Static Detection of TOCTOU Bugs Caused by Kernel Races

Gui-Dong Han
Beihang University

Jia-Ju Bai*
Beihang University

Qiu-Ji Chen
Beihang University

Jiqiang Lu
Beihang University

Abstract

The TOCTOU (Time Of Check to Time Of Use) bug is a well-known security issue in kernel code, because it bypasses security checks and leads to unexpected behaviors that can cause serious problems like system crashes and privilege escalation. According to our study on Linux kernel patches, kernel race is the most common root cause of kernel TOCTOU bugs. However, due to the complexity of kernel concurrency logic and non-determinism of thread scheduling, there is still no systematic approach that focuses on detecting TOCTOU bugs caused by kernel races.

In this paper, we design KERAT, the first systematic static approach for detecting TOCTOU bugs caused by kernel races. Indeed, such TOCTOU bugs are introduced by atomicity violations about the check-use operations of specific shared variables. Thus, KERAT performs bug detection by statically mining and checking the atomicity rules about shared variables from kernel code. Specifically, KERAT has two key techniques: (1) an *atomicity-rule mining method* to effectively identify which lock should protect the check-use operations of which shared variable; and (2) a *state-based validation strategy* to detect TOCTOU bugs that violate the mined atomicity rules based on state machines encoding of common bug patterns. We have evaluated KERAT on Linux-6.8 and FreeBSD-14.1, and found 351 real bugs. Among these bugs, 287 are identified as harmful, and 65 of them have been confirmed by kernel developers. 10 bugs have received CVE IDs.

1 Introduction

The operating system (OS) kernel that works as a fundamental part of a computer system manages various resources and devices. To perform safe and reliable management, an OS kernel must perform a large number of security checks on specific variables before their use. However, these security checks can be bypassed by race conditions involving the checked variables, leading to TOCTOU (Time Of Check to Time Of Use) bugs classified (CWE-367 [17]). Specifically, a variable can be modified between its check and use operations, causing the checked value and used value to be unexpectedly different. In this case, an attacker can deliberately construct a safe value to bypass a security check and then replace it with a malicious value before the variable is used to trigger dangerous vulnerabilities like buffer overflow or division by zero. Some known kernel vulnerabilities [14–16] are caused by TOCTOU bugs, and they cause serious security problems like system crashes and privilege escalation.

According to the possible sources of race conditions, kernel TOCTOU bugs can be classified into three kinds:

(1) User-space race TOCTOU bug (URT bug for short). This kind of bug is caused by a race condition in the data exchange between user space and kernel space. For example, a variable may be copied from user space and checked for security purposes, and then be copied from user space again and used without being rechecked; before the second copy, the attacker can modify the variable in user space. Some existing approaches can find such URT bugs via double-fetch [46, 47, 55] and lacking-recheck [50] detection.

(2) Hardware-I/O race TOCTOU bug (HRT bug for short). This kind of bug is caused by a race condition in the data exchange between the hardware I/O and kernel space. For example, an element of the DMA buffer mapped from the hardware may be checked for security purposes, and then it may be used by directly accessing the DMA buffer; before the element use, the attacker can modify this element by controlling the untrusted hardware. Some existing approaches are capable of detecting such HRT bugs via mapped-I/O [35] and DMA-access [6] validation techniques.

(3) Kernel-space race TOCTOU bug (KRT bug for short). This kind of bug is caused by a race condition in the concurrent execution of multiple kernel threads. For example, a shared variable may be checked in a kernel thread for security purposes, and then be used in the same thread; before the variable use, the attacker can modify this variable in another concurrent kernel thread. Existing approaches for kernel race detection [2, 19, 24, 25, 31, 45, 54] are able to find some data

*Corresponding author.

races that may lead to KRT bugs due to lacking necessary locks, but they still miss many other KRT bugs like those caused by atomicity violations. To our knowledge, *there is no systematic approach that focuses on detecting TOCTOU bugs caused by kernel races at present*, so it is important to design a new approach to fill this gap.

Concretely, KRT bugs are introduced by atomicity violations about the check-use operations of specific shared variables. The check and use operations of a shared variable should be atomically performed with the protection of a common lock, and otherwise a KRT bug can occur. Accordingly, to detect KRT bugs, there are two important challenges:

(C1) How to mine atomicity rules, i.e. which lock should protect the check-use operations of which shared variable? The most relevant approach is LR-Miner [31] that mines locking rules describing which lock should protect which shared variable. Although locking rules and atomicity rules are similar, they have very different targets of lock protection. The target of a locking rule is each access of a shared variable, while that of an atomicity rule is each check-use pair of a shared variable. In other words, obeying locking rules is not sufficient to ensure obeying atomicity rules. Thus, a new method is required to mine atomicity rules from the kernel code.

(C2) How to use the mined atomicity rules to effectively detect KRT bugs? Different from data races that have just one occurrence pattern namely lacking lock protection of a single variable access, KRT bugs have multiple occurrence patterns involving lock-acquire/release and variable-check/use operations in the corresponding atomicity rule. Thus, KRT bug detection is more complex than race detection. Moreover, the OS kernel has a very large code base, and it heavily uses pointers and data structures, so achieving both good accuracy and efficiency of atomicity rule validation is challenging.

To solve the two challenges and perform KRT bug detection, we propose two key techniques:

(T1) Atomicity-rule mining method. To solve *C1*, our method automatically mines accurate atomicity rules from the kernel code by statically analyzing the structure field relation between locks and shared variables. To handle the complex nested data structures prevalent in the kernel, our method leverages field graphs [31] to resolve alias relationships between structure fields. The mining process consists of two main steps. First, our method identifies possible security-checked variables by analyzing conditional statements. Second, for each checked variable, our method checks whether its modification is always protected by a specific lock. If so, our method infers that the check-use operations of the variable's structure field require the protection of the same lock, and thus extracts an atomicity rule. Such atomicity rules are used for subsequent KRT bug detection, and we believe they can also help to improve kernel documentation.

(T2) State-based validation strategy. To solve *C2*, we analyze all the possible patterns of lock and check-use operations, and identify four dangerous patterns that can cause

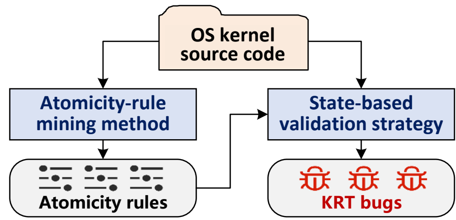


Figure 1: KERAT workflow.

KRT bugs. According to these four dangerous patterns, our strategy builds two finite state machines (FSMs) and uses them to detect KRT bugs. To improve accuracy, our strategy considers alias relationships about locks and checked/used variables by building and analyzing field graphs. To improve efficiency, our strategy uses function summaries to reduce repeated analysis for inter-procedural validation.

Based on these two techniques, we design KERAT, the first systematic static approach for detecting TOCTOU bugs caused by kernel races. As shown in Figure 1, KERAT has two stages. First, KERAT uses our atomicity-rule mining method to automatically mine atomicity rules about data fields and locks. Second, KERAT uses our state-based validation strategy to detect KRT bugs that violate the mined atomicity rules based on state machines encoding of common bug patterns. We have implemented KERAT with LLVM [34] for automated code analysis. Overall, we make four contributions:

- We study 203 Linux kernel patches fixing TOCTOU bugs within the last five years, and find that 68% of them are related to kernel concurrency, indicating that kernel races serve as the most common root cause of kernel TOCTOU bugs. Thus, KRT bugs should receive more attention.
- We reveal the challenges of KRT bug detection, and propose two solution techniques: (1) an *atomicity-rule mining method* to effectively identify which lock should protect the check-use operations of which shared variable; and (2) a *state-based validation strategy* to detect TOCTOU bugs that violate the mined atomicity rules based on state machines encoding of common bug patterns.
- Based on these key techniques, we design KERAT, the *first* systematic static approach for detecting TOCTOU bugs caused by kernel races, via mining and checking atomicity rules from the kernel code.
- We have evaluated KERAT on Linux-5.16, Linux-6.8 and FreeBSD-14.1, and found 320, 315 and 36 real TOCTOU bugs, respectively, with a false positive rate of 18.7%. We manually check these bugs and find that 76 in Linux-5.16 have been fixed in Linux-6.8. Among the real bugs found in Linux-6.8 and FreeBSD-14.1, 287 are identified as harmful, and 65 of them have been confirmed by kernel developers. 10 bugs have received CVE IDs. We experimentally compare KERAT to three related static approaches (LRSan [50], CPALockator [2] and LR-Miner [31]), and KERAT finds many real TOCTOU bugs missed by them.

```
FILE: Linux-6.2/drivers/md/dm-ioctl.c
1807. int check_version(..., struct dm_ioctl __user *user) {
        ......
        // version is fetched from the user-space
1812.   if (copy_from_user(version, user->version, ...)
        ......
1815.   if (version[0] != ...) { // Invalid check!
          ......
1822.     r = -EINVAL;
1823.   }
        ......
1834.   return r;
1835. }

1945. int ctl_ioctl(..., struct dm_ioctl __user *user) {
        ......
1969.   r = check_version(cmd, user);
1970.   if (r)
1971.     return r;
        ......
        // TOCTOU: user->version can be changed
1988.   r = copy_params(user, ...);
        ......
2017. }
```

(a) User-space race TOCTOU (URT) bug

```
FILE: Linux-5.10/drivers/nvme/host/pci.c
967. void nvme_handle_cqe(...) {
       ......
       // cqe points to DMA memory
969.   cqe = &nvmeq->cqes[idx];
       ......
       // Invalid check!
978.   if (nvme_is_aen_req(..., cqe->command_id)) {
         ......
981.     return;
982.   }
983.
       // TOCTOU: cqe->command_id can be changed
984.   req = blk_mq_tag_to_rq(..., cqe->command_id);
       ......
995. }

1474. int nvme_alloc_queue(...) {
        ......
        // nvmeq->cqes is mapped as coherent DMA
1483.   nvmeq->cqes = dma_alloc_coherent(...)
        ......
1507. }
```

(b) Hardware-I/O race TOCTOU (HRT) bug

```
FILE: Linux-6.6/drivers/ufs/core/ufshcd.c
3035. int ufshcd_wait_for_dev_cmd(...) {
        ......
3078.   spin_lock_irqsave(&hba->outstanding_lock, ...);
        ......
        // It can be concurrently modified
3082.   hba->dev_cmd.complete = NULL;
        ......
3086.   spin_unlock_irqsave(&hba->outstanding_lock, ...);
        ......
3119. }

5382. void ufshcd_compl_one_cqe(...) {
        ......
        // Invalid check!
5402.   if (hba->dev_cmd.complete) {
          ......
          // TOCTOU: hba->dev_cmd.complete can be NULL
5407.     complete(hba->dev_cmd.complete);
5408.     ufshcd_clk_scaling_update_busy(hba);
5409.   }
        ......
5411. }
```

(c) Kernel-space race TOCTOU (KRT) bug

Figure 2: Three types of example TOCTOU bugs in the Linux kernel.

