions to make precise race predictions over large corpuses of independently sampled seed execution traces.

Schedule Exploration. Schedule exploration methods search for races by systematically executing many different schedules, either by enumerating schedules bounded by a preemption count [36, 37], sampling a distribution of schedules [15, 53], fuzzing with a concurrency specific coverage metric [51], or performing targeted exploration of schedules based on static alias analysis [44] These approaches operate on one fixed concurrent program at a time, while PLA is designed to identify races between a large corpus of seed programs in the kernel.

Kernel Testing. Many concurrency testing approaches have been applied to the kernel such as random watchpoints with delays on memory accesses [18] and schedule exploration by sampling a distribution of schedules [21]. Targeted race fuzzers either static or dynamic alias analysis combined with dynamic tracing to identify possible races between input seeds, which it then combines for targeted fuzzing of the possible races [22, 25]. Concurrency fuzzers use a concurrency specific coverage metric to guide schedule fuzzing in conjuction with dynamic race predictors to detect observed races [26, 50]. PLA differs from existing kernel testing systems in that it performs race prediction over an entire corpus of seed programs subject to lock synchronization, and uses probabilistic prediction to accurately identify and prioritize races.

7. Limitations and Future Work

PLA targets races involving operations that are performed for most concurrent schedules and occur with high probability, but will ignore *schedule-dependent* races that only occur for specific schedules, since these will appear with low probability in PLA's sampling. This trade-off allows PLA to be both fast and accurate when performing analysis over billions of trace events, but means that PLA will not find schedule-dependent races, which can still potentially be exploited by attackers.

This naturally begs the question: is it possible to extend PLA to target schedule dependent races, while retaining the benefits in accuracy and scalability from PLA's probabilistic approach? We believe the answer to this question is *yes*: the probability of a memory access can also be conditioned on specific partial orderings on the execution schedule (conceptually, a probabilistic happens-before analysis). However, identifying and sampling relevant partial orders on schedules is much more challenging, because the space of possible partial orders on the schedule is exponential. We intend to explore this in future work.

8. Conclusion

We introduce Probabilistic Lockset Analysis (PLA), a form of race prediction analysis specifically designed to address the inherent challenges in predicting races in the kernel. PLA samples execution traces to estimate the probability of races between seeds in a fuzzer corpus, and can resolve predictions with greater precision by taking more samples. We use PLA to find 183 races in core kernel modules and show in an evaluation of kernel race testing methods that PLA finds races at more $3\times$ the rate of comparable systems. Although PLA's design is motivated by and applied to kernel race prediction, its approach can potentially be applied to testing any system that processes each input on a separate thread or process. We intend to explore applications of PLA's approach to testing other concurrent applications in future work.

Acknowledgements

We thank our shepherd and the anonymous reviewers for their constructive and valuable feedback. Gabriel Ryan is supported by an NDSEG Fellowship, and Abhishek Shah is supported by an NSF Graduate Research Fellowship. This work is supported partially by NSF grant CNS-21-54874; an NSF CAREER award; a Google Faculty Fellowship; and a Google ASPIRE award. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors, and do not necessarily reflect those of NSF, NDSEG, or Google.

References

[1] Kernel panic due to race condition. https://access.redhat.com/solutions/1593553, 2015.

[2] Dirty cow (cve-2016-5195). https://dirtycow.ninja/, 2016.

[3] Krace github issue number 2. https://github.com/sslab-gatech/krace/issues/2, 2020.

[4] Race condition in macos kernel (cve-2021-1782). https://nvd.nist.gov/vuln/detail/CVE-2021-1782, 2021.

[5] Conzzer binary release. https://oslab.cs.tsinghua.edu.cn/CONZZER/, 2022.

[6] Huawei kernel module race condition (cve-2022-31758). https://nvd.nist.gov/vuln/detail/CVE-2022-31758, 2022.

[7] Kernel concurrency sanitizer. https://www.kernel.org/doc/html/latest/dev-tools/kcsan.html, 2022.

[8] Kernel race exploit for denial-of-service (cve-2022-1652). https://www.cvedetails.com/cve/CVE-2022-1652/, 2022.

