A Sparse Learning Approach for Linux Kernel Data Race Prediction

Gabriel Ryan

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2024

Abstract

A Sparse Learning Approach for Linux Kernel Data Race Prediction

Gabriel Ryan

Operating system kernels rely on fine-grained concurrency to achieve optimal performance on modern multi-core processors [1]. However, heavy usage of fine-grained concurrency mechanisms make modern operating system kernels prone to data races, which can cause severe and often elusive bugs. In this thesis, I propose a new approach to identifying data races in OS Kernels based on learning a model to predict which memory accesses can be feasibly executed concurrently with one another. To develop an efficient learning method for memory access feasibility, I develop a novel approach based on encoding feasibility as a boolean indicator function of system calls and ordered memory accesses. A memory access feasibility function encoded this way will have a naturally sparse latent representation due to the sparsity of interthread communications and synchronization interactions, and can therefore be accurately approximated based on a small number of observed concurrent execution traces.

This thesis introduces two key contributions. First, Probabilistic Lockset Analysis (PLA), is a new analysis that exploits sparsity in input dependencies in conjunction with a conservative lockset analysis to efficiently predict data races in the Linux OS Kernel. Second, approximate happens-before analysis in the fourier domain (HBFourier) generalizes the approach used by PLA to reason about interthread memory communications and synchronization events through sparse fourier learning. In addition to being theoretically grounded, these techniques are highly practical: they find hundreds of races in a recent Linux development kernel, an order of

magnitude improvement over prior work, and find races with severe security impacts that have been overlooked by existing kernel testing systems for years.

Table of Contents

List of Figures

List of Tables

Acknowledgements

This work would not have been possible without the mentorship and guidance of my advisor, Suman Jana. During my time in graduate school I have had the pleasure of working with many incredible research mentors including Salvatore Stolfo, Baishahki Ray, Todd Mytkowicz, Shuvendu Lahiri, Ronghui Gu, and Eugene Wu, and many outstanding student collaborators including Abhishek Shah, Dongdong She, Elizabeth Dinella, Jianan Yao, Justin Wong, Preetam Dutta, and Abigail Mosca. Thank you all.

Dedication

To my wife Julia.

Chapter 1: Introduction

Operating system kernels have evolved to heavily rely on fine-grained concurrency to achieve optimal performance on modern multi-core processors [1]. By allowing many threads to execute simultaneously and access shared system resources concurrently, these kernels can efficiently utilize the computational resources of multi-core processors. However, the same fine-grained concurrent synchronization techniques that make modern operating system kernels efficient also make them particularly prone to race conditions. Race conditions occur when multiple threads concurrently access shared resources without synchronization, leading to unpredictable and erroneous behavior [2].

Research Problem. This thesis focuses on identifying a specific form of race conditions called data races, which occur when two memory accesses are performed concurrently to the same memory address. Identifying data races in operating system kernels is crucial due to their potential to introduce severe and often elusive bugs. These bugs can manifest in various forms, from system crashes and memory corruption to more insidious errors that result in security vulnerabilities. For instance, data races can lead to information leaks and privilege escalation attacks, which can compromise the integrity and confidentiality of a system [3, 4, 5, 6]. The complexity of kernel code and the non-deterministic nature of race conditions make them particularly challenging to diagnose and rectify, posing a continuous threat to system stability and security. Therefore, identifying and preventing data races in operating system kernels is vital for ensuring the security and reliability of modern computing systems.

Data race prediction for operating system kernels is exceptionally challenging due to the exponential space of potential system call combinations and concurrent thread interleavings that can give rise to data races [7]. To successfully identify a data race, finding a combination of system calls can race when executed concurrently is not sufficient–it is also necessary to identify the pre-

cise thread interleaving required for these system calls to execute racing memory accesses to a common address. This dual requirement makes data race prediction for kernel code challenging both in theory and practice. From a theoretical perspective data race prediction is NP-Hard, since there is an exponential space of system inputs and thread interleavings that can be executed [8]. As a result, practical solutions often heavily rely on specific heuristics, which can be brittle and limited to targeting specific classes of data races. Previous research has shown the prevalence of undiagnosed data races during development [9].

Prior Work. There are two predominant approaches for testing for data races: schedule exploration and dynamic race prediction. The former focuses on executing many different thread interleavings to detect data races. Schedule exploration approaches include systematically enumerating schedules based on the number of allowed preemptions [10], randomly sampling schedules from a targeted distribution [11, 12], or attempting to maximize concurrent coverage metrics [13]. In contrast, dynamic race prediction, rather than executing different schedules, reasons about potential memory access reschedulings based on a single execution trace of a concurrent program. Most dynamic race prediction methods use either analysis based on Lamport's happens-before relation [14] to predict races *soundly* (i.e., no false positives), or lockset analysis [15], which is *complete*(i.e., no false negatives) but prone to high false positive rates. SMT reductions have been constructed for dynamic race prediction, which are both sound and complete for an observed trace, but have limited scalability [16].

Systems designed for kernel race testing use a combination of schedule exploration and dynamic race prediction, but in a two-step manner. First, sequences of system calls are generated using a template-based fuzzer such as Syzkaller that maximizes code coverage. The corpus of generated fuzzer seeds is then used as inputs to the race testing procedure. Combinations of fuzzer seeds are selected from the corpus and executed together, and schedule exploration and dynamic race prediction are used to test for races. This seed selection is based on either alias analysis of the memory accesses performed by each seed [17, 7], or sampling diverse seed combinations based on a concurrent coverage metric [9, 18]. Schedule exploration and dynamic race prediction are

then used on each seed combination to test for races that can potentially occur. However, identifying possible racing memory accesses across different seed combinations is challenging, due to high false positive rates from alias analysis and variability in execution paths when seeds run concurrently, and existing approaches miss many data races as a result (See Section 2.5.2).

Approach. In this thesis, I propose a new approach to program analysis through sparse learning methods and apply it to data race prediction for OS Kernels. The core of the approach involves encoding the analysis into a query to a boolean indicator function f for a relevant program execution property (e.g., a given memory access being feasible), and defining two functions that are used to learn a model of f: a *trace encoding function* ϕ, and an *approximation function* f^*. ϕ defines a set of features that are used to learn a model of f by encoding a trace into a boolean vector, and f^* is used to learn an approximate model of f based on the encoding defined by ϕ. If we consider a program that operates on an input $x \in X$ to produce a trace $\tau \in T$, where X and T represent the sets of all possible inputs and traces, then f, ϕ, and f^* have the following type definitions:

$$f : X \times T \to Z_2 \qquad \phi : X \times T \to Z_2^n \qquad f^* : Z_2^n \to \mathbb{R} \tag{1.1}$$

Given an input x and corresponding execution trace τ, $f(x, \tau) = f^* \circ \phi(x, \tau)$. The objective is then to approximately learn f^* from a set of observed traces.

The approximation function f^* can then be used to reason about likely values of f for hypothetical program inputs x^*. The size of the input domain n of f can be very large for practical program analysis problems on a large real-world program like an OS kernel, but if a basis for f is known where the function has a sparse latent representation, the approximation f^* can be learned in this basis for a small fixed cost with strong probabilistic guarantees under mild uniform sampling assumptions [19]. This framework can be applied to many forms of program analysis through careful design of the trace mapping function ϕ and indicator function f. For example, ϕ can encode which interthread communications are observed in an execution trace, and f can indicate if a particular memory access of interest is executed in the trace (e.g., a memory access

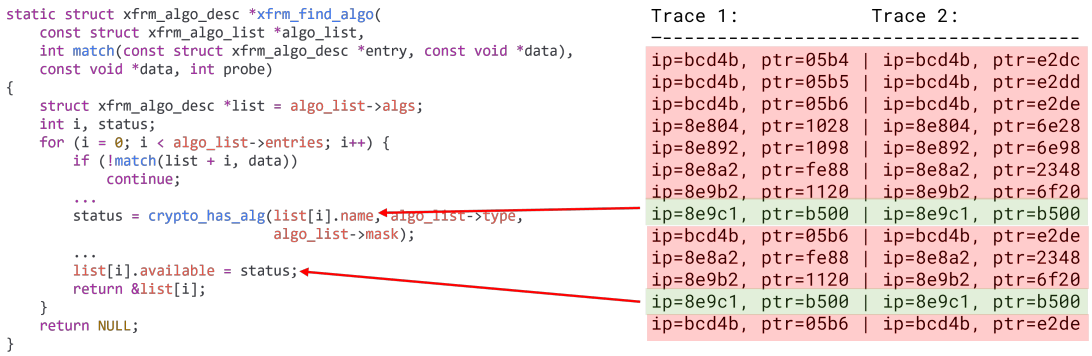


Figure 1.1: Example of sparsity in shared memory accesses in the Linux Kernel. Less than 1 in 1000 memory accesses are consistently performed to specific addresses over multiple executions. These represent accesses to shared data structures that form points of communication with other executing threads.

that could be involved in a data race). The learned approximation function f^* can be then used to predict which interthread communications could cause a potential data race to occur.

This approach is based on the observation that many seemingly computationally intractable problems in program analysis can be solved for practical real-world programs due to intrinsic sparsity in program structure. For example, the memory accesses in the kernel exhibit sparsity in their accesses to common addresses. A single system call may perform millions of memory accesses when it is executed, but only a very small fraction of these will access shared data structures that are points of potential communication with other executing threads. Figure 1.1 illustrates how only a small fraction of memory accesses performed by a method in the Linux `network/xrfm` subsystem access shared datastructures that do not change over multiple executions. Learning which memory accesses are involved in communications and targeting these can therefore reduce the cost of a pairwise analysis by a factor of millions (See Section 2.5.3 for detailed results).

Leveraging Sparsity in Race Prediction. Learning to predict which memory accesses are feasible to execute concurrently is a fundamentally hard problem due to the exponential number of possible interactions between synchronization primitives and inter-thread communications. However, the interactions between individual synchronization events and inputs for a set of concurrently executing threads exhibit the property that they are intrinsically sparse, since most instructions executed by the kernel do not involve explicit communication or synchronization. Prior work in race

prediction has observed that the sparsity in concurrent interactions means that many races can be exposed by limiting the search over the schedule space to a small number of thread preemptions, and focusing the search to testing schedules with fewer preemptions first. Chess [20] does this with a model-checking approach by enumerating possible thread interleavings to test for a concurrent program, starting with interleavings with only a single preemption, and then testing schedules with progressively more interleavings. PCT [11] uses this same approach in a randomized way, by sampling execution schedules to test from a distribution biased towards a small number of preemptions.

In this work, I leverage sparsity in inter-thread communications to make learning an accurate model of feasibility for concurrent memory accesses tractable. When encoded as a boolean indicator function f, a memory access feasibility model will have a sparse latent representation in an appropriate basis (e.g., the function's fourier expansion) due to the sparsity of interthread communications and synchronization interactions. Feasibility of memory accesses can therefore be accurately approximated by learning a small set of relevant memory communications that affect when specific memory accesses are feasible or infeasible. These relevant communications can then be learned based on a small number of observed concurrent execution traces.

Outline. The rest of this thesis is organized as follows. First, in Chapter 2, I introduce Probabilistic Lockset Analysis (PLA), an analysis that exploits sparsity in input dependencies in conjunction with a conservative lockset analysis to efficiently predict data races in the Linux OS Kernel. Second, in Chapter 3, I introduce HBFourier, which generalizes the approach used by PLA to reason about interthread memory communications and synchronization events through sparse fourier learning, and show that PLA can be expressed as a sparse fourier learning problem on memory access input dependencies. In addition to being theoretically grounded, these techniques are highly practical: they find hundreds of races in a recent Linux development kernel, an order of magnitude improvement over prior work, and find races with severe security impacts that have been overlooked by existing kernel testing systems for years.

Chapter 2: Probabilistic Lockset Analysis

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× higher rate over 24 hours. We use PLA to find 183 races, including 102 harmful races, in linux kernel v5.18-rc5, the most recent main line development kernel version at the time the experiments were conducted. 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.

2.1 Introduction

Data races are a source of serious security vulnerabilities in the OS kernels–many recent data-race-based exploits resulted in privilege escalation [5], denial of service [4], and leaking protected memory [21, 6]. Recent work has demonstrated that even races that appear unexploitable might be deterministically triggered by an attacker [22]. Moreover, even when data races do not immediately result in security vulnerabilities, they cause severe bugs that lead to memory corruption and undefined behavior [3, 23].

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 [8] 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 [24].

In general, there are two widely used approaches to search for races in arbitrary concurrent programs: *schedule exploration* searches by executing many different schedules [10, 13, 25], while *dynamic race prediction* reasons about possible reschedulings of memory accesses subject to synchronization to trigger races based on a single concurrent execution trace [26, 27, 15, 16]. 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 [17, 7] or a coverage metric for concurrent executions [9, 18], 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 thesis, 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 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 2.3.2 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 [28], 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 2.5.6). 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× 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 [29].

In summary, this thesis 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× 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.

[1] `www.github.com/gryan11/PLA`

(a) Function with racing global variable update.