2 Background and Motivation

In this section, we first introduce kernel TOCTOU bugs, then show our study of Linux kernel patches fixing such bugs, and finally analyze existing approaches for kernel TOCTOU detection to reveal their limitations in detecting KRT bugs.

2.1 TOCTOU Bugs in OS Kernels

Time-of-Check-to-Time-of-Use (TOCTOU) bug, classified as CWE-367 [17], represents a critical type of race condition bug. Specifically, a variable can be modified between its check and use operations, causing the checked value and used value to be unexpectedly different. In this case, the attacker can deliberately construct a safe value to bypass the security check and then replace it with a malicious value in the variable used to trigger dangerous vulnerabilities like buffer overflow or division by zero. Some known kernel vulnerabilities [14–16] are caused by TOCTOU bugs, and they cause serious security problems like system crashes and privilege escalation.

In an OS kernel, TOCTOU bugs can be caused by various race conditions during kernel execution. According to the possible sources of these race conditions, kernel TOCTOU bugs can be classified into three kinds:

(1) User-space race TOCTOU bug (URT bug). This kind of bug is caused by a race condition in the data exchange between the user space and kernel space. It is often introduced when the kernel checks an attribute of a resource whose state can be modified by a malicious user-space program after the check but before the kernel completes its intended use. Figure 2(a) illustrates a URT bug [20] in Linux 6.2. In the function `ctl_ioctl`, the kernel checks the data field `version` at Line 1815 after copying it from user space. After that, the kernel directly copies the same user-space data at Line 1988 without re-checking. A malicious user-space program can modify the data field `version` between its check and use, thus bypassing the check. An attacker could exploit this bug to make the kernel process an ioctl command with an incorrect interface version, potentially leading to memory corruption.

(2) Hardware-I/O race TOCTOU bug (HRT bug). This kind of bug is caused by a race condition in the data exchange between the hardware I/O and kernel space. The condition often stems from the kernel interaction with hardware-controlled resources like memory-mapped I/O or DMA buffers, and thus the untrusted hardware can change the interaction data after the check but before the kernel completes its intended use. Figure 2(b) illustrates a HRT bug [21] in Linux 5.10. In the function `nvme_alloc_queue`, a coherent DMA buffer is allocated and stored in the variable `nvmeq->cqes` at Line 1483. After that, in the function `nvme_handle_cqe`, the kernel checks `command_id` from a `cqe` structure located in the DMA buffer at line 978, and then it uses the identical data field `command_id` at Line 984. Because the hardware can synchronously modify the DMA memory, the value of `command_id` can be changed between the check and its use. An attacker can exploit this bug to pass an overlarge array index to the kernel, potentially leading to buffer overflow.

(3) Kernel-space race TOCTOU bug (KRT bug). This kind of bug is caused by a race condition in the concurrent execution of multiple kernel threads. It is often introduced when two kernel threads concurrently access the same shared variable. Specifically, one thread checks the validity of this variable and then uses it, while another thread modifies this variable between its check and use. Figure 2(c) illustrates a KRT bug [22] in Linux 6.6. In the function `ufshcd_compl_one_cqe`, the kernel checks whether the pointer `hba->dev_cmd.complete` is null at Line 5402, and then uses it at Line 5407 if it is non-null. However, another kernel thread concurrently executes the function `ufshcd_wait_for_dev_cmd` that sets this pointer to null at Line 3082. As a result, a malicious attacker can exploit this bug to trigger a null-pointer dereference, eventually leading to system crash.

Bug Type	Patch Category by Keyword			Total	Ratio
	TOCTOU	*Double Fetch*	*Invalid Check*		
URT	15	7	20	42	20.7%
HRT	6	3	14	23	11.3%
KRT	**25**	**0**	**113**	**138**	**68.0%**
Total	46	10	147	203	100.0%

Table 1: Study result of Linux kernel TOCTOU patches.

2.2 Study of Linux Kernel TOCTOU Patches

To better understand kernel TOCTOU bugs, we conduct an empirical study of Linux kernel patches from August 2020 to July 2025. There are over 300k patches in total, and thus an exhaustive manual review is infeasible. To select those involving TOCTOU bugs, we utilize a keyword-based search to identify relevant patches, focusing on three categories of keywords. The first category directly targets TOCTOU-related discussions (e.g., *"TOCTOU"*, *"time-of-check"*); the second category focuses on double fetch issues (e.g., *"double-fetch"*, *"second fetch"*), as many TOCTOU bugs are reported with such a description; and the third category includes keywords about invalid checks (e.g., *"recheck"*, *"invalid check"*), as many TOCTOU bugs make security checks invalid.

Patch identification. After the keyword-based search, we get 860 patches for manual inspection. As for the "TOCTOU" keyword category, there are 55 patches, and we find that 46 indeed fix TOCTOU bugs while 9 are unrelated to TOCTOU (e.g., discussion of preventive mechanisms). As for the double fetch category, we find that all the 10 patches indeed fix TOCTOU bugs. As for the "invalid check" keyword category, there are 795 patches, and we find that 147 indeed fix TOCTOU bugs while 648 are unrelated to TOCTOU. In total, we identify 203 patches fixing TOCTOU bugs for detailed analysis. For each identified patch, we analyze the main source of the fixed bug to classify it as a URT, HRT or KRT bug.

Study result. Table 1 shows our study result. From the table, we find that 68.0% of kernel patches fixing TOCTOU bugs target KRT bugs, indicating that *kernel race is the most common root cause of kernel TOCTOU bugs*. In addition to this, we analyze the lifetime of KRT bugs by studying the time of bug introduction and patch submission, and find that each KRT bug persists in the kernel over 5 years on average, indicating the difficulty of KRT bug detection and fixing. Based on these observations, KRT bugs should receive more attention from kernel security researchers.

2.3 Static Analysis of TOCTOU Detection and Concurrency Checking in OS Kernels

Static analysis is a classical technique of bug detection, and it can conveniently achieve high coverage without actual program execution. Existing static analysis approaches [7, 11, 33, 44, 49, 58] have found various kinds of dangerous kernel bugs (like memory leak, division by zero, null-pointer dereference and uninitialized-variable access), indicating the powerful capability of static analysis for kernel security.

Even though there is no static analysis approach explicitly designed for kernel TOCTOU detection, some existing approaches are proposed to statically detect double fetch issues [46, 47, 55], lacking-recheck bugs [50], unsafe hardware accesses [6, 35] or data races [2, 19, 31, 45], which can cause TOCTOU bugs in particular situations. In Table 2, we select six representative approaches to analyze their TOCTOU detection capabilities and the reasons for missing KRT bugs.

The approaches of double-fetch and lacking-recheck detection focus on checking the propagation of the data copied from the user space, by using pattern-based methods, error code analysis or symbolic execution. Some issues found by these approaches can indeed cause URT bugs, due to incorrect check-use operations of the user-space data. However, they are ineffective in detecting KRT bugs, as they do not consider kernel concurrency and shared variables.

The approaches of unsafe hardware access detection focus on identifying mapped-I/O or DMA accesses for security validation with pattern-based methods. Some issues found by these approaches can indeed cause HRT bugs, due to incorrect check-use operations of the hardware data. However, these approaches are also ineffective in detecting KRT bugs, as they also do not consider kernel concurrency and shared variables.

The existing approaches of data race detection focus on identifying shared variables and checking the lock protection of these variables, by using static lockset analysis or mining locking rules. Some issues found by these approaches can indeed cause KRT bugs, due to lacking lock protection for the check or the use operation of the shared variable. However, many KRT bugs are actually caused by atomicity violations, not data races. For example, supposing v is a shared variable, the operation sequence $\{lock(v) \rightarrow check(v) \rightarrow unlock(v) \rightarrow lock(v) \rightarrow use(v) \rightarrow unlock(v)\}$ does not cause a data race, but it causes an atomicity violation of the check-use operations on v, leading to a KRT bug. Moreover, among the reported data races, these approaches fail to automatically identify those that can cause KRT bugs, and they do not analyze the check-use operations of shared variables. Thus, identifying KRT bugs still requires much manual effort for checking the reported data races. These approaches are thus very limited in the task of KRT bug detection.

As described in Section 2.2, we find that over 68% of kernel TOCTOU bugs are actually KRT bugs, but there is still no systematic approach that focuses on detecting KRT bugs. Thus, it is important to design such a new approach to fill this research gap of kernel security.

