Static Race Detection in OS Kernels by Mining Locking Rules

Abstract

To take advantage of multiple-core architecture of modern CPUs, an operating system (OS) is expected to handle concurrency, inevitably introducing many concurrency issues. And data race is one of the most harmful concurrency issues that can trigger memory or logic bugs. Data races are caused by wrongly accessing shared data in different threads. To ensure correct access to shared data, fine-grained locking mechanisms are introduced. However, it is hard to determine whether a lock should be held when accessing a specific variable (locking rules) even for an expert developer, because of poor document and complicated logic of OS code. As a result, OS kernel is prone to data races, because necessary locks may be missed by mistake. Static analysis is a common technique to improve code quality, but it is quite challenging to detect data races in OS kernels automatically, due to lack of knowledge of locking rules and high complexity of concurrent execution.

In this paper, we design a practical static analysis approach named RaceMiner, to effectively detect data races and estimate their harmfulness in OS kernels by mining locking rules. RaceMiner first employs an alias-aware rule mining method to automatically deduce locking rules, and detects data races caused by violation of these rules. And then performs a lock-usage analysis to filter out false positives caused by code that can not execute concurrently. At last, RaceMiner extracts harmful data races from all detected data races through a pattern-based estimation. We have evaluated RaceMiner on Linux 6.2, and find 273 data races, with a false positive rate of 19.9%. Among these data races, 88 are estimated to be harmful. We have reported these harmful data races to Linux kernel developers, and 32 of them have been confirmed.

1. Introduction

To take advantage of multiple-core architecture of modern CPUs, an OS kernel is expected to handle concurrency. However, concurrent execution can inevitably introduce concurrency issues. Data race is one of the most harmful concurrency issues that occurs when multiple threads access a shared variable concurrently and at least one of the accesses is a write. Some races are harmful and can cause serious bugs like null-pointer dereferences, data inconsistencies and double fetches.

To detect data races effectively, some approaches [1, 2, 5, 6, 11, 14, 15, 38, 46] rely on developers to supply annotations that describe the locking discipline such as which lock is required for accesses to a specific variable, and detect data races by searching variable accesses that violate these disciplines. For example, Clang thread safety analysis [11] requires developers to label a variable and the lock to protect it with a `GUARDED_BY` attribute. And then detects data races by finding variable accesses that violate the given attribute. However, such annotation-based approaches are not suitable for race detection in OS kernels, because even an expert developer can not provide accurate annotations, due to poor document and complicated logic of OS codes.

Other static approaches [9, 13, 33, 37, 41] employ lockset-based analysis to detect data races automatically. For a given variable, they first collect locks that are acquired for each access to it, and then detect data races by checking whether the intersection of locks for different accesses is empty. However, they do not consider alias relationships [13, 41] or just use imprecise flow-insensitive alias analysis [9, 33, 37], and thus can introduce both false positives and negatives.

Dynamic analysis can acquire run-time information such as memory address, and thus complex alias analysis can be avoided. Therefore, some approaches [21, 27, 28, 29, 30] detect data races dynamically to improve precision. They mine implicit code rules in softwares by statistically analyzing execution traces, and then use the mined rules to detect related bugs. For example, LockDoc [28] records accesses to variables and lock acquisitions of an instrumented Linux kernel, and then infers locking rules from the recorded accesses. After that, LockDoc automatically locates variable accesses that violate the inferred locking rules to detect concurrency issues such as data races. However, dynamic analysis suffers from low code coverage, and thus the locking rules inferred through execution traces can be imprecise. Besides, dynamic analyses are hard to deploy, because they need to run OS kernels with instruments. At last, it is difficult for dynamic tools to estimate the harmfulness of a data race, because race conditions are hard to trigger by running the checked software [7, 16, 26, 45].

In this paper, we design a practical static analysis approach named RaceMiner, to automatically detect data races and estimate the harmfulness of these data races in OS kernels. RaceMiner consists of three key techniques:

(T1) RaceMiner uses an *alias-aware rule mining method* to automatically deduce locking rules about whether accesses to a specific data structure field should be protected and which data structure field the protecting lock exist in. We observe that the variable that is accessed and the lock to protect the access often exist in the same data structure. And this relationship between the accessed variable and the protecting lock can be effectively inferred from a special data structure named alias graph [24, 25]. Specifically, variables in the same data structure have a common ancestor in the alias graph, and thus whether an accessed variable and a specific lock are in the same data structure can be inferred through finding a common ancestor in the alias graph. Moreover, with benefits from precise field-sensitive alias relationships of alias graph, RaceMiner can effectively extract the accessed field and its corresponding lock field. After obtaining the accessed field and lock field pairs, RaceMiner calculates the proportion of field accesses protected by a specific lock field in all field accesses. If the proportion is larger than a given threshold, the accessed field is determined to be protected by the lock field for a specific data structure.

(T2) RaceMiner uses a *lock-usage analysis* to filter out false positives by validating concurrency of the kernel code. After mining locking rules, RaceMiner detects data races by checking whether a given variable access violates the locking rules. To reduce false negatives, RaceMiner conservatively assumes every two functions can execute concurrently, and thus can introduce false positives. However, we observe that each kernel module has an initialization phase, after which many functions can be called concurrently by other modules. When performing initialization, the kernel module serially initializes the locks and prepares other data for subsequent operations. And thus functions with the lock initialization and functions called by them tend not to execute concurrently. Based on this observation, RaceMiner extracts all functions that are reachable from lock initialization functions such as `spin_lock_init()` in the function call graph, and suppose that these functions can not execute concurrently.

(T3) RaceMiner uses a *pattern-based estimation* to extract harmful races that can trigger memory or logic bugs such as null-pointer dereferences, data inconsistencies and double fetches. Some data races are benign, and developers do not put effort into repairing them because they can not cause serious memory or logic bugs. Therefore, it is important to estimate the harmfulness a data race can bring. We observe that harmful data races can be estimated by specific patterns. For example, if a raced data performs as an operand of a dereference instruction, a null-pointer dereference can occur; if a raced data structure is accessed for more than once and each access gets its different fields, a data inconsistency can occur; if a data race is caused by an unprotected write, and it is likely to introduce double fetches.Based on this observation, RaceMiner exploits some patterns to extract harmful data races that can cause memory or logic bugs automatically.

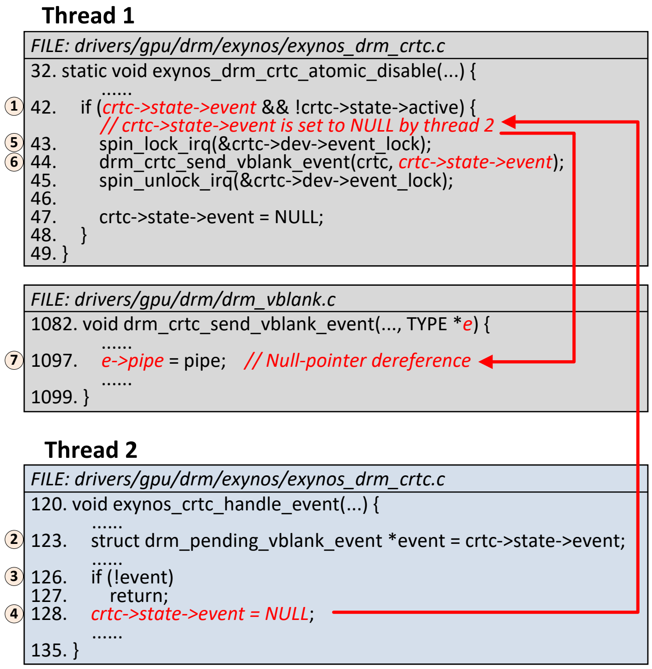
We have implemented RaceMiner with LLVM [10] and Z3 [43]. RaceMiner performs automated static analysis on the LLVM bytecode of the checked OS kernel. Overall, we make three main contributions in this paper:

- We analyze the challenges to detect races in kernels, and propose three key techniques to address these challenges: (T1) an *alias-aware rule mining method* to automatically deduce locking rules; (T2) a *lock-usage analysis* to filter out false positives caused by code that can not execute concurrently; (T3) a *pattern-based estimation* to extract harmful data races that can cause memory or logic bugs such as null-pointer dereferences, data inconsistencies and double fetches.
- Based on these three key techniques, we design a practical static analysis approach named RaceMiner, to effectively detect data races and estimate the harmfulness of these data races in OS kernels.
- We have evaluated RaceMiner on Linux 6.2, and find 273 real data races, with a false positive rate of 19.9%. Among these data races, 88 are estimated to be harmful. We have reported these harmful bugs to Linux kernel developers, and 32 of them have been confirmed.

The rest of this paper is organized as follows. Section 2 introduces the motivation and challenges of data race detection in OS kernels. Section 3 introduces our key techniques to address these challenges. Section 4 introduces RaceMiner. Section 5 shows our evaluation. Section 6 makes a discussion about RaceMiner. Section 7 presents related work, and Section 8 concludes this paper.