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

Figure 2.1: Simplified example of race found by PLA in `net/netfilters/`. The race occurs on the global variable `global_handle` shown in 2.1a, which can be concurrently modified by multiple threads when passed different `net` structs (which each have different per-thread `net->mutex`). 2.1b 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 2.2: Simplified kernel fuzzer seed used to trigger the race shown in Figure 2.1. The seed opens a socket and then sends a message that executes the `nf_newtable` function.

2.2 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.

2.2.1 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 [30]. Figure 2.1 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 2.1a that is guarded by mutex in a `net` struct. The memory access pair and their respective system calls are shown in 2.1b, 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 2.3 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 2.3, 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 2.2 shows an example syzkaller fuzzer seed that triggers the race shown in Figure 2.1. Kernel concurreny testing systems generate corpuses of kernel fuzzer seeds by either running a single threaded fuzzer and maximizing branch coverage [17, 7], or using concurrency specific coverage metrics [9, 18].

```
   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 2.3: 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.

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.

2.2.2 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.1 and A.2.

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 2.3 to avoid false positive predictions, with *lockset analysis*, which identifies locking violations such as the non-overlapping mutexes bug shown in Figure 2.1. 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 2.5.2).

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 [17] or dynamically [7]. 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,

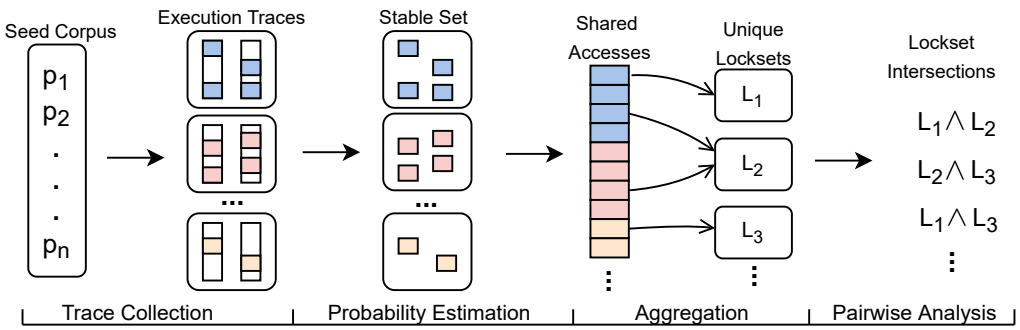


Figure 2.4: PLA's workflow. PLA collects traces independently from each seed in the corpus to identify its stable set of memory accesses. It then groups all memory accesses first by memory address and then by unique locksets, and performs pairwise intersections on the aggregated locksets. This procedure 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.

see Figure 2.3), 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 2.5.2).

2.3 Methodology

In this thesis, 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.

2.3.1 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 2.5.6).

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, 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 2.4 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 2.2.2). 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

Hybrid Dynamic Race Prediction on Corpus:

```
// Check O(n^2) 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 2.5: 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 2.3.2 for precise definitions of traces, α, and A_α.

random variable is estimated by independently sampling traces from each input, this only requires $O(n)$ traces for n seeds. Figure 2.5 illustrates the difference between PLA and hybrid race prediction when run a corpus of fuzzer seeds.

2.3.2 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 2.3.3. 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:

$$A_\alpha = \begin{cases} 1 & \text{if } \alpha \in \text{trace}(p_1, p_2, s) \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

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 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:

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

We evaluate settings of β in Section 2.5.5 and find that in practice setting β to 0.5 achieves accurate race predictions. Since β determines the set of potential memory accesses and races we consider for testing, if more compute resources are available, β can be lowered to increase the size of the stable set and potentially find more races, at a cost of lower prediction accuracy and more false positive predictions that need to be tested. Similarly, if compute resources are limited, β can be increased to reduce the size of the stable set and prioritize race predictions that are more accurate to test first.

Making predictions on the stable set drastically reduces the cost of PLA's analysis and makes it more accurate, since any pair of stable accesses are likely to have a feasible concurrent schedule (see evaluation in Section 2.5.3).

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:

$$\alpha_1.lockset \cap \alpha_2.lockset = \varnothing \ \wedge \ \alpha_1, \alpha_2 \in \mathcal{S} \tag{2.2}$$

Witness Schedule. Given two accesses α_1 and α_2 that satisfy Eq. 2.2 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:

$$\text{PLA-race-prediction} := (\alpha_1, \alpha_2, p_1, p_2, s) \tag{2.3}$$

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. 2.2 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 A.3 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 2.2 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 A.4 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 2.5.5).

2.3.3 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 2.5.4 and show that ablating any one of them prevents PLA from scaling effectively.

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 2.5.5).

For each access lockset α, we estimate the probability $\mathbf{P}\left[A_\alpha = 1 \mid p_1 = p\right]$ based on the exe-

Algorithm 1 Access Lockset Construction.

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

```
1: sample_accesses = {}
2: held_locks = {}
3: for t ∈ trace(p1, p2, s) do
4:     if is_lock_acquire(t) then
5:         held_locks = held_locks ∪ {t.lock_addr}
6:     if is_lock_release(t) then
7:         held_locks = held_locks\t.lock_addr
8:     if is_memory_access(t) then
9:         α = (t.ip, t.m, t.op, held_locks)
10:        sample_accesses = sample_accesses ∪ α
11: return sample_accesses
```

cution 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.

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.

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

Algorithm 2 Access Lockset Probability Estimation.

Input: $p \leftarrow$ Seed
$\mathbb{P} \leftarrow$ Seed Corpus
$N \leftarrow$ Seed Sample Count
$\beta \leftarrow$ Race Prediction Threshold

$\mathbb{M}_p$ = hashmap(default = $\varnothing$)
$access\_counts$ = hashmap(default = 0)
for $i \in \{1..N/2\}$ **do**
    p' = choose_random($\mathbb{P}$)
    for $s \in \{p\_first, p'\_first\}$ **do**
        $sample\_accesses$ = 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 $\mathbb{M}_p$

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:

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

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

Algorithm 3 Whole Corpus PLA.

Input: $\mathbb{P} \leftarrow$ Seed Corpus
$N \leftarrow$ Seed Sample Count
$\beta \leftarrow$ Race Prediction Threshold

$\mathbb{M}$ = hashmap(default = $\varnothing$)
for $p \in \mathbb{P}$ **do**
    $\mathbb{M}_p$ = 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

    $\mathbb{L}$ = 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} = select_races($accs_1, accs_2, C_m$)
            R_{new2}, C_{new2} = 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}

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}

must exceed β). Algorithm 4 describes this procedure.

2.4 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`) [31] 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 [32].

When tracing we use a modified syzkaller [33] 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 2.5.6).

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.

2.5 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.

2.5.1 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 [7].

Results. Table 2.1 summarizes the results with full details in Table A.1 in Appendix A.5. 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 [9] as well as the number of unique racing pairs of instructions based on Conzzer [18]. For a concrete example, race ID 48 from Table A.1 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. [9]. 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

Table 2.1: 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 [9]. We provide a full listing of races in Table A.1 in Appendix A.5.

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

(CVE-2022-3028) has been allocated based on our reports [29].

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 2.6 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 [29].

Use after free in mm. Figure 2.7 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.

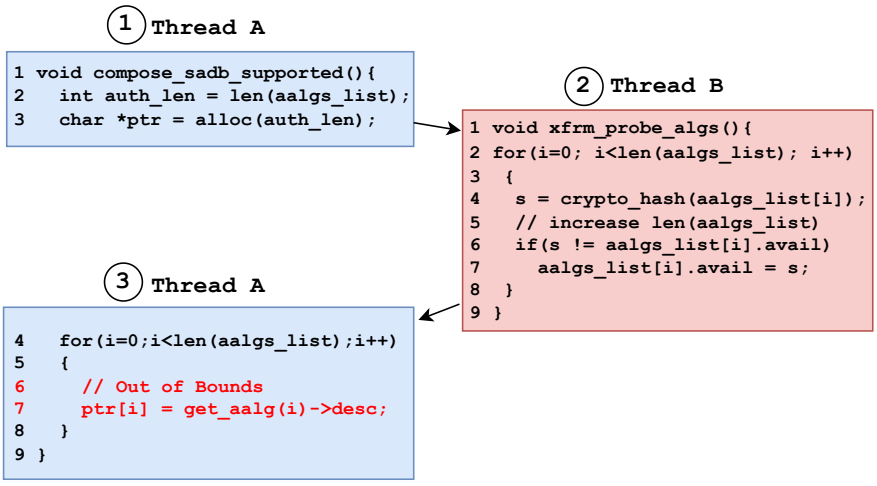


Figure 2.6: **A harmful data race in the net/xfrm kernel subsystem involving the `aalg_list[i].available` variable (ID 48 in Table A.1). The numbers (1), (2), (3) indicate the order of events in the data race that leads to an out-of-bounds write vulnerability.**

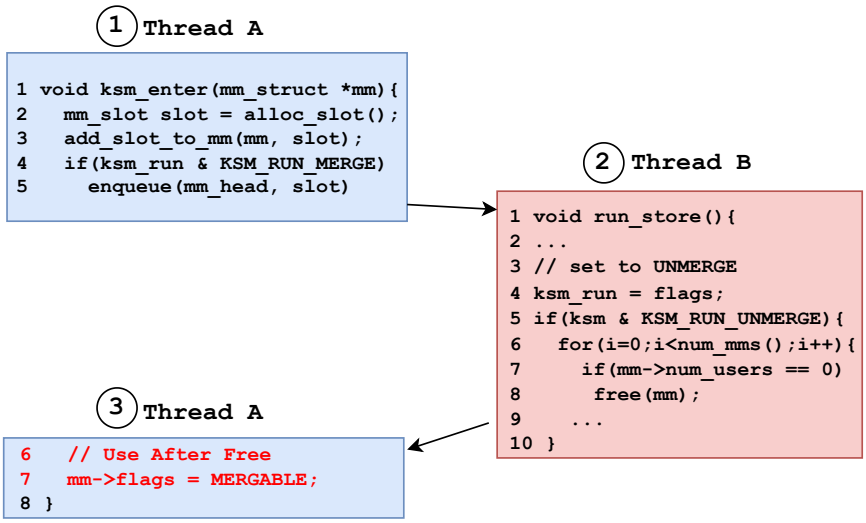


Figure 2.7: **A harmful data race in the mm kernel subsystem (ID 14 in Table A.1). This data race leads to a use-after-free vulnerability.**

2.5.2 Comparison with other Approaches

We evaluate PLA against other recent systems that target data race detection in the kernel based on their ability efficiently find kernel races with a 24 hour time budget.

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 [9] and Conzzer [18] are two recent kernel concurrency fuzzers.

Krace is open sourced [34], 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 [35]. We emailed the Krace authors to report the issue but did not receive a response. Conzzer has a binary-only release available from [36]. 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 [9] 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 [17] and Snowboard [7]. 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 [33], a standard kernel fuzzer that performs multithreaded fuzzing, using the kernel concurrency sanitizer (`kcsan`) [31], a watchpoint-based data race detector that is deployed for continuous linux kernel testing [37].

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.

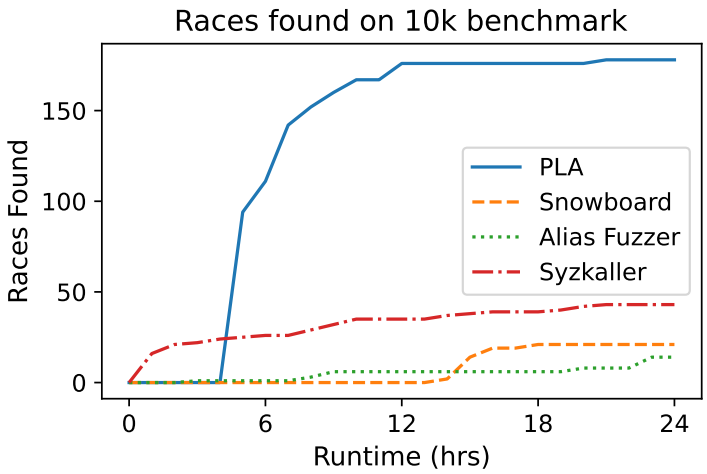


Figure 2.8: Evaluation of races found over five 24hr runs on benchmark of 10k minimized seeds. On average, PLA finds 164 races in total, Snowboard 21, Alias Fuzzer 15, and Syzkaller with Kcsan finds 43.

Results and Discussion

Figure 2.8 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.

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 [9], 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 [18] and finding specific races associated with CVEs [17]. 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`.

2.5.3 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 2.5.2) because the extremely high error rates of standard

Table 2.2: 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.