3 Basic Idea and Challenges

As KRT bugs are introduced by atomicity violations in the check-use operations of specific shared variables, our basic

Approach	Target Bug	TOCTOU Detection			Reason for Missing KRT Bugs
		URT	HRT	KRT	
Wang et al. [46]	Double fetch	●	○	○	Neglecting kernel concurrency and shared variables
DEADLINE [55]	Double fetch	●	○	○	Neglecting kernel concurrency and shared variables
LRSan [50]	Lacking recheck	●	○	○	Neglecting kernel concurrency and shared variables
SADA [6]	Unsafe DMA access	○	●	○	Neglecting kernel concurrency and shared variables
LR-Miner [31]	Data race	○	○	◐	Neglecting the atomicity of check-use operations
CPALockator [2]	Data race	○	○	◐	Neglecting the atomicity of check-use operations
KERAT (our work)	KRT bug	○	○	●	***Solution:*** Mining atomicity rules about kernel concurrency

Table 2: Comparison of state-of-the-art static approaches that can find kernel TOCTOU bugs.

idea for KRT bug detection is to statically detect such atomicity violations in kernel code. However, achieving this is still difficult and requires addressing two important challenges:

(C1) How to mine atomicity rules, i.e. which lock should protect the check-use operations of which shared variable? The most relevant approach is LR-Miner [31] that mines locking rules describing which lock should protect which shared variable. Although locking rules and atomicity rules are similar, they still have very different targets of lock protection. The target of a locking rule is each access of a shared variable, while the target of an atomicity rule is each check-use pair of a shared variable. In other words, obeying locking rules does not mean obeying atomicity rules. As a result, a new method is required to mine atomicity rules from kernel code.

(C2) How to use the mined atomicity rules to effectively detect KRT bugs? Different from data races that have just one occurrence pattern namely lacking lock protection of single variable access, KRT bugs have multiple occurrence patterns involving lock-acquire/release and variable-check/use operations about the corresponding atomicity rule. Thus, KRT bug detection is more complex than race detection. Moreover, the OS kernel has a very large code base (e.g., Linux-6.8 contains over 17 million lines of source code, according to the calculation result of CLOC [10]), and it heavily uses pointers and data structures, so achieving both good accuracy and efficiency of atomicity rule validation is challenging.

4 Key Techniques

To address the two challenges mentioned in Section 3, we propose two key techniques. To solve *C1*, we propose an atomicity-rule mining method to effectively identify which lock should protect the check-use operations of which shared variable; and to solve *C2*, we propose a state-based validation strategy to detect KRT bugs that violate the mined atomicity rules. We will introduce these two key techniques as follows:

4.1 Atomicity-Rule Mining Method

As locking rules are similar to atomicity rules, an intuitive way to perform KRT bug detection is to simply tune LR-Miner [31]. However, after analyzing its process of locking

Thread 1

```
void atomicity_violation(struct device *dev) {
  spin_lock(&dev->lock);
  if (dev->state == STATE_A) {   // Check with the lock
    spin_unlock(&dev->lock);
    ... // TOCTOU window
    spin_lock(&dev->lock);
    dev->state = STATE_C;   // Use with the lock
  }
  spin_unlock(&dev->lock);
}
```

Thread 2

```
void locked_state_update(struct device *dev) {
  spin_lock(&dev->lock);
  dev->state = STATE_B;   // Modification with the lock
  spin_unlock(&dev->lock);
}
```

Figure 3: KRT bug example that obeys the locking rule.

rule mining, we find this way infeasible, as LR-Miner has two inherent and significant limitations for KRT bug detection:

(L1) Locking rule fails to guarantee atomicity. A locking rule requires that each single access to a shared variable is protected by a lock. However, a KRT bug is actually caused by an atomicity violation, not just by a data race completely, where the check-use operations of a shared variable must be protected together. Figure 3 illustrates this problem. In Thread1 and Thread2, every individual access (the check and the use of `dev->state`) is protected by `dev->lock`. Thus, the code obeys the locking rule about `dev->state`. However, as the check and use occur in separate critical sections in Thread1, a TOCTOU window still exists so that Thread2 can change `dev->state`, leading to a KRT bug. Thus, locking rules are unsuitable for KRT bug detection.

(L2) Statistical method of locking rule mining can miss many real atomicity rules. For accuracy, LR-Miner uses a statistical method for locking rule mining, by collecting two key numbers for the given data field AP_v (using the access path form, namely *AP* in short) and lock field AP_l: the number num_{all} of all the calling contexts containing accesses to AP_v, and the number $num_{protected}$ of calling contexts containing the locking relation $\langle AP_v, AP_l \rangle$ that indicates the access to AP_v is protected by AP_l. If the proportion of $num_{protected}$ among num_{all} is larger than a threshold T (set to 0.7 in the LR-Miner paper), and at least one of all the accesses is a write, LR-Miner mines a locking rule $\langle AP_v, AP_l \rangle$ that the data

```
FILE: linux-5.16/drivers/tty/mxser.c
971. int mxser_put_char(struct tty_struct *tty, ...) {
           ......
           // Invalid check!
979.       if (info->xmit_cnt >= SERIAL_XMIT_SIZE - 1)
980.          return 0;
           ......
982.       spin_lock_irqsave(&info->slock, flags);
           ......
           // TOCTOU: xmit_cnt may exceed SERIAL_XMIT_SIZE
985.       info->xmit_cnt++;
986.       spin_unlock_irqrestore(&info->slock, flags);
           ......
989.  }

992. void mxser_flush_chars(struct tty_struct *tty) {
           ......
996.       if (!info->xmit_cnt || ...) // Read without lock protection
997.          return;
           ......
1001. }

1012. unsigned int mxser_chars_in_buffer(struct tty_struct *tty) {
1013.    struct mxser_port *info = tty->driver_data;
1014.    return info->xmit_cnt; // Read without lock protection
1015. }
NOTE: info->xmit_cnt is often read without lock protection!
```

Figure 4: A KRT bug missed by statistical mining.

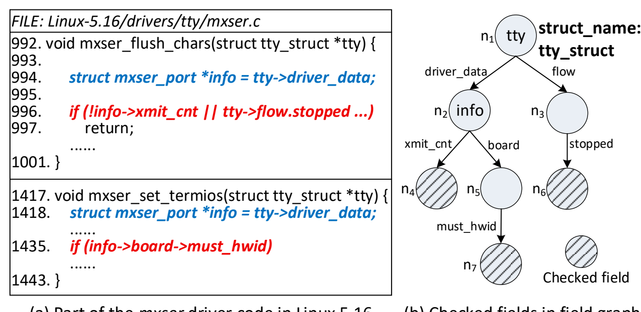


Figure 5: Example of identifying checked fields.

field AP_v should be protected by the lock field AP_l. However, this statistical method fails to consider that there may be many intentional reads of shared variables without lock protection, and thus can miss many real atomicity rules that have a low lock-protection proportion. Figure 4 shows a real-world example. The `mxser` driver contains numerous intentional reads of `info->xmit_cnt` for performance reasons (e.g., in `mxser_flush_chars`). These benign read-side races, which are intentionally performed without lock protection, cause the lock-protection proportion of `info->xmit_cnt` to fall well below the threshold T. As a result, the statistical method fails to identify the real atomicity rule about `info->xmit_cnt` and thus misses the real KRT bug in the code.

Method design. To solve these limitations, we give up mining locking rules or using statistical methods, and instead propose to mine *atomicity rules* in a more precise way. Accordingly, we specify the definition of an atomicity rule, namely *the check and subsequent use operations of a shared structure field must be protected together by a lock field*. In other words, an atomicity rule emphasizes the atomicity property of the check-use operations of a shared data field, which is different from a locking rule that focuses on the atomicity property of single access to a shared data field.

Overall, our method automatically mines atomicity rules from the kernel code, with two main steps: *(S1)* It first identifies the data fields that are used in security checks. *(S2)* It examines whether the modification of a checked data field is always protected by a specific lock field; if so, it considers the check-use operations of this checked field should be protected by the lock field, which is a mined atomicity rule. Different from the statistical method that requires a threshold, our method examines the consistency of lock protection for data field modification, to effectively reduce the imprecision introduced by intentional reads of the data field.

Note that to handle the complex nested data structures prevalent in the kernel, our method uses field graphs [31] to precisely identify whether a checked variable and a lock belong to the same parent data structure, even when they are deeply nested. Specifically, A field graph is defined as $FG = \langle N, E \rangle$, where N is a set of nodes, and each node represents a field that can be in the lower-layer structure. E is a set of edges, and each edge is labeled with a field and represents how a lower-layer structure field is accessed from a higher-layer structure.

S1: Checked-field identification. In this step, our method performs a flow-sensitive and inter-procedural analysis to identify all the data fields used in conditional statements that are possible security checks. The analysis traverses the kernel's control-flow graphs at the code-path level. When it processes an instruction in the code path, it simultaneously performs two key tasks, namely dynamically updating a field graph to map out structure field relationships, and inspecting conditional statements to find each checked variable. When a conditional statement is encountered, the analysis extracts the variables involved in the condition. Then, the analysis immediately uses the current state of the field graph to resolve these variables to their access paths, making the identification process field-sensitive. These resolved fields are recorded as "checked fields", which are the candidate targets of atomicity rule mining in the next step.

