Place Your Locks Well: Understanding and Detecting Lock Misuse Bugs

Yuandao Cai Peisen Yao Chengfeng Ye Charles Zhang
The Hong Kong University of Science and Technology
{*ycaibb, pyao, cyeaa, charlesz*}*@cse.ust.hk*

Abstract

Modern multi-threaded software systems commonly leverage locks to prevent concurrency bugs. Nevertheless, due to the complexity of writing the correct concurrent code, using locks itself is often error-prone. In this work, we investigate a general variety of lock misuses. Our characteristic study of existing CVE IDs reveals that lock misuses can inflict concurrency errors and even severe security issues, such as denial-of-service and memory corruption. To alleviate the threats, we present a practical static analysis framework, namely LOCKPICK, which consists of two core stages to effectively detect misused locks. More specifically, LOCKPICK first conducts path-sensitive typestate analysis, tracking lock-state transitions and interactions to identify sequential typestate violations. Guided by the preceding results, LOCKPICK then performs concurrency-aware detection to pinpoint various lock misuse errors, effectively reasoning about the thread interleavings of interest. The results are encouraging — we have used LOCKPICK to uncover 203 unique and confirmed lock misuses across a broad spectrum of impactful open-source systems, such as OpenSSL, the Linux kernel, PostgreSQL, MariaDB, FFmpeg, Apache HTTPd, and FreeBSD. Three exciting results are that those confirmed lock misuses are long-latent, hiding for 7.4 years on average; in total, 16 CVE IDs have been assigned for the severe errors uncovered; and LOCKPICK can flag many real bugs missed by the previous tools with significantly fewer false positives.

1 Introduction

As concurrency becomes a popular programming model for enhancing the performance of modern software, it is also the culprit of many subtle errors stemming from complex memory interactions between threads [31, 51]. Even worse, there is abundant evidence that concurrency attacks are both practical and harmful [18, 100, 103], resulting in cataclysmic disasters [41, 92]. To mitigate the threats, a plethora of research has focused on combating data race bugs by detecting shared memory accesses without the protection of identical locks [4, 7, 9, 29, 97, 98]. Unfortunately, less attention has been paid to using locks (APIs) itself, which, as we have noted, is often undisciplined (e.g., missing releases) and, consequently, harbors reliability and even security problems.

The error-prone nature of lock misuses stems from one primary factor: writing high-performance and secure synchronization code is extremely challenging, as witnessed by a long stream of literature [26, 30, 31, 104]. In particular, the developers require intricate *non-modular* reasoning regarding the concurrent semantics of the programs as well as *modular* reasoning about the critical regions protected by lock and unlock invocations.

To demystify common lock misuses and understand their security impacts, we performed a characteristic study on existing CVE IDs over the past decade and made several interesting findings. First, our investigation reveals a general variety of errors, as shown in Table 1, for which at least 32 CVE IDs are assigned. These bugs characterize misuses through initializing and using locks, considering thread interactions. Second, we inspect and present the characteristics of these errors, facilitating bug understanding and detection, as explained later. Third, lock misuses can incur critical security problems, such as denial-of-service, memory corruption, and privilege escalation. Besides, lock misuses are often related to other security flaws, such as data race bugs [59, 61, 63], resulting in many exploitable vulnerabilities. For example, by exploiting race conditions [59, 65], vulnerabilities CVE-2020-10573 and CVE-2014-9748 can cause not only a denial of service but also other unspecified consequences. More details of our study are presented in § 2. Therefore, it is becoming increasingly urgent to pinpoint lock misuses.