2. Motivation

2.1. A Motivating Example



```
Thread 1
FILE: drivers/gpu/drm/exynos/exynos_drm_crtc.c
    32. static void exynos_drm_crtc_atomic_disable(...) {
              ......
①  42.     if (crtc->state->event && !crtc->state->active) {
               // crtc->state->event is set to NULL by thread 2
⑤  43.         spin_lock_irq(&crtc->dev->event_lock);
⑥  44.         drm_crtc_send_vblank_event(crtc, crtc->state->event);
    45.         spin_unlock_irq(&crtc->dev->event_lock);
    46.
    47.         crtc->state->event = NULL;
    48.     }
    49. }

FILE: drivers/gpu/drm/drm_vblank.c
    1082. void drm_crtc_send_vblank_event(..., TYPE *e) {
              ......
⑦  1097.     e->pipe = pipe;    // Null-pointer dereference
              ......
    1099. }

Thread 2
FILE: drivers/gpu/drm/exynos/exynos_drm_crtc.c
    120. void exynos_crtc_handle_event(...) {
              ......
②  123.     struct drm_pending_vblank_event *event = crtc->state->event;
              ......
③  126.     if (!event)
    127.         return;
④  128.     crtc->state->event = NULL;
              ......
    135. }
```

Figure 1: A null-pointer dereference due to data race in Linux 6.2.

Figure 1 shows a real null-pointer dereference caused by a data race in the Linux DRM driver. In this example, We exploit ⓝ to represent the execution order of instructions and show one execution case on the left of Figure 1. In the DRM driver, the functions `exynos_drm_crtc_atomic_disable()` and `exynos_crtc_handle_event()` can execute concurrently. In Thread 1, the variables `crtc->state->event` and `crtc->state->active` are checked by an if statement in the function `exynos_drm_crtc_atomic_disable()`. After the condition is calculated to be true, the function `exynos_crtc_handle_event()` execute in Thread 2. In this function, the value of `crtc->state->event` is assigned to `event` (②) and then `event` is checked in an if statement (③). If it is not NULL, the variable `crtc->state->event` is assigned with NULL (④). Right after this assignment, the function `drm_crtc_send_vblank_event()` is called in Thread 1 (⑥) with the argument `crtc->state->event`, after acquiring the lock `crtc->dev->event_lock` (⑤). In the called function, the variable `crtc->state->event` is finally dereferenced through `e->pipe` (⑦). In this execution case, the variable `crtc->state->event` is first assigned with a NULL value and then dereferenced, and thus a null-pointer dereference can occur.

This bug is triggered only when `crtc->state->event` is set to NULL by Thread 2 right after the first condition of the if statement in Thread 1 is calculated to be true. Such a requirement is difficult to satisfy by executing existing test suites. In fact, this bug had existed for nearly 6 years since Linux 4.14 (Released in Nov. 2017), and it was fixed by us based on a report generated by RaceMiner.

2.2. Challenges

Detecting data races and estimating their harmfulness in OS kernels have three main challenges:

C1: Getting locking rules. The relationship between variables and locks is not well documented in OS kernels, making it hard to determine whether a specific variable should be protected by a lock and which lock is required, even for an expert developer. And thus existing annotation-basedapproaches [1, 2, 5, 6, 11, 14, 15, 38, 46] are difficult to apply to race detection in OS kernels. Other approaches [9, 13, 33, 37, 41] employ lockset-based analysis to detect data races automatically, but they do not consider alias relationships [13, 41] or just use imprecise flow-insensitive alias analysis [9, 33, 37]. However, due to the heavy use of pointers and data structure fields in kernel code, the alias relationships between variables can be very complex, and thus lacking effective alias analysis can introduce both false positives and false negatives.

C2: Dropping false data races. Due to lack of run-time information, static analysis suffers from false positives. For example, each data race involves more than one code path that should be able to concurrently execute. However, which code can execute concurrently is not well documented for an OS kernel, and it is also hard to determine concurrent code statically due to the complexity of OS code. Thus static analysis can report many false data races.

C3: Estimating harmfulness of data races. Many data races are benign or introduced by developers deliberately to improve the performance of OS kernels, and can not cause memory or logic bugs in fact. Therefore, developers are unwilling to put effort into repairing these data races. To automatically detect harmful data races, most approaches [22, 23, 34, 39] explore thread inter-leavings dynamically to trigger data races and observe their impact on the checked software. However, they suffer from low code coverage and thus can miss many real harmful data races.

3. Race Detection by Mining Locking Rules

To address the above challenges, we propose three key techniques. For *C1*, we propose an *alias-aware rule mining method* to automatically deduce locking rules. For *C2*, we propose a *lock-usage analysis* to filter out false positives caused by kernel code that can not execute concurrently. For *C3*, we propose a *pattern-based estimation* to extract harmful data races that can trigger memory or logic bugs such as null-pointer dereference, data inconsistencies and double fetches. We introduce them as follows:

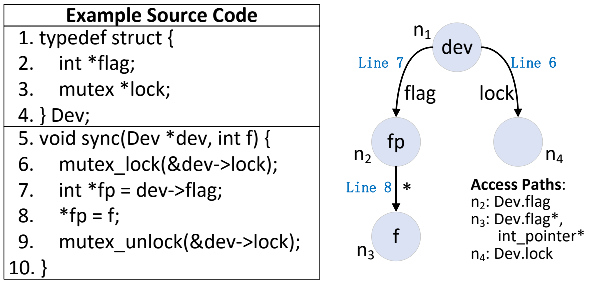
3.1. Alias-Aware Rule Mining Method

The relationship between variables and locks is not well documented in OS kernels, but it can be inferred from the kernel code. Specifically, a variable is often protected by the lock exist in the same data structure. And thus if a variable is accessed after acquiring a lock existing in the same data structure in most cases, it is likely to be protected by the lock. Whether a variable and the protecting lock exist in the same data structure can be determined through an alias graph [24, 25], by finding their common ancestor. Based on this insight, we propose an *alias-aware rule mining method* to deduce locking rules automatically. Moreover, with benefits from precise field-sensitive alias relationships of alias graph, our alias-aware rule mining method can effectively extract accessed filed and its corresponding lock field.

Alias Graph. It is an important data structure to infer relationship between variable and its protecting lock in our analysis, so we introduce it and its update first.

An alias graph is a 2-tuple $G = \langle N, E \rangle$, where N is a set of nodes, and each node n represents an alias set that points to one abstract object. E is a set of labeled edges. Each edge is labeled with a data structure field or a dereference operator "$*$", which represents how an abstract object is accessed. In an alias node, a variable residing in a node followed by a sequence of edge labels forms an access path [8, 24]. In this paper, we replace the variable in an access path with its structure name to represent a data structure field.

An alias graph is updated by handling four types of instructions that change alias relationships: MOVE($v_1 = v_2$), STORE ($*v_2 = v_1$), LOAD ($v_1 = *v_2$) and GEP ($v_1 = \&v_1 - > f$). We exploit n_x to represent the node whose representing alias set includes v_x, and introduce how the four types of instructions update alias graphs. For a MOVE operation, v_1 is moved from n_1 to n_2. After this operation, v_1 and v_2 are represented by the same node, which indicates they become aliases. For a STORE operation, the existing outgoing edge from n_2 is dropped first, and then a new edge labeled with $*$ from n_2 to n_1 is inserted. After this operation, v_1 and $*v_2$ are represented by the same node, which indicates they become aliases. For a LOAD operation, the analysis first finds the destination node of the edge that comes from n_2 and is labeled with $*$, and then moves v_1 to the destination node. And after this operation, v_1 and $*v_2$ are represented by the same node, which indicates they become aliases. GEP operation is similar to LOAD, expect that the edge is labeled with a data structure field f, instead of a dereference operator $*$.



Example Source Code

```
1. typedef struct {
2.     int *flag;
3.     mutex *lock;
4. } Dev;
5. void sync(Dev *dev, int f) {
6.     mutex_lock(&dev->lock);
7.     int *fp = dev->flag;
8.     *fp = f;
9.     mutex_unlock(&dev->lock);
10. }
```

(a) Souce code (b) Alias graph

Figure 2: Example of alias graph.