Example. Figure 5(a) shows a part of the *mxser* driver code in Linux 5.16. The field graph of the structure variable `tty` is incrementally built in Figure 5(b). For instance, when analyzing the instruction `info = tty->driver_data` at Line 994, our method creates an edge labeled `driver_data` from the existing node *n1* (`tty`) to a new node *n2* (`info`). Then, when reaching the conditional statement at Line 996, our method identifies that `info->xmit_cnt` and `tty->flow.stopped` are two checked fields, corresponding to the nodes *n4* and *n6* dynamically created in the field graph. Similarly, when reaching the conditional statement at Line 1435, our method identifies `info->board->must_hwid` as another checked field matching the node *n7* dynamically created in the field graph.

```
AtomicityRuleMining(CP_set, CF_set)
Input: CP_set: Set of the code paths, CF_set: Set of the checked fields identified in S1
Output: AtomicityRuleSet: Set of the mined atomicity rules
 1: CandPairSet ← ∅
 2: foreach code path cp in CP_set do
 3:     FieldGraph ← ∅; LockSet ← ∅
 4:     foreach inst in cp do
 5:         update FieldGraph and LockSet based on inst
 6:         if inst is a variable-modification instruction then
 7:             var ← GetModifiedVar(inst)
 8:             foreach lock in LockSet do
 9:                 Ancestor ← FindCommonAncestor(var, lock, FieldGraph)
10:                 if Ancestor ≠ NULL then
11:                     AP_var ← GetAccessPath(var, Ancestor, FieldGraph)
12:                     AP_lock ← GetAccessPath(lock, Ancestor, FieldGraph)
13:                     if IsCheckedField(AP_var, CF_set) then
14:                         Insert ⟨AP_var, AP_lock⟩ into CandPairSet
15:                     end if
16:                 end if
17:             end foreach
18:         end if
19:     end foreach
20: end foreach
21: AtomicityRuleSet ← CandPairSet
22: foreach code path cp in CP_set do
23:     FieldGraph ← ∅; LockSet ← ∅
24:     foreach inst in cp do
25:         update FieldGraph and LockSet based on inst
26:         if inst is a variable-modification instruction then
27:             var ← GetModifiedVar(inst)
28:             foreach ⟨AP_v, AP_l⟩ in CandPairSet do
29:                 if FieldFormMatch(var, AP_v, FieldGraph) then
30:                     if AP_l ∉ FieldFormMatchSet(LockSet, FieldGraph) then
31:                         Remove ⟨AP_v, AP_l⟩ from AtomicityRuleSet
32:                     end if
33:                 end if
34:             end foreach
35:         end if
36:     end foreach
37: end foreach
38: return AtomicityRuleSet
```

Figure 6: Atomicity-rule mining algorithm.

S2: Atomicity-rule mining. For each checked field identified in *S1*, this step determines whether it is consistently modified with lock protection, to mine the related atomicity rules. Figure 6 shows the detailed mining process, which consists of two main phases:

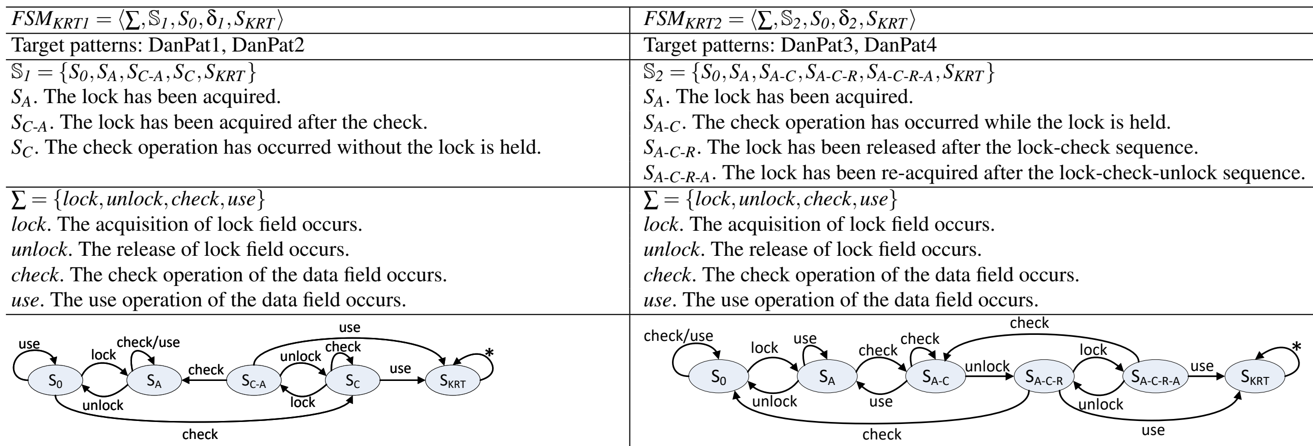
In the first phase (Lines 1–20), our method identifies each candidate pair of the modified data field and its protected lock field. It first initializes an empty set *CandPairSet* that stores all the candidate pairs. Then, it traverses the control-flow graphs of the kernel code via code paths. For each code path, it dynamically maintains a field graph *FieldGraph* and a lockset *LockSet* by analyzing each instruction in this code path. When encountering a variable-modification instruction, it gets the modified variable *var*. For this variable, our method gets each protecting lock *lock* in *LockSet*, and validates whether *var* and *lock* are different fields but in the same data structure by checking whether the nodes representing *var* and *lock* have a common ancestor (representing a common higher-layer structure) in *FieldGraph*. If so, our method gets the access paths AP_{var} and AP_{lock} of *var* and *lock*, respectively. If AP_{var} is a checked field in CP_{set} identified in *S1*, $\langle AP_{var}, AP_{lock} \rangle$ is considered as a candidate pair of the atomicity rule, and thus this pair is added to *CandPairSet*.

```
FILE: linux-5.16/drivers/tty/mxser.c
Simplified Code Path CP1:
mxser_activate
  -- spin_lock_irqsave(&info->slock, flags); [Line 730]   // Lock
  -- mxser_change_speed [Line 798]
    -- mxser_handle_cts [Line 636]
      -- tty->hw_stopped = 1; [Line 554]   // Modification
Simplified Code Path CP2:
mxser_set_termios
  -- tty->hw_stopped = 0; [Line 1427]   // Modification without info->slock
Inconsistent lock protection about tty->hw_stopped and info->slock
Simplified Code Path CP3:
mxser_interrupt
  -- spin_lock(&info->slock); [Line 1754]   // Lock
  -- mxser_port_isr [Line 1756]
    -- mxser_transmit_chars [Line 1661]
      -- info->xmit_cnt--; [Line 514]   // Modification
Simplified Code Path CP4:
mxser_put_char
  -- spin_lock_irqsave(&info->slock, flags); [Line 982]   // Lock
  -- info->xmit_cnt++; [Line 985]   // Modification
Simplified Code Path CP5:
mxser_write
  -- spin_lock_irqsave(&info->slock, flags); [Line 953]   // Lock
  -- info->xmit_cnt += c; [Line 956]   // Modification
The modification of info.xmit_cnt is always protected by info.slock
An atomicity rule is mined: <info.xmit_cnt, info.slock>
```

Figure 7: Example of mining atomicity rules.

In the second phase (Lines 21–38), our method checks each candidate pair collected in the first phase, to extract atomicity rules. It first puts all the candidate pairs in *CandPairSet* into *AtomicityRuleSet*. Similar to the first phase, our method analyzes each code path in the kernel code, dynamically maintains a field graph *FieldGraph* and a lockset *LockSet* in the code path, and handles each variable-modification instruction. For each modified variable *var*, our method analyzes each candidate pair $\langle AP_v, AP_l \rangle$ in *CandPairSet*, to check whether the accessed data field AP_v can match the data field of *var* according to *FieldGraph*. If so, our method checks whether the corresponding lock field AP_l has a matched lock in *LockSet*. If not, our method considers that the modification of AP_v is not always protected by AP_l, namely $\langle AP_v, AP_l \rangle$ cannot be identified as a valid atomicity rule, and thus this pair is removed from *AtomicityRuleSet*. After traversing the code paths, our method returns the final *AtomicityRuleSet* that stores all the mined atomicity rules.

Example. Figure 7 illustrates the process of atomicity rule mining in the *mxser* driver source code. As for the data field `tty->hw_stopped`, although its modification is protected by the lock field `info->slock` in the code path *CP1*, there still exists a code path *CP2* where `info->slock` does not protect its modification, so our method does not extract an atomicity rule about these two fields. As for another data field `info->xmit_cnt`, our method finds that its modification is always protected by the lock field `info->slock` in all the analyzed code paths like *CP3*, *CP4* and *CP5*, and thus our method extracts an atomicity rule ⟨`info.xmit_cnt`, `info.slock`⟩. This atomicity rule indicates that the check-use operations of the data field `info.xmit_cnt` should be protected together by the lock field `info.slock`.



$FSM_{KRT1} = \langle \Sigma, \mathbb{S}_1, S_0, \delta_1, S_{KRT} \rangle$	$FSM_{KRT2} = \langle \Sigma, \mathbb{S}_2, S_0, \delta_2, S_{KRT} \rangle$
Target patterns: DanPat1, DanPat2	Target patterns: DanPat3, DanPat4
$\mathbb{S}_1 = \{S_0, S_A, S_{C\text{-}A}, S_C, S_{KRT}\}$ S_A. The lock has been acquired. $S_{C\text{-}A}$. The lock has been acquired after the check. S_C. The check operation has occurred without the lock is held.	$\mathbb{S}_2 = \{S_0, S_A, S_{A\text{-}C}, S_{A\text{-}C\text{-}R}, S_{A\text{-}C\text{-}R\text{-}A}, S_{KRT}\}$ S_A. The lock has been acquired. $S_{A\text{-}C}$. The check operation has occurred while the lock is held. $S_{A\text{-}C\text{-}R}$. The lock has been released after the lock-check sequence. $S_{A\text{-}C\text{-}R\text{-}A}$. The lock has been re-acquired after the lock-check-unlock sequence.
$\Sigma = \{lock, unlock, check, use\}$ *lock*. The acquisition of lock field occurs. *unlock*. The release of lock field occurs. *check*. The check operation of the data field occurs. *use*. The use operation of the data field occurs.	$\Sigma = \{lock, unlock, check, use\}$ *lock*. The acquisition of lock field occurs. *unlock*. The release of lock field occurs. *check*. The check operation of the data field occurs. *use*. The use operation of the data field occurs.

Table 3: Two built FSMs of KRT bug detection.