		# Seeds in Benchmark				
Metric	Approach	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

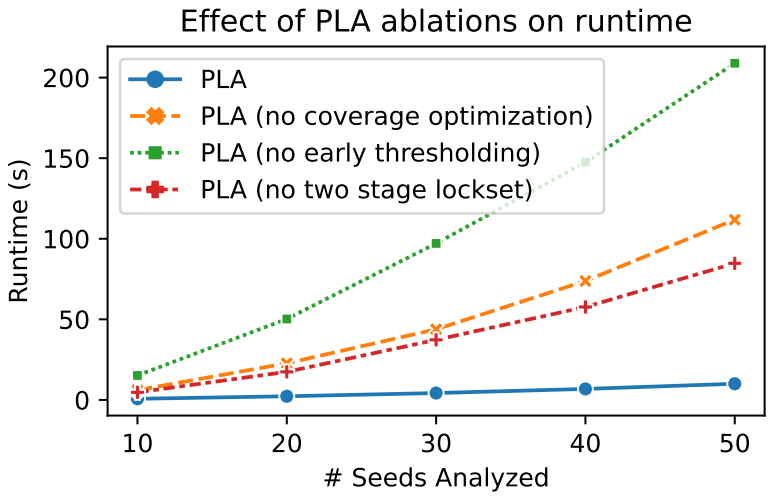


Figure 2.9: Impact of ablations on analysis runtime averaged over 5 randomly sampled benchmarks. On benchmarks of 50 seeds, ablations increase PLA's runtime between 8.5× and 21× and cause the analysis to scale superlinearly in the number of seeds.

lockset analysis make testing it on even small benchmarks prohibitively time consuming.

PLA vs. Lockset Analysis. Table 2.2 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 2.2 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 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 2.5.2 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.

2.5.4 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 2.9 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× 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× and 8.5× 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.

Table 2.3: 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

2.5.5 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 2.5.2. 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 2.3 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 β=0.5 setting for all experiments. See Appendix A.6 for detailed results.

2.5.6 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 2.5.2. As described in Section 2.4, PLA works in 3 states: tracing and sampling, race prediction analysis, and race checking. Table 2.4 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 2.4 illustrate that two design optimizations in PLA (Section 2.3.3) are

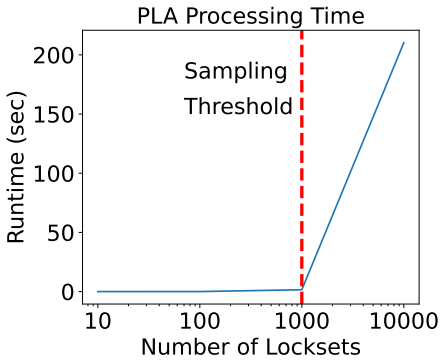
Table 2.4: Input sizes and runtimes for PLA on 10k inputs.

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

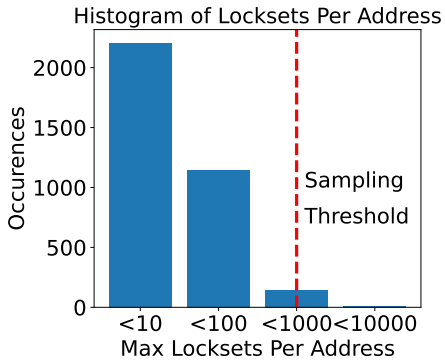
absolutely critical to its performance: 1.) Applying probability thresholding during initial trace collection reduces the number of events that must handled by the subsequent, more expensive, stages of the analysis by a factor of over 100. 2.) Applying coarse grained linear lockset analysis before running 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 2.10a. 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 2.4.

We found that only a very small number of memory addresses with lock violations have more than 1000 locksets. Figure 2.10b 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 [38], harmful data races usually involve rarely accessed memory, and all of the harmful races we found involved infrequently used memory addresses.



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



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

Figure 2.10: 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 2.10b.

2.6 Related Work

Dynamic Race Prediction Dynamic race prediction identifies possible data races based on concurrent program execution traces. Happens-before methods reason about partial orders on traces based on Lamport's happens-before relation on interthread communications to predict races soundly [14, 39, 40, 26]. Extensive work has focused on developing weakened partial orders that soundly predict more races from a trace [41, 42, 43, 44, 45, 27], or using SMT reductions, which are sound and complete with regard to the observed trace but limited in scalability [16, 46, 47, 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 [30, 15]. Therefore, many race predictors such as RaceTrack and Goldilocks combine happens-before and lockset analysis to make precise lockset-based race predictions [50, 51, 28, 52, 53]. 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 [20,

10], sampling a distribution of schedules [11, 12], fuzzing with a concurrency specific coverage metric [13], or performing targeted exploration of schedules based on static alias analysis [54] 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 [25] and schedule exploration by sampling a distribution of schedules [55]. 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 [17, 7]. Concurrency fuzzers use a concurrency specific coverage metric to guide schedule fuzzing in conjuction with dynamic race predictors to detect observed races [9, 18]. 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.

2.7 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.

2.8 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× 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.

Chapter 3: Spectral Race Prediction with HBFourier

Testing for data races in the Linux OS kernel is challenging because there is an exponentially large space of system calls and thread interleavings that can potentially lead to concurrent executions with races. In this work, we introduce a new approach for modeling execution trace feasibility and apply it to Linux OS Kernel race prediction. To address the fundamental scalability challenge posed by the exponentially large domain of possible execution traces, we decompose the task of predicting trace feasibility into independent prediction subtasks encoded as learning Boolean indicator functions, and apply a sparse fourier learning approach to learning each feasibility subtask.

Boolean functions that that are sparse in their fourier domain can be efficiently learned by estimating the coefficients of their fourier expansion. Since the feasibility of each memory access depends on only a few other relevant memory accesses or system calls (e.g., relevant inter-thread communications), trace feasibility functions have this sparsity property and can be learned efficiently. We use learned trace feasibility functions in conjunction with conservative alias analysis to implement a kernel race-testing system, HBFourier, that uses sparse fourier learning to efficiently model feasibility when making predictions. we evaluate our approach on a recent Linux development kernel and show it finds 44 more races with 15.7% more accurate race predictions than the next best performing system in our evaluation.

3.1 Introduction

Modern operating system (OS) kernels heavily rely on fine-grained concurrency to achieve optimal performance by utilizing the parallelism of multi-core processors [1]. However, the use of fine-grained concurrent synchronization can lead to race conditions, which are errors that occur due to multiple threads accessing shared resources concurrently [2]. In this work, we focus on

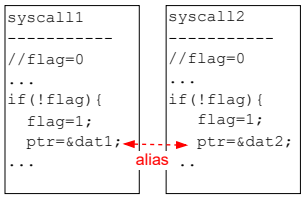
identifying data races, which are race conditions that occur in shared memory accesses. Data races in OS kernels can result in difficult-to-diagnose bugs that can cause various issues, such as crashes, memory corruption, and even security vulnerabilities like information leaks and privilege escalation attacks [3, 4, 5, 6].

Testing for data races in an OS kernel is a very challenging problem because there is an exponential space of possible system call combinations and concurrent thread interleavings that can result in races [7]. To expose a data race, one not only has to identify the correct combination of system calls to execute concurrently, but also to identify the specific thread interleaving necessary for those system calls to execute concurrently and lead to racing accesses. This requirement makes finding data races in the kernel a challenging problem both in theory and practice. From a theoretical perspective, race prediction is NP-hard [8], and therefore practical solutions often depend heavily on different brittle heuristics. Prior work has shown that many data races go undiagnosed during development and are patched later [9].

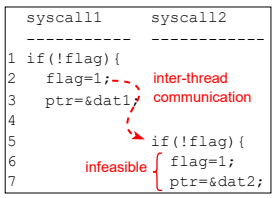
Kernel Race Prediction. Concurrent executions in the kernel typically arise from different user-land processes invoking various sequences of system calls. As a result, most practical kernel race detection systems use a two-step approach to search the vast system-call/interleaving space for races. Firstly, they generate a diverse corpus of system calls, known as kernel fuzzer seeds, using a code coverage-guided kernel fuzzer, such as syzkaller [33]. Secondly, they collect execution traces for those seed sequences of system calls and predict races based on shared memory accesses in the traces.

These predictions are usually based on identifying memory accesses that are aliased (i.e., access common addresses in the traces), and then prioritizing predictions on either aliases that occur infrequently in the trace set [7], or aliases that have a high estimated probability of being data races [56]. However, since alias analysis might produce many false positive predictions, each prediction is verified by executing the associated inputs together and attempting to force a race by controlling the executed thread interleavings.

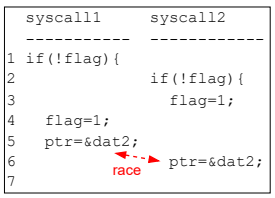
Limitations of Prior Work. However, the alias-analysis-based approaches mentioned above used



(a) Traces with aliased accesses.



(b) Interleaving 1: Infeasible data race.



(c) Interleaving 2: Feasible data race.

Figure 3.1: Example of a predicted race on that is only feasible under specific thread interleavings. Although the two traced memory accesses both access the same shared variable `ptr` in 3.1a, a race is infeasible in the thread interleaving in 3.1b due to an inter-thread communication. The thread executing `syscall2` cannot execute the predicted racing access to `ptr` on line 7 because the `syscall1` thread sets `flag=1` before the `if` check on line 5. However, if both `if` checks are executed first as shown in 3.1c, the race on `ptr` is still feasible.

for kernel race detection do not consider whether a thread interleaving in which a race occurs is feasible for the kernel. In practice, inter-thread communications can affect many memory accesses when different sequences of system calls are executed together, causing them to access different addresses or not be accessed at all. To illustrate this issue, consider the example shown in Figure 3.1, which demonstrates how inter-thread communications can make certain data races infeasible.

In the example, two system calls perform aliased accesses to a common shared variable `ptr` shown in 3.1a. If the two system calls are executed together according to a thread interleaving shown in 3.1b that causes the accesses to `ptr` to race, the second racing memory access on line 7 is not executed after the `if(!flag)` check on line 5 because `flag` is already set to 1 on line 2. However, the race on `ptr` is still feasible under a different thread interleaving shown in 3.1c, in which both `if(!flag)` checks are performed before either input thread sets `flag=1`. Therefore, correctly identifying the race on `ptr` requires accounting for inter-thread communication and deriving a feasible racing thread interleaving. Not accounting for inter-thread communications in prediction causes both false negatives when predicted races cannot be reproduced without a specific interleaving, as well as false positives when any racing interleaving is infeasible.

Happens-Before & its Limitations. The issue of interleaving feasibility is often tackled in the race prediction literature using Lamport's happens-before partial order [14]. Happens-before analysis determines the feasibility of reorderings of memory accesses in a single observed concurrent trace

by preserving the order of read-write accesses to shared addresses in the observed trace.

For instance, consider a scenario in which memory access on thread 2 can only occur after a shared variable has been written to by thread 1 due to an if-check in thread 2 (see Figure 3.1b). In such a case, a happens-before analysis on the trace shown in Figure 3.1b will correctly detect the race as infeasible for reorderings for this trace as it will maintain the same order of interthread communications as the observed trace. However, the race with the same system calls is feasible for another trace as shown in Figure 3.1c. From a bug detection perspective, what matters is whether the race is possible for any trace, not just for a single observed trace. A naive solution to address this issue is to run happens-before on a large number of traces, but this approach will still miss races because only a small fraction of the exponential number of possible thread interleavings and system call combinations can be tested for a realistic computational budget.

Our Approach. To address this challenge, in this work, we propose to approximately learn a function that can predict which execution traces will be feasible in a hypothetical interleaving based on a decomposition of the executing system calls, operations, and order of events in a trace. We present a sparse fourier learning that approach efficiently learns decomposed feasiblity functions from a set of observed traces to estimate their fourier expansions, and then reconstructs a combined trace feasibility function based on a convolution of the decomposed functions fourier expansions. We combine this trace feasibility estimation with a conservative alias analysis that identifies potential data races to perform highly accurate data race prediction, although we note there are other potential uses for trace feasibility estimation (e.g., prioritizing bug warnings for manual inspection.)

We model trace feasibility as a boolean function that outputs 1 if a memory access is feasible for a given interleaving. Each input of the function corresponds to a partial order appearing in the given interleaving (e.g., 1 if `r(x)` appears before `w(x)` in the interleaving, 0 otherwise). As the space of possible traces is very large (exponential in the number of memory accesses), learning such a function in the general case is very challenging. However, we notice that such functions tend to be sparse, i.e., the feasibility of any individual memory access in an interleaving depends

on only a few partial orders (e.g., relevant inter-thread communications, see Section 3.6). This implies that trace feasibility functions are often close to *fourier sparse*, a class of functions that can be efficiently learned with a small number of samples even if their domain is very large as they have a small number of nonzero coefficients in their Fourier domain [19]. Therefore, we learn trace feasibility functions by estimating their Fourier coefficients from a set of observed traces.

We use our sparse fourier learning approach to implement HBFourier, a kernel race prediction approach that directly models the feasibility of traces for its race predictions. HBFourier is able to find hard-to-detect races that only occur under specific thread interleavings, as well as prevent false positive predictions that appear due to ignoring inter-thread communications. We evaluate HBFourier against recent kernel race prediction approaches on traces from a corpus of 10,000 inputs and show its predictions are 15.7% more accurate and it finds 44 additional races.

In summary, we make the following contributions:

1. We propose to learn concurrent trace feasibility, a generalization of Lamport's Happens-Before partial order to reason about the feasibility of memory accesses appearing under hypothetical thread interleavings.

