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 concurrent issues. And data race is one of the most harmful concurrent issues that can trigger memory or logical bugs. A data race is 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 used for a specific variable (locking rules) even for an expert developer, due to poor documentation 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 help improve code quality, but it is quite challenging to detect data races in OS kernels automatically, because of 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 harmful races that can trigger memory or logical bugs 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 concurrency. 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 xxx data races, with a false positive rate of xxx. Among these data races, xxx are estimated to be harmful. We have reported these harmful bugs to Linux kernel developers, and xxx 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 the same variable concurrently and at least one of the accesses is a write. Some data races are quite dangerous and can cause serious bugs such as null-pointer dereferences, data inconsistency and infinite loops.

To detect data races effectively, some approaches [1, 2, 3, 4, 9, 12, 13, 29, 34] rely on developers to supply annotations that describe the locking discipline such as which lock protects a specific variable, and detect data races by exploiting variable accesses that violate these disciplines. For example, Clang thread safety analysis [9] 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 documentation and complicated logic of OS codes.

Some static approaches [7, 11, 26, 28, 31] employ lockset-based analysis to detect data races automatically. They first collect locks that are acquired when a given variable is accessed, and then mine locking rules about whether a given variable should be protected by a specific lock. At last, they detect data races by checking whether variable accesses violate the mined locking rules. However, they do not consider alias relations [11, 31] or just use imprecise flow-insensitive alias analysis [7, 26, 28], and thus can introduce both false positives and false negatives.

Dynamic analysis can acquire run-time information such as memory address, and thus complex alias analysis can be avoided. Therefore, some approaches [16, 22, 23, 24, 25] 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 [23] 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 variables access that violate the inferred locking rules to detect concurrency bugs 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 [5, 14, 21, 33].

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 effectively. RaceMiner consists of three key techniques:

(T1) RaceMiner uses an *alias-aware rule mining method* to automatically deduce locking rules. We observe that an access to a variable is often protected by the lock in the same data structure as the accessed field. And this relationship between accessed field and the protecting lock can be effectively inferred from a special data structure named alias graph [19, 20]. Specifically, fields in the same data structure have the same an-

cestor in the alias graph, and thus whether an accessed field and a specific lock are in the same data structure can be inferred by 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 protecting lock. After obtaining the accessed field and protecting lock pairs, RaceMiner calculates the proportion of field accesses protected by a specific lock in all field accesses. If the proportion is larger than a given threshold, the accessed field is determined to be protected by the protecting lock.

(T2) RaceMiner uses a *lock-usage analysis* to filter out false positives validating concurrency of code paths. 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 be executed 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 initialized the lock and prepare other data for subsequent operations. And thus functions with the lock initialization and functions called by them tend not to be executed concurrently. Based on this observation, RaceMiner extracts all functions reachable from the function with a lock initialization operation such as `spin_lock_init()`, and suppose that this function can not be executed in parallel.

(T3) RaceMiner uses a *pattern-based estimation* to extract harmful races that can trigger memory or logical bugs such as null-pointer dereferences and data inconsistency. Some data races are benign, and developers do not put effort into repairing them because they can not cause serious memory or logical bugs. Therefore, it is important to estimate the harm a data race can bring. We observe that harmful data races can be estimated by given 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 is accessed for more than once and data inconsistency can occur. Based on this observation, RaceMiner uses some patterns to extract harmful races that can cause memory or logical bugs automatically.