[9] Kernel race exploit leading to information leak, memory corruption (cve-2022-3028). https://nvd.nist.gov/vuln/detail/CVE-2022-3028, 2022.

[10] Krace open source release. https://github.com/sslab-gatech/krace, 2022.

[11] Linux kernel memory consistency model. https://github.com/torvalds/linux/blob/master/tools/memory-model/Documentation/explanation.txt, 2022.

[12] Syzbot reports. https://syzkaller.appspot.com/upstream, 2022.

[13] Syzkaller. https://github.com/google/syzkaller, 2022.

[14] Syed Mumtaz Ali and Samuel D Silvey. A general class of coefficients of divergence of one distribution from another. *Journal of the Royal Statistical Society: Series B (Methodological)*, 28(1):131–142, 1966.

[15] 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.

[16] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 85–96, 1991.

[17] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. In Klaus Havelund, Manuel Núñez, Grigore Roşu, and Burkhart Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification*, pages 193–208, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[18] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[19] Azadeh Farzan, P Madhusudan, and Francesco Sorrentino. Meta-analysis for atomicity violations under nested locking. In *International Conference on Computer Aided Verification*, pages 248–262. Springer, 2009.

[20] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[21] Pedro Fonseca, Rodrigo Rodrigues, and Björn B Brandenburg. Ski Exposing kernel concurrency bugs through systematic schedule exploration. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 415–431, 2014.

[22] 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*, pages 66–83, 2021.

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

[24] Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordehai. Toward integration of data race detection in dsm systems. *Journal of Parallel and Distributed Computing*, 59(2):180–203, 1999.

[25] 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.

[26] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. 2022.

[27] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. An empirical analysis of vulnerabilities in openssl and the linux kernel. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 105–112. IEEE, 2016.

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

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[30] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. Exprace: Exploiting kernel races through raising interrupts. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2363–2380. USENIX Association, 2021.

[31] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A study of linux file system evolution. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 31–44, San Jose, CA, February 2013. USENIX Association.

[32] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 134–143. ACM, 2009.

[33] 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.

[34] 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.

[35] Friedemann Mattern. Virtual time and global states of distributed systems. In *Proc. Workshop on Parallel and Distributed Algorithms,*, 1989.

[36] Madan Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, November 2007.

[37] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, volume 8, 2008.

[38] Robert HB Netzer and Barton P Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

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

[40] Eli Poznianski and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 287. IEEE Computer Society, 2003.

[41] 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.

[42] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. Generating data race witnesses by an smt-based analysis. In *NASA Formal Methods Symposium*, pages 313–327. Springer, 2011.

[43] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[44] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[45] Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu. Maximal causal models for sequentially consistent systems. In *International Conference on Runtime Verification*, pages 136–150. Springer, 2012.

[46] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 387–400. ACM, 2012.

[47] Francesco Sorrentino, Azadeh Farzan, and P Madhusudan. Penelope: weaving threads to expose atomicity violations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 37–46, 2010.

[48] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. Symbolic predictive analysis for concurrent programs. In *International Symposium on Formal Methods*, pages 256–272. Springer, 2009.

[49] Chao Wang, Rhishikesh Limaye, Malay Ganai, and Aarti Gupta. Trace-based symbolic analysis for atomicity violations. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 328–342. Springer, 2010.

[50] 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.

[51] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. OOPSLA '12, page 485–502, New York, NY, USA, 2012. Association for Computing Machinery.

[52] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, 2005.

[53] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. Partial order aware concurrency sampling. In *International Conference on Computer Aided Verification*, pages 317–335. Springer, 2018.

Appendix A. Dynamic Race Prediction

Dynamic race prediction seeks to predict data races based on a concurrent execution trace. A concurrent program P is composed of a set of threads $P = \{p_1, p_2, ...\}$ that can be executed concurrently according to a schedule s to generate a trace T:

$$\text{trace}(P; s) = T$$

A trace T is composed of events (denoted e) that are totally ordered by the schedule s:

$$T = [e_1, e_2, ...]$$

Each trace event e is a tuple composed of an executing thread p, relevant memory or lock address m, and operation type op (read, write, lock acquire, or lock release):

$$e = (p, m, op) \qquad op = r|w|acq|rel$$

We use $a \in T$ and $l \in T$ as shorthand for the memory accesses or locks operations in a trace. Other synchronization operations such as forks, joins, and barriers may also be included in a trace. We avoid them here for the sake of clarity.