2. We introduce HBFourier, a new kernel race prediction approach that uses sparse Fourier learning to efficiently model feasibility for its race predictions. We are working on an open source release of HBFourier and make it available for anonymous review[1].

3. We evaluate HBFourier against other kernel race prediction approaches and show it's predictions are 15.7% more accurate and it finds 44 additional races.

3.2 Problem Setting

3.2.1 Multi-Input Program Traces

Multi-Input Execution Traces. Kernel race prediction can be expressed as a more general problem of predicting races for a program $\mathcal{P}$ that takes multiple inputs and executes each input on a

[1] `https://osf.io/z2fra/?view_only=e6ba5524b4fd416da10a3036f70e88a0`

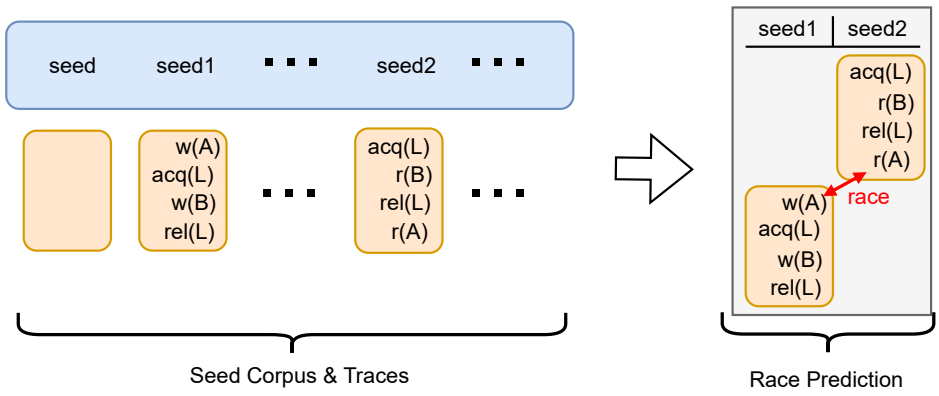


Figure 3.2: Race prediction on the kernel based on traces of corpus of fuzzer seeds. Correctly identifying a race requires both identifying the racing accesses and a feasible thread interleaving that can be used to reproduce the race.

separate thread. We base our definitions of traces and events on common usage in the dynamic race prediction literature [27, 57], but extend them to programs that concurrently execute a set of inputs. The program $\mathcal{P}$ executes a set of inputs on separate threads to generate a trace E of execution events:

$$E = e_0, e_1, e_2, \ldots$$

Each event in E is defined as a tuple of four elements: a unique position in the trace i, the thread that executed the event t, the input x executed on thread t, and an operation $\texttt{op}$.

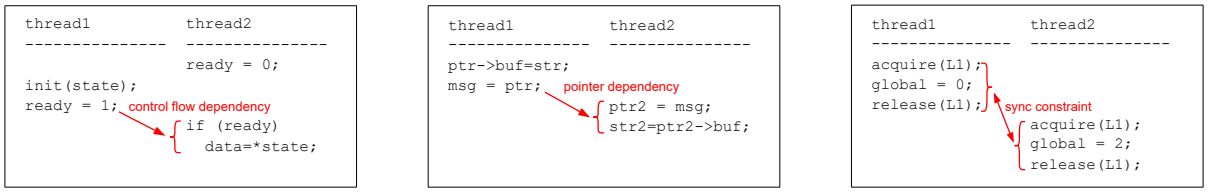
$$e = \langle i, t, x, \texttt{op} \rangle \tag{3.1}$$

An operation is either a memory access α or synchronization σ, where each is composed of a tuple of either a memory access instruction a and address m, or synchronization instruction s and lock l:

$$\texttt{op} := \alpha \mid \sigma$$

$$\alpha := \langle a, m \rangle \qquad a := w \mid r$$

$$\sigma := \langle s, l \rangle \qquad s := acq \mid rel$$



(a) Control flow data dependency. (b) Memory address data dependency. (c) Synchronization constraint.

Figure 3.3: Examples of control flow data dependencies, memory address data dependencies, and synchronization constraints that occur in concurrent programs. In 3.3a, the thread 2 read from `*state` has a control flow dependency on the global variable `ready`, which must be set by thread 1 before thread 2 read from `*state`. This prevents a race with the thread 1 `init(state)`. Similarly, in 3.3b, the thread 2 read from `ptr2->buf` is dependent on `ptr2` being read from `msg`, which is set by thread 1. In order for thread 2 to access the `ptr->buf` written to by thread 1, it first has to read the pointer address is that is communicated through the shared variable `msg`. This prevents the thread 1 write to `[ptr]` and thread 2 read from `ptr2` from racing. In 3.3c, accesses to the shared variable `global` are both guarded by a common lock `L1`, which prevents the accesses from racing.

Access instructions are either reads or writes (w or r), and synchronization instructions are either acquire (acq) or release (rel).

To simplify notation, we refer to memory access and synchronization events as simply α and σ, respectively, instead of $e.\alpha$ and $e.\sigma$ when it is clear from the context.

Notation. We use the following notation with regard to execution traces.

- *Set Definitions.* We use $\{i..j\}$ as shorthand for defining a set $\{i, i+1, ..., j\}$.
- *Sets of sets.* We use $\mathbb{G}$ to refer to a set of sets. $\mathbb{G}$ may be a *powerset*, denoted 2^G, which is the set of all subsets of a set G.
- *Tuple Elements.* We use dot notation to indicate an element of a tuple (e.g., $e.t$ denotes the executing thread of an event e.)
- *Restriction.* We use $E|_o$ to denote the restricted subset of events in E involving a concurrent object o. For example, $E|_t$ denotes events in E executed by thread t.
- *Slicing.* We use $E[i_1 : i_2]$ to denote the subset of events $e \in E$ where $e.i \in \{i_1..i_2\}$.

We represent abstract execution traces in figures as follows: Memory accesses to a variable `A` are denoted `w(A)` or `r(A)`, and acquiring or releasing a lock `L` is denoted `acq(L)` or `rel(L)`.

The input x being executed on each thread is indicated at the top of the trace. See Figure 3.4 for an example of abstract traces showing a race predicted across multiple traces.

Execution Trace Space. We denote the space of all possible execution traces up to length n for a program $\mathcal{P}$ as $\mathbb{E}_{\mathcal{P}}$. For a program executing different inputs on up to k threads drawn from a set of m total inputs, the set of all possible trace events $\mathbb{E}_{\mathcal{P}}$ is composed of the cross product of the possible trace positions $I = \{1..n\}$, executing threads $T = \{t_1..t_k\}$, possible inputs $X = \{x_1..x_m\}$, and O, all possible operations that can be performed by $\mathcal{P}$. O is composed of the union of possible memory accesses and synchronization operations that can be performed by $\mathcal{P}$, denoted $\texttt{Accesses}(\mathcal{P})$ and $\texttt{Syncs}(\mathcal{P})$ respectively: $O = \texttt{Accesses}(\mathcal{P}) \cap \texttt{Syncs}(\mathcal{P})$. Formally, the execution space of $\mathcal{P}$ is

$$\mathbb{E}_{\mathcal{P}} = Sym(I) \times 2^T \times 2^X \times 2^O \tag{3.2}$$

Where $Sym(I)$ is the symmetric permutation group over I, and 2^T, 2^X, and 2^O are powersets of T, X, and O, respectively. We refer to $\mathbb{E}_{\mathcal{P}}$ as $\mathbb{E}$ when $\mathcal{P}$ is clear from context.

3.2.2 Race Prediction on Multi-Input Programs

Concurrent Events and Data Races. When a multithreaded program executes, operations that are performed by different threads without synchronization happen concurrently (i.e., at the same time). A data race then occurs when memory access operations modify the same data concurrently, leading to undesired nondeterministic behavior.

In practice, when testing for data races, it is often more useful to identify pairs of memory access instructions that can race in the program rather than data races themselves, since a single unsynchronized instruction pair can generate thousands of data races during execution. We formally define concurrency, data races, and instruction pair races for multi-input program traces here:

- *Concurrent events.* Two events e_1 and e_2 in a trace E where $e_1.i < e_2.i$ are *concurrent* if

$e_1.t \neq e_2.t$ and $e_1 = last\_event(E[1 : e_2.i]|_{e_1.t})$.

- *Data races.* Two memory access events α_1 and α_2 are considered a *data race* in a trace E if they are concurrent in E, they $\alpha_1.m = \alpha_2.m$, and at least one of them is a write operation.

- *Racing instruction pairs.* We consider two memory access instructions a_1 and a_2 in $\mathcal{P}$ a *racing instruction pair* if there exists a $\mathcal{P}$-feasible *witness trace* E^* such that for two memory access events $\alpha_1, \alpha_2 \in E^*$, $\alpha_1.a = a_1$, $\alpha_2.a = a_2$, and α_1 and α_2 are a data race in E^*.

Multi-Trace Race Prediction. We define race prediction for a multi-input program $\mathcal{P}$ as follows: given a set of inputs X, thread count k, and set of traces $\mathbb{E}_{observed}$ collected by executing $\mathcal{P}$ on subsets of k inputs from X, a race prediction rp is composed of a pair of memory accesses α_1 and α_2 in $\texttt{Accesses}(\mathbb{E}_{observed})$ along with a hypothesized witness trace E^* in which α_1 and α_2 race:

$$rp := \langle \alpha_1, \alpha_2, E^* \rangle$$

A race prediction for two memory accesses α_1 and α_2 can be tested by extracting the executed inputs $\alpha_1.x$ and $\alpha_2.x$, and checking if α_1 and α_2 race when $\mathcal{P}$ is executed on their inputs according to the interleaving of their hypothesized witness trace E^*.

Feasible Race Predictions. The objective of multi-input race prediction is to identify all racing instruction pairs that appear in $\texttt{Accesses}(\mathbb{E}_{observed})$ for which a feasible witness trace exists. However, only a small subset of the possible $\mathbb{E}_{\mathcal{P}}$ represent traces that are feasible for $\mathcal{P}$ to execute. We use the definitions from [49] and denote a feasible execution trace as $\mathcal{P}$-feasible, where $feasible(\mathcal{P})$ is the set of all traces that are feasible for $\mathcal{P}$, and $\text{feas}_{\mathcal{P}} : 2^{\mathbb{E}_{\mathcal{P}}} \rightarrow \{0, 1\}$ is a boolean indicator function for a feasible trace on $\mathcal{P}$:

$$\text{feas}_{\mathcal{P}}(E) = 1 \iff E \in feasible(\mathcal{P}) \tag{3.3}$$

In order for an execution trace E to be $\mathcal{P}$-feasible, it must satisfy several sets of constraints. First, the trace must be *well-formed*, where each event must have a unique trace order i, and each

input x must be consistently executed on a specific thread t. Second, the operations performed on each thread in E must be *sequentially consistent* with $\mathcal{P}$, where the order of operations in $E|_t$ conform to the execution path followed by t on $\mathcal{P}$.

When threads are executing concurrently their execution paths can change due to inter-thread communications on shared variables that cause the execution to either take alternate branches or access different locks. Therefore, an execution trace must also satisfy two additional sets of constraints to be feasible:

- *Data Dependencies*. Data dependencies occur when events are dependent on specific values being read during previous memory accesses. These consist of *control flow dependencies*, where the values of specific read operations are used in branches that govern which operations are performed on the execution path, and *address dependencies*, where memory accesses are performed to addresses that are determined by prior read operations (e.g., pointer dereferences).

- *Synchronization*. The execution order must respect the constraints enforced by any synchronization in $\mathcal{P}$. In particular, any acquired lock in E must be released before it is acquired again.

Figure 3.3 shows examples of control flow dependencies, address dependencies, and synchronization constraints in concurrent execution traces.

Effective race prediction therefore requires being able to reason about the set possible of feasible traces as accurately as possible based on the the traces in $\mathbb{E}_{observed}$ based on their data-dependencies and synchronization. Over-approximating the feasible set leads to large numbers of incorrect predictions (i.e., false positives), which can lead to missed races when the computation budget is wasted testing incorrect predictions. Similarly, under-approximating the feasible set can also result in missed races (i.e., false negatives) since races may never be predicted.

3.2.3 Existing Approaches for Multi-Input Race Prediction

Sound Dynamic Race Prediction. Sound dynamic data race prediction methods under-approximate the feasible trace set $feasible(\mathcal{P})$ by predicting races on alternate reorderings of each individual trace $E \in \mathbb{E}_{observed}$ that are feasible for $\mathcal{P}$, $feasible(E)$. To enforce feasibility, the analysis preserves the order of all inter-thread communications based on Lamport's happens-before partial order for concurrent programs [14]. Sound dynamic race prediction has been applied to testing filesystems for data races [9, 18] by predicting races on each individual trace in an observed trace set $\mathbb{E}_{observed}$.

