# LR-Miner: Static Race Detection in OS Kernels by Mining Locking Rules

Tuo Li, *Tsinghua University;* Jia-Ju Bai and Gui-Dong Han, *Beihang University;* Shi-Min Hu, *Tsinghua University*

https://www.usenix.org/conference/usenixsecurity24/presentation/li-tuo

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

# LR-Miner: Static Race Detection in OS Kernels by Mining Locking Rules

Tuo Li
*Tsinghua University*

Jia-Ju Bai
*Beihang University*

Gui-Dong Han
*Beihang University*

Shi-Min Hu
*Tsinghua University*

## Abstract

Data race is one of the most common concurrency issues in OS kernels, and it can cause severe problems like system crashes and privilege escalation. Therefore, detecting kernel races is important and necessary. A critical step of kernel race detection is to identify locking rules that which variable should be protected by which lock. However, due to insufficient documents of kernel concurrency, it is challenging to identify accurate locking rules, causing existing approaches to produce many false results in kernel race detection.

In this paper, we design a new static analysis approach named LR-Miner, to effectively detect data races in OS kernels by mining locking rules from kernel code. LR-Miner consists of three key techniques: (1) a field-aware mining method that constructs and statistically analyzes the structure field relation between locks and accessed variables, to mine accurate locking rules from kernel code; (2) an alias-aware checking method to detect data races that violate the mined locking rules; (3) a pattern-based estimation strategy to estimate the security impact of the found races and identify harmful ones. We have evaluated LR-Miner on two popular OS kernels including Linux and FreeBSD, and it finds 306 real races with a false positive rate of 19.9%. Among these found races, 200 are estimated to be harmful, and 61 of them have been confirmed by kernel developers. 10 harmful races have been assigned with CVE IDs.

## 1 Introduction

To utilize multiple CPU cores in modern computer systems, OS kernel code is often concurrently executed for high performance, but inevitably introducing the risk of concurrency issues. Data race is one of the most common concurrency issues in OS kernels, and it can cause severe problems like system crashes and privilege escalation. Many well-known kernel vulnerabilities [22–25] are caused by data races. For example, Dirty COW [24] is a dangerous vulnerability caused by a data race in memory-management subsystem of Linux kernel. By exploiting Dirty COW, the attacker can maliciously corrupt critical memory and even obtain root privilege. Thus, for security and reliability, detecting data races in OS kernels is important and necessary.

Static analysis is a high-coverage and effective technique of bug detection, without actual execution of the tested program. Many existing approaches [1,2,10,31,32,43,52,61,70] focus on static analysis of user-level applications for race detection. Indeed, different from applications actively executing code, kernel code is often passively executed via system calls invoked by upper-level applications [9,67]. Thus, kernel concurrency is actually caused by application concurrency in many cases, without having explicit operations of thread creation and termination like applications do. However, these existing approaches require such thread operations to perform concurrency analysis, and thus they cannot effectively check kernel code to perform static race detection.

To detect kernel races, a few existing approaches [3, 14, 29, 64, 73, 74] perform static lockset analysis according to kernel concurrency. They assume all the functions can be concurrently executed or require the user to provide guidance about code concurrency, and then use static lockset analysis on concurrent functions to detect races. However, these approaches report many false races, and most of the found real races are benign and thus have no harmfulness. Indeed, their unsatisfactory results are caused by three main limitations:

*(L1) They fail to identify accurate locking rules, namely which variable should be protected by which lock.* In fact, a data race is often caused by missing necessary lock protection, and thus identifying locking rules is a critical step of race detection. If locking rules are unavailable or inaccurate, static lockset analysis will blindly search code paths and thus produce many false results in race detection. However, due to insufficient documents of kernel concurrency and complicated logics of kernel code [6,54,60], it is difficult for these existing approaches to statically identify accurate locking rules.

*(L2) They fail to consider accurate alias relationships in static lockset analysis.* Such alias relationships involve both locks and accessed variables, which are main objects handled

by static lockset analysis. If such alias relationships are neglected or inaccurately identified, static lockset analysis will make mistakes in checking the accessed variables protected by the related locks, and thus produce many false results in race detection. However, due to complex data structure usages and complicated control flows in kernel code, it is difficult for these existing approaches to identify and consider accurate alias relationships during lockset analysis.

*(L3) They never estimate the security impact of the found races.* A data race can be benign or harmful [44, 62]. Benign races are deliberately introduced by developers to improve the efficiency of concurrent execution, and they are expected to cause no security problem when being triggered. Only harmful races have security impact and can cause security problems, and thus identifying such races is valuable for concurrency bug detection. However, these existing approaches fail to check the influence of racy variables on possible code execution, and thus they cannot automatically identify harmful races from the produced results.

To solve the above limitations and improve static analysis of kernel race detection, we propose three key techniques:

**(1) Field-aware mining method.** To solve *L1*, our method automatically mines accurate locking rules from kernel code, by constructing and statistically analyzing the structure field relation between locks and accessed variables. In OS kernels, data structures are commonly used to share data between concurrent functions [54, 63]. In this case, *a shared variable and its protection lock are often represented as different fields but in the same data structure*. Based on this concurrency feature, our method analyzes each lock usage and variable access involving data structure field, to identify *locking relation*, namely which variable is actually protected by which lock in the code. To improve analysis accuracy of data structure fields, our method identifies locking relations based on *field graphs* that represent access paths [17, 45]. Then, for each variable in locking relations, this method statistically calculates the proportion of the accesses to this variable protected by each involved lock among all the accesses to this variable. If the proportion is large enough, indicating it is very likely that this variable should be protected by the involved lock, which is the *locking rule* mined by our method. Such locking rules are used for subsequent kernel race detection, and they can also help improve the documents of kernel concurrency.

**(2) Alias-aware checking method.** To solve *L2*, our method performs static lockset analysis to detect data races that violate the mined locking rules. To improve accuracy, our method performs flow/context/field-sensitive and inter-procedural alias analysis, to identify and utilize accurate alias relationships involving both locks and accessed variables. To improve efficiency, our method creates and uses function summaries containing alias relationships to perform inter-procedural checking. Moreover, our method performs SMT-based validation of code-path feasibility for each reported race, to reduce false positives in race detection.

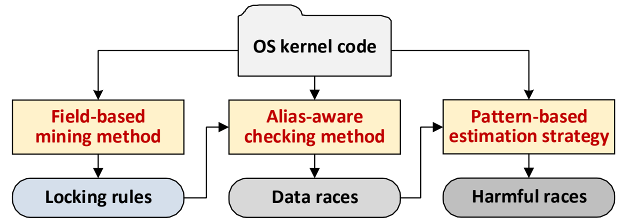



Figure 1: LR-Miner workflow.

**(3) Pattern-based estimation strategy.** To solve *L3*, our strategy checks the racy-variable usage of each reported race, based on some representative patterns that can cause security bugs like null-pointer dereferences, double fetch issues, etc. Our strategy can automatically estimate the security impact of the reported races and identify harmful ones.

Based on the above three techniques, we design LR-Miner (Locking Rule Miner), a new static analysis approach, to effectively detect data races in OS kernels by mining locking rules from kernel code. LR-Miner has three stages shown in Figure 1. LR-Miner first uses our field-aware mining method to mine accurate locking rules from the kernel code; then uses our alias-aware checking method to detect races according to the mined locking rules; and finally uses our pattern-based estimation strategy to identify harmful races from the found races. We have implemented LR-Miner with LLVM [53], and it performs automated analysis on the LLVM bytecode. Overall, we make three technical contributions in the paper:

- To improve static analysis of kernel race detection, we propose three key techniques: (1) a field-aware mining method that constructs and statistically analyzes the structure field relation between locks and accessed variables, to mine accurate locking rules from kernel code; (2) an alias-aware checking method to detect data races that violate the mined locking rules; (3) a pattern-based estimation strategy to estimate the security impact of the found races and identify harmful ones.
- Based on these key techniques, we design a novel static analysis approach named LR-Miner, to effectively detect kernel races by mining locking rules from kernel code. To our knowledge, *LR-Miner is the first systematic static analysis approach that detects kernel races by mining accurate locking rules and identifies harmful races*.
- We have evaluated LR-Miner on two popular OS kernels including Linux and FreeBSD. It in total mines 2.7K locking rules from kernel code, and finds 306 real races with a false positive rate of 19.9%. Among the found races, 200 are estimated to be harmful as they can cause security problems, and 61 of them have been confirmed by kernel developers. 10 harmful races have been assigned with CVE IDs. We also perform experimental comparison to multiple existing static approaches of kernel race detection (including Relay [74], RacerX [29] and CPALockator [3]). The results indicate that LR-Miner finds many real races missed by these approaches, with fewer false positives.

## 2 Background and Motivation

In this section, we first introduce kernel race and its static detection, then study data structure usages for kernel concurrency, and finally introduce how to use locking rules in static analysis for kernel race detection.

### 2.1 Kernel Race and Static Detection

**Kernel race.** To guarantee the correctness and security of kernel concurrency, a shared variable accessed by concurrently-executed threads should be protected by necessary synchronization primitives like locks. Otherwise, when at least one thread writes the shared variable during concurrent accesses, a data race can occur, causing the value of this shared variable to be uncertain.

In fact, a data race can be benign or harmful [44, 62]. Benign races are deliberately introduced by developers to improve the efficiency of concurrent execution, and they are expected to cause no security problem when being triggered. In comparison, harmful races are dangerous and can cause security problems. These harmful races may corrupt critical kernel data and affect important execution paths, and thus can lead to system crashes, privilege escalation, etc. Many well-known kernel vulnerabilities [22–25] are actually caused by data races. Thus, for security and reliability, detecting data races in OS kernels is important and necessary.

**Static detection of kernel races.** Static analysis is a popular and effective technique of bug detection, and it can conveniently analyze all the possible code paths of the tested program without actual execution. For this reason, static analysis can achieve higher detection coverage and find many bugs missed by runtime testing. In fact, many existing approaches of static analysis have produced promising results of kernel bug detection, but most of them [8, 34, 51, 57, 59, 71, 76, 85] focus on detecting bugs in sequential code, and thus cannot detect kernel races in concurrent code.

A few existing approaches [3, 14, 29, 64, 73, 74] use static lockset analysis to detect kernel races, but they report many false races, and most of the found real races are benign and thus have no harmfulness. Indeed, their unsatisfactory results are caused by three main limitations: *(L1)* lacking the identification of accurate *locking rules* (namely which variable should be protected by which lock), *(L2)* neglecting accurate alias relationships in static lockset analysis, and *(L3)* lacking the estimation about the security impact of the found races.