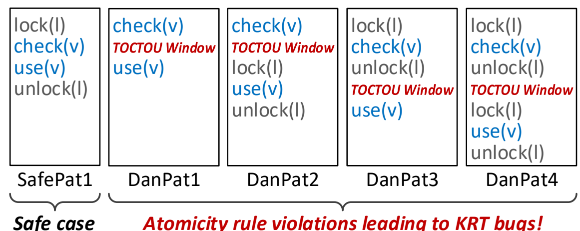


Figure 8: Five patterns of lock and check-use operations.

4.2 State-Based Validation Strategy

Pattern study. Section 4.1 mines atomicity rules by validating whether the modification of a checked data field is always protected by a specific lock field. In other words, for a mined atomicity rule, the check-use operations of the data field should be protected together by the lock field. Otherwise, an atomicity rule violation can lead to a KRT bug. To detect atomicity rule violations, we first study all the possible patterns of lock and check-use operations involving the data and lock fields. We find that there are five possible patterns, as shown in Figure 8.

The pattern *SafePat1* is the safe case as it obeys the atomicity rule, and the other four patterns are dangerous cases as they violate atomicity rules and can cause KRT bugs:

- *DanPat1*: Neither the check nor use operation is protected by the lock, so a TOCTOU window exists between the check and use operations.
- *DanPat2*: The use operation is protected by the lock, but the check operation is not protected, so a TOCTOU window exists between the lock-acquire and use operations.
- *DanPat3*: The check operation is protected by the lock, but the use operation is not protected, so a TOCTOU window exists between the lock-release and use operations.
- *DanPat4*: The check and use operations are separately protected by the lock, but not together, so a TOCTOU window exists between the lock-release and lock-acquire operations.

State tracking. To effectively detect KRT bugs that are caused by these four dangerous patterns, we design a state-based strategy that builds and uses two finite state machines (FSMs) to track the program state using dataflow analysis. The two FSMs are defined in Table 3. Specifically, FSM_{KRT1} is used to detect *DanPat1* and *DanPat2*, while FSM_{KRT2} is used to detect *DanPat3* and *DanPat4*. In the two FSMs, all the states are named based on the sequence of the already performed operations (*A* for lock acquiring, *C* for variable check, and *R* for lock release). Specifically, the state transitions of detecting the four dangerous patterns based on the two FSMs are explained as follows:

- *DanPat1*: $\{S_0 \xrightarrow{check} S_C \xrightarrow{use} S_{KRT}\}$ in FSM_{KRT1}.
- *DanPat2*: $\{S_0 \xrightarrow{check} S_C \xrightarrow{lock} S_{C\text{-}A} \xrightarrow{use} S_{KRT}\}$ in FSM_{KRT1}.
- *DanPat3*: $\{S_0 \xrightarrow{lock} S_A \xrightarrow{check} S_{A\text{-}C} \xrightarrow{unlock} S_{A\text{-}C\text{-}R} \xrightarrow{use} S_{KRT}\}$ in FSM_{KRT2}.
- *DanPat4*: $\{S_0 \xrightarrow{lock} S_A \xrightarrow{check} S_{A\text{-}C} \xrightarrow{unlock} S_{A\text{-}C\text{-}R} \xrightarrow{lock} S_{A\text{-}C\text{-}R\text{-}A} \xrightarrow{use} S_{KRT}\}$ in FSM_{KRT2}.

The two FSMs also handle other cases that never cause KRT bugs. For example, as the safe pattern *SafePat1*, its state transitions in FSM_{KRT1} are $\{S_0 \xrightarrow{lock} S_A \xrightarrow{check} S_A \xrightarrow{use} S_A \xrightarrow{unlock} S_0\}$, which never reaches the buggy state S_{KRT}. Note that although the two FSMs involve lock-related operations, they are just used for KRT bug detection in our strategy, and they are not applicable to detecting other lock-misuse bugs such as the double locks and missing unlock, due to lacking the corresponding state transitions.

In fact, we believe it is possible to combine the two FSMs into a large and unified one, but there would be over ten states and dozens of state transitions in this combined FSM, making it too complex to build, track or explain. During dataflow analysis, our strategy executes two separate tasks to simultaneously track the program state based on the two FSMs. If any task reaches the buggy state S_{KRT}, our strategy reports a possible KRT bug.

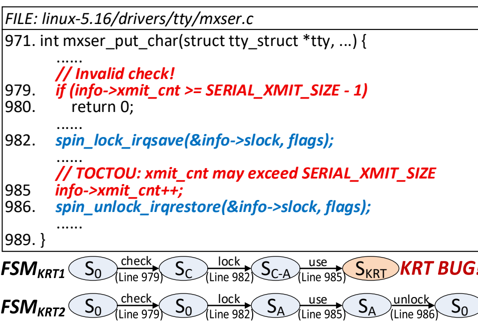


Figure 9: Example of state-based validation.

Alias relationship handling. Because a KRT bug involves a checked/used variable and a lock variable, the alias relationships of these two variables should be analyzed. Otherwise, false positives and negatives can occur for KRT bug detection. For example, suppose the lock field is assigned to a local variable, and then this local variable is used for lock-acquiring/release operations. If the alias relationship between the lock field and the local variable is neglected, FSM_{KRT1} can report a false bug involving *DanPat1*, even though the true case is *SafePat1*, as FSM_{KRT1} fails to find the lock field in the lock-acquiring operation. Another example is that, the data field is assigned to a local variable, then this local variable is checked without lock protection, and finally the data field is used without lock protection. If the alias relationship between the data field and local variable is neglected, FSM_{KRT1} cannot track the program state to reach S_{KRT} and thus would miss this real bug involving *DanPat1*.

To handle the above alias relationships, similar to Section 4.1, our strategy also builds and analyzes field graphs during dataflow analysis. Moreover, inspired by recent alias-based static approaches [32,33,43,56], our strategy maintains the program state at the granularity of aliased sets, not variables. Specifically, it identifies the aliased variables from the built field graphs, and stores them in an alias set. During state tracking, our strategy maintains the state of each alias set, and thus it does not need to synchronize the states of all the aliased variables, which is time-consuming and error-prone.

Inter-procedural analysis. In some cases, lock-related and variable-check/use operations are in different functions, and thus it is necessary to perform state tracking across functions. To accelerate the datatlow analysis of large-scale kernel code, our strategy uses function summaries to reduce repeated analysis for inter-procedural validation. Specifically, after the intra-procedural analysis of a function, our strategy creates a summary for it. This summary stores an ordered list of pairs $\langle operation, AP \rangle$, where *operation* is one of the four actions (*lock*, *unlock*, *check* and *use*) and *AP* is the access path of the involved variable or lock. When our strategy encounters a call to a function that already has a summary, it directly instantiates this summary using the actual arguments of the call site, instead of re-analyzing the function body.

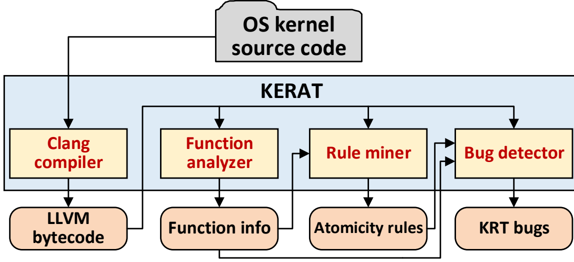


Figure 10: KERAT architecture.

Example. Figure 9 illustrates how our strategy detects the KRT bug presented in Figure 4. According to the atomicity rule ⟨`info.xmit_cnt`,`info.slock`⟩ mined in Figure 7, the check-use operations of `info.xmit_cnt` should be protected together with `info.slock`, and thus our strategy instantiates the two FSMs in Table 3 using the two fields and performs state tracking in the function `mxser_put_char`. There are four key operations in order, as shown in the figure, and thus our strategy finds that the program state reaches S_{KRT} in FSM_{KRT1}, and accordingly reports a KRT bug.

5 KERAT Approach

Based on the two key techniques introduced in Section 4, we design KERAT, the *first* systematic static approach for detecting TOCTOU bugs caused by kernel races. We have implemented KERAT with about 12K lines of C++ code based on the Clang compiler [9], and it performs automated static analysis on the kernel LLVM bytecode. Figure 10 shows the architecture of KERAT, which has three phases:

Phase1: Code compilation. The *Clang compiler* first compiles the kernel source files into LLVM bytecode files. Then, the *function analyzer* collects each function's information (including the name and body location) in a database, for inter-procedural analysis in the subsequent phases.

Phase2: Atomicity-rule mining. The *rule miner* uses our atomicity-rule mining method in Section 4.1, to mine atomicity rules from kernel code. Note that to handle global locks and variables that are not passed as function arguments, it introduces a virtual root node in the field graph to serve as the common ancestor for global data and function arguments.

Phase3: KRT bug detection. The *bug detector* uses our state-based validation strategy in Section 4.2, to detect KRT bugs that violate the mined atomicity rules. To drop false positives, the detector checks the code-path feasibility using an SMT solver Z3 [57], and neglects module initialization and removal functions by search for the keywords like "probe" and "remove", as there is no concurrency during the execution of such functions [19,31,45]. To avoid duplicate reports for the same bug found via different code paths, the detector guarantees that each reported bug has unique locations of the related check-use operations in the source code.

OS	Version	Release year	Source files (*.c)	LOC
Linux	5.16	2022	30.7K	15.9M
	6.8	2024	33.7K	17.8M
FreeBSD	14.1	2024	19.9K	9.3M

Table 4: Information about the checked OS kernels.

Description		Linux-5.16	Linux-6.8	FreeBSD
Code analysis	Source files (analyzed/all)	21.3K/30.7K	23.8K/33.7K	4.2K/19.9K
	Code lines (analyzed/all)	13.0M/15.9M	14.7M/17.8M	3.4M/9.3M
Rule mining	Checked data fields	120.1K	133.8K	29.3K
	Mined atomicity rules	1.9K	1.9K	0.4K
Bug detection	Handled state transitions	1.1M	1.2M	0.2M
	Violated atomicity rules	242	252	22
	Dropped false bugs	448	398	35
	Found KRT bugs (real/all)	320/392	315/388	36/45
Time usage	Rule mining	15h27m	19h41m	4h19m
	Bug detection	5h36m	8h57m	2h18m
	Total time	21h03m	28h38m	6h37m

Table 5: Analysis results of the three OS kernels.