Feasible Schedules. For a schedule to be feasible on a program it must satisfy two ordering constraints: (i) *thread order*, the instructions in each thread must be executed in order and (ii) *synchronization order*, it must not violate the order imposed by synchronization primitives in each thread (e.g., a lock cannot be acquired twice without first being released). We denote a feasible schedule for a program P as $\text{feas}_P(s)$.

Concurrent Events. Two events are considered *concurrent* in a schedule if their positions in the schedule are interchangeable: either can be executed at a given location without violating either thread order or synchronization order. We define two events as concurrent for a program P and schedule s if exchanging their positions does not make the schedule infeasible:

$$\text{concurrent}_P(e_i, e_j, s) := \text{feas}_P(\text{exchange}(e_i, e_j, s)), \\ e_i, e_j \in \text{trace}(P; s)$$

where *exchange* indicates swapping two events in the schedule.

Data Races. Two memory accesses are considered a *conflict* if they are both memory accesses to the same address and at least one is a write:

$$\text{conflicting}(a_i, a_j) := a_i.m = a_j.m \wedge \\ (a_i.op = w \vee a_j.op = w)$$

A pair of conflicting memory accesses in a trace is then considered a *data race* for a program P if they are concurrent in the trace schedule:

$$\text{race}_P(a_i, a_j, s) := \text{concurrent}_P(a_i, a_j, s) \\ \wedge \text{conflicting}(a_i, a_j)$$

Predicted Races. We denote the set of synchronization primitives that guard two memory accesses by *sync*, where two accesses are considered unsynchronized if $sync(a_i, a_j) = \varnothing$. Any predicted race will always be on two unsynchronized events:

$$\text{pred\_race}(a_i, a_j) \implies \text{sync}(a_i, a_j, s) = \varnothing$$

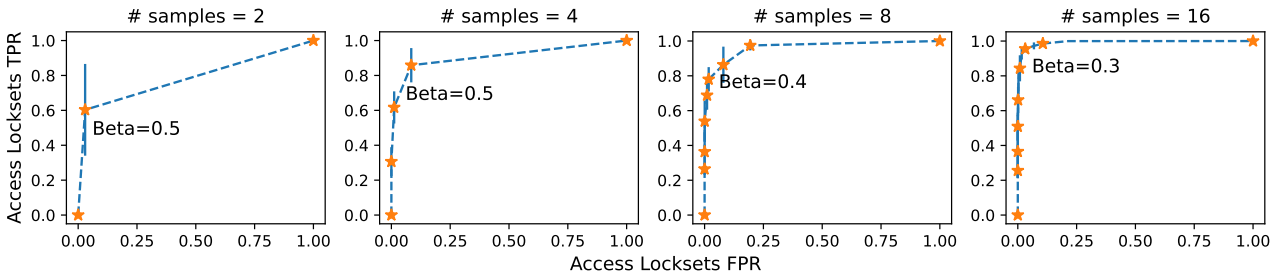


Figure 11: ROC curves for access lockset prediction using varying numbers of samples evaluated on 5 sets of 50 randomly selected seeds with shown std. deviation. For each curve, the classification threshold parameter β giving the best performance is annotated based on F1 score.

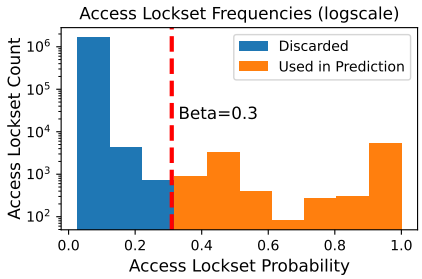


Figure 12: Distribution of access locksets probabilities shown with log scale, where access locksets with probability exceeding β are marked orange. The vast majority of access locksets ($> 99.9\%$) occur with very low probability ($< 0.05\%$), therefore identifying high probability access locksets is critical to making accurate race predictions.