Among the three limitations, we believe *L1* is the most important one, because locking rules directly reflect kernel concurrency and thus determine basic analysis process of race detection. However, there are lots of variables and locks in kernel code, and thus it is error-prone and time-consuming to consider all these variables and locks when identifying locking rules. *L2* heavily affects the accuracy of race detection, and *L3* is related to the importance of the found races.

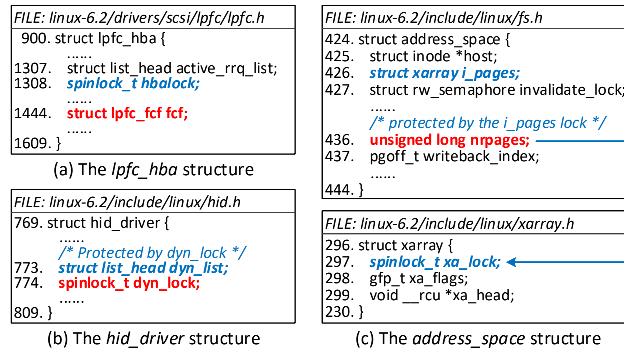



Figure 2: Example structures about kernel concurrency.

### 2.2 Structure Usages for Kernel Concurrency

In OS kernels, data structures are commonly used to share data between concurrent functions [54, 63]. In this case, *a shared variable and its protection lock are often represented as different fields but in the same data structure*. Figure 2 shows three examples of this concurrency feature in Linux:

In Figure 2(a), the structure `lpfc_hba` is used by Linux *lpfc* SCSI drivers. In this structure, the lock field `hbalock` actually protects some other fields, including `fcf`, according to code implementation and our discussion with kernel developers.

In Figure 2(b), the structure `hid_driver` is used by the human interface drivers in the Linux kernel. In this structure, the lock field `dyn_lock` protects the list field `dyn_list`, as described in its definition comment.

In Figure 2(c), the structure `address_space` is used by the filesystems in the Linux kernel. In this structure, the integer field `nrpages` is protected by the lower-layer nested field `i_pages->xa_lock`, as described in its definition comment.

We believe this feature can help to describe locking rules in a simpler way, without considering all the accessed variables and locks in kernel code. Specifically, when a lock and a variable are different fields in the same data structure, they may have protection relation. In this way, ***a locking rule is specifically described as: which data field should be protected by which lock field***. As for the accessed variable and locks in unrelated data structures, we consider that they have no protection relation and thus neglect them when identifying locking rules. Though using this simpler way, identifying locking rules from kernel code still has two main difficulties:

*(D1)* As shown in Figure 2(b) and Figure 2(c), the code comments clearly describe the locking rules of some structure fields. However, many parts of kernel code are not sufficiently commented or documented [6, 54, 60] (Figure 2(a) is such an example that has no code comment about locking protection), and thus just analyzing code comments and kernel documents is insufficient and infeasible to identify locking rules.

*(D2)* As shown in Figure 2(a) and Figure 2(b), the lock field and its protected data field are in the same layer of data structure. However, this phenomenon is not always correct. For example, as shown in Figure 2(c), the protection lock

```
FILE: linux-6.2/drivers/scsi/lpfc/lpfc_hbadisc.c
1230. int lpfc_linkdown(struct lpfc_hba *phba)
      ......
1251.   spin_lock_irq(&phba->hbalock);
1252.   phba->fcf.fcf_flag &= ...;
1253.   spin_unlock_irq(&phba->hbalock);
      ......
1323. }
```

```
FILE: linux-6.2/drivers/scsi/lpfc/lpfc_hbadisc.c
1597. void lpfc_mbx_cmpl_reg_fcfi(struct lpfc_hba *phba, ...)
      ......
1612.   spin_lock_irq(&phba->hbalock);
1613.   phba->fcf.fcf_flag |= ...;
1614.   spin_unlock_irq(&phba->hbalock);
      ......
1640. }
```

```
FILE: linux-6.2/drivers/scsi/lpfc/lpfc_hbadisc.c
1858. void lpfc_register_fcf(struct lpfc_hba *phba)
      ......
1864.   spin_lock_irq(&phba->hbalock);
1865.   if (!(phba->fcf.fcf_flag & ...)) {...}
1886.   spin_unlock_irq(&phba->hbalock);
      ......
1908. }
```

```
FILE: linux-6.2/drivers/scsi/lpfc/lpfc_hbadisc.c
6961. void lpfc_unregister_fcf_rescan(struct lpfc_hba *phba)
      ......
        // No protection of phba->hbalock!
6979.   phba->fcf.fcf_flag = 0; // Race!
      ......
7009. }
```

Figure 3: Example of locking rule usage.

field `i_pages->xa_lock` is a lower-layer nested field of the data field `nrpages`, but they are both in the same structure `address_space`. Thus, it is necessary to consider different layers of nested structures when identifying locking rules.

### 2.3 Race Detection Using Locking Rules

We illustrate how to utilize locking rules for kernel race detection, by using the *lpfc* SCSI driver in Linux 6.2. From Figure 2(a), we have a locking rule that the data field `fcf` should be protected by the lock field `hbalock` in the structure `lpfc_hba`. In Figure 3, the three functions `lpfc_linkdown`, `lpfc_mbx_cmpl_reg_fcfi` and `lpfc_register_fcf` all protect the variable `phba->fcf.fcf_flag` using the lock `phba->hbalock`, and thus they conform to the locking rule. However, in the function `lpfc_unregister_fcf_rescan`, the variable `hba->fcf.fcf_flag` is not protected by the lock `phba->hbalock` in any calling context of this function, which violates the locking rule, and thus a data race occurs. This race is found by our approach LR-Miner, and has been confirmed and fixed by the related kernel developers.

Inspired by the example, we can detect kernel races by checking the lock usages and variable accesses according to locking rules; if there is a violation of the rules, a possible data race will be reported. However, achieving this idea still faces three main challenges:

*(C1) How to identify accurate locking rules?* As described in Section 2.2, identifying locking rules is difficult, due to insufficient kernel documents/comments and complex structure layers. LockDoc [54] is the sole existing systematic approach of identifying locking rules in OS kernels for race detection, and it is based on dynamic analysis. It analyzes the execution traces of the provided workloads, to identify locking rules. However, due to the limited code coverage of the provided workloads, LockDoc misses many execution situations for the analyzed traces, which affects the accuracy of the mined locking rules. In our opinions, static analysis can conveniently analyze all the possible execution situations without actual kernel execution, so it can be used to identify accurate locking rules. RacerX [29] is the sole static approach that can identify simple locking rules from kernel code, but its used techniques (like field-insensitive and non-alias analysis) are imprecise and non-systematic for locking-rule mining, and thus it has a high false positive rate of over 40% in its race-detection experiments. As a result, it is important but challenging to systematically mine *accurate* locking rules in OS kernels.

*(C2) How to effectively check locking rules?* In kernel code, due to heavy use of pointers and data structures, the alias relationships between variables can be very complex. On the one hand, if such alias relationships are neglected or inaccurately identified when checking locking rules, many false results would be produced in race detection. On the other hand, the OS kernel is large-size and has lots of functions, and thus checking locking rules by identifying and considering accurate alias relationships can be quite time-consuming. For these reasons, improving both the accuracy and efficiency of locking-rule checking with alias relationships is challenging for kernel race detection.

*(C3) How to identify harmful races from the results?* As described in Section 2.1, a data race can be benign or harmful, and only harmful races can cause security problems. Several approaches [44, 56, 62, 77] control thread scheduling and analyze execution traces of the tested programs, to reproduce the given races and estimate their security impact. However, because thread scheduling of concurrent programs has much non-determinism, these approaches cannot always stably reproduce and effectively analyze all the given races. In our opinions, static analysis is a feasible way of identifying harmful races, because it does not require race reproduction and can achieve high analysis coverage. However, we find that this way has not been well explored so far.

## 3 Key Techniques

To address the three main challenges mentioned in Section 2.3, we propose three key techniques. For *C1*, we propose a *field-aware mining method* that constructs and statistically analyzes the structure field relation between locks and accessed variables, to mine accurate locking rules from kernel code. For *C2*, we propose an *alias-aware checking method* to detect data races by checking whether a variable access violates the mined locking rules. For *C3*, we propose a *pattern-based estimation strategy* to estimate the security impact of the found races and identify harmful ones. We will introduce these three techniques as follows:

## 3.1 Field-Aware Locking-Rule Mining

**Method design.** We design our method inspired by an existing dynamic analysis approach LockDoc [54], which analyzes the execution traces to identify locking rules about structure fields. Similar to LockDoc, our method has two basic steps: *(S1)* It first analyzes each lock usage and variable access involving data structure field, to identify *locking relation*, namely which data field is actually protected by which lock field in the code. *(S2)* For each data field in locking relations, it statistically calculates the proportion of the accesses to this data field protected by each involved lock field among all the accesses to this data field; if the proportion is large enough, it considers that this data field should be protected by the involved lock field, which is a mined locking rule.

However, our method is based on static analysis, and has two significant differences compared to LockDoc:

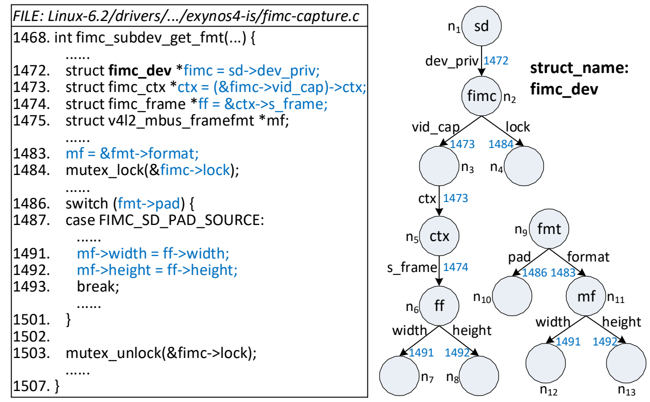
(1) LockDoc requires the user to provide various workloads for trace analysis; but due to limited code coverage of these workloads, LockDoc misses many execution situations for the analyzed traces, which affects the accuracy of the mined locking rules. To solve this problem, our method should statically mine locking rules from kernel code, without the requirement of workloads or execution traces.

(2) LockDoc never considers nested structures, so it cannot identify the locking rules involving the fields in different layers of nested structures. However, such locking rules are common in kernel code, like the example in Figure 2(c). To mine such locking rules, our method should accurately analyze the fields in different layers of nested structures. For this purpose, we use a new description form named *field graph* to describe the relation between each data field and lock field in structures (including nested ones).

**Field graph.** This form is based on access path [17, 45], to represent the relations between different structure fields (including nested ones). We introduce it as follows.