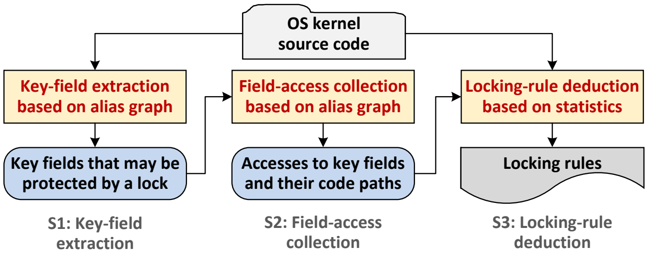
Example. Figure 2 shows a piece of driver-like source code and its alias graph. In this example, after a GEP (&dev->lock) operation at Line 6, an edge labeled with `lock` from node n_1 to node n_4 is inserted. Similarly, an edge labeled with `flag` from node n_1 to node n_2 is inserted after Line 7. At last, an edge labeled with a dereference operator ($*$) is inserted after the STORE (*fp = f) operation at Line 8. The final alias graph is shown in Figure 2(b), and access paths are shown in the bottom left corner. Take node n_3 as an example, it can represent two fields, one is `Dev.flag*`, and the other is `int_pointer*` (we exploit int_pointer to represent a pointer points to an integer, and regard it as a data structure for convenience).

```
define: GetProtectedFieldAccess(var_node, lock_node, alias_graph)
1: ancestor_node := GetCommonAncestor(var_node, lock_node);
2: if ancestor_node is NULL then
3:     return <NULL, NULL>;
4: end if
5: var_field := GetAccessPath(ancestor_node, var_node);
6: lock_field := GetAccessPath(ancestor_node, lock_node);
7: return <var_field, lock_field>;
```

Figure 3: Pseudocodes to get accessed field and protecting lock.

Given an alias graph, whether a variable and a lock exist in the same data structure can be determined by finding a common ancestor. If they are in the same data structure, the variable is likely to be protected by the lock. Figure 3 shows the pseudocode to get the field of the accessed variable and the field of the protecting lock in the form of access path, if they exist in the same data structure. Given a node of an accessed variable and a node of a lock variable, the analysis first gets the common ancestor of the two nodes (Line 1). And then, if the common ancestor does not exist, the analysis returns a NULL pair (Lines 2-3). Otherwise, the analysis gets the access paths for the node of the accessed variable and the node of the protecting lock from the common ancestor (Lines 5-7).

Take the alias graph in Figure 2 as an example, the accessed variable `dev->flag` is represented by node n_2, and the lock `dev->lock` is represented by node n_4. The two nodes have a common ancestor n_1, and thus the accessed variable `dev->flag` and the lock `dev->lock` can be inferred to exist in the same data structure (namely `Dev`). Therefore, the structure field Dev.flag is likely to be protected by the lock stored in the structure field Dev.lock.



OS kernel source code
Key-field extraction based on alias graph
Field-access collection based on alias graph
Locking-rule deduction based on statistics
Key fields that may be protected by a lock
Accesses to key fields and their code paths
Locking rules
S1: Key-field extraction
S2: Field-access collection
S3: Locking-rule deduction

Figure 4: Workflow of locking-rule mining.

Based on the alias graph, our alias-aware rule mining method performs an inter-procedural, path-based [25], field-sensitive and alias-aware analysis to effectively mine locking rules about whether accesses to a specific data structure field should be protected and which data structure field the protecting lock exist in. Overall, our alias-aware rule mining method has three main stages shown in Figure 4. In Stage 1, it extracts key fields that may be protected by a lock. In Stage 2, it collects all accesses to key fields as well as code paths these accesses exist in. In Stage 3, it deduces locking rules by calculating the proportion of field accesses protected by a specific lock in all field accesses.

S1: Key-field extraction. The OS kernel has a large code base with numerous variables. However, only a small part of variables should be protected by a specific lock, and thus collecting all variable accesses can introduce much unnecessary overhead. Generally, given a data structure field, only a small portion of accesses to it can miss necessary protecting locks, due to the carelessness of developers. Based on this insight, our analysis first extracts key fields that may need to be protected by a specific lock, by performing a lock-set analysis to find whether a given variable is accessed after acquiring a lock in the same data structure in any code path.

Figure 5 shows the lock-set analysis based on alias graph. The analysis first gets the operand of the instruction (Line 1), and then gets the node of the operand from the alias graph (Line 2). If the instruction is a lock operation, the node of the operand is inserted into the lock set, which stores all nodes of the acquired locks (Lines 3-4). Otherwise, if the instruction is an unlock operation, the node of the operand is removed from the lock set (Lines 5-6).

```
define: UpdateLockSet(inst, lock_set, alias_graph)
1:  lock_var := GetOperand(inst);
2:  lock_node := GetAliasNode(lock_var, alias_graph);
3:  if inst is a lock opeartion then
4:      insert lock_node into lock_set;
5:  else if inst is an unlock opeartion then
6:      remove lock_node from lock_set;
7:  end if
8:  return lock_set;
```

Figure 5: Pseudocodes of lock-set analysis.

```
define: HandleInstForExtraction(inst, alias_graph, lock_set, key_fields)
1:  alias_graph' := UpdateAliasGraph(alias_graph, inst);
2:  lock_set' := UpdateLockSet(inst, lock_set, alias_graph');
3:  if inst is a write or read operation then
4:      var := GetOperand(inst);
5:      var_node := GetAliasNode(var, alias_graph');
6:      foreach lock_node in lock_set' do
7:          <var_field, lock_field> := GetProtectedFieldAccess(
8:                                      var_node, lock_node, alias_graph');
9:          if var_field is not NULL then
10:             insert var_field into key_fields;
11:         end if
12:     end foreach
13: end if
14: return < alias_graph', lock_set', key_fields>;
```

```
define: ExtractKeyField ()
15: key_fields := ∅;
16: foreach func in OS code without a caller function do
17:     foreach code_path in GetCodePath(func) do
18:         alias_graph := ∅;
19:         lock_set := ∅;
20:         foreach inst in GetInstructions(code_path) do
21:             < alias_graph, lock_set, key_fields> :=
22:                     HandleInstForExtraction(
23:                     inst, alias_graph, lock_set, key_fields);
24:         end foreach
25:     end foreach
26: end foreach
27: return key_fields;
```

Figure 6: Pseudocodes of key-field extraction.

Figure 6 shows the pseudocode to extract key fields that may be protected by a specific lock, based on the lock-set analysis and the alias graph. The analysis starts from each function without a caller function (Lines 16-26) and performs a path-based analysis [25] (Lines 17-25). For each instruction in the code path, the analysis first updates the alias graph according to the instruction with the four operations (MOVE, STORE, LOAD and GEP) that can change alias relationships (Line 1), and then performs a lock-set analysis (Line 2) to get all acquired locks. After updating the alias graph and the lock set, if the instruction is a write or a read, the analysis first gets the node of the operand with the new alias graph (Lines 4-5). And then, for each node of the acquired lock in the lock set, the analysis uses the alias graph to extract the protected data structure field (Lines 6-12). If the field is not NULL, it is inserted into the set of key fields (Lines 9-11). Note that to perform inter-procedural analysis, the analysis copies each function definition into its call sites, but this operation is not shown in Figure 6 for convenience.

S2: Field-access collection. With the key fields extracted in Stage 1, the analysis only needs to collect accesses to the data structure field that are protected in some cases, because if a field is never protected by any lock when is accessed, it is less likely to be shared by different threads, and thus can not introduce any concurrency issue.

```
define: HandleInstForCollection(inst, alias_graph, lock_set, key_fields)
1:  alias_graph' := UpdateAliasGraph(alias_graph, inst);
2:  lock_set' := UpdateLockSet(inst, lock_set, alias_graph');
3:  field_access->var_field := NULL;
4:  field_access->lock_field := NULL;
5:  if inst is a write then
6:      field_access->access_type = write;
7:  else
8:      field_access->access_type = read;
9:  else
10:     return <alias_graph', lock_set', NULL>;
11: end if
12: var := GetOperand(inst);
13: var_node := GetAliasNode(var, alias_graph');
14: // Checking whether the accessed field is protected by any lock.
15: foreach lock_node in lock_set' do
16:     <var_field, lock_field> := GetProtectedFieldAccess(
17:                                 var_node, lock_node, alias_graph');
18:     if var_field is not NULL then
19:         field_access->var_field := var_field;
20:         field_access->lock_field := lock_field;
21:         return <alias_graph', lock_set', field_access>;
22:     end if
23: end foreach
24: // Checking whether the accessed var exist in any key field.
25: foreach var_field in key_fields do
26:     if var exists in the data structure field var_field then
27:         field_access->var_field := var_field;
28:         field_access->lock_field := NULL;
29:         return <alias_graph', lock_set', field_access>;
30:     end if
31: end foreach
32: return <alias_graph', lock_set', NULL>;
```

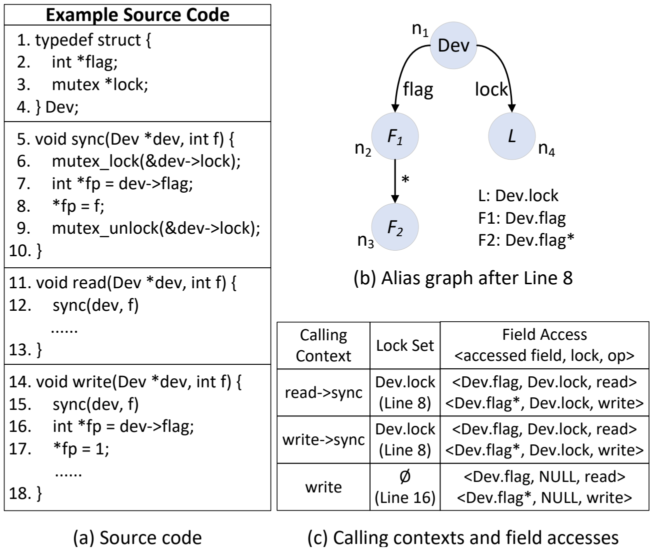
```
define: CollectFieldAccess ()
33: key_fields := ExtractKeyField();
34: field_access_rec := ∅;
35: foreach func in OS code without a caller function do
36:     foreach code_path in GetCodePath(func) do
37:         alias_graph := ∅;
38:         lock_set := ∅;
39:         foreach inst in GetInstructions(code_path) do
40:             <alias_graph, lock_set, field_access> :=
41:                     HandleInstForCollection(
42:                     inst, alias_graph, lock_set, key_fields);
43:             if field_access is not NULL then
44:                 insert <code_path, field_access> into field_access_rec
45:         end foreach
46:     end foreach
47: end foreach
48: return field_access_rec;
```

Figure 7: Pseudocodes of field-access collection.