Alias Analysis and Lockset Analysis. Unlike dynamic race prediction, race prediction approaches that use alias analysis over-approximate the feasible set by assuming any execution trace E that can be generated by concurrently executing two inputs in X is $\mathcal{P}$-feasible. Potential races are then predicted on all aliased memory accesses that appear in $\mathbb{E}_{observed}$. In multi-input race prediction, inputs with instructions that can potentially race are identified using either static [17] or dynamic [7] alias analysis, where any pair of accesses α_1, α_2 where $\alpha_1.m = \alpha_2.m$ is considered an alias. Aliased accesses and then executed together to attempt to trigger a race, where E^* is either constructed by executing each input thread up to the predicted race and preempting. Alias analysis has also been combined with lockset analysis [56] (checking if common locks are held by two memory accesses when they were executed) to account for synchronization constraints in prediction.

Limitations of Existing Approaches. Using sound dynamic race prediction individually on each observed trace results in a sound under-approximation of the feasible set–the analysis won't make any incorrect predictions, but it will miss any feasible races that do not appear in the sound reorderings of the observed traces. This leads to missed races in two particular cases which are illustrated in Figures 3.4 and 3.5: when feasible races occur between accesses observed in separate traces (Figure 3.4), and when feasible races appear in reorderings of the observed traces that do not follow the happens-before constraints imposed by the observed communications (Figure 3.5).

Alias analysis and lockset analysis can be used to predict races between independently observed

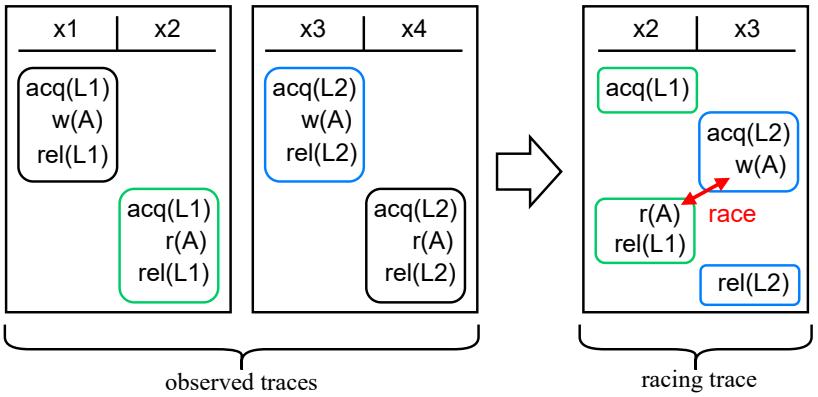
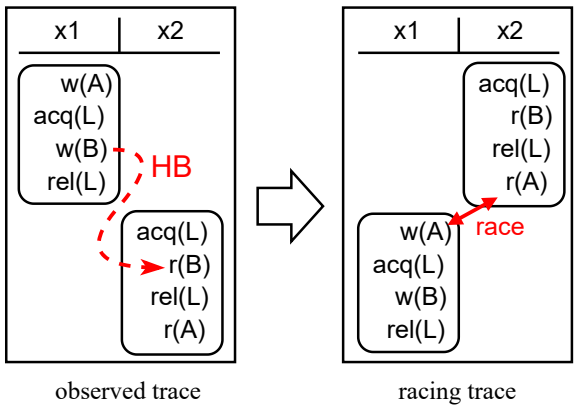


Figure 3.4: Example of a race between inputs from independently collected execution traces, which cannot be identified with dynamic race prediction. The accesses to shared variable `A` in the observed trace of `x1` and `x2` share a common lock `L1` and cannot race, and the accesses to `A` for `x3` and `x4` are also locked. However, `x2` and `x3` can be combined to form a new trace with a feasible race.

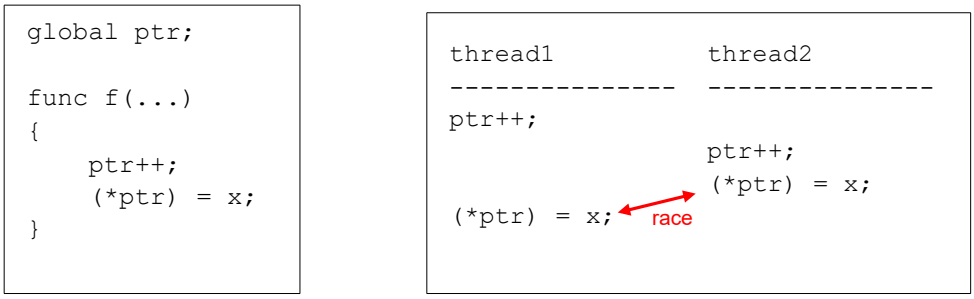
```
thread1          thread2
-------------    -------------
A = msg1;
acquire(L);
B = 1;
release(L);
                 acquire(L);
                 localB = B;
                 release(L);
                 msg2 = A;
```

(a) Two threads with communication on shared variable `B`.



(b) Observed trace with HB constraint, and reordering with feasible race.

Figure 3.5: Example of how the happens-before constraints used in sound dynamic race prediction can lead to missed races. The thread execution shown in 3.5a has a communication on shared variable `B` before `A` is read by thread 2, which generates a happens-before (HB) constraint in sound dynamic race prediction and prevents the analysis from predicting a race on shared variable `A`. However, the memory accesses to `A` have no data dependency on `B`, so a feasible trace that violates the happens-before constraint with a race on `A` can be constructed as shown 3.5b.



(a) Reentrant function.

(b) Execution interleaving with race.

Figure 3.6: Example of a race on a write with an address dependency to a global pointer that is incremented immediately before the write. The race only occurs for execution interleavings that perform both pointer increments before writing to the pointer address.

traces and is not prone to under-approximating the feasible trace set. This means that it won't miss the races shown in Figures 3.4 and 3.5. However, race predictions made with alias analysis and lockset analysis still do not take data dependencies into account. This can lead to missed races when the racing memory accesses have data dependencies that must be taken into account in order to generate a feasible witness trace, as well as incorrect predictions when races are infeasible due to data dependencies. Figure 3.6 shows an example of a race that only happens on execution interleavings that account for data dependencies.

3.3 Theory

In this section, we describe a general approach to modeling the feasibility of races for a multi-input program based on a decomposition of the operations, inputs, executing threads, and their respective orderings in traces. We first develop an approximate feasibility function based on the partial orders that appear in a trace. We then describe how fourier learning can be used to efficiently learn an approximate model of feasibility based on observed traces.

3.3.1 Feasibility Modeling

Constructing Traces. Multi-trace race prediction requires reasoning about constructing new traces E^* from the set of observed traces $\mathbb{E}_{observed}$. When constructing a new trace E^* from events in traces in $\mathbb{E}_{observed}$, we make the following changes:

- *Event Order.* All events orders in E^* are renumbered from $e.i = 1$ to $e.i = n$ for in $e \in E^*$ and $n = |E^*|$.

- *Thread Identifiers.* Events drawn from distinct traces are assigned distinct execution threads in the new constructed trace, so that for any two events e_i, e_j in E^* originating from separate traces, $e_i.t \neq e_j.t$ in E^*.

For example, events drawn from the first ten events of thread 1 in two different traces E_1 and E_2 would be assigned threads 1 and 2 and ordered 1 through 20 when combined in new trace E^*.

Approximating Feasiblity. Given a set of traces $\mathbb{E}_{observed}$, any pair of memory access events α_1, α_2 in the observed traces that access the same memory address can potentially race. Determining if α_1 and α_2 can race requires determining if there exists a witness trace E^* in which α_1 and α_2 race that is also $\mathcal{P}$-feasible. An initial hypothesized witness trace E^* in which α_1 and α_2 race can be constructed from traces in which α_1 and α_2 appear, denoted respectively E_1 and E_2:

$$E^* = \texttt{concat}(E_1[1 : \alpha_1.i]|_{\alpha_1.t}, E_2[1 : \alpha_2.i]|_{\alpha_2.t}) \tag{3.4}$$

Alias-analysis based race testing methods such as Razzer [17], Snowboard [7], and PLA [56] implicitly do this when they attempt to trigger a race during execution based on an aliased pair of accesses. However, an E^* constructed this way may not be $\mathcal{P}$-feasible, and in practice predicting all possible racing pairs this way leads to both overwhelming numbers of false positives as well as false negatives (missed races), since for a predicted E^* that is infeasible, there may be another combination of E_1 and E_2 that is feasible and the race does occur. Therefore, accurately predicting races on $\mathbb{E}_{observed}$ requires accurately estimating the function $\text{feas}_{\mathcal{P}}$ in order to construct feasible witness traces.

We represent $\text{feas}_{\mathcal{P}}$ from Eq. 3.3 as follows: first, we define a boolean feasibility function for each memory access operation α, $\text{feas}_{\mathcal{P}_\alpha} : \mathbb{E} \rightarrow \{0, 1\}$, which represents if an operation α is feasible after a sequence of operations is performed in the trace. For clarity we drop $\mathcal{P}$ when it is

clear from context and use feas_α. Formally:

$$\text{feas}_\alpha(E) = \begin{cases} 1 & \text{if feas}(E) \wedge \alpha \in E \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

Feasibility Modeling in Race Prediction. The trace feasibility function $\text{feas}_\alpha(P)$ approximates the feasibility of a racing trace based on any control flow and address dependencies the operations may have. This can generalize to reasoning about the feasibility hypothetical traces that do not appear in the observed set $\mathbb{E}_{observed}$. This is used in race prediction in two ways: first, the feasibility model can improve accuracy by identifying when predicted races are unlikely to be feasible (e.g., when a data dependency exists between two accesses to a common address). Second, the feasibility model can be used to construct witness traces that account for dependencies to make races feasible.

However, learning a model of trace feasibility from a set of observed traces is challenging because $\mathbb{E}$ is exponential in the length of the traces, possible operations, inputs, and threads in $\mathcal{P}$. In order to learn a model of trace feasibility efficiently, we base our approach on the sparsity of inter-thread data dependencies. Since most memory accesses between threads do not access common addresses, they do not interact with one another and therefore can be ordered arbitrarily in a trace without effecting feasibility.

As a result, trace feasibility functions exhibit the property that they are close to *fourier sparse*, meaning that they can be learned and expressed efficiently with a small number of nonzero coeffients in the fourier domain, even if they are defined over an exponentially large input domain. We find that in practice, more than 99.7% of the fourier domain coefficients are very close to 0 for boolean access feasibility functions (Section 3.6.3). Many search problems over large discrete spaces exhibit sparse fourier spectra and can be learned efficiently, and sparse fourier learning has been previously applied to problems in engineering and discrete optimization such as identifying optimal software system configurations [58] and modeling distributed system reliability in the presence of component failures [59]. Therefore, we use sparse fourier learning to model trace

feasibility efficiently. We provide background fourier learning in the following section, and then describe how we apply it to efficiently modeling traces.

3.3.2 Fourier Learning

Background on Fourier Expansion. It is well known in the discrete analysis literature that many computational problems can be expressed as boolean functions that are sparse in their fourier domain and can efficiently learned and analyzed using the fourier expansion. We summarize the fourier expansion for background, and we encourage readers to explore more details in Analysis of Boolean Functions Chapter 1 [19]. The fourier expansion for boolean functions typically defined on a real-valued function f that operates on n boolean inputs using the encoding $True := -1$ and $False := +1$. We note that this encoding is unintuitive from the perspective of boolean logic, but is used conventionally in the boolean fourier analysis literature because it simplifies definitions for the boolean fourier expansion. We summarize the standard definitions here:

$$f : \{+1, -1\}^n \rightarrow \mathbb{R}$$

The fourier expansion of f is then defined as a weighted sum of functions called *fourier characters* denoted χ weighted with constants called *fourier coefficients* denoted $\hat{f}$ over the domain of f:

$$f(x) = \sum_{S \subseteq [n]} \hat{f}(S)\chi_S(x) \tag{3.6}$$

Where $[n]$ is the set $\{1, 2, ..., n\}$. Each fourier character χ_S is a periodic function on a subset of the inputs determined by the elements in S, where $\chi_\varnothing$ is defined to be 1:

$$\chi_S(x) = \prod_{i \in S} x_i \tag{3.7}$$

Each fourier coefficient $\hat{f}(S)$ corresponds to χ_S's overall contribution to approximating f. While the full fourier expansion of f has 2^n fourier coefficients, functions that are close to fourier sparse

can be closely approximated and learned efficiently with a small number of nonzero fourier coefficients. Each coefficient can be estimated by sampling f:

$$\hat{f}(S) = \sum_{x \in \{+1,-1\}^n} f(x)\chi_S(x) = \mathbf{E}[f(x)\chi_S(x)] \tag{3.8}$$

Fourier Domain Sparsity. A function is considered to be *k-sparse* in the fourier domain if it has no more than k nonzero fourier coefficients, and is considered to ϵ-close to k-sparse if the function can be accurately approximately with an error bound of ϵ with k fourier coefficients. When a function is close to k-sparse, both learning the function's fourier coefficients can be accurately approximated in $O(k)$ operations. *Low degree* fourier sparse functions form a subclass of k-sparse functions where the k heavy fourier coefficents correspond to low degree sets S where $|S| < d$ for a low degree d [**kushilevitz1991learning**]. In the following section, we describe how we use low degree sparse fourier learning to make trace feasibility modeling tractable.