A field graph is defined as $FG = < N, E >$, where $N$ is a set of nodes, and each node represents a field. Note that a field can be in the lower-layer structure. $E$ is a set of edges, and each edge is labeled with a field and represents how a lower-layer field is accessed from a higher-layer structure. For convenience, a basic data type like *char*, *int* or *float* is regarded as a special structure that only contains a single field. For a nested structure with multiple layers, accessing a lower-layer field from a higher-layer structure can be expressed as an access path from the node representing the higher-layer structure to the node representing the lower-layer field. Fields in different functions with the same access path are regarded as identical fields. In a field graph, two fields in different layers are in the same structure if the nodes representing them have a common ancestor.

Field graph is updated by handling each arrow operator (->) and dot operator (.). For example, for each instruction like $v_2 = v_1$->$f$, our method inserts an edge labeled with $f$ from the node representing $v_1$ to the node representing $v_2$. After this operation, $v_2$ can be expressed as an access path $v_1.f$




(a) Part of the *exynos4-is* driver code in Linux 6.2 (b) Final field graph of the code

Figure 4: Example of field graph.

***Example.*** Figure 4(a) shows a part of the *exynos4-is* driver code in Linux 6.2. We use this example to illustrate how to build a field graph and how to check whether two fields in different layers are actually in the same structure with the built field graph. Take the instruction `fimc = sd->dev_priv` at Line 1472 as an example, it gets the field `dev_priv` of `sd` (represented by $n_1$), and then assigns this field to `fimc` (represented by $n_2$), so our method inserts an edge labeled with `dev_priv` from $n_1$ to $n_2$, indicating that the lower-layer field `fimc` can be accessed through the access path `sd.dev_priv` from the higher-layer structure `sd`. After handling all arrow operators in the code, our method figures out the final field graph shown in Figure 4(b). Take the fields `ff->width` and `fimc->lock` as an example, they are represented by $n_7$ and $n_4$ separately, and these two nodes have a common ancestor $n_2$ which represents the higher-layer structure named `fimc_dev`. Therefore, the two fields are identified to be in this common structure. To indicate the relation between the two fields, we use access paths to them from their common ancestor (namely `fimc_dev.vid_cap.ctx.s_frame.width` and `fimc_dev.lock`) to represent these fields. For convenience, in the remaining examples of this paper, the inner access path `vid_cap.ctx.s_frame` is omitted, and the structure fields such as `fimc_dev.vid_cap.ctx.s_frame.width` will be represented as `fimc_dev...width`.

**S1: Locking relation construction.** Based on field graph, locking relations can be identified by checking whether the accessed variable of each variable-access instruction is in the same data structure with the held locks. To get the held locks at each variable-access instruction, our method performs a static lockset analysis. Typically, lockset analysis maintains a set of locks held at each program site, and updates the set according to lock-acquire/release function calls. Specifically, when encountering a lock-acquire function call, our method adds the acquired lock of this call into the lockset; and when encountering a lock-release function call, our method drops

```
CollectLockingRelation(func)
Input: func – Analyzed function in the kernel code
Output: LR_set – Set of the collected locking relations
1:  field_graph := ∅;
2:  lock_set := ∅;
3:  LR_set := ∅;
4:  foreach code path cp in func do
5:    foreach instruction inst in cp do
6:      if inst is a lock-acquire/release function call then
7:        lock_set := LockSetAnalysis(inst);
8:      else if inst is an arrow operation then
9:        field_graph := UpdateFieldGraph(inst);
10:     else if inst is a variable-access instruction then:
11:       var := GetAccessedVariable(inst);
12:       access_type := GetAccessType(inst);
13:       foreach lock in lock_set do
14:         anc := FindCommonAncestor(var, lock, field_graph);
15:         if anc is not NULL then
16:           AP_var := GetAccessPath(var, anc, field_graph);
17:           AP_lock := GetAccessPath(lock, anc, field_graph);
18:           insert [<AP_var, AP_lock>, access_type, cp] into LR_set;
19:         end if
20:       end foreach
21:     end if
22:   end foreach
23: end foreach
24: return LR_set;
```

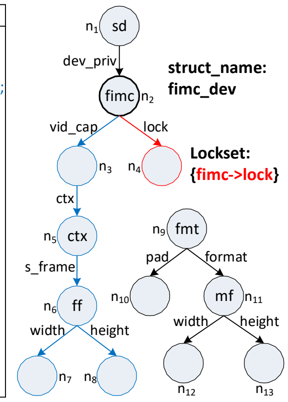
Figure 5: Process of collecting locking relations.

the handled lock of this call from the lockset. To collect locking relations, when encountering a variable-access instruction, for each lock $l$ in the lockset, our method checks whether the accessed variable $v$ and the lock are different fields but in the same data structure, by checking whether the nodes representing $v$ and $l$ have a common ancestor (representing a common higher-layer structure) in the field graph. If so, our method records a locking relation <$AP_v$, $AP_l$>, where $AP_v$ means the access path from the common higher-layer structure to the variable $v$, and $AP_l$ means the access path from the common higher-layer structure to the lock $l$.

Figure 5 shows the process of collecting locking relations. It creates a locking relation set *LR_set*, which stores all locking relations. Each element in the set is a triple [*lr*, *type*, *cp*], where *lr* is the constructed locking relation, *type* is how the data field involving in the locking relation is accessed (either a write or a read) and, *cp* is the code path of the locking relation. Like existing approaches [6, 51], this process starts from each function that has no explicit caller function in kernel code. Note that this process performs inter-procedural analysis, by combining the code paths of the callee and caller functions to get complete code path. Given a code path of the analyzed function, for each instruction in the code path, this process performs lockset analysis, updates field graph or constructs locking relations according to different types of the instruction (Lines 5–22). Specifically, for a lock-acquire/release function call, this process updates the lockset (Line 7). For an arrow operation, this process updates the field graph according to the handled instruction (Line 9). For a variable-access instruction, this process first gets the accessed variable and the access type (Lines 11–12). Then, for each lock in the lockset, this

```
FILE: Linux-6.2/drivers/.../exynos4-is/fimc-capture.c
1468. int fimc_subdev_get_fmt(...) {
        ......
1472.   struct fimc_dev *fimc = sd->dev_priv;
1473.   struct fimc_ctx *ctx = (&fimc->vid_cap)->ctx;
1474.   struct fimc_frame *ff = &ctx->s_frame;
1475.   struct v4l2_mbus_framefmt *mf;
        ......
1483.   mf = &fmt->format;
1484.   mutex_lock(&fimc->lock);     // Lock
        ......
1486.   switch (fmt->pad) {
1487.   case FIMC_SD_PAD_SOURCE:
          ......
1491.     mf->width = ff->width;       // Access
1492.     mf->height = ff->height;     // Access
1493.     break;
          ......
1501.   }
1502.
1503.   mutex_unlock(&fimc->lock);
        ......
1507. }
```

(a) Part of the *exynos4-is* driver code in Linux 6.2




(b) Locking relations in field graph

Figure 6: Example of constructing locking relations.

process traverses the field graph to find a common ancestor of the accessed variable and the lock (Line 14). If a common ancestor is found (Line 15), this process gets the access paths from the ancestor to the accessed variable and the lock (Lines 16–17), and records them as a locking relation (Line 18). Besides, the access type and the code path are also recorded for subsequent locking rule mining.

***Example.*** Figure 6 illustrates how our method constructs locking relations for the example code in Figure 4. In Figure 6(a), in the code path 1468→1486→1487→1491 (denote as `CP1`), the lock `fimc->lock` is acquired at Line 1484, and thus it is added into the lockset. Then, the code accesses five fields including `fmt->pad`, `mf->width`, `mf->height`, `ff->width` and `ff->height`. However, in Figure 6(b), only the two nodes representing the data fields `ff->width` and `ff->height` ($n_7$ and $n_8$), and the node representing the lock field `fimc->lock` ($n_4$) have a common ancestor ($n_2$). Take `ff->width` and `fimc->lock` as an example, the access paths from the common ancestor to the nodes representing them are `fimc_dev...width` and `fimc_dev.lock`, and the access is a read. Thus, our method inserts [<`fimc_dev...width`, `fimc_dev.lock`>, `read`, `CP1`] into the locking relation set. Similarly, our method also inserts [<`fimc_dev...height`, `fimc_dev.lock`>, `read`, `CP1`] into the locking relation set for the access to `fimc_dev...height`.

**S2: Locking rule mining.** After collecting locking relations, for each data field involved in them, our method statistically calculates the proportion of the accesses to this data field protected by each involved lock field among all the accesses to this data field, to mine locking rules. We observe different functions can have quite different numbers of code paths. Thus, distinguishing locking relations by code paths may be unfair for diverse functions, which can reduce the accuracy of locking rule mining. To address this problem, our method distinguishes locking relations by calling contexts, like existing approaches [42, 58]. Specifically, given a data field $AP_v$ and a lock field $AP_l$, our method first counts the number $num_{protected}$ of calling contexts containing the locking relation <$AP_v$, $AP_l$>

```
FILE: linux-6.2/drivers/media/platform/samsung/exynos4-is/fimc-capture.c

Simplified Code Path CP1:
fimc_subdev_set_selection
  -> mutex_lock(&fimc_dev.lock); [Line 1645]   // Lock
  -> set_frame_crop [Line 1665]
    -> fimc_dev.vid_cap.ctx.s_frame.width = width [Line 514]   // Write
    -> fimc_dev.vid_cap.ctx.s_frame.height = height [Line 515] // Write
Locking relation: <fimc_dev...width, fimc_dev.lock, Write, CP1>
                  <fimc_dev...height, fimc_dev.lock, Write, CP1>

Simplified Code Path CP2:
fimc_subdev_get_selection
  -> mutex_lock(&fimc_dev.lock); [Line 1588]   // Lock
    -> r.width = fimc_dev.vid_cap.ctx.s_frame.width [Line 1619]   // Read
    -> r.height = fimc_dev.vid_cap.ctx.s_frame.height [Line 1620] // Read
Locking relation: <fimc_dev...width, fimc_dev.lock, Read, CP2>
                  <fimc_dev...height, fimc_dev.lock, Read, CP2>

Simplified Code Path CP3:
fimc_subdev_set_selection
  -> mutex_lock(&fimc_dev.lock)  [Line 1645]   // Lock
  -> fimc_capture_try_selection [Line 1646]
    -> tmp_min_h = ffs(fimc_dev.vid_cap.ctx.s_frame.width) - 3 [Line 660]  // Read
    -> tmp_min_v = ffs(fimc_dev.vid_cap.ctx.s_frame.height) - 1 [Line 661] // Read
Locking relation: <fimc_dev...width, fimc_dev.lock, Read, CP3>
                  <fimc_dev...height, fimc_dev.lock, Read, CP3>

Simplified Code Path CP4:
fimc_subdev_set_fmt
  -> mf->width = fimc_dev.vid_cap.ctx.s_frame.width [Line 1548]   // Read
  -> mf->height = fimc_dev.vid_cap.ctx.s_frame.height [Line 1549] // Read
Locking relation: <fimc_dev...width, NULL, Read, CP4>
                  <fimc_dev...height, NULL, Read, CP4>

Mined Locking Rule: <fimc_dev...width, fimc_dev.lock>
```

Figure 7: Example of mining locking rules.

from the locking relation set returned by *CollectLockingRelation()* in Figure 5, and then counts the number $num_{all}$ of all the calling contexts containing accesses to $AP_v$ using a dataflow analysis. If the proportion of $num_{protected}$ among $num_{all}$ is larger than a threshold $T$ (set to 0.7 in this paper, as described in Section 5.1), and at least one of all the access is a write, our method mines a locking rule $<AP_v, AP_l>$ that the data field $AP_v$ should be protected by the lock field $AP_l$.