Unfortunately, detecting lock misuses, in general, inherits the significant innate challenge of analyzing concurrent programs, which need to account for explosive

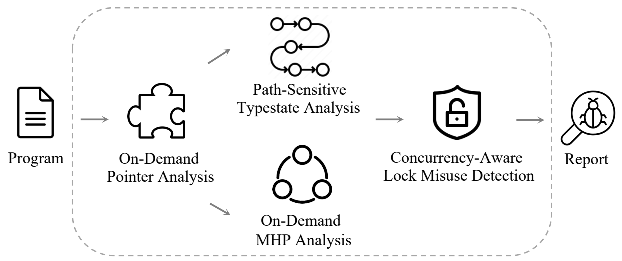
Table 1: Definitions on five lock misuses.

No.	Misuse Pattern	Bug Description	Concurrency
①	Missing lock releases	A lock is not released after its effective lifetime.	
②	Double locking	A lock is acquired twice.	
③	Using uninitialized locks	A lock is not initialized before using it. A concurrency error occurs when the lock is initialized non-deterministically.	✓
④	Releasing unacquired locks	A lock is released without acquiring it first. A concurrency error occurs when there is another thread holding the lock.	✓
⑤	Cyclic lock acquisitions	Different locks are not acquired in the same order. A concurrency error occurs when each thread in a set waits for the other to release a lock.	✓

thread interleavings together with complicated sequential reasoning. For instance, given T threads with S statements each executed concurrently, the control-flow execution within a thread and thread interleavings [16, 25, 83] can generate $O(TS)$ and $O((TS)^2)$ edges, respectively. The previous data-flow analysis for concurrent programs [7, 10, 22, 39, 83, 84] is integrated by alternating between reasoning over intra-thread and inter-thread semantics for correctness (e.g., to avoid missing concurrency errors). However, on the downside, we note that these previous approaches may reason about thread interleavings irrelevant to lock misuse detection and, thus, become inefficient for large-scale software systems.

In this paper, we present LOCKPICK, a static analysis framework for effectively uncovering both the sequential and concurrent lock misuses listed in Table 1. Our key insight is that sequential lock typestate violations can help reduce unrelated concurrency reasoning. For example, a lock is initialized non-deterministically ③, which can be captured by tracking the lock's state (e.g., the lock is uninitialized after creating a child thread using the lock). More specifically, unlike the previous approaches that carry out the integrated concurrent data-flow analysis [7, 22, 39, 83, 84], we can separate the sequential typestate analysis from the concurrency reasoning. As a consequence, LOCKPICK is guided to efficiently identify the may-happen-in-parallel relations for a few statements with interesting lock typestates. Furthermore, owing to the reduction in computation of thread interleavings, we can employ a path-sensitive sequential data-flow analysis to track lock typestates, which achieves high effectiveness compared to the previous FSM-based typestate analysis (e.g., lacking alias information [40], partial analysis of a single translation unit [11], path-insensitive ones [85], pattern-matching ones [1]).

Figure 1 depicts the workflow of LOCKPICK, which consists of two key collaborative stages and synergizes multiple techniques (i.e., pointer analysis, typestate analysis, and may-happen-in-parallel (MHP) analysis). First, LOCKPICK performs a domain-specific typestate analysis to track lock typestates at associated program locations *path-sensitively*. Second, guided by the discovered typestate violations on related statements, LOCKPICK employs concurrency-aware lock misuse detection by lazily querying MHP relationships among statements, reducing costly and unnecessary concurrency reasoning.


Program
On-Demand Pointer Analysis
Path-Sensitive Typestate Analysis
On-Demand MHP Analysis
Concurrency-Aware Lock Misuse Detection
Report
Figure 1: LOCKPICK framework design.