Figure 7 shows the pseudocode to collect all accesses to key fields. Similarly to the key-field extraction, this stage also performs a path-based analysis. For each instruction in the code path (Lines 39-45), the analysis first updates the alias graph and the lock set according to the handled instruction (Lines 1-2). And then, the access type (either a write or a read) of a field access is set according to the instruction (Lines 5-8). However, if the instruction is not an access operation, the function returns a NULL value for the field access (Line 10). Otherwise, the analysis first gets the node of the operand with the new alias graph (Lines 12-13), and then for each node of the acquired lock in the lock set, the analysis uses the alias graph to extract the fields that the accessed variable and the acquired lock stored in (Lines 16-17). If the fields are found, the field access is returned with the found fields (Lines 18-22). Otherwise, the accessed field is not protected by any lock. In this case, if the accessed variable is stored in a key field, the field access is returned with a NULL lock (Lines 25-30).

S3: Locking-rule deduction. After collecting all accesses to key fields, the analysis deduces locking rules based on statistics. However, on the one hand, distinguishing different accesses to the same data structure field by different program sites is not fine-grained enough, because the access to a field and the acquirement to a specific lock are often packaged in a special function, and all accesses under different calling context are regarded as the same access in this strategy. On the other hand, distinguishing accesses by different code paths can also suffer from inaccuracy when a function contains many branch statements, because accesses to the same field is regarded as different in different code paths, causing numerous accesses to the same field due to large quantities of code paths. Based on this consideration, the analysis distinguishes accesses to the same data structure field by different calling contexts. Specifically, given a key field *f* and a lock *l*, the analysis first finds all access to *f* with lock *l* from the collected field accesses in Stage 2, and gets the number *locked_access_num* by counting different calling contexts extracted from the code paths of these field accesses. Then, the analysis finds all access to *f* (no matter whether a lock is held), and gets the number *all_access_num* in the same way as *locked_access_num*. If the ratio *locked_access_num* / *all_access_num* is larger than a given threshold, and there is at least one write access to *f*, the data structure field *f* is inferred to be protected by the lock *l*. After deducting locking rules, the analysis detects data races by checking whether a field access validates any locking rules.

Example. We use an example in Figure 8 to illustrate how to mine locking rules with the three stages. The analysis first performs a path-based analysis to extract key fields. Take the code path `Line11, Line 12, Line 5, Line 6, Line 7, Line 8, Line 9` as an example. The analysis first updates the alias graph by handling two GEP operations (*&dev->lock* and *fp = dev->flag*) and a STORE operation (**fp = f*), and the final alias graph is shown in Figure 8(b). And then at the same time as alias analysis, the analysis records the acquired lock *Dev.lock* into the lock set after the lock instruction at Line 6. When analyzing the read instruction at Line 7, the analysis finds the node of the accessed field (n_2) and the node of the lock stored in the lock set (n_4) has the same ancestor (n_1), and thus *Dev.flag* is a key field. Similarly, *Dev.flag** is also a key field, due to the write instruction at Line 8.



```
Example Source Code
1. typedef struct {
2.    int *flag;
3.    mutex *lock;
4. } Dev;

5. void sync(Dev *dev, int f) {
6.    mutex_lock(&dev->lock);
7.    int *fp = dev->flag;
8.    *fp = f;
9.    mutex_unlock(&dev->lock);
10. }

11. void read(Dev *dev, int f) {
12.    sync(dev, f)
       ......
13. }

14. void write(Dev *dev, int f) {
15.    sync(dev, f)
16.    int *fp = dev->flag;
17.    *fp = 1;
       ......
18. }
```

(b) Alias graph after Line 8

Calling Context	Lock Set	Field Access <accessed field, lock, op>
read->sync	Dev.lock (Line 8)	<Dev.flag, Dev.lock, read> <Dev.flag*, Dev.lock, write>
write->sync	Dev.lock (Line 8)	<Dev.flag, Dev.lock, read> <Dev.flag*, Dev.lock, write>
write	Ø (Line 16)	<Dev.flag, NULL, read> <Dev.flag*, NULL, write>

(a) Source code (c) Calling contexts and field accesses

Figure 8: Example of locking-rule mining.

After extracting key fields, the method collects all accesses to key fields with lock-set analysis and the alias graph. Take the code path `Line 14, Line 15, Line 5, Line 6, Line 7, Line 8, Line 9, Line 16 and Line 17` as an example, through a lock instruction at Line 6, the analysis records the acquired lock *Dev.lock* into the lock set. When analyzing the read instruction at Line 7, the analysis finds the node of the accessed field (n_2) and the node of the lock stored in the lock set (n_4) has the same ancestor (n_1), and thus records a field access <*Dev.flag*, *Dev.lock*>. Similar to the read instruction at Line 7, the method also records a field access <*Dev.flag**, *Dev.lock*> for the write instruction at Line 8. Then through an unlock operation at Line 9, the analysis removes the lock *Dev.lock* from the lock set. As a result, the key fields *Dev.flag* and *Dev.flag** are accessed without acquiring any lock at Lines 16 and 17, and thus the analysis records two field accesses <*Dev.flag*, *NULL*> and <*Dev.flag**, *NULL*>. The final field accesses are shown in Figure 8(c).

After collecting all field accesses, the analysis finds that the fields *Dev.flag* and *Dev.flag** are accessed under three calling contexts, and two of them are protected by the lock *Dev.lock*. Besides, there is a write to *Dev.flag** at Line 8, and thus the field *Dev.flag** is deduced to be protected by the lock *Dev.lock* (assume the threshold of the ratio *locked_access_num* / *all_access_num* is 0.6). However, the access to *Dev.flag** at Line 17 is not protected by *Dev.lock*, causing a data race.

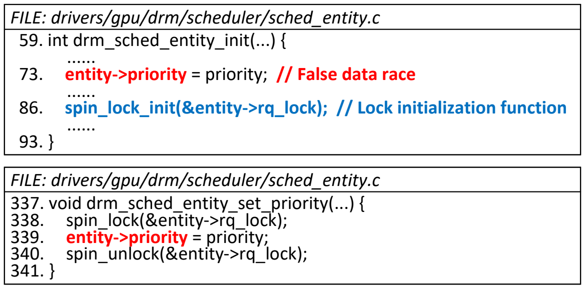
3.2. Lock-Usage Analysis

Our analysis takes functions that have no caller function as the entries like existing work [25], and performs a path-based analysis starting from these entry functions. It assumes that all codes reaching from entry functions can execute concurrently, to reduce false negatives. However, this assumption is too conservative and can introduce many false positives because not all entry functions can execute concurrently in fact. We observe that each kernel module has an initialization phase, after which many functions can be called concurrently by other modules. When performing initialization, the kernel module serially initializes the locks and prepares other data for subsequent operations. And thus functions with the lock initialization and functions called by them tend not to execute concurrently. Based on this observation, we propose a lock-usage analysis to filter out false positives caused by code that can not execute concurrently.

```
define: GetInitFunc(init_lock_func)
 1:  init_funcs := ∅;
 2:  foreach func in the OS kernel do
 3:      foreach call instruction call in func do
 4:          called_func := GetCalledFunc(call);
 5:          UnionSet(func, called_func);
 6:      end foreach
 7:  end foreach
 8:  foreach func in the OS kernel do
 9:      if FindSet(init_lock_func) == FindSet(func) then
10:          insert func into init_funcs;
11:      end if
12:  end foreach
13:  return init_funcs;
```

Figure 9: Pseudocodes of lock-usage analysis.