3.3.3 Sparse Fourier Learning on Traces

We develop a general approach for applying sparse fourier learning to approximating trace feasibility as follows: Given an indicator function f for a relevant trace property (e.g., a given memory access being feasible), we define two functions, a *trace encoding function* ϕ, and an *approximation function* f^*, where ϕ encodes a trace into a boolean vector, and f^* approximates f based on the encoding defined by ϕ:

$$f : \mathbb{E} \to Z_2 \qquad \phi : \mathbb{E} \to Z_2^n \qquad f^* : Z_2^n \to R \tag{3.9}$$

where $f(E) = f^* \circ \phi(E)$. The objective is then to approximately learn f^* from a set of observed traces. To approximately learn f^*, we estimate each fourier coefficient $\hat{f}^*(S)$ using Equation 3.8 and compose the encoding function ϕ with χ_S so each trace is encoded as a boolean vector before being input to the fourier character function χ_S:

$$\hat{f}^*(S) = \mathbf{E}[f(E)(\chi_S \circ \phi)(E)] \tag{3.10}$$

Trace Communication Encoding. In order to model the types of interthread communication dependencies shown in Figure 3.3, we define the trace encoding function ϕ to encode inter-thread communications that appear in a trace. We define communications as ordered pairs of memory accesses α_1, α_2 to the same address, and encode the presence of each possible communication to $\{0, 1\}$:

$$\text{encode}_{\alpha_1 \times \alpha_2}(E) = \begin{cases} 1 & \text{if } \alpha_1, \alpha_2 \text{ appear in } E \text{ in order} \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

For a set of traces $\mathbb{E}_{observed}$, we define ϕ using `encode` in Equation 3.11 as encoding all possible pairs of accesses $\alpha_1, \alpha_2 \in \texttt{Accesses}(\mathbb{E}_{observed})$ into a boolean vector, where each index ϕ_i represents a possible inter-thread communication.

Learning Trace Feasibility. For a given trace feasibility function feas_α, we define feas^*_α based on our communication encoding ϕ, where feas^*_α and feas_α are analogous to f^* and f in Equation 3.9. Given a set of traces $\mathbb{E}_{observed}$, we can then estimate each fourier coefficient of feas^*_α based on Equation 3.10:

$$\widehat{\text{feas}}^*_\alpha(S) \approx \frac{1}{|\mathbb{E}_{observed}|} \sum_{E \in \mathbb{E}_{observed}} \text{feas}_\alpha(E)(\chi_S \circ \phi)(E) \tag{3.12}$$

Each fourier coefficent $\widehat{\text{feas}}^*_\alpha(S)$ then represents the effect of a particular set of communications encoded by S on the feasibility of the memory access α.

Low Degree Sparse Fourier Learning. In order to learn feas^*_α efficiently, we apply sparse fourier learning in two ways. First, we only learn low degree fourier coefficents, where $|S| < d$ for a small upper bound d [60]. Second, while performing the coefficient estimation in Equation 3.12 over a

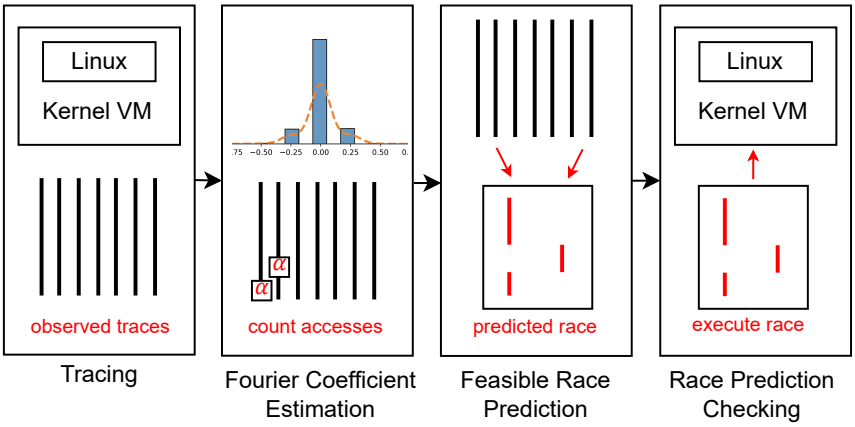


Figure 3.7: Overview of HBFourier's approach. HBFourier operates on a collection of execution memory access traces obtained from a linux kernel vm and estimates fourier coefficients for feasibility functions for each memory access that appears in the traces. It then uses the learned feasibility model to generate thread interleavings with predicted races.

set of observed traces, we dynamically update the set of estimated fourier coefficients to discard low magnitude fourier coefficients and retain high magnitude fourier coefficents. We describe this procedure in detail in the next section.

We note that more efficient methods for learning fourier sparse functions exist when samples can be selected adaptively [60]. We believe implementing a kernel scheduler for adaptive sampling of specific trace orderings on the kernel is a promising direction for future work.

3.4 Methodology

In this section we describe how we use sparse fourier learning to make accurate race predictions. Given a set of observed traces, we perform race prediction in four stages: feasibility learning, input feasible race prediction, order feasible trace construction, and race prediction checking.

Feasibility Learning. First, we learn approximate feasiblity functions for each access based on both inputs and partial orders. Algorithm 5 describes how we perform sparse fourier learning for feasibility function on the set of observed traces. We learn first degree ($d = 1$) fourier approximations for feasibility for each access α based on the factored feasibility approximation functions for operation order, thread, inputs, and operations present in the trace.

Race Candidate Generation. We generate a set of race candidate pairs of memory accesses in

the traces through a conservative analysis (i.e., no missed races) that checks for aliased accesses to common addresses that are not guarded by explicit synchronization (i.e., locks). See Algorithm 6, lines 1-6.

Feasible Race Prediction. We then construct a hypothetical witness trace E^* according to Eq. 3.4 for each race candidate of memory accesses α_1 and α_2. We then filter out race candidates that predicted to be infeasible based on their inputs and the partial orders that appear in E^* (Algorithm 6, lines 8-10).

Feasible Trace Construction. Third, for each race candidate that is filtered out as infeasible based on the partial orders in E^*, we attempt to make a feasible version of E^* by altering the execution interleaving. If there are specific inter-thread communications that make the race infeasible based on the fourier coefficients of their operations under a specific ordering, we check if it is possible to modify the interleaving to prevent the communications from occurring in $E*$ by adding a preemption to switch the execution thread immediately before the communication would occur. To identify an infeasible communication, given a set of memory access pairs P_{comms} representing inter-thread communications in a trace, a target access α, and a set of fourier coefficients $\widehat{\text{feas}}_\alpha$, we identify the largest negative contributing communication, denoted ρ:

$$\rho_{infeas} = \arg\min_{\rho} \widehat{\text{feas}}_\alpha(\rho)\chi_\rho(E^*) \tag{3.13}$$

Given a ρ_{infeas} composed of two operations $\texttt{op}_1$ and $\texttt{op}_2$ representing an inter-thread communication and a hypothesized trace E^* constructed from two observed traces E_1 and E_2, we attempt to create a new trace as follows:

$$\begin{aligned} E^* = \texttt{concat}(&E^*[1 : \texttt{op}_1.i - 1]|_{\texttt{op}_1.t}, \\ &E^*[1 : \texttt{op}_2.i]|_{\texttt{op}_2.t}, \\ &E^*[\texttt{op}_1.i : \texttt{op}_2.i]|_{\texttt{op}_1.t}, \\ &E^*[\texttt{op}_2.i + 1 : n]) \end{aligned} \tag{3.14}$$

If the new E^* with the added preemption is predicted to be feasible, we include it in the predicted races (Algorithm 6, lines 11-15).

Race Prediction Checking. Once the race predictions have been checked for feasiblity wrt. both inputs and partial orders in the witness trace, we check each prediction by executing it on the kernel. We enforce the thread execution interleaving to follow the same interleaving in the witness trace by instrumenting each memory access and making each thread wait before performing an access when another thread is interleaving to execute in the witness trace. We then check if the predicted race between the two memory accesses occurs during the controlled execution.

Algorithm 5 Sparse Fourier Feasibility Learning.

Input: $\mathbb{E}_{observed} \leftarrow$ Observed Traces
$n \leftarrow$ Function domain size
$d \leftarrow$ Maximum Coefficient Degree

$\mathbb{S}_d = \{S : S \subseteq [n], |S| \leq d\}$
initialize $\hat{f}_\alpha(\gamma) = 0$ for all $S \in \mathbb{S}_d$, $\alpha \in \texttt{Accesses}(\mathbb{E}_{observed})$
for $E \in \mathbb{E}_{observed}$ **do**
    for $\alpha \in \texttt{Accesses}(\mathbb{E}_{observed})$ **do**
        for $S \in \mathbb{S}_d$ **do**
            $\text{feas}_\alpha(E) = +1$ if $\alpha \in E$, -1 otherwise
            $\hat{f}_\alpha(S)$ += $\frac{1}{|\mathbb{E}_{observed}|} \text{feas}_\alpha(E)\chi_S(E)$
return All estimated $\hat{f}_\alpha$

3.5 Connection Between HBFourier and PLA

The sparse fourier learning approach used by HBFourier can also be used to express the learning performed by PLA as a first degree fourier expansion of a memory access indicator function.

Each input seed in PLA's notation is denoted p, therefore we can define the input x as the set of seeds that are executed from a corpus:

$$x = \{p_i, p_j, ...\} \tag{3.15}$$

Using the framework with the indicator function f, trace encoding ϕ, and approximation function f^* defined in Eq. 3.9, the trace encoding function ϕ is defined to encode which inputs p are

Algorithm 6 Race Prediction.

Input: $\mathbb{E}_{observed} \leftarrow$ Observed Traces
feas $\leftarrow$ Feasibility Model

RPs = { }
for $\alpha_1, \alpha_2 \in$ `Accesses`$(\mathbb{E}_{observed})$ **do**
  if $\alpha_1.m = \alpha_2.m$ **and** $lockset(\alpha_1) \cap lockset(\alpha_2) = \varnothing$ **then**
    E^* = hypothesize_witness(α_1, α_2) ▷ See Eq. 3.4
    $rp = (\alpha_1, \alpha_2, E^*)$
    RPs.add(rp)

for $rp \in$ RPs **do**
  if not feas$_{\alpha_1}(E^*)$ **or not** feas$_{\alpha_2}(E^*)$ **then**
    ρ_{infeas} = get_infeas_order(E^*) ▷ See Eq. 3.13
    reorder E^* to invalidate ρ_{infeas} ▷ See Eq. 3.14
    if not feas$_{\alpha_1}(E^*)$ **or not** feas$_{\alpha_2}(E^*)$ **then**
      drop rp from RPs and continue
return RPs

executed with input x, where

$$\phi_i(x, \tau) = \begin{cases} 1 & \text{if } p_i \in x \\ 0 & \text{otherwise} \end{cases} \tag{3.16}$$

We then the define the indicator function for a memory access α, f_α, to indicate if a given memory access appears in the trace τ:

$$f_\alpha(x, \tau) = \begin{cases} 1 & \text{if } \alpha \in \tau \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

The fourier expansion of f^*, defined on the encoding of inputs ϕ, will have a first degree fourier coefficient corresponding to each input. For an input p_i:

$$\hat{f^*}(S) = \mathbf{E}[f_\alpha(x, \tau)\chi_S(\phi(x))] \text{ where } S = \{i\} \tag{3.18}$$

The projection of f_α on to the single fourier character χ_S then becomes:

$$f_\alpha(x,\tau)\chi_S(\phi(x)) = \begin{cases} 1 & \text{if } \alpha \in \tau \text{ when } p_i \text{ executes} \\ 0 & \text{otherwise} \end{cases} \tag{3.19}$$

This is equivalent to the definition of the indicator function A_α learned in PLA in Eq. 2.1, therefore the feasibility model expressed in the learned first degree fourier expansion of f_α^* is equivalent to PLA.

3.6 Evaluation

Research Questions. We address the following research questions in our evaluation:

1. **Race Prediction Accuracy.** How do the accuracy of HBFourier's race prediction's compare to other applicable approaches?

2. **Race Testing.** Is HBFourier effective at finding new kernel races?

3. **Approximation Quality.** Does HBFourier's sparsity assumption hold, and does it closely approximate feasibility?

Evaluation Environment. We perform all evaluations on an Ubuntu 22.04 server with a Ryzen Threadripper 2970WX CPU and 128Gb of memory. These are a CPU/memory server configuration selected to cost effectively support compute and memory intensive applications.

Implementation. We implement HBFourier using an LLVM pass to collect traces of memory accesses and check for races, and use syzkaller's (a kernel fuzzer) executor to execute inputs. VM's for tracing and race checking are run with Qemu. Given a set of traces, HBFourier's analysis is implemented in three distributed stages: it first performs fourier learning and collects candidate racing memory accesses into sets, then refines the the predictions for each set based on lock synchronization and feasiblity, and finally aggregates the resulting predictions and removes duplicate

predictions to the same memory access instruction pairs (races may be predicted between two instructions multiple times on different memory addresses).