We have implemented RaceMiner with LLVM [8] and Z3 [32]. 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 of data race detection in OS 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 concurrency; (T3) a *pattern-based estimation* to extract harmful data races that can trigger memory or logical bugs such as null-pointer dereferences and data inconsistency.
- 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 xxx data races, with a false positive rate of xxx. Among these data races, xxx are estimated to be harmful. We have reported these harmful bugs to Linux kernel developers, and xxx 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

```
FILE: drivers/gpu/drm/exynos/exynos_drm_crtc.c (Thread 1)
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 (Thread 1)
1082. void drm_crtc_send_vblank_event(..., TYPE *e) {
        ......
⑦ 1097.   e->pipe = pipe;   // Null-pointer dereference
        ......
1099. }

FILE: drivers/gpu/drm/exynos/exynos_drm_crtc.c (Thread 2)
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 the DRM driver, the functions `exynos_drm_crtc_atomic_disable()` and `exynos_crtc_handle_event()` can execute concurrently. We exploit ⓝ to represent the execution order of instructions and show one execution case in the left of Figure 1. In Thread 1, the variables `crtc->state->event` and `crtc->state->active` is 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()` is executed in Thread 2. In this function, the value of `crtc->state->active` 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->active`, after acquiring the lock `crtc->dev->event_lock` (⑤). In the called function, the variable `crtc->state->event` is dereferenced through `e->pipe` (⑥). In this execution case, the data structure `crtc->state->event` is first assigned with NULL 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 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 are 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 (locking rules), even for an expert developer. And thus existing annotation-based approaches [1, 2, 3, 4, 9, 12, 13, 29, 34] are difficult to apply to race detection in OS kernels. Other approaches [7, 11, 26, 28, 31] employ lockset-based analysis to detect data races automatically, but they do not consider alias relations [11, 31] or just use imprecise flow-insensitive alias analysis [7, 26, 28]. However, due to the heavy use of pointers and data structure fields in OS code, the alias relationships between variables can be very complex, and thus lacking effective alias analysis can introduce many false locking rules.

C2: Dropping false data races. Static analysis suffers from false positives. For example, each data race involves more than one code paths 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, and can not cause memory or logic bugs, and thus developers are unwilling to put effort into repairing them. To automatically detect harmful data races, most approaches [17, 18, 27, 30] estimate harmfulness of data races through dynamic analysis, and explore thread interleavings to trigger data races to estimate their harmfulness. 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 data races by validating concurrency of code paths. For *C3*, we propose a pattern-based estimation to extract harmful races that can trigger memory or logical bugs such as null-pointer dereference and data inconsistency. We introduce them as follows:

3.1. Alias-Aware Rule Mining Method

The relationship between variables and locks are not well documented in OS kernels, but it can be inferred from the kernel code. Specifically, a variable is often protected by the lock stored 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 though an alias graph [19, 20] 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 find data structure filed and its protecting lock effectively.

Alias Graph. It is an important data structure to infer relationships 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 form an access path [6, 19]. 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 move 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 $*$.

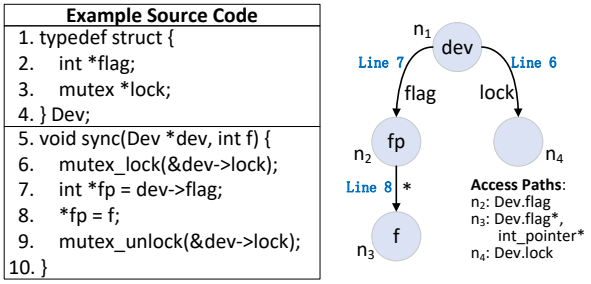

Figure 2: Example of alias graph.

Example. Figure 2 shows a piece of driver-like source code and its alias graph. In this example, after an 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 lock is likely to protect the variable when it is accessed. 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 path for the node of the accessed variable and the node of the lock variable from the common ancestor (Lines 5-7).

Take the alias graph in Figure 2 as an example, `&dev->flag` is represented by node n_2, and `&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 protecting 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 the lock stored in the structure field Dev.lock.

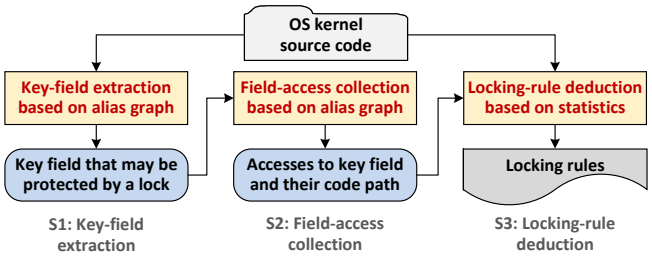

Figure 4: Workflow of locking-rule mining.

Based on the alias graph, our alias-aware rule mining method performs an inter-procedural path-based [20], field-sensitive and alias-aware analysis to effectively mine locking rules about whether a specific variable should be protected by a lock and which lock is required. Overall, our alias-aware rule mining method has three main stages shown in Figure 4. In Stage 1, it extract key variable that may be protected by a lock. In Stage 2, it collect all accesses to key fields as well as code paths these accesses exist in. In Stage 3, it deduce locking rules by calculate 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, only a few field accesses can miss necessary protecting lock, due to 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 as the accessed field in any code path.

```
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.