Our lock-usage analysis uses the union-find set [18] to get all functions that are reachable from the lock initialization functions such as *spin_lock_init()* in the function call graph. Figure 9 shows the pseudocodes of the lock-usage analysis. For each analyzed function in the OS kernel, the analysis first gets all the called functions of it (Lines 3-4), and then union the analyzed function and the called function to update the union-find set (Line 5). After processing all the call instructions, the analysis judges whether a function executes in the initialization phase by checking whether it exists in the same set with the given lock initialization function (Lines 9-11).



```
FILE: drivers/gpu/drm/scheduler/sched_entity.c
 59. int drm_sched_entity_init(...) {
       ......
 73.    entity->priority = priority;  // False data race
       ......
 86.    spin_lock_init(&entity->rq_lock);  // Lock initialization function
       ......
 93. }
```

```
FILE: drivers/gpu/drm/scheduler/sched_entity.c
337. void drm_sched_entity_set_priority(...) {
338.    spin_lock(&entity->rq_lock);
339.    entity->priority = priority;
340.    spin_unlock(&entity->rq_lock);
341. }
```

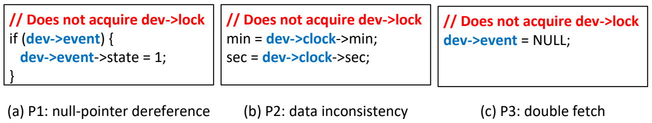
Figure 10: A false data race filtered out by our lock-usage analysis.

Example. Figure 10 shows a false data race filtered out by our lock-usage analysis in the Linux DRM scheduler. For example in *drm_sched_entity_set_priority()*, the accesses to *entity->priority* is protected by the lock *entity->rq_lock* in most cases. Therefore, *entity-priority* is deduced to be protected by the lock *entity->rq_lock*. But in *drm_sched_entity_init()*, *entity->priority* is written without acquiring the lock *entity->rq_lock* and thus introducing a possible data race. However, the lock initialization function *spin_lock_init()* is called at Line 86 by *drm_sched_entity_init()*. And thus *drm_sched_entity_init()* is inferred to execute in the module initialization phase and can not execute concurrently with other functions, and this is a false data race.

3.3. Pattern-Based Estimation

Many data races are benign or introduced by developers deliberately to improve the performance of OS kernels. They can not cause memory or logic bugs in fact, and thus developers are unwilling to put effort into repairing them. We observe that harmful data races match some typical patterns, and propose a *pattern-based estimation* to extract data races that can cause memory or logic bugs.

At present, we propose three patterns that can cause null-pointer dereference, data inconsistencies and double fetches, because these three patterns are common and dangerous in OSes. The three patterns are shown in Figure 11, and we assume that accesses to the fields of *dev* should be protected by *dev->lock*.



```
// Does not acquire dev->lock
if (dev->event) {
  dev->event->state = 1;
}
```

(a) P1: null-pointer dereference

```
// Does not acquire dev->lock
min = dev->clock->min;
sec = dev->clock->sec;
```

(b) P2: data inconsistency

```
// Does not acquire dev->lock
dev->event = NULL;
```

(c) P3: double fetch

Figure 11: Three harmful patterns of data races.

- **P1: Null-pointer dereference.** The two accesses to *dev->event* are not protected by *dev->lock*. This can cause a null-pointer dereference if *dev->event* is set to NULL right after the condition of the if statement is checked to be true. To recognize such a pattern, the analysis first locates the data race, if it is checked by an if statement and then dereferenced, a possible null-pointer dereference can occur.
- **P2: Data inconsistency.** Two different fields of the same data structure are accessed without acquiring *dev->lock*. This can cause a data inconsistency when the data structure field *dev->clock* is changed by another thread right after the access to *dev->clock->min*. To recognize such a pattern, the analysis detects whether multiple fields of the same data structure are accessed without acquiring the protecting lock.
- **P3: Double fetch.** The write access to *dev->event* is not protected by *dev->lock*. This is dangerous because the value of *dev->event* can be modified at any time when other threads access it. If it is changed during two accesses in another thread, a double fetch can occur, introducing unpredictable behavior. The analysis detects such a pattern by judging whether a data race occurs in a write access.

4. RaceMiner Approach

Based on the three key techniques in Section 3, we develop a practical static approach named RaceMiner, to detect data races in OS kernels, and estimate the harmfulness of these data races. We have implemented RaceMiner with Clang [10] and Z3 [43]. RaceMiner can automatically mine locking rules, detects data races and estimates the harmfulness of detected data races. Figure 12 shows the architecture of RaceMiner, which has four phases:

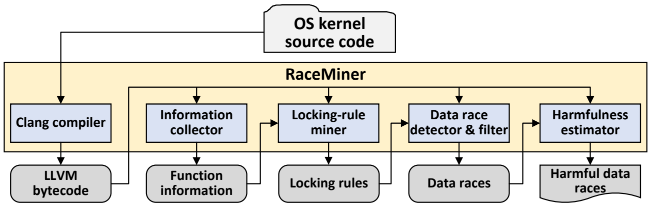


Figure 12: RaceMiner architecture.

P1: Source-code compilation. The Clang compiler compiles the OS source code into LLVM bytecode, and then the information collector scans each LLVM bytecode file to record function information (including the position of each function definition and function name, etc.) in a database. Such information is used in subsequent code analysis for inter-procedural analysis across source files.

P2: Locking-rule Mining. The locking-rule miner uses our alias-aware rule mining method to automatically deduce locking rules about whether accesses to a specific data structure field should be protected and which data structure field the protecting lock exist in

P3: Data race detection and false-positive filtering. The data race detector detects data races by checking whether a given access violates the rules mined by our locking-rule miner. After finding a possible data race, the data race filter uses our locking-usage analysis to determine whether it is a false positive. Besides, for a given data race, there may be multiple code paths from the entry function to its problematic instruction, and thus many repeated data races can be reported. To drop repeated data races, for a new possible data race, the data race filter checks whether its problematic instruction is identical to any existing data race. If so, this possible data race is regarded as repeated and dropped.

P4: Harmfulness estimation. The harmfulness estimator uses our pattern-based estimation to estimate the harmfulness of detected data races, and reports data races that can cause memory or logic bugs. With these results, developers can focus on those harmful data races.

5. Evaluation

To validate the effectiveness of RaceMiner, we evaluate it on the code of Linux kernel 6.2. We run the evaluation on a regular x86-64 desktop with sixteen Intel i7-10700 CPU@2.90GHz processors and 64GB physical memory. We use the kernel configuration *allyesconfig* to enable all kernel code for the x86-64 architecture.

Table 1: Detection results of Linux 6.2.

Description		RaceMiner
Code analysis	Source files (analyzed/all)	xxxK/xxxK
	Source code lines (analyzed/all)	xxxM/xxxM
Locking-rule mining	Key fields / total variables	
	Mined locking rules	
Data race detection	Detected data races (real / all)	273 / 341
	Dropped data races by lock-usage analysis	63
Harmfulness estimation	Null-pointer dereference (confirmed / all)	15 / 20
	Data inconsistency (confirmed / all)	7 / 10
	Double fetch (confirmed / all)	10 / 57
	Total harmful data races (confirmed / all)	32 / 88
Time usage	Key-field extraction	
	Data-race detection	
	Total time	

5.1. Bug Detection

We configure RaceMiner with common lock-acquiring/release functions (like `spin_lock` and `spin_unlock`) to perform lock-set analysis to extract key fields and collect field accesses, and lock-initialization functions (like `spin_lock_init`) to filter our false data races caused by code that can not execute concurrently. And then run RaceMiner to automatically check the kernel source code. We manually check all the data races found by RaceMiner, and Table 1 shows the results, and source code lines are counted by CLOC [12]. From the results, we have the following findings:

Code analysis. RaceMiner can scale to large code bases of OS kernels, and it in total analyzes xxxM lines of code in xxxK source code files within xxx hours. The remaining xxxM lines of code in xxxK source files are not analyzed, as they are not enabled by the *allyesconfig* for the x86-64 architecture. We believe that RaceMiner can also find more data races in other architectures with proper configuration.

Locking-rule mining. An OS kernel has a large code base with numerous variables. Handling all variables when mining locking rules can introduce much overhead. However, we observe that the variable that is accessed and the lock to protect the access often exist in the same data structure. Based on this observation, our locking-rule mining method extracts key fields by finding whether there exists any access to it that is protected by a lock stored in the same data structure. This method drops xxx% variables (xxx out xxx) that need to be handled when mining locking rules, and thus can reduce overhead significantly. After extracting key fields, our locking-rule mining method collects all accesses to these key fields, and then deduces locking rules based on statistics. In this paper, given a data structure field, we set the threshold of the ratio of accesses protected by a specific lock to all accesses to 0.6. Our alias-aware rule mining method can drop many false rules and thus only mines xxx locking rules, which can effectively reduce false data races.