***Example.*** In Figure 7, our method collects four accesses ($num_{all}$) to `fimc_dev...width`, including one write and three reads, three ($num_{protected}$) of which are protected by the lock `fimc_dev.lock`. Thus, the proportion of $num_{protected}$ among $num_{all}$ is 0.75. As the proportion is large, our method infers the data field `fimc_dev...width` should be protected by the lock field `fimc_dev.lock`, and mines a locking rule <`fimc_dev...width`, `fimc_dev.lock`>. Similarly, our method mines another locking rule <`fimc_dev...height`, `fimc_dev.lock`> for accesses to `fimc_dev...height`.

## 3.2 Alias-Aware Race Checking

**Method design.** Due to heavy use of pointers and data structures in large-scale kernel code, static race detection can be inaccurate and time-consuming. Inspired by DLOS [7] that detects kernel deadlocks, we use function summaries to reduce the time of inter-procedural analysis. However, DLOS uses an intra-procedural and flow-insensitive alias analysis, which can introduce much inaccuracy. To improve accuracy and efficiency, we propose an alias-aware checking detection based on field graph (in Section 3.1). It uses a flow/field-sensitive intra-procedural analysis to construct field graphs representing alias relationships, detect races in single function and create a function summary containing alias relationships. When detecting races across functions, it performs a context-sensitive inter-procedural analysis with such summaries.

**Representation of aliased variables.** Our method focuses on alias relationships involving locks and accessed variables, based on their field graphs. Specifically, in a field graph, the aliased variables are represented by different access paths ending with the same node. However, for lockset analysis involving alias relationships, it is necessary to use one common access path to represent multiple aliased variables. Indeed, the locks and accessed variables shared by different threads often come from function arguments [5,64]. Thus, our method selects one access path starting from the node representing function argument, to represent these aliased variables.

**Intra-procedural analysis.** Based on the above representation of aliased variables, our method performs lockset analysis and race detection by handling the following instructions:

- For each lock-acquire/release function call, our method first extracts the access path $AP_l$ of the handled lock $l$, and then performs lockset analysis with $AP_l$ for this call.
- For each variable-access instruction, our method first extracts the access path $AP_v$ of the handled variable $v$. Then, for each locking rule $<AP_v, AP_{lx}>$ mined in Section 3.1, our method checks whether $AP_{lx}$ is in the lockset. If not, a rule violation is reported as a possible race.

After analyzing a function, our method creates and maintains a function summary for inter-procedural analysis. This function summary records the access paths and related instructions for all the lock-acquire/release function calls and variable accesses in the analyzed function. Formally, a function summary can be defined as a set of pairs $<inst, AP_v>$, where *inst* is an instruction of lock-acquire/release function call or variable access, and $AP_v$ is the access path of the variable handled by *inst*.

**Inter-procedural analysis.** When encountering a function call, our method first looks for the summary of the called function. If not found, our method performs intra-procedural analysis of the called function and creates its function summary. Then, our method instantiates the function summary by replacing each formal argument with actual argument. Finally, the instantiated summary of the called function is spliced to the call site of the caller function and stored as a part of the summary of the caller function.

During inter-procedural analysis, our method uses lockset analysis and detects the violations of the mined locking rules as possible races, like intra-procedural analysis.

***Example.*** Figure 8 illustrates how our method detects races across multiple functions, using the mined locking rule in Figure 7. This figure shows three functions and their partial instructions. Suppose that the analysis order of our method is `set_frame_crop` → `__fimc_capture_set_format` → `fimc_subdev_set_fmt`. After analyzing `set_frame_crop`,

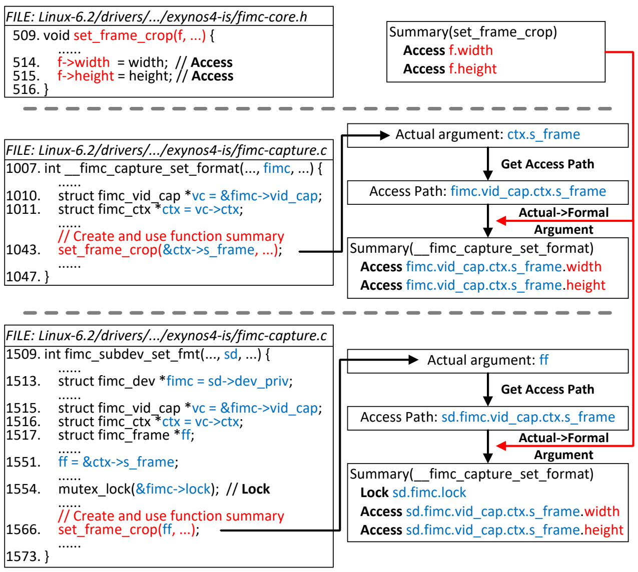



Figure 8: Example of alias-aware race checking.

our method records two variable accesses at Lines 514 and 515 in its function summary. When analyzing the function call to `set_frame_crop` at Line 1043 in the function `__fimc_capture_set_format`, our method first calculates the access path of the actual argument `ctx->s_frame`, namely `fimc.vid_cap.ctx.s_frame`, and then replaces the formal argument `f` in the function summary of `set_frame_crop` with the calculated access path to instantiate variable accesses in the summary. Finally, our method splices the instantiated variable accesses to the function summary of the caller function `__fimc_capture_set_format`, and then checks whether the instantiated accesses violate the minded locking rules. In this example, the two accesses to the fields `fimc_dev...width` and `fimc_dev...height` are not protected by the lock `fimc_dev.lock`, which violates the mined locking rules in Figure 7. Thus, our method reports two possible races here. When analyzing the function call at Line 1566 in `fimc_subdev_set_fmt`, the function summary of `set_frame_crop` is reused to avoid repeated analysis of the definition of `set_frame_crop`, which can reduce the time usage of race detection. In `fimc_subdev_set_fmt`, the accesses are all protected by the lock `fimc_dev.lock` acquired at Line 1554, and thus our method reports no race here.

## 3.3 Pattern-Based Harmfulness Estimation

**Method design.** Inspired by Portend [44] that reproduces and identifies harmful races, we estimate the security impact of the found races by analyzing code information. Different from Portend that executes the tested program to analyze trace information, we aim to statically analyze source code for security estimation without program execution. For this purpose, we propose a pattern-based estimation strategy that performs propagation analysis of the racy variables, according to some representative patterns that can cause security problems. Overall, our strategy has two main steps:

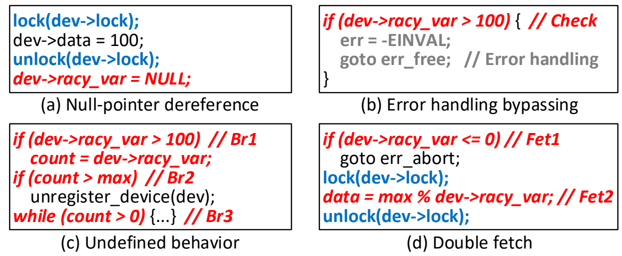



Figure 9: Four patterns of identifying harmful races.

**S1: Identifying racy-variable accesses.** The alias-aware checking method in Section 3.2 reports the variable-access instruction of each found race. If this instruction is a read, this step just passes this access into the next step. If this instruction is a write, namely other reads of the racy variable can be affected due to the uncertain value caused by this write, this step identifies the racy variable, and uses a field-based analysis [6, 37] of the kernel code to identify other accesses that possibly read the racy variable. These accesses are all handled in the next step.

**S2: Checking usage patterns from racy-variable accesses.** For each racy-variable access identified by *S1*, this step performs a flow/field/context-sensitive inter-procedural analysis starting from the access, to analyze the propagation of the racy variable and check its usage. The found race is considered to be harmful, if the racy variable's usage satisfies one of the following patterns:

*P1) Null-pointer dereference (NPD): affecting NULL assignment or check.* When the racy variable is a pointer that is assigned or checked by NULL, the concurrent access of this pointer may cause a null-pointer dereference. We select this pattern because many reported kernel vulnerabilities (like CVE-2023-31081 [26] and CVE-2023-46862 [27]) are caused by this pattern, and they can lead to DoS attacks. Figure 9(a) shows an example of this pattern.

*P2) Error handling bypassing (EHB): affecting error check.* When the racy variable affects a branch check of error handling, the race can make the kernel abnormally bypass error checks to access the resources already released by other threads due to error occurrence. We select this pattern because error handling are necessary but error-prone in OS kernels [41, 57], and error handling bugs can lead to DoS attacks, memory corruption and other security problems. Figure 9(b) shows an example of this pattern.

*P3) Undefined behavior (UB): affecting multiple branches.* When the racy variable affects multiple ($\geq$3) branches, the race can cause kernel control flow to be non-deterministic at runtime, which may lead to undefined behaviors. We select this pattern because some existing fuzzing approaches [42, 79] for kernel race detection reveal the security impact of such races. Figure 9(c) shows an example of this pattern.

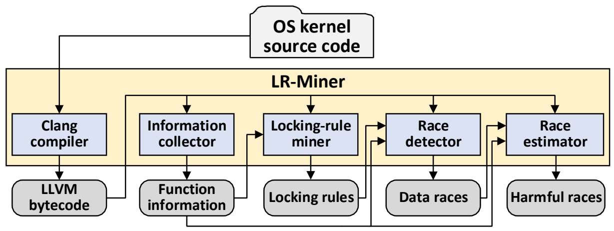


Figure 10: LR-Miner architecture.