6 Evaluation

We utilize KERAT to check three OS kernels: Linux-5.16, Linux-6.8 and FreeBSD-14.1. Among them, Linux-6.8 and FreeBSD-14.1 are the latest minor kernel release versions as of the KERAT implementation, and Linux-5.16 is selected to validate whether KERAT can find known bugs. Table 4 describes the kernels, including the release year and the number of source code lines as counted by CLOC [10]. For the Linux kernels, we use the *allyesconfig* kernel configuration to enable as much code as possible for the x86-64 architecture. Because the FreeBSD kernel provides no such a configuration, we use its default *GENERIC* configuration file. We run the evaluation on a regular x86-64 PC equipped with sixteen Intel Xeon CPU@2.10GHz cores and 128GB of physical memory.

6.1 Bug Detection

We run KERAT to automatically mine atomicity rules and detect KRT bugs in the three target OS kernels. Then, we manually study all the bugs found by KERAT to check whether they are real. Table 5 summarizes the analysis results. We make the following observations:

Code analysis. KERAT is scalable to a large code base, and it in total analyzes 31.1M lines of code across 49.3K source files within less than 57 hours. The remaining source files are not analyzed because they are not enabled by the selected configurations. We believe that KERAT could find more bugs if these files are enabled with suitable kernel configurations.

Atomicity-rule mining. Across the three kernels, KERAT identifies over 283K data fields used in possible security checks. By analyzing the modification and lock usage of these data fields, it successfully mines 4.2K atomicity rules. Besides KRT bug detection, we believe that these mined atomicity rules can also help to improve kernel concurrency documentation and develop secure kernel code.

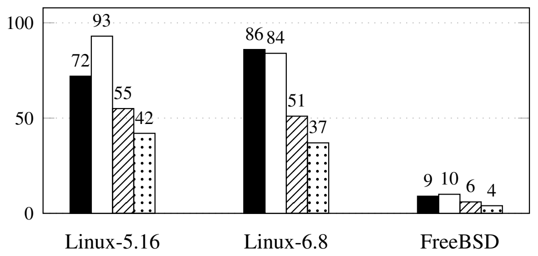


Figure 11: Security impact of KRT bugs across kernels.

KRT bug detection. According to the mined atomicity rules, KERAT handles 2.5M state transitions during state tracking with the two FSMs in Table 3. Among the 4.2K mined atomicity rules, KERAT discovers that 516 are violated, leading to 825 KRT bugs found in total. Two master's students spent 24 hours studying these found bugs, and identified 671 of them to be real bugs. Note that KERAT drops 881 false bugs due to their infeasible code paths and non-concurrency functions, and it achieves a low false positive rate of 18.7%.

Among the 320 real bugs found in Linux-5.16, 76 have been fixed in Linux-6.8, indicating that KERAT can find known KRT bugs. Among the 315 real bugs found in Linux-6.8, 244 are inherited from Linux-5.16 (so they have been present for at least two years), and 71 are introduced in newly-developed kernel code, indicating KERAT can find new KRT bugs.

Bug features. We manually analyze the 671 real KRT bugs found by KERAT, and find some interesting features. First, 78% of the found bugs are in drivers, and the remaining are in the sound subsystem, filesystems and network modules, indicating drivers may be more error-prone than other kernel parts. Second, 15% of the found bugs involve multiple function calls between the check and use operations, indicating the complexity of these bugs. Finally, 65%, 30% and 5% of the found bugs occur due to variable reads, dereferences and writes for the use operations, respectively, indicating check-read pairs may be more error-prone in kernel TOCTOU situations.

Security impact of the found bugs. We manually check the source code of the 671 real KRT bugs to identify their security impact. Among them, 122 bugs are considered to have no security impact, as the involved shared variables are used for non-core functionalities like log printing and code debugging. The remaining 549 bugs are considered to be harmful, and we classify them into four categories:

C1) Memory corruption. 167 bugs (including 72 for Linux-5.16, 86 for Linux-6.8 and 9 for FreeBSD-14.1) can cause use-after-free, out-of-bounds-write and double-free issues, as the involved shared variables are used for memory-free and buffer-bound-check operations. The attacker can exploit these bugs to corrupt memory and cause security problems like privilege escalation.

C2) System crash. 187 bugs (including 93 for Linux-5.16, 84 for Linux-6.8 and 10 for FreeBSD-14.1) can cause null-pointer dereferences and division-by-zero issues, as the involved shared variables are used for null-check and integer-division operations. The attacker can exploit these bugs to crash the kernel and cause denial of service.

C3) Undefined behavior. 112 bugs (including 55 for Linux-5.16, 51 for Linux-6.8 and 6 for FreeBSD-14.1) can cause overlength bit shifting, signed integer overflow and other undefined behavior issues, as the involved shared variables are used for bit-shift and special-value-check operations. The attacker can exploit these bugs to make kernel execution unexpected and thus lead to denial of service.

C4) Logic error. 83 bugs (including 42 for Linux-5.16, 37 for Linux-6.8 and 4 for FreeBSD-14.1) can cause unexpected execution of kernel logic, as the involved shared variables are used for important semantics. The attacker can exploit these bugs to cause kernel malfunctions and denial of service.

Bug reporting. For the 287 harmful KRT bugs in Linux-6.8 and FreeBSD-14.1, we have reported them to kernel developers, and 65 of them have been confirmed. Our 21 patches fixing 36 bugs have been applied, and 55 confirmed bugs have been fixed. We are still waiting for the response on the remaining bugs. Furthermore, 10 confirmed bugs have been assigned CVE IDs. Some kernel developers also express their willingness to use KERAT for continuous kernel analysis and to add the mined atomicity rules into kernel documentation.

6.2 False Positives and Negatives

False positives. KERAT still reports 154 false bugs in the three checked OS kernels, for three main reasons:

First, 76 false bugs are introduced due to mining incorrect atomicity rules about lock over protection. Indeed, in some cases, multiple data fields are always modified together with lock protection in the code, but only one of them is actually the protection target. KERAT then mines incorrect atomicity rules about the other data fields. We infer that lock over protection occurs because the developers are uncertain about which data field should be protected by which lock, and thus just blindly use lock protection for all the shared data fields.

Second, 40 false bugs are introduced due to redundant variable checks that have no meaningful relation to subsequent variable uses. For example, a null check is placed before the memory-free operation due to defensive programming style, and KERAT reports a bug where a TOCTOU window can occur between the check and memory-free operation. However, it is safe to free a null pointer, so this reported bug is false.

Finally, 38 false bugs are introduced due to complex situations that KERAT does not handle, including multiple nested loops and array element accesses with non-constant indices.

False negatives. KERAT can still miss some real KRT bugs for two main reasons:

First, KERAT mines an atomicity rule only when the modification of a data field is always protected by a lock field. However, in some cases, the modification of a data field is protected by non-lock synchronization primitives (like wait queues and reference counts) that are neglected by KERAT. Thus, KERAT fails to mine the atomicity rules about these primitives and thus misses the related KRT bugs.

Second, KERAT does not handle the functions and macros for atomic variable modification (like `atomic_set` in Linux). As it fails to mine atomicity rules about atomic variables, it misses the related KRT bugs.

6.3 Case Studies of the Found KRT Bugs

Figure12 shows three KRT bugs found by KERAT in Linux and FreeBSD that have been confirmed.

Memory corruption in Linux SCSI driver. In Figure 12(a), Thread1 assigns the pointer `nvmebuf` with the shared buffer pointer `ctxp->rqb_buffer` (①) that is non-null, and passes the null check of this pointer (②). After that, Thread2 assigns the pointer `nvmebuf` with the shared buffer pointer `ctxp->rqb_buffer` (③), and also assigns this shared pointer with null (④). It then invokes the pointer's member function `rqb_free_buffer` to free the buffer memory of `nvmebuf` (⑤). Afterwards, Thread1 uses the member of `nvmebuf` (⑥), causing a use-after-free issue. This bug can be exploited to corrupt the memory of a data buffer and inject malicious data.

System crash in FreeBSD LAGG network module. In Figure 12(b), Thread1 checks whether the shared variable `sc->sc_proto` is *PROTO_NONE* representing the empty protocol (①), and this check is passed because the protocol is not empty. Thread2 calls the function `lagg_proto_detach` (②) that assigns the shared variable `sc->sc_proto` with *PROTO_NONE* to clear the protocol (③). Then, Thread1 calls the function `lagg_proto_start` (④) that uses the member `pr_start` of the element `lagg_protos[sc->sc_proto]` (⑤). Because `sc->sc_proto` is *PROTO_NONE* at this time, `lagg_protos[sc->sc_proto]` is a null pointer due to the empty protocol, causing a null-pointer dereference. This bug can be exploited to cause system crash and denial of service.

Undefined behavior in Linux HWMON driver. In Figure 12(c), Thread1 checks whether the shared integer variable `data->fan_source[chan]` is *SOURCE_INVALID* that represents the invalid source namely *0xff* (①), and this check is passed because the fan source is valid. Thread2 assigns the shared variable `data->fan_source[chan]` with *SOURCE_INVALID* to set an invalid state of the fan source (②). Then, Thread1 utilizes `data->fan_source[chan]` as the length of bit shifting for an unsigned integer 1 (③). Because this variable is *0xff* at this time, which exceeds the length of the unsigned integer, an undefined behavior occurs. This bug can be exploited to interfere driver execution and cause denial of service.