Data race detection. RaceMiner reports 341 data races in the kernel source code. We spent 15 hours on checking these data races and identify that 273 of them are real, with a false positive rate of 19.9%. Besides, our lock-usage analysis drops

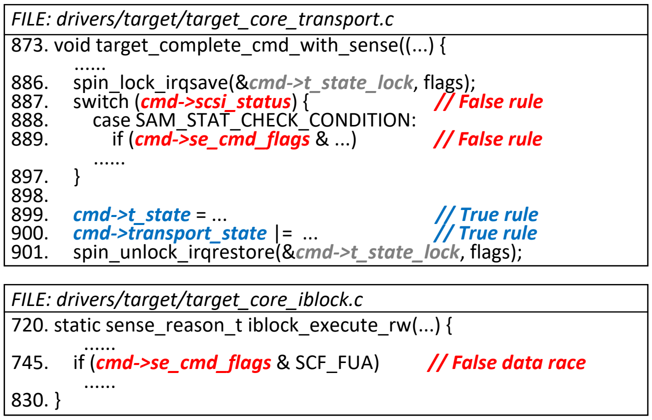
63 false data races and the results show that this strategy can reduce false positives significantly.

Data race estimation. Many data races are benign and can not cause memory or logic bugs, and thus developers are unwilling to put effort into repairing them. We exploit three patterns to detect null-pointer dereferences, data inconsistencies and double fetches as introduced in Section 3.3, and find 88 data races in these patterns. We report them to developers and 32 of them have been confirmed and fixed by them. We still wait for the response of other data races. Moreover, one of the developers wonders if RaceMiner can be used in their CI to detect these problems. The results show that our pattern-based estimation can extract harmful data races effectively, and can considerably reduce the workload of developers.

5.2. False Positives and Negatives

False positives. RaceMiner reports 68 false data races, and through manually checking these false data races, we find that they are introduced for three main reasons:

First, RaceMiner employs an alias-aware rule mining method to deduce locking rules. To improve the precision of mined rules, it assumes that the accessed variable and the protecting lock are stored in the same data structure. Even so, RaceMiner can also deduce some false rules because some developers do not use locks properly. They take a lock/unlock pair to protect accesses to all fields in the same data structure, instead of the exact field that should be protected for convenience. As a result, our alias-aware rule mining method infers that accesses to all fields surrounded by the lock/unlock pair should be protected by the lock. This reason causes RaceMiner to report 47 false data races.



```
FILE: drivers/target/target_core_transport.c
873. void target_complete_cmd_with_sense((...) {
         ......
886.     spin_lock_irqsave(&cmd->t_state_lock, flags);
887.     switch (cmd->scsi_status) {              // False rule
888.        case SAM_STAT_CHECK_CONDITION:
889.          if (cmd->se_cmd_flags & ...)        // False rule
            ......
897.     }
898.
899.     cmd->t_state = ...                       // True rule
900.     cmd->transport_state |= ...              // True rule
901.     spin_unlock_irqrestore(&cmd->t_state_lock, flags);
```

```
FILE: drivers/target/target_core_iblock.c
720. static sense_reason_t iblock_execute_rw(...) {
         ......
745.     if (cmd->se_cmd_flags & SCF_FUA)       // False data race
          ......
830. }
```

Figure 13: A false data race caused by an incorrect locking rule.

Figure 13 shows a false data race caused by an incorrect locking rule. In this example, only accesses to *cmd->t_state* and *cmd->transport_state* should be protected by the lock *cmd->t_state_lock*. However, the lock operation is put ahead of the switch statement by developers, making our alias-aware rule mining method deduce that *cmd->scsi_status* and *cmd->se_cmd_flags* also need to be protected by *cmd->t_state_lock* mistakenly. Based on this incorrect locking rule, RaceMiner reports a false positive at Line 745 when *cmd->se_cmd_flags* is accessed in an if statement. Although this data race is a false positive, it can cause performance degradation because the critical zone protected by the lock/unlock pair should have been limited to Lines 899-900.

Second, in order to reduce memory overhead and improve the performance of data passing among different functions, an integer can be divided into several bit vectors to represent different data structure fields. However, in the LLVM bytecode, accesses to these vectors are divided into a load operation and several bit operations, and thus RaceMiner can not distinguish accesses to different fields and reports false data races, because not all these fields should be protected by a lock. This reason causes RaceMiner to report 14 false data races.

Apart from false data races caused by the two main reasons, there are still 7 other data races. RaceMiner regards functions without caller functions in a kernel module as the entry functions and starts analysis from these entries. However, in 4 cases, an assertion is put at the beginning of the entry function, to guarantee that the required lock is held, but RaceMiner does not consider assertions and the subsequent accesses are regarded as unprotected. The other 3 false data races are accessed with *READ_ONCE* or *WRITE_ONCE*, and can not cause serious issues.

False negatives. RaceMiner may still miss some real data races for three main reasons:

First, our alias-aware mining method only considers accessed variables and protecting locks that exist in the same data structure. However, on the one hand, some accesses to variables may be protected by a global lock in a module, and this lock is used to protect accesses to various variables and does not exist in any data structure. On the other hand, some special lock -acquiring/release functions (such as *rcu_read_lock* and *rcu_read_unlock*) do not have an argument and are not handled in an alias graph. Therefore, locking rules in the above two cases can be missed by our alias-aware mining method. As a result, data races that violate these locking rules can not be detected by RaceMiner.

Second, RaceMiner does not handle function-pointer calls, and thus it can not build complete call graphs for interprocedural analysis. However, RaceMiner starts from functions without caller functions and performs analysis along call graphs. And as a result, functions that are only called indirectly through function pointers can not be analyzed by RaceMiner, and thus data races involving code reached through function-pointer calls can be missed.

Finally, RaceMiner exploits a lock-usage analysis to filter out false data races caused by code that can not execute concurrently. Specifically, the lock-usage analysis extracts all functions that are reachable from the lock initialization functions in a call graph with the union-find set, and assumes that these functions can not execute concurrently. However, some functions can be called at multiple program sites, and some of the program sites may be not reachable from the lock initialization functions, but data races that exist in these functions are all dropped by our lock-usage analysis.

5.3. Case Studies of the Found Harmful Data Races

Figure 14 shows three data races found by RaceMiner in Linux 6.2, and they have been confirmed by Linux kernel developers.

Null-pointer dereference in the HDA sound driver. In Figure 14(a), the functions *snd_hdac_regmap_exit()* and *snd_hdac_regmap_sync()* can execute concurrently. In Thread 2, the variable *codec->regmap* is checked by an if statement in the function *snd_hdac_regmap_sync()*. Right after the condition of the if statement is calculated to be true, the variable *codec->regmap* is set to be NULL by the function *snd_hdac_regmap_exit()* in Thread 1. And then the function *regcache_sync()* is called in Thread 2, with the argument *codec->regmap*, after acquiring the lock *codec->regmap_lock*. In the called function, the variable *codec->regmap* is dereferenced through *map->lock()*. In this execution case, the data structure field *codec->regmap* is first assigned with NULL in Thread 1, and then dereferenced in Thread 2, and thus cause a null-pointer dereference.

Data inconsistency in the GFS2 file system. In Figure 14(b), the functions *gfs2_show_options()* and *gfs2_reconfigure()* can execute concurrently. In Thread 1, several fields such as *gt_statfs_quantum* and *gt_quota_quantum* of *sdp->sd_tune* are accessed and their values are printed to logs. However, if the value of *gt->gt_quota_quantum* is updated by the function *gfs2_reconfigure()* in Thread 2 right before the access to it in Thread 1, the values of different fields recorded in logs can be inconsistent.

Double fetch in the LPFC SCSI driver. In Figure 14(c), the value of the shared variable *phba->fcf.fcf_flag* is set to 0 by the function *lpfc_unregister_fcf_rescan()* in Thread 2, without acquiring the lock *phba->hbalock*, and this can cause double fetches in several program sites. For example, if *phba->fcf.fcf_flag* is set to 0 by Thread 2 between the two accesses to it in the function *lpfc_register_fcf()* in Thread 1 (Lines 1873 and 1874), the values of the two accesses can be different, causing unpredictable behaviors.

6. Discussion

Benefits in other aspects. RaceMiner uses an alias-aware mining method to deduce precise locking rules about whether accesses to a specific data structure field should be protected and which data structure field the protecting lock exist in. Such rules can benefit in various aspects.

First, with these rules, developers can improve the reliability and performance of kernel code. On the one hand, developers can carefully check whether necessary locks are used when writing code, and avoid serious bugs caused by data races in the early developing stage. On the other hand, developers can limit the critical zone protected by a lock/unlock pair to the exact accesses that should be protected, and let other code execute concurrently. And these locking rules can also help improve documents of OS kernels, which is beneficial for long term evolution of kernel codes.

Second, key fields extracted by RaceMiner are data structure fields that are protected by some locks, and this means that these fields are likely to be concurrently accessed by different threads. Therefore, key fields can be used in detecting various types of concurrency bugs such as use-after-free and uninitialized-variable access. Take concurrency use-after-free as an example, if a key field is accessed in one thread and freed in another thread, a concurrency use-after-free can occur.