*P4) Double fetch (DF): affecting variable check and usage without a common lock.* When the racy variable is checked before being used, but this check and use are not protected by a common lock, the race can cause a double fetch issue where the checked value and used values are inconsistent. We select this pattern because many double fetch issues in OS kernels are caused by code concurrency, and they can cause DoS attacks, restriction bypassing and other security problems [75, 80]. Figure 9(d) shows an example of this pattern.

These four bug types are selected, as many approaches [8, 38, 42, 51, 79, 81] have proven their harmfulness and use different ways to detect them. However, these approaches focus on static analysis of sequential code [8, 38, 51, 81] or dynamic analysis [42, 79] with limited coverage. In comparison, our strategy focuses on static analysis of concurrent code, and thus has significant difference from these approaches.

## 4 LR-Miner Approach

Based on the three key techniques introduced in Section 3, we design a new static analysis approach named LR-Miner, to effectively detect data races in OS kernels by mining locking rules from kernel code. We have implemented LR-Miner with the Clang compiler [18] to perform static analysis on the LLVM bytecode of OS kernels. Figure 10 shows the architecture of LR-Miner, which has four phases:

**Phase1: Code compilation.** First, the *Clang compiler* compiles the kernel source code into LLVM bytecode. During the compilation, the operators "." and "->" are all compiled to LLVM getelementptr (GEP) instructions. Then, the *information collector* scans each LLVM bytecode file to record the information about each function (including function name and function definition position) in a database. The information is used in the subsequent phases for inter-procedural analysis across source files.

**Phase2: Locking-rule mining.** The *locking-rule miner* uses our field-aware mining method to mine locking rules from kernel code. The mined locking rules are recorded in a readable form, to help detect races and understand the restriction of kernel concurrency.

**Phase3: Race detection.** The *race detector* uses our alias-aware checking method to detect the violations of the mined locking rules as data races. Note that for a given race, there may be multiple code paths from the entry function to its problematic instruction, and thus many repeated races can be reported. To drop repeated results, for a new possible race, the detector checks whether its problematic instruction is identical to any existing race. If so, this possible race is considered to be repeated and dropped.

**Phase4: Harmfulness estimation.** The *race estimator* uses our pattern-based estimation strategy to estimate the security impact of the found races and identify harmful ones. Note that a race can simultaneously satisfy two or more patterns presented in Section 3.3, so the estimator matches it with the most suitable pattern according to the usage of racy variable.

***Global variable handling.*** In kernel code, some global locks are used to protect global variables, and they are unrelated to function arguments. To handle such situations, LR-Miner specifically introduces a virtual node in the field graph, and this node is the ancestor of all global variables and locks. In this way, the accessed global variable and the global protection lock have a common ancestor, and thus they can be normally handled during locking rule mining and race detection.

***Atomic access handling.*** To ensure the atomicity of variable accesses in concurrent execution, some special functions and macros (like `atomic_inc` and `READ_ONCE` in the Linux kernel) are used. If such a function or macro is encountered during static analysis, LR-Miner neglects the related variable accesses in race detection.

***False positive dropping.*** On the one hand, similar to existing approaches of kernel bug detection [7, 8, 51], LR-Miner uses an SMT solver Z3 [84] to validate code-path feasibility during static analysis, which can drop false races caused by infeasible code paths. Specifically, for each possible race, LR-Miner translates instructions in the related code path into constraints using the Z3 grammars, and then checks whether these constraints can be satisfied. If not, LR-Miner regards the possible race as a false positive and thus drops it. On the other hand, similar to existing approaches of kernel race detection [29, 64, 74], LR-Miner drops the races occurring in the functions for kernel-module initialization and removal, by matching function names with keywords like "init" and "remove", as such functions are expected to have no concurrency during module execution.

## 5 Evaluation

We evaluate LR-Miner on the two OS kernels including Linux (version 6.2) and FreeBSD (version 14.0), and their versions are the latest minor releases as of our evaluation. Table 1 shows their information, and source code lines are counted by CLOC [19]. For the Linux kernel, we use the kernel configuration *allyesconfig* to enable all kernel code for the x86-64 architecture. For the FreeBSD kernel, we use the *GENERIC* configure file for the x86-64 architecture. We run the evaluation on a regular x86-64 desktop with sixteen Intel i7-10700 CPU@2.90GHz processors and 64GB physical memory.

| OS | Version | Source files (*.c) | LOC |
|---|---|---|---|
| Linux | 6.2 | 28.3K | 14.2M |
| FreeBSD | 14.0 | 19.6K | 9.2M |

Table 1: Information about the two checked OS kernels.

| Description | | Linux | FreeBSD |
|---|---|---|---|
| *Code analysis* | Source files (analyzed/all) | 22.5K/28.3K | 4.2K/19.6K |
| | Source code lines (analyzed/all) | 13.9M/14.2M | 3.3M/9.2M |
| *Locking-rule mining* | Identified locking relations | 52.1M | 16.8M |
| | Mined locking rules | 1.6K | 1.1K |
| *Race detection* | Times of handling called functions | 57.1M | 12.8M |
| | Times of reusing function summaries | 53.3M | 12.7M |
| | Dropped false races (path/concurrency) | 4.4K/1.2K | 4.5K/1K |
| | Found races (real/all) | 273/341 | 33/41 |
| *Harmfulness estimation* | Handled racy-variable accesses | 1,381K | 127K |
| | Identified harmful races | 173 | 27 |
| | Harmful patterns (NPD/EHB/UB/DF) | 26/59/50/38 | 11/6/10/0 |
| *Time usage* | Locking-rule mining | 1h33m | 56m |
| | Race detection | 10h43m | 4h3m |
| | Harmfulness estimation | 7h26m | 3h35m |
| | Total time | 21h48m | 9h32m |

Table 2: Analysis results of the two OS kernels.

## 5.1 Bug Detection

We run LR-Miner to automatically mine locking rules, detect kernel races and estimate their harmfulness. The proportion threshold $T$ used by our field-aware mining method is set to 0.7 in the evaluation. We select this value as it can help LR-Miner achieve good accuracy of race detection, according to our experience of kernel development and detection results of small-scale code bases. The user can conveniently change this threshold as needed. Then, we manually check all the races found by LR-Miner. Table 2 summarizes the results, and we make the following observations:

**Code analysis.** LR-Miner in total analyzes 17.2M lines of code in 26.7K source files, within 32 hours. The remaining 6.2M lines of code in 21.2K source files are not analyzed, because they are not enabled by the configurations for the x86-64 architecture. We believe that LR-Miner can find more kernel races, if these source files can be compiled with proper configurations for other architectures.

**Locking-rule mining.** LR-Miner identifies 68.9M locking relations from the code of the two kernels. By analyzing these locking relations and the related variable accesses, LR-Miner mines 2.7K locking rules (including 1.6K in Linux and 1.1K in FreeBSD) indicating which data field should be protected by which lock field. Besides race detection, we believe that these locking rules can also help to improve the documents about kernel concurrency and benefit the secure development of new kernel code.

**Race detection.** Based on the mined locking rules, LR-Miner finds 382 kernel races, including 341 in Linux and 41 in FreeBSD. 11.1K false races are dropped, because their code paths are infeasible (8.9K) or their caller functions have no concurrency (2.2K). We spent 18 hours on checking these races, and identified 306 of them (including 273 in Linux and 33 in FreeBSD) to be real. Thus, LR-Miner achieves a false positive rate of 19.9%, which is lower than many existing static approaches of kernel race detection [3, 29, 64, 73, 74].

| Part | Filesystem | Network | Driver | | | | Others |
|---|---|---|---|---|---|---|---|
| | | | *All* | *Network* | *SCSI* | *Others* | |
| **Races** | 29 (9%) | 36 (12%) | 198 (65%) | 43 | 42 | 113 | 43 (14%) |
| **Harmful** | 19 (10%) | 29 (15%) | 126 (63%) | 28 | 27 | 71 | 26 (13%) |

Table 3: Distribution of the found races.

**Efficiency improvement.** During race detection, LR-Miner uses function summaries to avoid repeated analysis of the same functions. Specifically, to perform inter-procedural analysis, LR-Miner handles called functions for 69.9M times, 66M (94%) of which are handled by reusing function summaries, without the need of analyzing function definitions again. Thus, race-detection efficiency is largely improved.

**Harmfulness estimation.** For the 306 real races, LR-Miner first identifies 1,508K accesses to the racy variables in kernel code, and then estimates the security impact of these races by analyzing the identified accesses according to four patterns introduced in Section 3.3. LR-Miner identifies 200 harmful races (173 in Linux and 27 in FreeBSD). Specifically, 37 can cause null-pointer dereferences (NPD), 65 can cause error handling bypassing (EHB), 60 can cause undefined behaviors (UB), and 38 can cause double fetch (DF) issues.

**Race distribution.** We classify all the real and harmful races found by LR-Miner, according to the category of the kernel part containing the race. Table 3 shows the distribution results of the real and harmful races. We find that drivers have over 60% of the real and harmful races, indicating that drivers are more error-prone than other kernel parts and thus deserve more attention in race detection. We also classify the driver races by driver class, and show the result in Table 3. We find that network and SCSI drivers together have over 40% of the real and harmful races in all drivers, possibly because these drivers have more concurrent code than other driver classes.

**Harmful race reporting.** We reported all the 200 harmful races to kernel developers, and 61 of them (including 51 in Linux and 10 in FreeBSD) have been confirmed. We are still waiting for the response of the remaining ones. Moreover, 10 harmful races have been assigned with CVE IDs. Some kernel developers also expressed their interests of integrating LR-Miner in their continuous integration (CI) testing systems to help detect races during kernel development.

**Security impact of the found harmful races.** We manually check these races and find that: 37 NPD races can cause null-pointer dereferences, leading to DoS attacks; 65 EHB races can bypass error handing to cause use-after-free vulnerabilities due to accessing the resources released by other threads, leading to memory corruption; 60 UB races can make kernel control flow non-deterministic to cause undefined behaviors, leading to DoS attacks; 38 DF races can cause double fetch issues where the checked value and used value are inconsistent, leading to restriction bypassing and memory corruption.

## 5.2 False Positives and Negatives

**False positives.** LR-Miner reports 76 false races in the two kernels, for four main reasons:

*R1)* LR-Miner mines some false locking rules due to over lock protection of the data field that should not be protected. In a data structure, if a lock field is often used to protect many other fields, LR-Miner will mine the related locking rules. But in fact, only one specific field should be protected while the other fields are not, and thus some mined locking rules are actually false, causing 53 false races to be reported.