However, null synchronization is a necessary but not sufficient condition for a race. If any schedule that triggers the race would cause the program not to execute the relevant memory accesses, then the race prediction is a false positive.

Feasible Races. For a predicted race to be feasible there must be a feasible schedule under which the two events still appear (i.e., P still executes the conflicting memory accesses) and race with each other:

$$\text{feas\_race}_P(a_i, a_j) := \exists s^* : \text{race}_P(a_i, a_j, s^*) \ \wedge \ \text{feas}(s^*, P)$$

Task Definition. Dynamic race prediction seeks to predict all feasible racing pairs of memory accesses in a trace from program P executing a given schedule s:

$$\begin{aligned} &\text{input:} && \text{program } P, \ \text{trace } T \\ &\text{output:} && \text{race prediction } a_i, a_j, s^* \end{aligned} \tag{3}$$

Appendix B. Dynamic Race Prediction Approaches

Happens Before Analysis. Happens before analysis uses partial orders defined on memory accesses and synchronization events to perform *sound* dynamic data race prediction (i.e., predict only feasible races). When a race is predicted between a pair of events a schedule and trace s^*, T^* must also be found that preserves the read/write happens-before relation in the observed trace T:

$$\forall r \in T^* : last\_write(r, T^*) = last\_write(r, T)$$

where $last\_write$ indicates the most recent write to a read address of r in a trace.

Preserving the read-write partial order ensures that the program will follow the same execution path for s^* as the original schedule s. This guarantees that predicted races will be feasible, and has the additional benefit that s^* can be used as a witness schedule to reproduce the race. However, happens-before analysis requires a reference trace T in order to define a sound partial order.

Lockset Analysis. Lockset analysis ignores the order in the observed trace and instead checks exclusively for commonly held locks on each shared memory access. Ignoring ordering makes lockset analysis *complete* but *unsound*. Any observed memory accesses that can race will be predicted as races, but the predicted races are not guaranteed to be feasible.

The lockset algorithm checks for commonly held locks by performing an intersection over the held locks for each memory access to a given address. It marks a memory access a as potentially racing if the the following condition is met:

$$\text{lockset\_violation}(a) := \bigcap_{\hat{a} \in T} \text{lockset}(\hat{a}) = \varnothing : \hat{a}.m = a.m$$

where *lockset* indicates the set held locks by a thread when a memory access was performed:

$$\text{lockset}(a) := \{l : last\_acq_l(a, T) > last\_rel_l(a, T)\}$$

and $last\_acq_l$ and $last\_rel_l$ indicate the most recent acq or rel operation for a lock l and memory access a in trace T.

Lockset analysis is fast and scalable because it uses cheap set intersections to perform its analysis. However, it is also prone to extremely high false positive rates, and the races it predicts cannot be checked automatically because it does not generate a witness schedule s^*.

Hybrid Happens-Before Lockset Therefore, lockset analysis is usually used in conjunction with happens-before analysis [17, 50, 52], which prevents false positives and generates witness schedules for each predicted race.

Appendix C. Theorem 1 Proof

Theorem 1 statement: For a threshold β, relative error bound $0 < \delta < 1$, and two access locksets α_1 and α_2 with non-intersecting locksets and random variables A_{α_1} and A_{α_2} sampled N times such that $\alpha_1, \alpha_2, \beta$ satisfy Eq. 1 and $\mathbf{P}\left[A_{\alpha_1} = 1 \cap A_{\alpha_2} = 1\right] \geq \beta$, then with probability $e^{-\delta^2 N\beta/(2-\delta)}$, the probability of a false positive is bounded by:

$$\mathbf{P}[A_{\alpha_1} = 0 \cup A_{\alpha_2} = 0] < 1 - \beta(1+\delta)$$

Proof. Let $\mathcal{A}$ be a random variable such that

$$\mathcal{A} = \begin{cases} 1 & \text{if } A_{\alpha_1} = A_{\alpha_2} = 1 \\ 0 & \text{otherwise} \end{cases}$$

and $\mu = \mathbf{E}[\mathcal{A}] < \beta$ and let $\hat{\delta} = \frac{\beta(1+\delta)-\mu}{\mu}$. Let $\mathcal{A}_i$ be sample of $\mathcal{A}$ that is obtained by independently sampling A_{α_1} and A_{α_2}. Then probability of the false positive rate exceeding $1 - \beta(1+\delta)$ for A_{α_1} and A_{α_2} is given by:

$$\mathbf{P}\left[\sum_i^N \mathcal{A}_i \geq N\beta(1+\delta)\right] = \mathbf{P}\left[\sum_i^N \mathcal{A}_i \geq N\mu(1+\hat{\delta})\right]$$