Figure 5 shows the lock-set analysis based on alias graph. The analysis first gets the operand of the instruction (Line

1), and then get 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 (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: 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 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 [20] (Lines 17-25). For each instruction in the code path, the analysis first update the alias graph according to the instruction with the four operations (MOVE, STORE, LOAD and GEP) (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 lock variable 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). For inter-procedural analysis, the analysis copies function definition into its call sites and this operation is not shown in Figure 6 for convenience.

S2: Field-access collection. With the key fields extracted in Stage 1, the analysis can only collects accesses to the variables that is protected in some cases, because if a variable is never protected by any lock when is accessed, it is less likely to be shared by different threads, and thus can not introduce no 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:  // Finding whether the accessed variable is protected by a 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:  // Finding whether the accessed var exist in a 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 key-field extraction.

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) is set according to the instruction (Lines 5-8). However, if the instruction is not an access operation, the function returns with 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 lock variable in the lock set, the analysis uses the alias graph toextract the fields that the accessed variable and the lock variable stored in (Lines 16-17). If the fields are found, the field access is returned with the found fields (Lines 18-22). If the fields are not found, the accessed variable does 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 variable (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 variable 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. On the other hand, distinguishing different accesses by different code paths can also suffer from inaccuracy when a function contains many branch statement and cause numerous code paths for the same variable access. Based on these consideration, the analysis distinguishes different 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 which lock is accessed), 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 these 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 detecting 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 perform a path-based analysis to extract key fields. Take the code path `Line 12, Line 6, Line 7, Line 8, Line 9` as an example. The analysis updates the alias graph by handling two GEP operations (*&dev->lock* and *fp = dev->flag*) and a STORE operation (**fp = f*). The final alias graph is shown in Figure 8(b). At the same time of 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.

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 15, Line 6, Line 7, Line 8,`

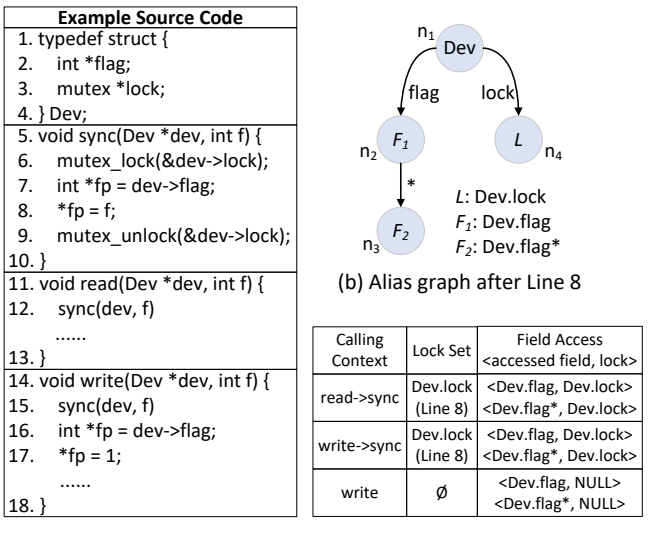

Figure 8: Example of locking-rule mining.

`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, 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*. And thus the two fields *Dev.flag* and *Dev.flag** is deduce 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 accesses at Lines 16 and 17 are not protected by *Dev.lock* and thus causing two data races.

3.2. Lock-Usage Analysis

Our analysis takes functions that has no caller function as the entry as existing work [20], and performs a path-based analysis. 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 function can be called concurrently by other modules. When performing initialization, the kernel module serially initialized the lock and prepare other data for subsequent operations. And thus functions with