We infer that over lock protection is introduced, as the related developers fail to identify which data fields should be protected by a given lock field, and thus just blindly add lock protection for extra fields together. Although over lock protection hardly causes security bugs, it can reduce kernel concurrency and degrade OS performance. Thus, it is meaningful to detect over lock protection in kernel code.

*R2)* In order to reduce memory overhead and accelerate data passing across different functions, an integer can be divided into several bit vectors to represent different data structure fields in kernel code. However, in the LLVM bytecode, the accesses to these vectors are divided into a load and several bit operations. At present, LR-Miner cannot correctly distinguish the accesses to different structure fields in this case, causing 14 false races to be reported.

*R3)* In some kernel functions, special assertions are used at the entry of these functions, to guarantee that the required lock is held. However, LR-Miner does not specially handle these assertions, and thus maintains incorrect locksets during analysis, causing 6 false races to be reported.

*R4)* LR-Miner still errs when handling some complicated cases, like checking the accesses to array elements via non-constant indexes and validating path feasibility related to complex arithmetic conditions. This reason causes 3 false races.

**False negatives.** LR-Miner may still miss some real races for three main reasons:

*R1)* LR-Miner maintains and checks locksets by analyzing the arguments of lock-acquire/release function calls, but some lock-acquire/release functions (such as `rcu_read_lock` and `rcu_read_unlock` in the Linux kernel) have no argument in the calls to them. Thus, LR-Miner neglects the calls to these functions during locking-rule mining and race detection, which may miss the related real races.

*R2)* Some locks are represented as array elements that are used via non-constant indexes (such as `dev->locks[i]`) in kernel code, and analyzing these locks is error-prone. Thus, LR-Miner neglects such locks during lockset analysis to reduce false positives, but may miss the related real races.

*R3)* Some locks are customized in special forms like reference counts and condition variables, instead of calling lock-acquire/release functions. Thus, LR-Miner cannot handle such locks, which may miss the related real races.

## 5.3 Case Studies of the Found Harmful Races

Figure 11 shows four harmful races found by LR-Miner in Linux and FreeBSD. All of these races have been confirmed.

**NPD race in the FreeBSD firewire driver.** In Figure 11(a), the pointer field `ir->stproc` is dereferenced without holding lock in the function `fw_read` at Line 363. However, in many other functions, this pointer field is accessed with the lock acquired by calling `FW_GLOCK`, like Line 471 in the function `fw_write` (note that `it->stproc` in this function and `ir->stproc` in `fw_read` are actually identical). Thus, there is a race at Line 363. Moreover, `it->stproc` is checked with NULL at Line 417, namely it can be NULL, and thus `ir->stproc` can be also NULL at Line 363, causing a null-pointer dereference. The attacker can exploit this harmful race to crash the kernel and perform DoS attacks.

**EHB race in the Linux media driver.** In Figure 11(b), the data field `dmxdev->exit` is checked for error handling without holding lock in the function `dvb_dvr_read` at Line 271. However, in many other functions, this data field is accessed with the lock `dmxdev->mutex`, like Line 1456 in the function `dvb_dmxdev_release`. Thus, there is a race at Line 271. Meanwhile, we observe that in the function `dvb_dmxdev_release`, after `dmxdev->exit` is set to 1 at Line 1456, `dvb_unregister_device` is called at Line 1470 to release the data buffer used in the driver; in the function `dvb_dmxdev_release`, if `dmxdev->exit` is 0 in the error check at Line 271, `dvb_dmxdev_buffer_read` is subsequently called at Line 273 to read the data buffer. Thus, in a special execution order in form of Lines 1455 $\rightarrow$ 271 $\rightarrow$ 1456 $\rightarrow$ 1457 $\rightarrow$ 1470 $\rightarrow$ 273, the race can bypass the error check at Line 271 and cause that `dvb_dmxdev_buffer_read` is executed after `dvb_unregister_device`, namely the data buffer is read after being released, leading to a use-after-free vulnerability. The attacker can exploit this harmful race to corrupt the memory of data buffer and inject malicious data.

**UB race in the Linux SCSI driver.** In Figure 11(c), the data field `phba->fcf.fcf_flag` is written without holding lock in the function `lpfc_unregister_fcf_rescan` at Line 6979. However, in many other functions, this data field is accessed with the lock `phba->hbalock`, like Lines 6791, 6797, 6834 and 6942 in the function `lpfc_sli4_async_fip_evt`. Thus, there is a race at Line 6979. Moreover, in the function `lpfc_sli4_async_fip_evt`, there are at least four branches are directly affected by `phba->fcf.fcf_flag`, and thus this race can disorder the code execution to cause undefined behavior of the driver. The attacker can exploit this harmful race to interfere driver execution and perform DoS attacks.

**DF race in the Linux sound subsystem.** In Figure 11(d), the data field `card->total_pcm_alloc_bytes` is checked without holding lock in the function `do_alloc_pages` at Line 41. However, in many other functions, this data field is accessed with the lock `card->memory_mutex`, like Lines 62 and 63 in the function `do_free_pages`. Thus, there is a

**Thread 1**

```
FILE: FreeBSD-14.0/sys/dev/firewire/fwdev.c
314. int fw_read(struct cdev *dev, ...) {
         ......
325.     d = dev->si_drv1;
326.     fc = d->fc;
327.     ir = d->ir;
         ......
362.     FW_GUNLOCK(fc);  // Unlock
         // Race: ir->stproc can be NULL!
363.     fwdma_v_addr(ir->stproc->poffset, ...);
         ......
388. }
```

**Thread 2**

```
FILE: FreeBSD-14.0/sys/dev/firewire/fwdev.c
447. int fw_write(struct cdev *dev, ...) {
         ......
459.     d = dev->si_drv1;
460.     fc = d->fc;
461.     it = d->it;
         ......
469.     FW_GLOCK(fc);  // Lock
471.     if (it->stproc == NULL) // Can be NULL
         ......
518. }
```

(a) Null-pointer dereference in FreeBSD

**Thread 1**

```
FILE: Linux-6.2/drivers/media/.../dmxdev.c
265. ssize_t dvb_dvr_read(struct file *file, ...) {
         ......
269.     dvbdev = file->private_data;
270.     dmxdev = dvbdev->priv;
         // Race: error check can be bypassed!
271.     if (dmxdev->exit)  // No lock is held
272.         return -ENODEV;  // Error handling
         // Read the buffer
273.     return dvb_dmxdev_buffer_read(...);
         ......
277. }
```

**Thread 2**

```
FILE: Linux-6.2/drivers/media/.../dmxdev.c
1454. void dvb_dmxdev_release(...) {
1455.     mutex_lock(&dmxdev->mutex);
1456.     dmxdev->exit = 1;
1457.     mutex_unlock(&dmxdev->mutex);
          ......
          // Release the buffer
1470.     dvb_unregister_device(...);
          ......
1475. }
```

(b) Error handling bypassing in Linux

**Thread 1**

```
FILE: Linux-6.2/drivers/scsi/lpfc/lpfc_hbadisc.c
6961. int void lpfc_unregister_fcf_rescan(...) {
          ......
          // Race: multiple branches are affected!
6979.     phba->fcf.fcf_flag = 0;  // No lock is held
          ......
7009. }
```

**Thread 2**

```
FILE: linux-6.2/drivers/scsi/lpfc/lpfc_init.c
6741. void lpfc_sli4_async_fip_evt(...)
          ......
6785.     spin_lock_irq(&phba->hbalock);
          ......
6791.     if (phba->fcf.fcf_flag & ...) // Branch1
          ......
6797.     if (phba->fcf.fcf_flag & ...) // Branch2
          ......
6834.     if (phba->fcf.fcf_flag & ...) // Branch3
          ......
6942.     if (phba->fcf.fcf_flag & ...) // Branch4
          ......
6948.     spin_unlock_irq(&phba->hbalock);
          ......
6982. }
```

(c) Undefined behavior in Linux

**Thread 1**

```
FILE: Linux-6.2/sound/core/pcm_memory.c
34. int  do_alloc_pages(struct snd_card *card, ...) {
        ......
        // Race: inconsistent value in check and use
41.     if (card->total_pcm_alloc_bytes + size > ...)
        ......
50.     mutex_lock(&card->memory_mutex);
51.     card->total_pcm_alloc_bytes += ...;
52.     mutex_unlock(&card->memory_mutex);
        ......
55. }
```

**Thread 2**

```
FILE: Linux-6.2/sound/core/pcm_memory.c
57. int  do_free_pages(struct snd_card *card, ...) {
        ......
61.     mutex_lock(&card->memory_mutex);
62.     WARN_ON(card->total_pcm_alloc_bytes < ...);
63.     card->total_pcm_alloc_bytes -= ...;
64.     mutex_unlock(&card->memory_mutex);
        ......
67. }
```

(d) Double fetch issue in Linux

Figure 11: Four example harmful races found by LR-Miner.

race at Line 41. Moreover, in the function `do_alloc_pages`, `card->total_pcm_alloc_bytes` is used with the lock at Line 51, after being checked at Line 41, so the checked and used values of this field can be inconsistent. Note that this field is used to represent the total size of the PCM (Pulse Code Modulation) buffers in sound subsystem, so this race can cause buffer overflow when the value is changed after the check. The attacker can exploit this harmful race to corrupt the memory of PCM buffers and inject malicious sound data.

## 5.4 Comparison Experiment

We aim to compare with existing static approaches of kernel race detection, including Relay [74], RacerX [29], CPALockator [3], Goblint [73], Locksmith [64] and Lockpick [14]. We select Linux 5.17 to check, as many of them can only check old Linux kernel versions including 5.17 but excluding 6.2.

As for Relay, we build it from its source code [65] with minor modification to parse Linux 5.17 code. As for RacerX, its source code is not available, so we tried our best to implement a RacerX-like tool referring to its paper. As for CPALockator, we build it from its source code [21] based on the race detection part of CPAchecker [20]. As for Goblint and LockSmith, we downloaded their source code from their repositories [35, 55], but encountered too many error when parsing Linux 5.17 code. We tried our best to solve these errors but failed. We contacted the authors of these approaches, and they also had no solution of these errors. As for Lockpick, it is not publicly available. Thus, we make a methodology comparison with Goblint, LockSmith and Lockpick.

Note that LR-Miner uses our field-aware mining method and alias-aware checking method to improve the accuracy of kernel race detection, as well as the function summary to improve the efficiency of data race detection. To validate the value of these three key techniques, we implement three tools by modifying LR-Miner: (1) LR-Miner$_{FieldBased}$ that uses a field-based analysis when mining locking rules, without using field graph to check whether the data field and lock field are in the same structure; (2) LR-Miner$_{NonAlias}$ that neglects alias relationships when detecting races with the locking rules mined by LR-Miner; (3) LR-Miner$_{NonSum}$ that performs inter-procedural analysis without using function summaries.