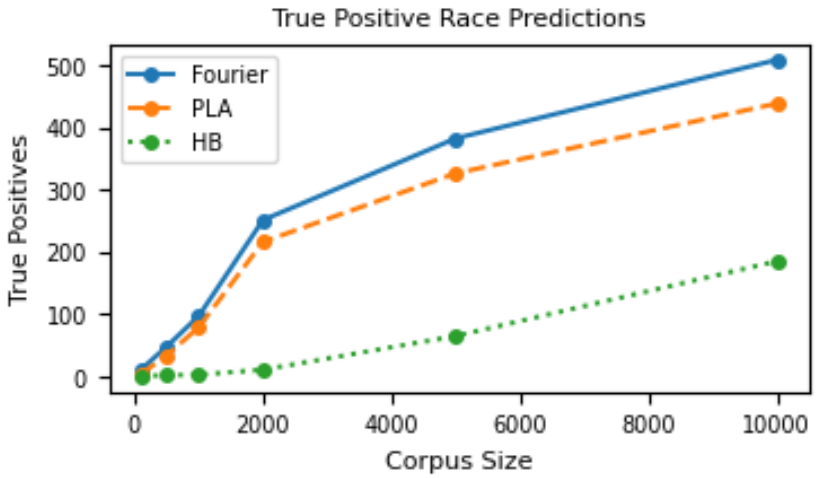
3.6.1 Race Prediction Accuracy

We evaluate the accuracy of HBFourier's race predictions against other race prediction approaches that are used in the kernel. We evaluate against two other approaches that can predict races on sets of observed traces, Probabilistic Lockset Analysis (PLA) [56], which estimates the probability of predicted races based on lockset analysis and prioritizes high probability races, and happens-before dynamic race prediction (HB), which analyzes each trace individually to predict races using Lamport's happens-before partial order to prevent false positive predictions, and is used by most recent kernel fuzzers [9, 18].

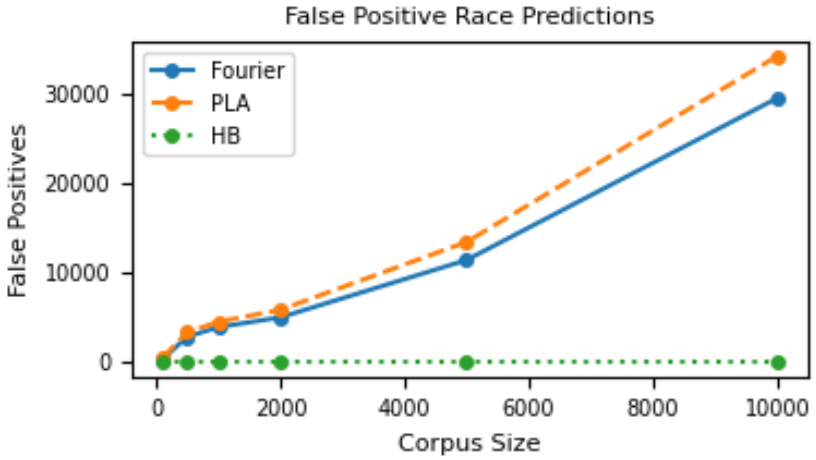
We compare the evaluated approaches on traces from five corpuses of fuzzer seeds varying size: 100 seeds, 500 seeds, 1k seeds, 5k seeds, and 10k seeds, where 8 traces are collected from each seed executing concurrently with another seed randomly sampled from the corpus. For each approach, we measure how many of its predictions are true positives (i.e., they result in a race being found) and how many are false positives (i.e., the predicted race cannot be confirmed by executing).

Figure 3.8 summarizes the results of the accuracy comparison. For all size corpuses, HBFourier makes more true positive predictions than PLA or HB, and makes fewer false positive predictions than PLA. On the 10k seed corpus, HBFourier makes 508 true positive predictions (corresponding to 332 unique instruction pairs) and 29,463 false positive predictions, while PLA makes 438 true positive predictions (corresponding to 276 unique instruction pairs) and 34,167 false positive predictions, and HB analysis makes 185 true positive predictions (corresponding to 72 races on unique instruction pairs).

These results illustrate the benefits of learning and directly modeling feasibility due to data dependencies in race prediction. HBFourier is able to make more correct predictions and find more races while making fewer false positive predictions than PLA by accounting for data dependencies



(a) True positive race predictions.



(b) False positive race predictions.

Figure 3.8: True positive and false positive race predictions for HBFourier, HBFourier (infeasible only), PLA, and sound dynamic race prediction on traces of 100, 500, 1k, 5k, and 10k fuzzer seeds. As the size of the trace set increases, PLA and HBFourier both many more false positive predictions, but HBFourier feasibility modeling makes it more accurate, with 14% fewer false positive predictions and 16% more true positive race predictions. Unlike HBFourier and PLA, sound dynamic race prediction does not make false positive predictions but can only identify 185 races that are directly observed in the traces, compared to 332 races found by HBFourier.

Table 3.1: Races found by HBFourier, PLA, and HB race prediction on 10k seed corpus, shown per kernel subsystem. In total HBFourier finds 332 races, PLA finds 276 races, and HB finds 72 races.

Kernel Subsystem	SYS	PLA	HB
kernel/	16	12	3
net/	85	65	23
mm/	26	20	11
fs/	42	29	8
security/	50	50	7
sound/	12	10	1
drivers/	74	65	15
include/*	15	15	0
block/	7	5	2
arch/*	2	2	1
lib/*	3	3	1

*the races found in include/, /arch, and /lib are related to different subsystems.

in candidate race predictions. Being able to make precise race predictions on large trace sets (i.e., without too many false positives) is essential to finding races effectively, since each prediction must be tested individually. As the size the corpus increases, the number of false positive predictions for both HBFourier and PLA increases substantially, but HBFourier makes more than 4700 fewer false positive predictions than PLA on the traces from the 10k seed corpus.

In contrast, HB dynamic race prediction does not make any false positive race predictions, but only makes 185 true positive predictions on the 10k seed corpus traces that appear directly in the traces. This result illustrates the benefits of multi-trace race prediction when testing for races on the kernel or other multi-input programs. Although HB dynamic race prediction does not make any false positive race predictions, a multi-trace analysis can find more races and can be paired with automatic prediction checking to ensure only confirmed races are reported to a user while finding more races overall.

3.6.2 Race Testing

In addition to evaluating the accuracy of HBFourier's predictions, we also evaluate HBFourier's effectiveness as a race testing system. We compare the races found by HBFourier to PLA and HB in Table 3.1. In total on the traces of the 10k seed corpus, HBFourier finds 332 races, PLA finds 276

races, and HB finds 72 races. HBFourier is able to effectively find more races across subsystems, and notably finds races in the `kernel` `net`, `mm`, `fs`, `sound`, `drivers`, and `block` subsystems that are not found by the other approaches.

We characterize the found races based on whether they occur in the observed traces or they occur between accesses in distinct traces, as well as if they have data dependencies that must be satisfied for a race to occur. PLA only finds races that do not have data dependencies (Data Ind.), while HB only finds races that appear in the observed trace set (In-Trace). HBFourier is able to find more races than either approach specifically because it can in infer data dependencies from its learned fourier coefficients, while still making accurate inter-trace race predictions.

3.6.3 Fourier Domain Sparsity

HBFourier's feasibility learning and analysis relies on a sparsity assumption for feasibility based on partial orders. Therefore we evaluate the sparsity of the fourier coefficients by directly evaluating coefficients for 50 sampled accesses. Figure 3.9 shows how fourier coefficient values converge to a sparse representation as the number of sampled traces increases. The results confirm that access feasibility is sparse in the fourier domain: the vast majority ($> 99.7\%$) of coefficients are 0 or have magnitude less than 0.1, indicating that they correspond to operation orders that have no effect on whether a given memory access is feasible or not, while a very small fraction ($< 0.0006\%$) have large coefficients close to -1.0 or 1.0. The extreme sparsity of large coefficients in the feasiblity fourier domain reflects the fact that most operations do not directly affect one another: only a very small fraction of operations will directly make another operation infeasible or feasible. Since most coefficients are very small, the large coefficents can be accurately resolved with only a small number of samples.

3.7 Threats to Validity

We consider two potential sources of bias that could threaten the validity of our results: dataset bias and sample bias. Our experiments are performed on a dataset of 10k kernel fuzzer seeds

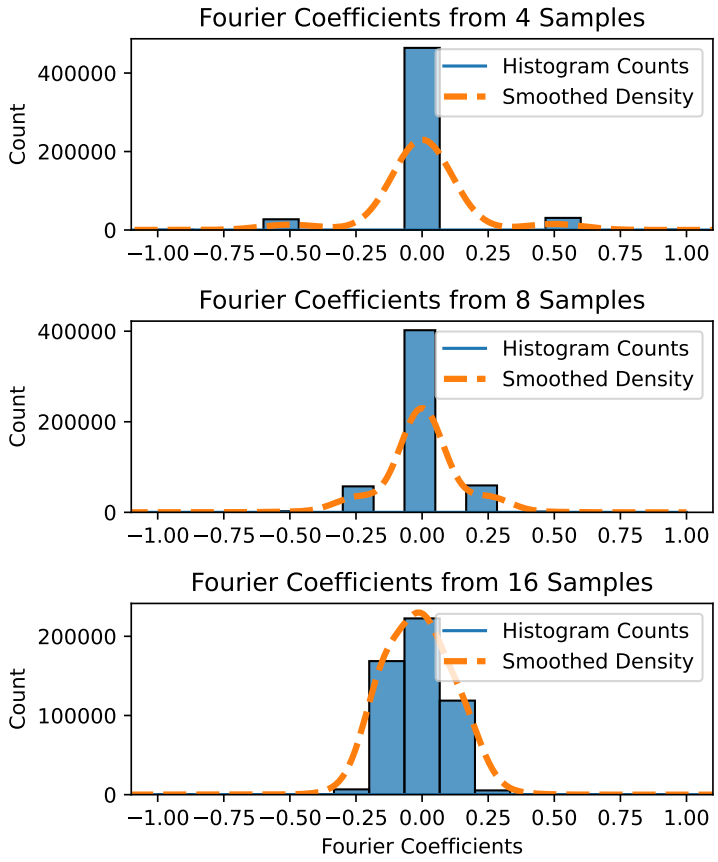


Figure 3.9: Distribution of estimated fourier coefficients, averaged over 50 accesses. As the number of sampled traces increases, most (99.7%) coefficients converge to 0, indicating that their corresponding operations have no significant impact on feasibility for a given memory access. A very small number of coefficients (0.0006%) converge to 1.0 or -1.0, which indicates they correspond to the trace being feasible (1.0) or infeasible (-1.0) for the memory access feasibility function. Sparsity in the fourier coefficients of partial order feasibility functions makes learning and analysis with them tractable.

generated with syzkaller [33]. Bias in the generated seeds towards certain kernel functionalities and types of races could effect the validity of our results for RQ1 and RQ2, although we note this dataset is large and exposes races in most core kernel modules, and is therefore likely to reflect the overall distribution of racing behavior in the linux kernel.

Sample bias could affect RQ3, since the results are based on a set of 50 randomly sampled accesses from the set of all accesses in the observed traces. We observed consistent results across sampled accesses, which suggests the sample is reflective of the overall distribution of accesses.

3.8 Limitations & Future Work

Although HBFourier has more accurate predictions and finds more races than other approaches in our evaluation, it is clearly possible to model race feasibility more accurately to further reduce erroneous predictions. HBFourier's approach does not fully utilize the power of fourier learning on sparse functions because it learns from a fixed set of samples and uses first degree coefficients in its race predictions. Adaptive sampling strategies can be used to learn sparse fourier functions with higher degrees more efficiently [60], which will result in more accurate feasibility predictions. We believe this is a promising direction for future work.

Another limitation of HBFourier's feasiblity modeling is that it is based purely on inputs and order-of-operations. However, many other events in the kernel can potentially be used to predict whether a hypothetical execution trace is feasible, such as RCU callbacks, worker thread events, etc. Therefore improving the kernel representation used in the feasibility model is another direction that could facilitate more accurate race predictions.

3.9 Related Work

Dynamic Race Prediction. Dynamic race prediction methods predict races based on a single observed execution trace. Most dynamic race predictors use Lamport's happens-before partial order to make sound predictions [14, 39, 26]. Recent work has focused on efficient variations of partial orders [41, 42, 43, 44, 45, 27], SMT reductions [47, 48, 49], as well as well as sampling

accesses from a single trace [61]. In contrast, HBFourier focuses on accurately predicting races across multiple traces, and allows false positives in its predictions that are ruled out during a final race checking step.

Kernel Race Testing. Kernel race testing approaches either use interleaving fuzzing in conjunction with dynamic race prediction [9, 18], or alias analysis over a set of seed traces [17, 7]. Probabilistic Lockset Analysis extends alias analysis by checking lock synchronization on accesses to common addresses and estimating the probability aliases may race through execution sampling. HBFourier also checks lock synchronization on shared memory accesses, but in addition uses sparse fourier learning to model the feasibility of its predictions.