the lock initialization and functions called by them tend not to be executed concurrently. Based on the 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 [15] to get all functions that reachable from the lock initialization function such as *mutex_init()* and *spin_lock_init()*. 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 function 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 judging whether a function executes in the initialization phase by checking whether it exist 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 function, 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 kernel performance. They can not cause memory or logic bugs, 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 four patterns that can cause null-pointer dereference, infinite loop, data inconsistency and unprotected write, because these four patterns are common and dangerous in OSes. The four 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
while (dev->cnt) {
  dev->cnt--;
}
```

(b) P2: infinite loop

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

(c) P3: data inconsistency

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

(d) P4: unprotected write

Figure 11: Three harmful patterns of data races.

- **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 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.
- **Infinite loop.** The condition of the while statement does not protected by *dev->lock*. This can cause a infinite loop when *dev->cnt* is assigned with a non-zero value every time before it is checked by the while statement. The analysis detects such bugs by judge whether the data race acts as a condition of a loop statement.
- **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 pattern, the analysis detect whether multiple fields of the same data structure are accessed without acquiring the protecting lock.
- **Unprotected write.** The write access to *dev->event* is not protected by *dev->lock*. This is dangerous because the value *dev->event* can be modified at any time when other threads access it, and thus introduce unpredictable behavior. The analysis detect such pattern by judge 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 harmfulness of these data races. We have implemented RaceMiner with Clang [8] and Z3 [32]. RaceMiner automatically mines locking rules, detects data races and estimates the harmfulness of detected data races. Figure 12 shows the architecture of RaceMiner, which has four phases:

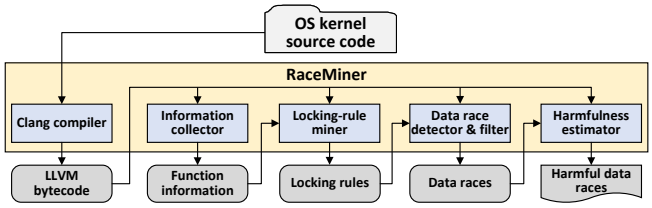

Figure 12: RaceMiner architecture.

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 deduce locking rules about whether a variable access should be protected by a lock and which lock is required.

P3: Data race detection and false-positive filtering. The data race detector detecting 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	
	Protected field accesses / all accesses	0.6
Data race detection	Detected data races (real / all)	273 / 341
	Dropped data races by lock-usage analysis	63
Data race estimation	Null-pointer dereference (confirmed / all)	15 / 20
	Infinite loop (confirmed / all)	0 / 1
	Data inconsistency (confirmed / all)	7 / 10
	Unprotected write (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 mine locking rules, and lock-initialization functions (like `spin_lock_init`) to filter our false data races caused by code paths 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 [10]. From the results, we have the following findings:

Code analysis. RaceMiner can scale to large code bases of OS kernels, and it in total analyze 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 a variable tend to be protected by the lock stored in the same data structure as the accessed variable. Based on this observation, our locking-rule mining method first extract a key field 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 handled when mining locking rules, and thus can reduce overhead significantly. After extracting key fields, our locking-rule mining method collect all accessed to these key fields, and then deducing locking-rule based on statistic. 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 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 shows that it 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 four patterns to detect null-pointer dereferences, infinite loop, data inconsistency and unprotected write 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 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

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 does 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 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.

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 deduces 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 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,

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.

an integer can be divided into several bit vector to represent different data structure fields. However,

6. Discussion

7. Related Work

8. Conclusion

References

[1] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: checking data sharing strategies for multithreaded C. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 149–158, 2008. https://doi.org/10.1145/1379022.1375600.

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

[3] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages (OOPSLA)*, 2:1–28, 2018. https://doi.org/10.1145/3276514.

[4] 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. https://doi.org/10.1145/582419.582440.

[5] 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.

[6] Ben-Chung Cheng and Wen-Mei W Hwu. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation (PLDI)*, pages 57–69, 2000. https://doi.org/10.1145/349299.349311.

[7] 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 ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), pages 258–269, 2002. https://doi.org/10.1145/512529.512560.

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

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

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

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

[12] Cormac Flanagan and Stephen N Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming Language Design and Implementation (PLDI)*, pages 219–232, 2000. https://doi.org/10.1145/349299.349328.

[13] 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.

[14] 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 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 221–230. IEEE, 2010. https://doi.org/10.1109/DSN.2010.5544315.

[15] 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.

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

[17] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: telling the difference with portend. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS12)*, pages 185–198, 2012. https://doi.org/10.1145/2150976.2150997.

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

[19] 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.

[20] 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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 859–872, 2022. https://doi.org/10.1145/3503222.3507770.

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

[22] 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.

[23] 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 2019*, pages 1–15, 2019. https://doi.org/10.1145/3302424.3303948.

[24] Jie Lu, Feng Li, Lian Li, and Xiaobing Feng. Cloudraid: hunting concurrency bugs in the cloud via log-mining. In *Proceedings of the 2018 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.

[25] 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 ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 103–116, 2007. https://doi.org/10.1145/1294261.1294272.

[26] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 308–319, 2006. https://doi.org/10.1145/1133981.1134018.

[27] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 22–31, 2007. https://doi.org/10.1145/1250734.1250738.

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

[29] 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.

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

[31] 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.

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

[33] 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.

[34] 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.