We run the three existing static approaches and three implemented tools to check Linux 5.17 with a timeout of 7 days (168h), and summarize the results in Table 4. We have the following observations from the table:

(1) LR-Miner finds all the real and harmful races found by Relay, RacerX-like and CPALockator, and it also finds more real and harmful races missed by these approaches, with a lower false positive rate. Note that Relay and RacerX-like report too many races (12,291 and 20,457) that require much manual work to check, so we randomly select 300 of them to check. We analyze the methodology of the three existing approaches according to their papers, and summarize the following reasons why LR-Miner produces better results:

Relay uses classical lockset analysis [69] to detect races by checking the intersection of locksets in different code paths, without using locking rules about kernel concurrency. Moreover, it uses flow/field-insensitive alias analysis when maintaining locksets, and neglects code-path feasibility during analysis. In comparison, LR-Miner accurately mines locking rules and detects races, with flow/field-sensitive alias analysis and SMT-based path-feasibility validation.

RacerX uses imprecise dataflow analysis that can statistically identify simple locking rules from kernel code, without checking whether the data field and lock field are in the same structure. Moreover, RacerX neglects alias relationships and code-path feasibility during analysis. Thus, RacerX produces many incorrect locking rules and false races. In comparison, LR-Miner uses field graph and considers alias relationships to more accurately mine locking rules and detect races.

CPALockator combines imprecise but fast classical lockset analysis and precise but slow model checking, without using locking rules. However, classical lockset analysis assumes that all the functions can be concurrently executed, which is actually incorrect during kernel execution [6]. More-

| Description | Relay | RacerX-like | CPALockator | LR-Miner$_{FieldBased}$ | LR-Miner$_{NonAlias}$ | LR-Miner$_{NonSum}$ | LR-Miner |
|---|---|---|---|---|---|---|---|
| Found races | 12,291 | 20,457 | 782 | 3,171 | 742 | N/A | 385 |
| Real races | 8 in 300 | 6 in 300 | 21 | 185 | 51 | N/A | 274 |
| Harmful races | 4 in 300 | 3 in 300 | 9 | 154 | 40 | N/A | 188 |
| Time usage | 1h7m | 4h38m | 126h15m | 5h25m | 5h32m | >168h | 5h51m |

Table 4: Comparison results of Linux 5.17.

over, to accelerate model checking for concurrency analysis, CPALockator sacrifices analysis accuracy by pruning many code paths, simplifying path constraints, etc. In comparison, LR-Miner mines accurate locking rules from kernel code, without using the incorrect assumption used by CPALockator; and then it uses alias relationships and function summaries to improve both accuracy and efficiency of race detection.

(2) LR-Miner spends more time than Relay and RacerX-like, as it has more complicated analyses of building field graphs, computing accurate alias relationships, etc. LR-Miner spends less time than CPALockator, as CPALockator heavily suffers from concurrency state explosion of model checking, despite achieving some acceleration by sacrificing accuracy.

(3) LR-Miner produces fewer false positives and negatives than LR-Miner$_{FieldBased}$ and LR-Miner$_{NonAlias}$, indicating our field-aware mining method and alias-aware checking method can indeed help improve the accuracy of race detection. LR-Miner$_{NonSum}$ cannot finish analysis within the timeout in our evaluation, indicating that summary-based analysis is very important to improving the efficiency of race detection.

***Goblint, Locksmith and Lockpick.*** We make methodology comparison with them, according to their papers. Goblint analyzes just driver code in the Linux kernel, by assuming all driver interface functions can be concurrently executed. However, this assumption is actually incorrect during kernel execution [6], causing many false results. Locksmith checks the intersection of locksets in different code paths, but using the user's annotations. However, the annotations can be wrong provided by the inexperienced user, affecting the accuracy of concurrency analysis. Lockpick requires the user to provide lock specifications, to detect lock misuses including some kinds of races. However, the user can provide wrong lock specifications due to misunderstanding code logics, which can cause many false results. By contrast, LR-Miner detects races by mining locking rules that reflect kernel concurrency conventions, without concurrency assumption, user annotation or locking specifications, so it can achieve better accuracy.

## 6 Discussion

**Field graph.** LR-Miner exploits field graph to conveniently describe the relation between each data field and lock field in structures. Compared to existing field-based analysis [12, 31, 68] that uses just structure type and field name to identify aliased structure fields, LR-Miner can achieve better accuracy in lockset analysis by using field graph.

**Exploitability of the found harmful races.** After knowing the locations of these harmful races, the attacker can intentionally trigger them by preparing the related concurrency workloads, and then can transform them into deterministic vulnerabilities like null-pointer dereferences and use-after-free issues, which can be exploited to perform DoS attacks, privilege escalation, etc. Several works [49, 82, 86] have explored the vulnerability exploitation of such harmful races.

**Detecting other concurrency issues.** Besides races, we believe LR-Miner can be extended to detecting other concurrency issues in OS kernels, like atomicity violations, deadlocks and concurrency use-after-free issues. For example, LR-Miner can mine lock-order rules like that *lock A should be acquired before lock B when the two locks need to be held together*, and such rules can be used to detect kernel deadlocks caused by ABBA locking cycles. Besides, LR-Miner can also identify concurrency use-after-free by checking whether a racy variable is freed by the kernel.

**Limitations and future works.** First, LR-Miner still reports some false results due to neglecting bit vectors in structures, omitting special assertions about locks, etc. Thus, we plan to handle these cases in LR-Miner to further reduce false positives. Second, LR-Miner fails to handle special locks, like RCU and customized locks (including reference counts, condition variables, etc), so it may miss some races involving these locks. Thus, we plan to study the usage of these special locks, and improve LR-Miner to detect the related races. Third, LR-Miner detects races caused by missing lock protection at present, and fails to detect races involving non-lock concurrency mechanisms like wait queue and completion mechanisms. Thus, we plan to improve LR-Miner to support the analysis of these mechanisms and detect the related races. Fourth, LR-Miner just considers the sequential propagation of racy variables in harmful estimation, without checking their concurrent propagation across different functions, and thus may introduce some inaccuracy. To solve this problem, we plan to analyze possible thread interleavings and check the related concurrent propagation, to improve the accuracy of harmful estimation. Finally, besides races, we plan to extend LR-Miner to detect other concurrency issues in OS kernels.

## 7 Related Work

**Dynamic race detection.** Many dynamic approaches detect races based on address watchpoint [30, 40, 46], lockset analysis [4, 16, 66, 69, 87], happens-before relation [11, 13, 33, 50, 83], or hybrid of them [28, 47, 48, 72, 78]. However, these approaches require the tested program to actually cover different thread interleavings for deep race detection. To solve this problem, several dynamic approaches [15, 36, 39, 42, 79] introduce coverage-guided fuzzing into race detection. These approaches automatically control thread scheduling or inject random delays, guided by new concurrency metrics that reflect the covered thread interleavings during program execution. Despite using concurrency fuzzing, these approaches still miss some program code and infrequent thread interleavings, due to limited test cases and testing time, causing many real races to be missed.

Different from the above approaches, LockDoc [54] first analyzes kernel execution traces to identify locking rules, and then detects the violations of these rules as kernel races. However, identical to the above approaches, LockDoc also suffers from limited code coverage of kernel execution, and thus it misses many execution situations for the analyzed traces, which affects the accuracy of the mined locking rules.

Different from LockDoc, LR-Miner statically analyzes kernel source code, to conveniently handle many more possible execution situations, without actual kernel execution. Due to higher analysis coverage, LR-Miner can achieve better accuracy of locking-rule mining to benefit race detection and detect more kernel races.

**Static race detection.** Many static approaches [1, 2, 10, 31, 32, 43, 52, 61, 70] focus on race detection in user-level applications. Indeed, different from applications actively executing code, kernel code is often passively executed via system calls invoked by upper-level applications [9, 67]. Thus, kernel concurrency is actually caused by application concurrency in many cases, without having explicit operations of thread creation and termination like applications do. However, these approaches require such thread operations to perform concurrency analysis, and thus they cannot effectively check kernel code to perform static race detection.

A few static approaches [3, 14, 29, 64, 73, 74] can detect kernel races, based on the assumption that all the functions can be concurrently executed or using the user's annotations about code concurrency. As this assumption is actually incorrect during kernel execution and the inexperienced user can provide wrong annotations, these approaches can report many false races. Moreover, some of these approaches [29, 64, 74] use inaccurate lockset analysis that neglects field information about accessed variables and protected locks, which can produce many false results. Besides, these approaches fail to estimate the security impact of the found races. In comparison, LR-Miner mines locking rules and detects races with precise field information and alias relationships, without concurrency assumption or user annotation, so it can achieve better accuracy in race detection. Besides, LR-Miner identifies harmful races, according to representative patterns that can cause security problems.

# 8 Conclusion

We design a novel static analysis approach named LR-Miner, to detect races in OS kernels by mining locking rules from kernel code. Among the found races, it can identify harmful ones, according to representative patterns that can cause security problems. In the evaluation on Linux and FreeBSD, LR-Miner finds 306 real races, 200 of which are estimated to be harmful. 61 of the harmful races have been confirmed. In experimental comparison to existing approaches, LR-Miner finds more real races with better accuracy. LR-Miner is available on https://sites.google.com/view/LR-Miner/.


# Acknowledgment

We thank anonymous reviewers for their helpful advice on the paper. We also thank Linux kernel developers, who gave useful feedback and advice to us.


# References


[1] Martin Abadi, Cormac Flanagan, and Stephen N Freund. Types for safe locking: static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006.

[2] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: checking data sharing strategies for multithreaded C. In *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)*, pages 149–158, 2008.

[3] Pavel Andrianov, Vadim Mutilin, and Alexey Khoroshilov. CPALockator: thread-modular analysis with projections. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 423–427, 2021.

[4] Yoshitaka Arahori. RangeLocker: adaptive range-sensitive lockset analysis for precise dynamic race detection. In *Proceedings of the 19th International Symposium on High Assurance Systems Engineering (HASE)*, pages 184–191, 2019.

[5] Jia-Ju Bai, Qiu-Liang Chen, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Hybrid static-dynamic analysis of data races caused by inconsistent locking discipline in device drivers. *IEEE Transactions on Software Engineering (TSE)*, 48(12):5120–5135, 2022.

[6] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. Effective static analysis of concurrency use-after-free bugs in Linux device drivers. In *Proceedings of the 2019 USENIX Annual Technical Conference*, pages 255–268, 2019.