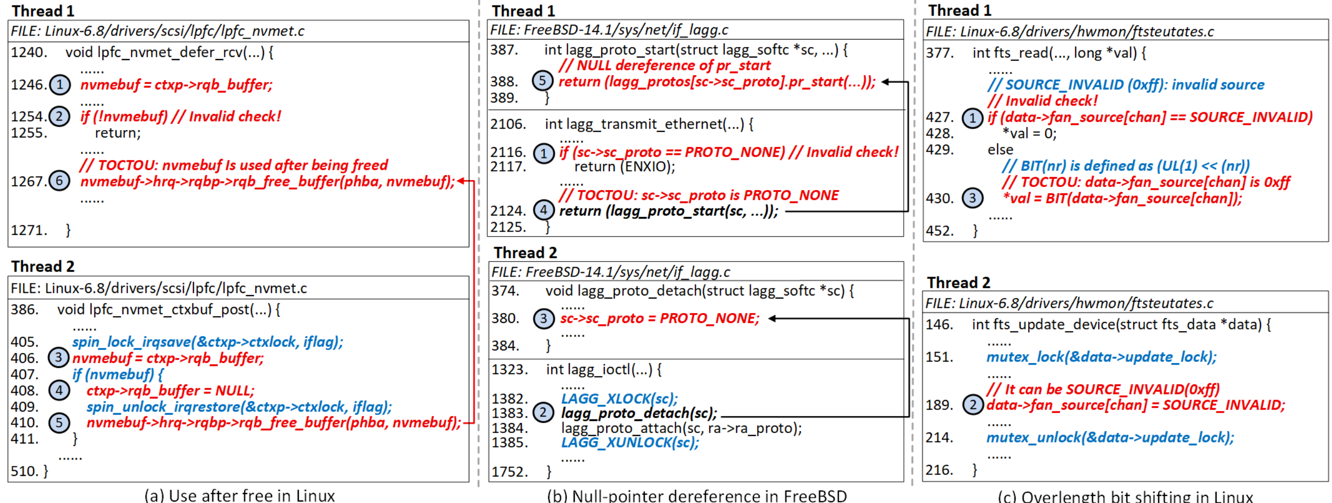


Figure 12: Three example KRT bugs found by KERAT.

6.4 Comparison Experiment

To our knowledge, before KERAT, there has been no systematic static approach that focuses on detecting KRT bugs. Thus, in experimental comparison, we select three state-of-the-art static analysis approaches LRSan [50], CPALockator [2] and LR-Miner [31] for two reasons: (1) they are designed to detect lacking-rechecking bugs or data races, which are very relevant to KRT bugs; and (2) they use security check validation or concurrency analysis, which are relevant to the key techniques of KERAT. On the other hand, for existing static approaches of double-fetch detection [46, 47, 55] and hardware access validation [6, 35], they do not analyze security checks or kernel concurrency, making them irrelevant to KRT bug detection.

In the experiment, we build LRSan [48] and CPALockator [12, 13] from their source code, and use the publicly available artifact of LR-Miner [30]. We select Linux 5.16 as the checked OS kernel, as LRSan and CPALockator can only check old kernel versions and produces many fatal errors when analyzing Linux-6.8 and FreeBSD-14.1. We run each compared approach and manually study its reports to identify real bugs, especially KRT bugs. Table 6 summarizes the analysis results, and we make the following observations:

(1) CPALockator and LR-Miner find 5 and 12 real KRT bugs from the hundreds of found data races, respectively. In fact, we identify these KRT bugs as the racy variables happen to be read in some security checks from the race reports. Though CPALockator and LR-Miner fail to validate the check-use operations of racy variables, we still consider the two approaches are useful to find these KRT bugs. By contrast, LRSan reports over 3,600 lacking-rechecking bugs that require much manual work to check completely, so we randomly select 300 ones for manual checking, and identify only 3 are real. However, none of these real bugs is related to a KRT bug, as they do not involve kernel concurrency.

Description	LRSan	CPALockator	LR-Miner	KERAT
Target bug	Lacking recheck	Data race	Data race	KRT bug
Reported bugs	3,652	773	373	392
Real bugs	3 in 300	23	257	320
Real KRT bugs	0 in 300	5	12	320
Time usage	5h21m	132h21m	17h24m	21h03m

Table 6: Comparison results of Linux-5.16.

(2) KERAT finds all the real KRT bugs found by the three compared approaches, and it also finds more KRT bugs missed by these approaches. The methodology of KERAT gives better results of KRT bug detection for the following reasons:

First, compared to LRSan that neglects kernel concurrency, KERAT performs accurate concurrency analysis of the kernel code by considering lock usages and shared structure fields.

Second, compared to CPALockator that uses classical lockset analysis and model checking with a simple but unreal assumption that all the functions can be concurrently executed, KERAT improves the accuracy and efficiency of concurrency analysis by mining atomicity rules about kernel concurrency.

Finally, LR-Miner is the most relevant approach that mines locking rules for race detection. However, the target of locking rule is each access of a shared variable, while the target of atomicity rule is each check-use pair of a shared variable. In other words, obeying locking rules does not mean obeying atomicity rules, as Figure 3 illustrates. Moreover, LR-Miner uses a statistical method to mine locking rules, and this method fails to consider intentional reads of shared variables without lock protection, which can introduce much inaccuracy for atomicity rule mining. By contrast, KERAT mines atomicity rules by analyzing the consistency of lock protection for data field modification, and detects KRT bugs caused by atomicity rule violations with accurate state tracking.

(3) KERAT is slower than LRSan and LR-Miner, as it has more complicated analyses of field modification handling for atomicity rule mining and state tracking for bug detection. KERAT is faster than CPALockator, as CPALockator suffers from the concurrency state explosion of model checking.

7 Discussion

Exploitability of the found KRT bugs. With the knowledge of the code information about KRT bugs, an attacker can construct special workloads that triggers these bugs. Then by utilizing the related TOCTOU windows, the attacker can transform these bugs into deterministic vulnerabilities like use after free and buffer overflow, which can be exploited for privilege escalation, information leak, etc. Prior studies [28, 40] have studied techniques for enlarging the race window to exploit such concurrency bugs.

Limitations and future works. KERAT can be improved from several aspects. First, KERAT produces some false positives, mainly due to lock over protection and redundant variable checks in the kernel code. We plan to handle these situations to reduce these false positives. Second, KERAT misses some real KRT bugs, due to neglecting non-lock synchronization primitives and atomic variable accesses. We plan to handle these special operations to find the related KRT bugs. Third, we plan to study the actual triggering conditions of the found KRT bugs, to construct the related PoC inputs (like system calls) automatically or semi-automatically, for reproducing bugs reliably and for precise security impact inspection. Finally, we plan to explore the possibility of automatic patch generation and automatic bug report analysis. For patch generation, it might be feasible to use the required locks identified by KERAT, while checking deadlock conditions to address liveness. For bug report analysis, LLM assistance may help check the reports and reduce manual effort.

8 Related Work

TOCTOU detection. As for user-level applications, some approaches focus on detecting TOCTOU bugs related to concurrent threads [60], file operations [29, 51], and specialized environments like Intel SGX [8]. Because OS kernels have a different concurrency model than user-level applications, these approaches are inapplicable to detecting kernel TOCTOU bugs. To solve this problem, some approaches of kernel static analysis have been designed to detect double-fetch issues [46, 47, 55], lacking-recheck [50] bugs and unsafe hardware accesses [6, 35], which can involve TOCTOU cases. However, they focus on user-space and hardware-I/O race conditions, without the awareness of kernel concurrency, so they still miss many TOCTOU bugs caused by kernel races.

To our knowledge, KERAT is the first systematic static analysis approach of detecting TOCTOU bugs caused by kernel races. Different from the above approaches of kernel static analysis, KERAT sufficiently considers kernel concurrency and mines atomicity rules about lock protection of shared variables from kernel code. Then, it tracks the program state based on state machines encoding of common bug patterns, to detect the violations of these rules as TOCTOU bugs.

TOCTOU attacks and defenses. Some approaches [3, 27] investigate TOCTOU attacks of remote attestation protocols in IoT network scenarios and propose various defense mechanisms based on certificates or zero-knowledge proofs. Some other approaches focuses on studying the attacks and defenses of TOCTOU bugs related to file operations [39], Intel SGX enclaves [52] and code compilers [53].

For OS kernels, some approaches explore the attack and defense techniques of double-fetch issues [18, 42] and unsafe DMA accesses [1, 38], which can lead to kernel TOCTOU bugs caused by user-space and hardware-I/O race conditions, respectively. Besides, some approaches [28, 40] finds that the race window of kernel concurrency bugs can be enlarged using inter-processor interrupts, to increase the exploitability of kernel TOCTOU bugs caused by kernel races. We believe these approaches can help to understand the exploitability and mitigation of the KRT bugs found by KERAT.

Kernel concurrency analysis. Some approaches [23–26, 37, 41, 54] utilize dynamic analysis to detect kernel concurrency bugs with memory watchpoints, happens-before relations, coverage-guided fuzzing, etc. However, due to insufficient testing time and a limited number of test cases, they fail to cover many complex and infrequent concurrent execution situations, and thus miss many real concurrency bugs. By contrast, some approaches [2, 4, 5, 19, 31, 36, 45, 56, 59] use static analysis to validate kernel concurrency without actual execution of the OS kernel, and thus they can conveniently achieve high code coverage and detect some complex bugs. However, these approaches focus on detecting common kinds of concurrency issues like data races [2, 19, 31, 45], deadlocks [5, 19, 56], concurrency use-after-free bugs [4, 59] and multi-variable related bugs [36] without detecting atomicity violations about the check-use operations of shared variables, so they are ineffective in KRT bug detection. To solve this problem, KERAT mines atomicity rules about lock protection of the shared variables from kernel code, and then detects the violations of these rules as TOCTOU bugs, based on state machines encoding of common bug patterns.