We apply the Chernoff bound [14] on μ and $\hat{\delta}$:

$$\mathbf{P}\left[\sum_i^N \mathcal{A}_i \geq N\mu(1+\hat{\delta})\right] \leq e^{-\hat{\delta}^2 N\mu/(2-\hat{\delta})}$$

From this we obtain a bound in terms of β and δ:

$$e^{-\hat{\delta}^2 N\mu/(2-\hat{\delta})} < e^{-\delta^2 N\beta/(2-\delta)}$$

□

Appendix D. Theorem 2 Proof

Theorem 2 statement: For a threshold β, relative error bound $0 < \delta < 1$, and two access locksets α_1 and α_2 with non-intersecting locksets and random variables A_{α_1} and A_{α_2} sampled N times such that α_1 and α_2 do not satisfy equation 1 and $\mathbf{P}\left[A_{\alpha_1} = 1 \cap A_{\alpha_2} = 1\right] < \beta$, then with probability $e^{-\delta^2 N\beta/2}$, the probability of a false negative is bounded by:

$$\mathbf{P}[A_{\alpha_1} = 1 \cap A_{\alpha_2} = 1] < \beta(1-\delta)$$

Proof. Let $\mathcal{A}$ be a random variable such that

$$\mathcal{A} = \begin{cases} 1 & \text{if } A_{\alpha_1} = A_{\alpha_2} = 1 \\ 0 & \text{otherwise} \end{cases}$$

and $\mu = \mathbf{E}[\mathcal{A}] > \beta$ and let $\hat{\delta} = -\frac{\beta(1-\delta)-\mu}{\mu}$. Let $\mathcal{A}_i$ be sample of $\mathcal{A}$ that is obtained by independently sampling A_{α_1} and A_{α_2}. Then probability of the false negative rate exceeding $\beta(1-\delta)$ for A_{α_1} and A_{α_2} is given by:

$$\mathbf{P}\left[\sum_i^N \mathcal{A}_i \leq N\beta(1-\delta)\right] = \mathbf{P}\left[\sum_i^N \mathcal{A}_i \leq N\mu(1-\hat{\delta})\right]$$

We apply the Chernoff bound on μ and $\hat{\delta}$:

$$\mathbf{P}\left[\sum_i^N \mathcal{A}_i \leq N\mu(1-\hat{\delta})\right] \leq e^{-\hat{\delta}^2 N\mu/2}$$

From this we obtain a bound in terms of β and δ:

$$e^{-\hat{\delta}^2 N\mu/2} < e^{-\delta^2 N\beta/2}$$

□

Appendix E. Data Races Found by PLA

Table 5 lists all of the races found by PLA in our evaluation.

Appendix F. Impact of Parameter Choices

We evaluate PLA's access lockset classification accuracy on seeds drawn from the benchmark corpus used in Section 5.2. For each seed, we vary the threshold parameter β used to classify consistent access locksets and number of samples used to estimate access lockset probabilities. We then measure on a set of test samples whether the predicted stable access locksets are present in each sample.

Figure 11 shows ROC curves that illustrate the tradeoff in True Positive Rate (ratio of predicted access locksets present in each test sample) and False Positive Rate (ratio of predicted access locksets not present each test sample) when varying the threshold parameter β for different numbers of samples, based on 5 randomly selected seed benchmarks used in 5.4. Standard deviations over the 5 seed benchmarks are also shown. Increasing the number of samples allows PLA to learn a better classifier with more consistent performance (i.e., lower std. deviation), but at a cost of increased sampling time, which we show in Section 5.6 is the most time consuming stage of PLA. In practice when running PLA we use 4 samples with $\beta = 0.5$, which provides a good tradeoff between accuracy and runtime.