[7] Jia-Ju Bai, Tuo Li, and Shi-Min Hu. DLOS: effective static detection of deadlocks in OS kernels. In *Proceedings of the 2022 USENIX Annual Technical Conference*, pages 367–382, 2022.

[8] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. Static detection of unsafe DMA accesses in device drivers. In *Proceedings of the 30th USENIX Security Symposium*, pages 1629–1645, 2021.

[9] Jia-Ju Bai, Yu-Ping Wang, and Shi-Min Hu. AutoPA: automatically generating active driver from original passive driver code. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, pages 288–299, 2018.

[10] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. In *Proceedings of the 33th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 1–28, 2018.

[11] Michael D Bond, Katherine E Coons, and Kathryn S McKinley. PACER: proportional detection of data races. In *Proceedings of the 31st International Conference on Programming Language Design and Implementation (PLDI)*, pages 255–268, 2010.

[12] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–230, 2002.

[13] Yan Cai, Jian Zhang, Lingwei Cao, and Jian Liu. A deployable sampling strategy for data race detection. In *Proceedings of the 2016 International Symposium on the Foundations of Software Engineering (FSE)*, pages 810–821, 2016.

[14] Yuandao Cai, Peisen Yao, Chengfeng Ye, and Charles Zhang. Place your locks well: understanding and detecting lock misuse bugs. In *Proceedings of the 32nd USENIX Security Symposium*, pages 3727–3744, 2023.

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

[16] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 366–376, 2019.

[17] Ben-Chung Cheng and Wen-Mei W Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the 21st International Conference on Programming Language Design and Implementation (PLDI)*, pages 57–69, 2000.

[18] Clang compiler. https://clang.llvm.org/.

[19] CLOC tool. https://cloc.sourceforge.net/.

[20] CPAchecker: configurable verification platform. https://cpachecker.sosy-lab.org/.

[21] CPALockator code in CPAChecker. https://github.com/sosy-lab/cpachecker/tree/trunk/src/org/sosy_lab/cpachecker/cpa/datarace.

[22] CVE-2009-1894: race in Linux Pulse Audio sound server for privilege escalation. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1894.

[23] CVE-2013-0871: race in Linux Ptrace part for privilege escalation. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0871.

[24] CVE-2016-5195: Dirty COW in Linux. https://www.cve.org/CVERecord?id=CVE-2016-5195.

[25] CVE-2023-2006: race in Linux RxRPC network protocol for arbitrary code execution. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-2006.

[26] CVE-2023-31081: concurrency null-pointer derference in Linux vidtv driver. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-46862.

[27] CVE-2023-46862: concurrency null-pointer derference in Linux io_uring driver. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-46862.

[28] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: efficiently computing the happens-before relation using locksets. In *Proceedings of the 2006 International Workshop on Formal Approaches to Software Testing*, pages 193–208, 2006.

[29] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th International Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003.

[30] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th International Conference on Operating Systems Design and Implementation (OSDI)*, pages 151–162, 2010.

[31] Cormac Flanagan and Stephen N Freund. Type-based race detection for Java. In *Proceedings of the 21th International Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000.

[32] Cormac Flanagan and Stephen N Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 International Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 90–96, 2001.

[33] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI)*, pages 121–133, 2009.

[34] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *Proceedings of the 40th International Conference on Programming Language Design and Implementation (PLDI)*, pages 1069–1084, 2019.

[35] Goblint repository. https://github.com/goblint/.

[36] Sishuai Gong, Dinglan Peng, Deniz Altınbüken, Pedro Fonseca, and Petros Maniatis. Snowcat: efficient kernel concurrency testing using a learned coverage predictor. In *Proceedings of the 29th International Symposium on Operating Systems Principles (SOSP)*, pages 35–51, 2023.

[37] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of the 22nd International Conference on Programming Language Design and Implementation (PLDI)*, pages 254–263, 2001.

[38] David Hovemeyer and William Pugh. Finding more null pointer bugs, but not too many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 9–14, 2007.

[39] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, pages 754–768, 2019.

[40] Yunyun Jiang, Yi Yang, Tian Xiao, Tianwei Sheng, and Wenguang Chen. DRDDR: a lightweight method to detect data races in Linux kernel. *The Journal of Supercomputing*, 72(4):1645–1659, 2016.

[41] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using context-sensitive software fault injection. In *Proceedings of the 29th USENIX Security Symposium*, pages 2595–2612, 2020.

[42] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Context-sensitive and directional concurrency fuzzing for data-race detection. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS)*, 2022.

[43] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 2009 International Symposium on the Foundations of Software Engineering (FSE)*, pages 13–22, 2009.

[44] Baris Can Cengiz Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with Portend. In *Proceedings of the 17th International Conference On Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 185–198, 2012.

[45] George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. An efficient data structure for must-alias analysis. In *Proceedings of the 27th International Conference on Compiler Construction (CC)*, pages 48–58, 2018.

[46] KCSAN: kernel concurrency sanitizer. https://github.com/google/ktsan/wiki/KCSAN.

[47] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. Data race detection on compressed traces. In *Proceedings of the 2018 International Symposium on the Foundations of Software Engineering (FSE)*, pages 26–37, 2018.

[48] KTSAN: kernel thread sanitizer. https://github.com/google/kernel-sanitizers/tree/ktsan.

[49] Yoochan Lee, Changwoo Min, and Byoungyoung Lee. ExpRace: exploiting kernel races through raising interrupts. In *Proceedings of the 30th USENIX Security Symposium*, pages 2363–2380, 2021.

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

[51] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. Path-sensitive and alias-aware typestate analysis for detecting OS bugs. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 859–872, 2022.

[52] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. When threads meet events: efficient and precise static race detection with origins. In *Proceedings of the 42nd International Conference on Programming Language Design and Implementation (PLDI)*, pages 725–739, 2021.

[53] LLVM compiler infrastructure. https://llvm.org/.

[54] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. LockDoc: trace-based analysis of locking in the Linux kernel. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, pages 1–15, 2019.

[55] Locksmith code repository. https://github.com/polyvios/locksmith/.

[56] Kai Lu, Zhendong Wu, Xiaoping Wang, Chen Chen, and Xu Zhou. RaceChecker: efficient identification of harmful data races. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 78–85, 2015.

[57] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *Proceedings of the 28th USENIX Security Symposium*, pages 1769–1786, 2019.

[58] Shan Lu, Soyeon Park, and Yuanyuan Zhou. Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering (TSE)*, 38(4):844–860, 2011.

[59] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. DR.CHECKER: a soundy analysis for Linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium*, pages 1007–1024, 2017.

[60] Paul E McKenney and Jonathan Walpole. Introducing technology into the Linux kernel: a case study. *ACM SIGOPS Operating Systems Review*, 42(5):4–17, 2008.

[61] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th International Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319, 2006.

[62] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th International Conference on Programming Language Design and Implementation (PLDI)*, pages 22–31, 2007.

[63] Brian Norris and Brian Demsky. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 28th international Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 131–150, 2013.

[64] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. Locksmith: practical static race detection for C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(1):1–55, 2011.

[65] Relay code repository. https://cseweb.ucsd.edu/~jvoung/race/.

[66] Gabriel Ryan, Abhishek Shah, Dongdong She, and Suman Jana. Precise detection of kernel data races with probabilistic lockset analysis. In *Proceedings of the 44th IEEE Symposium on Security and Privacy*, pages 2086–2103, 2023.

[67] Leonid Ryzhyk, Yanjin Zhu, and Gernot Heiser. The case for active device drivers. In *Proceedings of the 1st Asia-Pacific Workshop on Systems (APSys)*, pages 25–30, 2010.

[68] Amit Sasturkar, Rahul Agarwal, Liqiang Wang, and Scott D Stoller. Automated type-based analysis of data races and atomicity. In *Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 83–94, 2005.

[69] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[70] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th International Symposium on Principles of Programming Languages (POPL)*, pages 387–400, 2012.

[71] Xin Tan, Yuan Zhang, Xiyu Yang, Kangjie Lu, and Min Yang. Detecting kernel refcount bugs with two-dimensional consistency checking. In *Proceedings of the 30th USENIX Security Symposium*, pages 2471–2488, 2021.

[72] ThreadSanitizer: a data race detector for C/C++ programs. https://github.com/google/sanitizers/wiki/ThreadSanitizerCppManual.

[73] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pages 391–402, 2016.

[74] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the 2007 International Symposium on The Foundations of Software Engineering (FSE)*, pages 205–214, 2007.

[75] Pengfei Wang, Jens Krinke, Kai Lu, Gen Li, and Steve Dodier-Lazaro. How double-fetch situations turn into double-fetch vulnerabilities: A study of double fetches in the Linux kernel. In *Proceedings of the 26th USENIX Security Symposium*, pages 1–16, 2017.

[76] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check it again: Detecting lacking-recheck bugs in OS kernels. In *Proceedings of the 25th International Conference on Computer and Communications Security (CCS)*, pages 1899–1913, 2018.

[77] Zhendong Wu, Kai Lu, Xiaoping Wang, Xu Zhou, and Chen Chen. Detecting harmful data races through parallel verification. *The Journal of Supercomputing*, 71:2922–2943, 2015.

[78] Xinwei Xie, Jingling Xue, and Jie Zhang. Acculock: accurate and efficient detection of data races. *Software: Practice and Experience (SPE)*, 43(5):543–576, 2013.

[79] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. KRACE: data race fuzzing for kernel file systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 1643–1660, 2020.

[80] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in OS kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy*, pages 661–678, 2018.

[81] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. VFix: Value-flow-guided precise program repair for null pointer dereferences. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 512–523, 2019.

[82] Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan. Concurrency attacks. In *Presented as part of the 4th USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2012.

[83] Kunpeng Yu, Chenxu Wang, Yan Cai, Xiapu Luo, and Zijiang Yang. Detecting concurrency vulnerabilities based on partial orders of memory and thread events. In *Proceedings of the 2021 International Symposium on the Foundations of Software Engineering (FSE)*, pages 280–291, 2021.

[84] Z3: a theorem prover from Microsoft Research. `https://github.com/Z3Prover/z3`.

[85] Dongyang Zhan, Xiangzhan Yu, Hongli Zhang, and Lin Ye. ErrHunter: detecting error-handling bugs in the Linux kernel through systematic static analysis. *IEEE Transactions on Software Engineering (TSE)*, 49(2):684–698, 2022.

[86] Shixiong Zhao, Rui Gu, Haoran Qiu, Tsz On Li, Yuexuan Wang, Heming Cui, and Junfeng Yang. OWL: understanding and detecting concurrency attacks. In *Proceedings of the 48th International Conference on Dependable Systems and Networks (DSN)*, pages 219–230, 2018.

[87] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: hardware-assisted lockset-based race detection. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132, 2007.