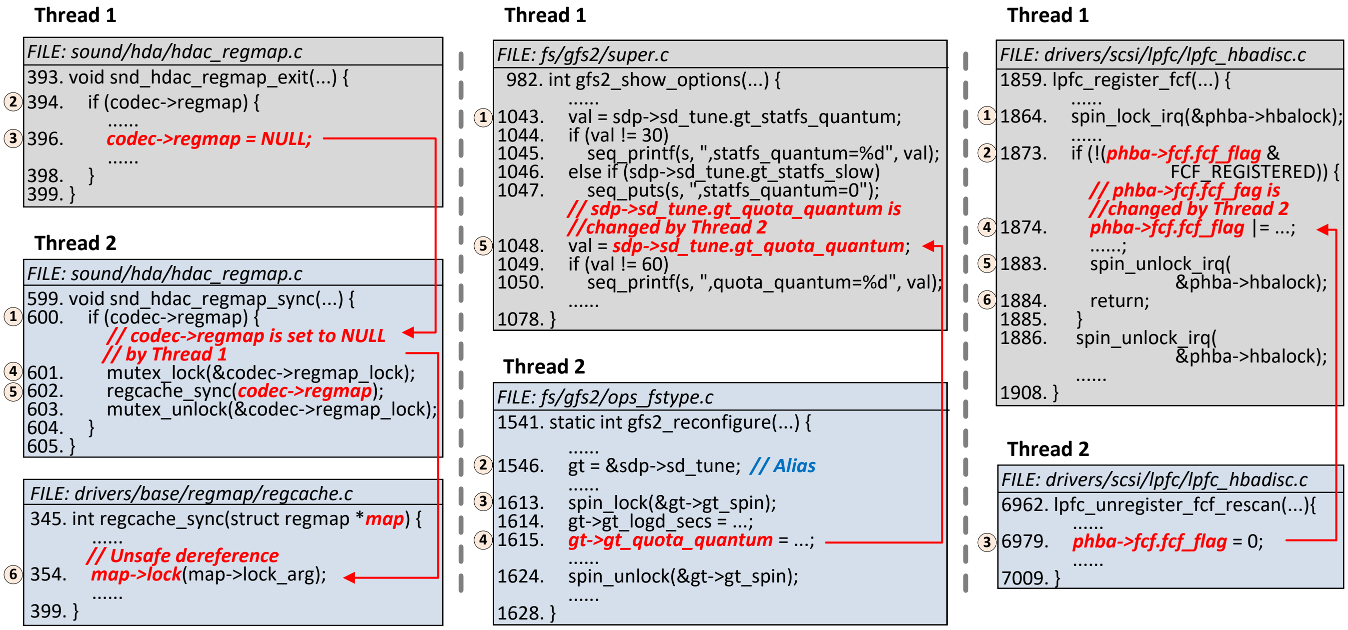
Finally, the locking rules deduced by our alias-aware rule mining method can enhance other annotation-based analyses such as Clang thread safety analysis [11]. Specifically, developers can use instrumentation to annotate necessary locks for specific variables in the code automatically, according to the locking rules generated by RaceMiner, and run other annotation-based analysis to detect data races.

Limitations and future works. RaceMiner can be improved in some aspects. First, our alias-aware mining method only considers accessed variables and protecting locks that exist in the same data structure, and can miss some locking rules involving special locks such as RCU locks. To overcome this limitation, we plan to create a common virtual node for RCU lock in the alias graph, and regard it as the ancestor of all other nodes. In this way, all variables have a common ancestor with the RCU lock, and can be inferred to be protected by the RCU lock, because the accessed variable and the RCU lock are existing in the identical data structure in our alias-aware rule mining method. Second, RaceMiner does not handle function-pointer calls, and thus it can not build complete call graphs, and thus can miss data races reached from function-pointer calls. To relieve this limitation, we plan to apply existing function-pointer analysis [19, 20, 44] in RaceMiner, to detect more data races in functions that are called through function pointers and reduce false negatives. Third, RaceMiner performs a path-based analysis and suffers from high time overhead, to accelerate its analysis, we plan to employ summer-based analysis [3, 4] to avoid re-analysis of the same function definition. Finally, we plan to apply RaceMiner to detect data races in other OS kernels such as FreeBSD [17] and NetBSD [35], and use it to detect more types of concurrency bugs including use-after-free and uninitialized-variable access.

7. Related Work

7.1. Static Analysis of Data Races

Many approaches [1, 2, 5, 6, 9, 11, 13, 14, 15, 33, 37, 38, 41, 46] use static analysis to detect data races, without running the checked program. Some of them [1, 2, 6, 11, 14, 15, 38, 46] perform annotation-based analysis, and request developers to provide locking rules about which lock is required when a



```
Thread 1
FILE: sound/hda/hdac_regmap.c
  393. void snd_hdac_regmap_exit(...) {
② 394.     if (codec->regmap) {
               ......
③ 396.         codec->regmap = NULL;
               ......
  398.     }
  399. }

Thread 2
FILE: sound/hda/hdac_regmap.c
  599. void snd_hdac_regmap_sync(...) {
① 600.     if (codec->regmap) {
               // codec->regmap is set to NULL
              // by Thread 1
④ 601.         mutex_lock(&codec->regmap_lock);
⑤ 602.         regcache_sync(codec->regmap);
  603.         mutex_unlock(&codec->regmap_lock);
  604.     }
  605. }

FILE: drivers/base/regmap/regcache.c
  345. int regcache_sync(struct regmap *map) {
            ......
          // Unsafe dereference
⑥ 354.     map->lock(map->lock_arg);
            ......
  399. }
```

(a) Null-pointer dereference in the HDA driver

```
Thread 1
FILE: fs/gfs2/super.c
   982. int gfs2_show_options(...) {
            ......
① 1043.     val = sdp->sd_tune.gt_statfs_quantum;
  1044.     if (val != 30)
  1045.         seq_printf(s, ",statfs_quantum=%d", val);
  1046.     else if (sdp->sd_tune.gt_statfs_slow)
  1047.         seq_puts(s, ",statfs_quantum=0");
            // sdp->sd_tune.gt_quota_quantum is
            //changed by Thread 2
⑤ 1048.     val = sdp->sd_tune.gt_quota_quantum;
  1049.     if (val != 60)
  1050.         seq_printf(s, ",quota_quantum=%d", val);
            ......
  1078. }

Thread 2
FILE: fs/gfs2/ops_fstype.c
  1541. static int gfs2_reconfigure(...) {

            ......
② 1546.     gt = &sdp->sd_tune;  // Alias
            ......
③ 1613.     spin_lock(&gt->gt_spin);
  1614.     gt->gt_logd_secs = ...;
④ 1615.     gt->gt_quota_quantum = ...;
            ......
  1624.     spin_unlock(&gt->gt_spin);
            ......
  1628. }
```

(b) Data inconsistency in the GFS2 file system

```
Thread 1
FILE: drivers/scsi/lpfc/lpfc_hbadisc.c
  1859. lpfc_register_fcf(...) {
            ......
① 1864.     spin_lock_irq(&phba->hbalock);
            ......
② 1873.     if (!(phba->fcf.fcf_flag &
                             FCF_REGISTERED)) {
              // phba->fcf.fcf_fag is
              //changed by Thread 2
④ 1874.       phba->fcf.fcf_flag |= ...;
              ......;
⑤ 1883.       spin_unlock_irq(
                             &phba->hbalock);
⑥ 1884.       return;
  1885.     }
  1886.     spin_unlock_irq(
                             &phba->hbalock);
            ......
  1908. }

Thread 2
FILE: drivers/scsi/lpfc/lpfc_hbadisc.c
  6962. lpfc_unregister_fcf_rescan(...){
            ......
③ 6979.     phba->fcf.fcf_flag = 0;
            ......
  7009. }
```

(c) Double fetch in the LPFC SCSI driver

Figure 14: Three real data races found by RaceMiner in Linux 6.2.

specific variable is accessed, and then detect data races that violate the provided rules. Clang thread safety analysis [11] needs developers to annotate code with a `GUARDED_BY` attribute to indicate which lock is required for a specific variable, and then works like a type system for multi-threaded programs [6, 32, 42] to detect data races that violate the provided attribute. Flanagan et al. [14] design a type system and rely on developers to provide additional type annotations to associate a protecting lock with each field declaration, and track the set of locks held at each program point. And then it identifies whether a given program is race free by checking whether necessary locks are indeed held at each program site. However, such annotation-based approaches are not suitable for race detection in OS kernels, because even an expert developer can not provide accurate annotations, because the locking rules of kernel code are not well documented, and the logic of kernel code tends to be very complex. Other approaches [5, 9, 13, 33, 37, 41] employ lockset-based analysis to detect data races automatically. They deduce locking rules based on statistics instead of relying on manual annotations. But they do not consider alias relationships [13, 41] or just employ imprecise flow-insensitive alias analysis [9, 33, 37]. RacerX [13] performs a lockset analysis to detect data races in three modes, including simple checking, simple statistical and precise statistical from least to most precise. However, even if for the most precise mode, RacerX does not employ any alias analysis, and thus introduces many false positives and negatives.