9 Conclusion

According to our study on Linux kernel patches, kernel race is the most common root cause of kernel TOCTOU bugs. However, there is still no systematic approach that focuses on detecting TOCTOU bugs caused by kernel races. To fill this research gap, we have designed KERAT, the first systematic static approach for detecting TOCTOU bugs caused by kernel

races. Specifically, KERAT has two key techniques: (1) an *atomicity-rule mining method* to effectively identify which lock should protect the check-use operations of which shared variable; and (2) a *state-based validation strategy* to detect TOCTOU bugs that violate the mined atomicity rules based on state machines encoding of common bug patterns. We have evaluated KERAT on Linux-6.8 and FreeBSD-14.1, and found 351 real bugs, 65 of which have been confirmed.

Acknowledgment

We thank the anonymous reviewers and the shepherd for their helpful comments, the kernel developers for useful feedback, and Julia Lawall for her revision suggestions. This work was supported by Smart Grid-National Science and Technology Major Project with No.2025ZD0808500 and National Natural Science Foundation of China under Grant No.62572021.

Ethical Considerations

Stakeholders. The stakeholders of our work may include the Linux and FreeBSD kernel communities, commercial companies, end users of Linux or FreeBSD, and our research team.

Impacts. We perform offline static analysis on public Linux and FreeBSD kernel source code and check the produced bug reports, without interacting with live systems, running test cases, publishing PoCs or generating exploitation code. For the kernel communities, we reported the found bugs to kernel developers by strictly following the reporting and disclosure processes of Linux and FreeBSD, and we also received much kind guidance from kernel developers. We never contacted commercial companies and end users about our approach and the found bugs, and they can update the kernel versions from the official websites of Linux and FreeBSD to fix the bugs found by us. We never exploit the found bugs to attack real-world systems. Our goal is to help the kernel communities find real bugs and fix them, which can improve kernel security. In fact, our approach is based on static analysis, without producing any PoC or exploitation code of the found bugs. The attacker may study our bug reports and the related bug-fix patches to manually construct the related PoCs and exploitation code to attack the operating system. This problem is also applicable to all bug reports and bug-fix patches, and it can be solved by patch release and kernel update to some degree. For these reasons, we believe that tangible harms are minimal and the likelihood of them being realized is low, and the intangible harms are minimal and controlled. For our research team, we follow the ethical principles in all the steps and face no harm.

Mitigations. We have reported the found bugs to developers, without publishing PoCs or generating exploitation code. We strictly follow the reporting and disclosure processes of Linux and FreeBSD, and help the kernel developers to fix these bugs.

Decision. While some bugs remain unfixed, since the kernel developers have acknowledged the found bugs and we actively assist in bug-fix patching, the potential harm should be minimal. Our work detects many harmful TOCTOU bugs caused by kernel races. This result indicates that such bugs are quite a few. Thus, the kernel developers might pay more attention to detecting and fixing such bugs, after reading our paper. Consequently, publishing this work reveals the importance of detecting and fixing such bugs, which is important to kernel security. In conclusion, we believe the ethical benefits of our work are greater than the harms. Thus, we decided to submit this paper and expect to publish our work.

Open Science

The artifact of KERAT is available on `https://doi.org/10.5281/zenodo.17898451`, including its source code, executable binary and usage instructions. Therefore, we believe this paper has good compliance with the open science policy.

References

[1] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. Characterizing, exploiting, and detecting DMA code injection vulnerabilities in the presence of an IOMMU. In *Proceedings of the 16th European Conference on Computer Systems*, pages 395–409, 2021.

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

[3] Fenhua Bai, Zikang Wang, Kai Zeng, Chi Zhang, Tao Shen, Xiaohui Zhang, and Bei Gong. ZKSA: secure mutual attestation against TOCTOU zero-knowledge proof based for iot devices. *Computers & Security*, 148:104136, 2025.

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

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

[6] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. Static detection of unsafe DMA accesses in device drivers. In

Proceedings of the 30th USENIX Security Symposium, pages 1629–1645, 2021.

[7] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: static driver verification with under 4% false alarms. In *Proceedings of the 2010 International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 35–42, 2010.

[8] Liheng Chen, Zheming Li, Zheyu Ma, Yuan Li, Baojian Chen, and Chao Zhang. EnclaveFuzz: finding vulnerabilities in SGX applications. In *Proceedings of the 31st Network and Distributed System Security Symposium (NDSS)*, 2024.

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

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

[11] Coverity: a commercial static analysis tool, 2021. https://scan.coverity.com/.

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

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

[14] CVE-2020-25212: TOCTOU mismatch in the NFS client code. https://nvd.nist.gov/vuln/detail/CVE-2020-25212.

[15] CVE-2021-29657: TOCTOU race condition in the KVM module. https://nvd.nist.gov/vuln/detail/CVE-2021-29657.

[16] CVE-2024-43882: TOCTOU between perm check and set-uid/gid usage. https://nvd.nist.gov/vuln/detail/CVE-2024-43882.

[17] CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition. https://cwe.mitre.org/data/definitions/367.html.

[18] Victor Duta, Mitchel Josephus Aloserij, and Cristiano Giuffrida. SafeFetch: practical double-fetch protection with kernel-fetch caching. In *Proceedings of the 33rd USENIX Security Symposium*, pages 1207–1224, 2024.

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

[20] dm-ioctl: avoid possible double-fetch issue of version. https://github.com/torvalds/linux/commit/249bed821b4d.

[21] nvme: avoid possible double-fetch issue when handling CQE. https://github.com/torvalds/linux/commit/62df80165d7f.

[22] scsi: ufs: core: fix a race condition related to the device commands. https://github.com/torvalds/linux/commit/20b97acc4caf.

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

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

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

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

[27] Anum Khurshid and Shahid Raza. AutoCert: automated TOCTOU-secure digital certification for iot with combined authentication and assurance. *computers & security*, 124:102952, 2023.

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

[29] Kyung-Suk Lhee and Steve J Chapin. Detection of file-based race conditions. *International Journal of Information Security*, 4(1):105–119, 2005.

[30] Tuo Li. LR-Miner artifact. https://sites.google.com/view/LR-Miner/.

[31] Tuo Li, Jia-Ju Bai, Gui-Dong Han, and Shi-Min Hu. LR-Miner: static race detection in OS kernels by mining locking rules. In *Proceedings of the 33rd USENIX Security Symposium*, pages 6149–6166, 2024.

[32] 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.

[33] Tuo Li, Jia-ju Bai, Yulei Sui, and Shi-min Hu. SPATA: effective OS bug detection with summary-based, alias-aware, and path-sensitive typestate analysis. *ACM Transactions on Computer Systems (TOCS)*, 42(3-4):1–40, 2024.

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

[35] Kai Lu, Peng-Fei Wang, Gen Li, and Xu Zhou. Untrusted hardware causes double-fetch problems in the I/O memory. *Journal of Computer Science and Technology (JCST)*, 33(3):587–602, 2018.

[36] 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 the 21st International Symposium on Operating Systems Principles (SOSP)*, pages 103–116, 2007.

[37] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. *ACM SIGOPS Operating Systems Review*, 40(5):37–48, 2006.

[38] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 249–262, 2016.

[39] Mathias Payer and Thomas R Gross. Protecting applications against TOCTTOU races by user-space caching of file metadata. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 215–226, 2012.

[40] Hany Ragab, Andrea Mambretti, Anil Kurmus, and Cristiano Giuffrida. GhostRace: exploiting and mitigating speculative race conditions. In *Proceedings of the 33rd USENIX Security Symposium*, pages 6185–6202, 2024.

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

[42] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated detection, exploitation, and elimination of double-fetch bugs using modern CPU features. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 587–600, 2018.

[43] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. Pinpoint: fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 693–706, 2018.

[44] Smatch: a static bug-finding tool for C, 2021. http://smatch.sourceforge.net/.

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

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

[47] Pengfei Wang, Kai Lu, Gen Li, and Xu Zhou. A survey of the double-fetch vulnerabilities. *Concurrency and Computation: Practice and Experience*, 30(6):e4345, 2018.

[48] Wenwen Wang. LRSan source code. https://github.com/kengiter/lrsan.

[49] Wenwen Wang. MLEE: effective detection of memory leaks on early-exit paths in OS kernels. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, pages 31–45, 2021.

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

[51] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-Style file systems: an anatomical study. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*, pages 155–167, 2005.

[52] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: exploiting synchronisation bugs in intel SGX enclaves. In *Proceedings of the 21st European Symposium on Research in Computer Security*, pages 440–457, 2016.

[53] Jianhao Xu, Luca Di Bartolomeo, Flavio Toffalini, Bing Mao, and Mathias Payer. WarpAttack: bypassing CFI through compiler-introduced double-fetches. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy*, pages 1271–1288, 2023.

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

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

[56] Chengfeng Ye, Yuandao Cai, and Charles Zhang. When threads meet interrupts: effective static detection of interrupt-based deadlocks in Linux. In *Proceedings of the 33rd USENIX Security Symposium*, pages 6167–6184, 2024.

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

[58] Yizhuo Zhai, Yu Hao, Hang Zhang, Daimeng Wang, Chengyu Song, Zhiyun Qian, Mohsen Lesani, Srikanth V Krishnamurthy, and Paul Yu. UBITect: a precise and scalable method to detect Use-before-Initialization bugs in Linux kernel. In *Proceedings of the 28th International Symposium on the Foundations of Software Engineering (FSE)*, pages 221–232, 2020.

[59] Hang Zhang, Jangha Kim, Chuhong Yuan, Zhiyun Qian, and Taesoo Kim. Statically discover cross-entry use-after-free vulnerabilities in the Linux kernel. In *Proceedings of the 32nd Network and Distributed System Security Symposium (NDSS)*, 2025.

[60] Yung-Yu Zhuang and Yao-Nang Tseng. A novel detection method for the security vulnerability of Time-of-Check to Time-of-Use. *Journal of Information Science & Engineering*, 38(6), 2022.