Sparse Fourier Learning. Sparse fourier learning on boolean-valued functions has been applied to many applications such as reliability analysis [59], and approximating expensive objective functions in machine learning [62]. In the area of software engineering, sparse fourier learning has been used for efficiently optimizing the parameters of complex software systems from few samples [58]. HBFourier applies sparse fourier learning to modeling race prediction feasibility, based on an encoding of partial orders in traces to boolean domain functions.

3.10 Conclusion

This thesis proposes HBFourier, a novel approach for detecting data races in modern operating system kernels. By efficiently learning and leveraging factored trace feasibility functions through sparse Fourier learning, HBFourier makes 15.7% more accurate data race predictions relative to prior work and identifies 44 additional races by taking into account inter-thread communications and avoiding both false negatives and false positives. We believe sparse fourier learning in concurrent program analysis is promising direction for future work and can be applied to both further improvements in OS Kernel race prediction as well as other testing other concurrent systems.

Conclusion

This thesis introduces a new approach to data race prediction for OS Kernels based on learning a model concurrent memory accesses feasibility by exploiting the natural sparsity of interthread communications through sparse fourier learning. First, PLA learns interactions between inputs and memory accesses to efficiently predict races with a conservative lockset analysis. HBFourier generalizes the approach developed in PLA by applying sparse learning to interthread memory communications. These approaches are both theoretically novel and highly effective: HBFourier finds hundreds of races in a recent Linux development, and security analysis on the races found by PLA demonstrated that it found 102 harmful races, including one data race with severe security impact that has been overlooked by existing methods for nearly 10 years.

Future Work. The new techniques for data race prediction developed in this thesis point towards a new general class of approaches for program analysis that exploit the intrinsic sparsity in program execution semantics. PLA and HBFourier work because input dependencies and interthread communications dependencies are sparse and can therefore be approximated with sufficient accuracy to make effective predictions based on a relatively small number of observed samples. However, many other interactions in programs exhibit similar patterns of sparsity–for example, dependencies between individual instructions in a program will tend to be sparse since each individual executed instruction only directly depends on few other instructions. In addition, functions can be defined on more complex groups than the boolean domain and still learned efficiently through their fourier expansions.

Dataflow Analysis. Dataflow analysis is usually performed using boolean indicators of whether

dataflow from a given source is present for a given instruction and therefore naturally lends itself to encoding through boolean indicator functions. These functions share the property with the memory access feasibility functions defined in this thesis that they will tend to be sparse in the fourier domain, since program instructions will receive dataflows from a few instructions on any given execution. This means that boolean dataflow indicator functions can be efficiently learned in their fourier domain. These models could then be applied to challenging open problems in dataflow analysis, such as predicting when dataflows will be present between pointer operations for a given set of program inputs.

Beyond Boolean Groups. Although the model functions learned in this thesis all operate on boolean groups and boolean output, sparse fourier learning can be applied to functions with many other possible group structures on their domains. Learning accurate and efficient models of functions with more general group structures could be useful in many domains. For example, learning functions on the integer groups could be used to model program branches and guide input generation for fuzzers and other software testing tools, while learning functions on permutation groups could be used to further refine data race predictions based on the ordering of operations in a concurrent trace. Finally, the selection of groups that define a function's fourier expansion can also be formulated as a learning problem instead of being manually. Learning an optimal basis for the fourier expansion could then resolved through application of other learning methods, for example by using the embedding space learned by an auto-encoding neural network.

References

[1] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos) the case for a scalable operating system for multicores," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, 2009.

[2] R. Netzer and B. P. Miller, "Detecting data races in parallel program executions," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1989.

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

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

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

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

[7] S. Gong, D. Altinbüken, P. Fonseca, and P. Maniatis, "Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 66–83.

[8] R. H. Netzer and B. P. Miller, "What are race conditions? some issues and formalizations," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, no. 1, pp. 74–88, 1992.

[9] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2020, pp. 1643–1660.

[10] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs.," in *OSDI*, vol. 8, 2008.

[11] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 1, pp. 167–178, 2010.

[12] X. Yuan, J. Yang, and R. Gu, "Partial order aware concurrency sampling," in *International Conference on Computer Aided Verification*, Springer, 2018, pp. 317–335.

[13] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," ser. OOPSLA '12, New York, NY, USA: Association for Computing Machinery, 2012, 485–502, ISBN: 9781450315616.

[14] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[16] C. Wang, S. Kundu, M. Ganai, and A. Gupta, "Symbolic predictive analysis for concurrent programs," in *International Symposium on Formal Methods*, Springer, 2009, pp. 256–272.

[17] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 754–768.

[18] Z.-M. Jiang, J.-J. Bai, K. Lu, and S.-M. Hu, "Context-sensitive and directional concurrency fuzzing for data-race detection," 2022.

[19] R. O'Donnell, *Analysis of boolean functions*. Cambridge University Press, 2014.

[20] M. Musuvathi, S. Qadeer, and T. Ball, "Chess: A systematic testing tool for concurrent software," Tech. Rep. MSR-TR-2007-149, 2007, p. 16.

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

[22] Y. Lee, C. Min, and B. Lee, "Exprace: Exploiting kernel races through raising interrupts," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds., USENIX Association, 2021, pp. 2363–2380.

[23] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, San Jose, CA: USENIX Association, Feb. 2013, pp. 31–44, ISBN: 978-1-931971-99-7.

[24] M. Jimenez, M. Papadakis, and Y. Le Traon, "An empirical analysis of vulnerabilities in openssl and the linux kernel," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2016, pp. 105–112.

[25] J. Erickson, M. Musuvathi, S. Burckhardt, and K. Olynyk, "Effective data-race detection for the kernel," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[26] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, M. Hind and A. Diwan, Eds., 2009.

[27] U. Mathur, A. Pavlogiannis, and M. Viswanathan, "Optimal prediction of synchronization-preserving races," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021.

[28] Y. Yu, T. Rodeheffer, and W. Chen, "Racetrack: Efficient detection of data race conditions via adaptive tracking," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005, pp. 221–234.

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

[30] A. Dinning and E. Schonberg, "Detecting access anomalies in programs with critical sections," in *Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, 1991, pp. 85–96.

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

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

[33] *Syzkaller*, `https://github.com/google/syzkaller`, 2022.

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

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

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

[37] *Syzbot reports*, `https://syzkaller.appspot.com/upstream`, 2022.

[38] D. Marino, M. Musuvathi, and S. Narayanasamy, "Literace: Effective sampling for lightweight data-race detection," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, M. Hind and A. Diwan, Eds., ACM, 2009, pp. 134–143.

[39] F. Mattern, "Virtual time and global states of distributed systems.," in *Proc. Workshop on Parallel and Distributed Algorithms,*, 1989.

[40] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai, "Toward integration of data race detection in dsm systems," *Journal of Parallel and Distributed Computing*, vol. 59, no. 2, pp. 180–203, 1999.

[41] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan, "Sound predictive race detection in polynomial time," in *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, J. Field and M. Hicks, Eds., ACM, 2012, pp. 387–400.

[42] D. Kini, U. Mathur, and M. Viswanathan, "Dynamic race prediction in linear time," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 157–170, 2017.

[43] U. Mathur, D. Kini, and M. 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*, vol. 2, no. OOPSLA, pp. 1–29, 2018.

[44] A. Pavlogiannis, "Fast, sound, and effectively complete dynamic race prediction," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, pp. 1–29, 2019.

[45] J. Roemer, K. Genç, and M. D. Bond, "Smarttrack: Efficient predictive race detection," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 747–762.

[46] C. Wang, R. Limaye, M. Ganai, and A. Gupta, "Trace-based symbolic analysis for atomicity violations," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2010, pp. 328–342.

[47] M. Said, C. Wang, Z. Yang, and K. Sakallah, "Generating data race witnesses by an smt-based analysis," in *NASA Formal Methods Symposium*, Springer, 2011, pp. 313–327.

[48] T. F. Şerbănuţă, F. Chen, and G. Roşu, "Maximal causal models for sequentially consistent systems," in *International Conference on Runtime Verification*, Springer, 2012, pp. 136–150.

[49] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, 2014, pp. 337–348.

[50] E. Poznianski and A. 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*, IEEE Computer Society, 2003, p. 287.

[51] T. Elmas, S. Qadeer, and S. Tasiran, "Goldilocks: Efficiently computing the happens-before relation using locksets," in *Formal Approaches to Software Testing and Runtime Verification*,

K. Havelund, M. Núñez, G. Roşu, and B. Wolff, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 193–208, ISBN: 978-3-540-49703-5.

[52] A. Farzan, P Madhusudan, and F. Sorrentino, "Meta-analysis for atomicity violations under nested locking," in *International Conference on Computer Aided Verification*, Springer, 2009, pp. 248–262.

[53] F. Sorrentino, A. 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*, 2010, pp. 37–46.

[54] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 11–21.

[55] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, "Ski exposing kernel concurrency bugs through systematic schedule exploration," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 415–431.

[56] G. Ryan, A. Shah, D. She, and S. Jana, "Precise detection of kernel data races with probabilistic lockset analysis," in *2023 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2023.

[57] M. A. Thokair, M. Zhang, U. Mathur, and M. Viswanathan, "Dynamic race detection with o(1) samples," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, 2023.

[58] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki, "Performance prediction of configurable software systems by fourier learning (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2015, pp. 365–373.

[59] Y. Crama and P. L. Hammer, *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press, 2011.

[60] Y. Mansour, "Learning boolean functions via the fourier transform," *Theoretical advances in neural computation and learning*, pp. 391–424, 1994.

[61] M. A. Thokair, M. Zhang, U. Mathur, and M. Viswanathan, "Dynamic race detection with o (1) samples," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 1308–1337, 2023.

[62] P. Stobbe and A. Krause, "Learning fourier sparse set functions," in *Artificial Intelligence and Statistics*, PMLR, 2012, pp. 1125–1133.

[63] S. M. Ali and S. D. Silvey, "A general class of coefficients of divergence of one distribution from another," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 28, no. 1, pp. 131–142, 1966.

Appendix A: PLA Appendices

A.1 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):

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

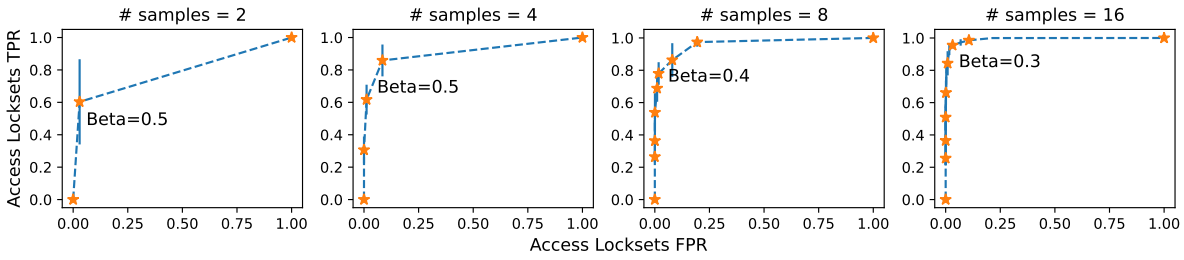


Figure A.1: ROC curves for access lockset prediction using varying numbers of samples evaluated on 5 sets of 50 randomly selected seeds with shown std. deviation. For each curve, the classification threshold parameter β giving the best performance is annotated based on F1 score.

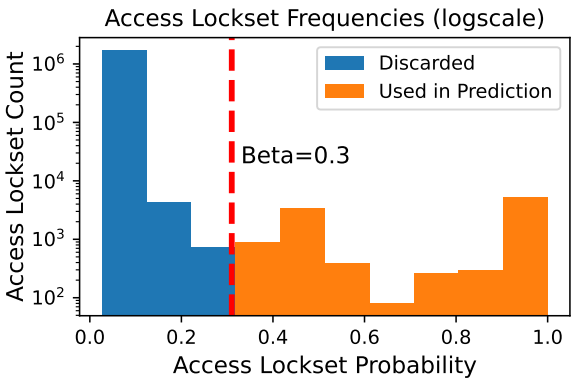


Figure A.2: Distribution of access locksets probabilities shown with log scale, where access locksets with probability exceeding β are marked orange. The vast majority of access locksets ($> 99.9\%$) occur with very low probability ($< 0.05\%$), therefore identifying high probability access locksets is critical to making accurate race predictions.

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:

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

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:

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

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:

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

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:

$$\begin{array}{ll} \text{input:} & \text{program } P, \text{ trace } T \\ \text{output:} & \text{race prediction } a_i, a_j, s^* \end{array} \tag{A.1}$$

A.2 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 [28, 51, 9], which prevents false positives and generates witness schedules for each predicted race.

A.3 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. 2.2 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 [63] 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:

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

□

A.4 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 2.2 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:

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

□

A.5 Data Races Found by PLA

Table A.1 lists all of the races found by PLA in our evaluation.

A.6 Impact of Parameter Choices

We evaluate PLA's access lockset classification accuracy on seeds drawn from the benchmark corpus used in Section 2.5.2. 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 A.1 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 2.5.4. 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 2.5.6 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 A.2 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 A.1: 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