Different from the above static analysis, RaceMiner employs the alias graph to mine locking rules automatically. Benefiting from the path-based and field-sensitive alias relationships generated by alias graphs, RaceMiner can deduce locking rules effectively. With these locking rules, RaceMiner can detect data races accurately. Moreover, RaceMiner exploits a pattern-based estimation to extract harmful data races automatically, and can help developers focus on data races that can indeed introduce serious bugs.

7.2. Dynamic Analysis of Data Races

The runtime analysis does not suffer from the pointer alias problems, and thus many approaches [21, 27, 28, 29, 30, 31, 36, 40] detects data races through dynamic analysis. LockDoc [28] records accesses to variables and lock acquisitions of an instrumented Linux kernel in logs, and then infers locking rules from the logs. After that, LockDoc automatically detect data races by checking whether a given variable access violates the inferred locking rules. O'callahan et al. [36] combine lockset-based detection and happens-before-based detection to detect data races, which can get fewer false positives and less overhead than previous dynamic analyses. In order to reduce runtime overhead further, LiteRace [31] employs a sampling-based approach based on the hypothesis that data races are likely to occur when a thread is executing an infrequently accessed region in the program. It instruments sampled memory accesses, and performs happens-before-based analysis to detect data races, according to the logs generated by the instrumentation.

However, dynamic analysis suffers from low code coverage, and the locking rules inferred through execution traces such as logs can be imprecise. Besides, dynamic analyses need to run OS kernels with instrumentation and are hard to deploy.

8. Conclusion

In this paper, we develop a novel static approach named RaceMiner to detect data races in OS kernels by mining locking rules. It consists of three key techniques, including an alias-aware rule mining method to automatically deduce locking rules, a lock-usage analysis to filter out false positives caused by code that can not execute concurrently and a pattern-based estimation to extract harmful data races that can trigger memory or logic bugs such as null-pointer dereferences, data inconsistencies and double fetches. In the evaluation, RaceMiner finds 88 real harmful data races, and 32 of them have been confirmed by the developers.

References

[1] 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. https://doi.org/10.1145/1375581.1375600.

[2] Zachary R Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI)*, pages 98–109, 2009. https://doi.org/10.1145/1542476.1542488.

[3] 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 (USENIX ATC)*, pages 255–268, 2019. https://www.usenix.org/conference/atc19/presentation/bai.

[4] 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 (USENIX ATC)*, pages 367–382, 2022. https://www.usenix.org/conference/atc22/presentation/bai.

[5] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. *Proceedings of the 33th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, 2:1–28, 2018. https://doi.org/10.1145/3276514.

[6] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th International Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 211–230, 2002. https://doi.org/10.1145/582419.582440.

[7] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010. https://doi.org/10.1145/1736020.1736040.

[8] 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. https://doi.org/10.1145/349299.349311.

[9] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the 23th International Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269, 2002. https://doi.org/10.1145/512529.512560.

[10] Clang: an LLVM-based C/C++ compiler, 2021. http://clang.llvm.org/.

[11] A C++ language extension which warns about potential race conditions in code, 2021. https://clang.llvm.org/docs/ThreadSafetyAnalysis.html.

[12] CLOC: count lines of code, 2021. https://cloc.sourceforge.net.

[13] Dawson Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 9th International Symposium on Operating Systems Principles (SOSP)*, pages 237–252, 2003. https://doi.org/10.1145/945445.945468.

[14] Cormac Flanagan and Stephen N Freund. Type-based race detection for Java. In *Proceedings of the 21st International Conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000. https://doi.org/10.1145/349299.349328.

[15] Cormac Flanagan and Stephen N Freund. Detecting race conditions in large programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 90–96, 2001. https://doi.org/10.1145/379605.379687.

[16] Pedro Fonseca, Cheng Li, Vishal Singhal, and Rodrigo Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of the 2010 International Conference on Dependable Systems and Networks (DSN)*, pages 221–230. IEEE, 2010. https://doi.org/10.1109/DSN.2010.5544315.

[17] FreeBSD: The FreeBSD project, 2023. https://www.freebsd.org/.

[18] Bernard A Galler and Michael J Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964. https://doi.org/10.1145/364099.364331.

[19] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the 22nd International Conference on Programming Language Design and Implementation (PLDI)*, pages 24–34, 2001. https://doi.org/10.1145/378795.378802.

[20] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(4):848–894, 1999. https://doi.org/10.1145/325478.325519.

[21] Pallavi Joshi and Koushik Sen. Predictive typestate checking of multithreaded Java programs. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE)*, pages 288–296. IEEE, 2008. https://doi.org/10.1109/ASE.2008.39.

[22] Baris 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. https://doi.org/10.1145/2150976.2150997.

[23] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the 24th International Symposium on Operating Systems Principles (SOSP)*, pages 406–422, 2013. https://doi.org/10.1145/2517349.2522736.

[24] 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. https://doi.org/10.1145/3178372.3179519.

[25] 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. https://doi.org/10.1145/3503222.3507770.

[26] Peng Liu, Omer Tripp, and Charles Zhang. Grail: Context-aware fixing of concurrency bugs. In *Proceedings of the 22nd International Symposium on Foundations of Software Engineering (FSE)*, pages 318–329, 2014. https://doi.org/10.1145/2635868.2635881.

[27] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS checker: combating bugs in distributed systems. In *Proceedings of the 4th USENIX conference on Networked Systems Design and Implementation (NSDI)*, pages 19–19, 2007. https://dl.acm.org/doi/abs/10.5555/1973430.1973449.

[28] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. LockDoc: Trace-based analysis of locking in the Linux kernel. In *Proceedings of the 14th EuroSys Conference*, pages 1–15, 2019. https://doi.org/10.1145/3302424.3303948.

[29] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 3–14, 2018. https://doi.org/10.1145/3236024.3236071.

[30] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A Popa, and Yuanyuan Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *Proceedings of 21st International Symposium on Operating Systems Principles (SOSP)*, pages 103–116, 2007. https://doi.org/10.1145/1294261.1294272.

[31] Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. LiteRace: Effective sampling for lightweight data-race detection. In *Proceedings of the 30th International Conference on Programming Language Design and Implementation (PLDI)*, pages 134–143, 2009. https://doi.org/10.1145/1542476.1542491.

[32] Jean-Yves Marion and Romain Péchoux. Complexity information flow in a multi-threaded imperative language. In *Proceedings of the 11th International Conference on Theory and Applications of Models of Computation (TAMC)*, pages 124–140. Springer, 2014. https://doi.org/10.1007/978-3-319-06089-7_9.

[33] 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. https://doi.org/10.1145/1133981.1134018.

[34] 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. https://doi.org/10.1145/1250734.1250738.

[35] NetBSD: The NetBSD project, 2023. https://www.netbsd.org/.

[36] Robert O'callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 167–178, 2003. https://doi.org/10.1145/781498.781528.

[37] Polyvios Pratikakis, Jeffrey S Foster, and Michael Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Proceedings of the 27th International Conference on Programming Language Design and Implementation*, pages 320–331, 2006. https://doi.org/10.1145/1133981.1134019.

[38] Caitlin Sadowski and Jaeheon Yi. How developers use data race detection tools. In *Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, pages 43–51, 2014. https://doi.org/10.1145/2688204.2688205.

[39] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th International Conference on Programming Language Design and Implementation (PLDI)*, pages 11–21, 2008. https://doi.org/10.1145/1375581.1375584.

[40] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *Proceedings of the 2009 Workshop on Binary Instrumentation and Applications (WBIA)*, pages 62–71, 2009. https://doi.org/10.1145/1791194.1791203.

[41] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 205–214, 2007. https://doi.org/10.1145/1287624.1287654.

[42] Xuan-Tung Vu, Mai Thuong Tran, Anh-Hoang Truong, and Martin Steffen. A type system for finding upper resource bounds of multi-threaded programs with nested transactions. In *Proceedings of the 3rd Symposium on Information and Communication Technology (SoICT)*, pages 21–30, 2012. https://doi.org/10.1145/2350716.2350722.

[43] Z3: a theorem prover, 2021. https://github.com/Z3Prover/z3.

[44] Wei Zhang and Yu Zhang. Lightweight function pointer analysis. In *Proceedings of the 11th International Conference on Information Security Practice and Experience (ISPEC)*, pages 439–453. Springer, 2015. https://doi.org/10.1007/978-3-319-17533-1_30.

[45] Bo Zhou, Iulian Neamtiu, and Rajiv Gupta. Predicting concurrency bugs: how many, what kind and where are they? In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering EASE*, pages 1–10, 2015. https://doi.org/10.1145/2745802.2745807.

[46] Diyu Zhou and Yuval Tamir. PUSh: Data race detection based on hardware-supported prevention of unintended sharing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 886–898, 2019. https://doi.org/10.1145/3352460.3358317.