Access Lockset Distribution. Figure 12 shows PLA's sampling classification on the distribution of access locksets probabilities, where access locksets with probability exceeding β are marked orange. PLA is effective because the vast majority of access locksets ($> 99.9\%$) occur with very low probability ($< 1.0\%$), therefore only predicting races when the relevant access locksets have high probability is critical to making accurate race predictions without overwhelming numbers of false positives.

Table 5: Full Listing of Races found by PLA. Note that, for the variable column, we list the macro when LLVM instrumentation failed to identify the corresponding source code variable.

ID	subsystem	variable	number of instruction pairs	category
0	kernel	variable: `ns->pid_allocated`	1	harmful
1	kernel	variable: `nr_threads`	1	harmful
2	kernel	variable: `lowest_to_date`	1	harmful
3	kernel	macro: `pr_info_once`	8	benign
4	kernel/time	macro: `printk_once`	4	benign
5	kernel/cgroup	variable: `cgrp_dfl_visible`	2	harmful
6	kernel	variable: `audit_cmd_mutex.owner`	2	harmful
7	kernel/events	variable: `sysctl_perf_event_sample_rate`	1	harmful
8	mm	variable: `pcpu_nr_populated`	1	harmful
9	mm	macro: `pr_warn_once`	21	benign
10	mm	variable: `h->resv_huge_pages`	4	benign
11	mm	variable: `h->free_huge_pages`	3	benign
12	mm	variable: `h->nr_huge_pages`	2	harmful
13	mm	variable: `h->surplus_huge_pages`	1	benign
14	mm	variable: `ksm_run`	1	harmful
15	fs	variable: `loop_check_gen`	2	harmful
16	security/keys	variable: `key_gc_next_run`	2	harmful
17	security/keys	variable: `user->qnkeys`	3	benign
18	security/keys	variable: `user->qnbytes`	4	benign
19	security/keys	variable: `ns->persistent_keyring_register`	1	harmful
20	arch/x86	macro: `alternative_call_2`	1	benign
21	drivers/pci	variable: `vga_arbiter_used`	4	harmful
22	drivers/tty	variable: `vt_dont_switch`	2	harmful
23	drivers/tty	variable: `shift_state`	1	harmful
24	drivers/tty	variable: `kbd->ledflagstate`	4	harmful
25	drivers/tty	variable: `kbd->kbdmode`	6	benign
26	drivers/tty	variable: `kbd->default_ledflagstate`	4	benign
27	drivers/tty	variable: `kbd->modeflags`	2	benign
28	drivers/tty	variable: `do_poke_blanked_console`	1	harmful
29	drivers/tty	variable: `want_console`	1	harmful
30	drivers/char	variable: `last_value`	2	benign
31	drivers/base	variable: `fw_fallback_config.loading_timeout`	4	harmful
32	drivers/misc	variable: `context->notify`	1	harmful
33	drivers/scsi	macro: `pr_err_once`	6	benign
34	drivers/net	variable: `crc_force`	3	harmful
35	drivers/input	variable: `input_devices_state`	1	harmful
36	sound/core	variable: `card_requested[card]`	2	harmful
37	sound/core	variable: `client_usage.cur`	2	benign
38	sound/core	variable: `client_usage.peak`	1	benign
39	sound/core	variable: `num_queues`	2	harmful
40	sound/core	variable: `max_midi_devs`	1	harmful
41	net/core	variable: `warned`	3	harmful
42	net/llc	variable: `llc_ui_sap_last_autoport`	2	benign
43	net/netfilter	variable: `table->handle`	2	harmful
44	net/ipv4	variable: `tcp_md5sig_pool_populated`	1	harmful
45	net/ipv4	variable: `challenge_timestamp`	2	harmful
46	net/ipv4	variable: `ca->flags`	5	harmful
47	net/xfrm	variable: `idx_generator`	3	harmful
48	net/xfrm	variable: `aalg_list[i].available`	22	harmful
49	net/xfrm	variable: `ealg_list[i].available`	21	harmful
50	net/xfrm	variable: `calg_list[i].available`	4	harmful
51	net/unix	variable: `user->unix_inflight`	2	harmful