Result highlights. We thoroughly evaluated LOCKPICK across a broad gamut of open-source, popular, and well-checked C/C++ systems (e.g., OpenSSL, MariaDB, Curl). We highlight five inspiring results as follows:

1. *Numerous confirmed bugs*: By using LOCKPICK, we have discovered 203 confirmed lock misuses, among which 184 bugs have been fixed by the software developers or our patches.
2. *Many Vulnerabilities*: Many confirmed bugs have severe security impacts, resulting in system hangs, memory corruption, and program crashes. Specifically, 12 vulnerabilities in PJSIP are assigned a unified CVE ID and marked by the developers as having high severity. Moreover, 16 CVE IDs had been assigned for some errors at the time of writing, accounting for approximately *one-third* of the total lock-misuse CVE IDs over the past decade.
3. *Considerable long-latent bugs*: The 203 confirmed errors are difficult to uncover, hiding for an average of 7.4 years in the corresponding software.
4. *Diverse bugs*: At least ten bugs are detected and confirmed for each type of lock misuses in Table 1. In total, 30 of the 203 lock misuse errors are related to thread interactions, incurring concurrency errors.
5. *Usability*: When compared to the three previous tools SVF [86], CSA [11], and L2D2 [40], LOCKPICK can analyze millions of lines of code in a rea-

sonable amount of time and detect more types of lock misuses with the lowest false positive rate.

The list of confirmed bugs is displayed online [49]. All code examples used in this paper are from the bugs discovered by LOCKPICK, which have been safely fixed. This paper makes the following contributions:

- A taxonomy of lock misuses with findings about their bug characteristics and security impacts.
- A practical static detection framework, LOCKPICK, for both sequential and concurrent lock misuses.
- A thorough evaluation of LOCKPICK's effectiveness compared to the state-of-the-art approaches.

2 A Characteristic Study

We first present the basic study background (§ 2.1). Then, based on a study of existing CVE IDs, we elaborate on a common variety of lock misuses, investigating their characteristics (§ 2.2) and security impacts (§ 2.3).

2.1 Background

We first describe the problem scopes, study motivation, dataset collection, and classification criteria.

Lock discipline violations and problem scopes. Locks are common synchronization primitives to prevent race conditions in modern multithreaded software. Importantly, locks have explicit disciplines for initialization, use, and destruction. Systems that violate these locking disciplines may exhibit unexpected behaviors and potentially exploitable states. Our work investigates lock discipline violations from a less-explored but critical aspect, i.e., *using locks (APIs) itself in an undisciplined way*. In contrast, most previous research [7, 9, 29, 97, 98] focuses on identifying concurrent accesses to shared memory locations without being protected by an identical lock. We believe these two aspects are orthogonal and complementary to ensure software reliability and security.

The lock disciplines in systems written in different languages may differ slightly. For example, locks like `synchronized` keywords in Java are coupled and cannot incur missing lock releases. In contrast, locks like Pthread APIs in C code are decoupled and more prone to errors [33, 34, 38], because the locks are not necessarily released in the reverse order in which they were acquired. We focus on the C and C++ languages, which are widely-adopted in low-level systems and contain abundant decoupled locks, such as Pthread APIs and spinlocks. To demystify common lock misuses and understand their security impacts, we are motivated to conduct an empirical characteristic study on existing CVE IDs.

Dataset collection and classification criteria. We searched in the CVE database with the keywords "mutex" and "lock", and examined the vulnerabilities from 2010 to 2021. By focusing on the misuses of lock APIs, initially, we found 38 related CVE IDs. Without losing generality, we excluded three vulnerabilities strongly related to the ad-hoc knowledge of systems. For instance, CVE-2017-8071 existed in the old Linux kernel because a spinlock (which cannot sleep) was used without considering that sleeping was possible in a USB HID request callback. It is difficult to detect this vulnerability automatically without such domain knowledge. Besides, we excluded three vulnerabilities caused by incorrect interactions between locks and other primitives (e.g., sockets and events). For example, CVE-2015-8767 was induced by the improper management between a lock and a socket. As a result, we were left with 32 CVEs to study [13], identifying five general locking discipline violations under both sequential and concurrent circumstances. The five categories are shown in Table 1.

```
373 static int open_console (UI *ui){
375    if (!CRYPTO_THREAD_write_lock(ui->lock))
376       return 0;
483 }
552 static int close_console (UI *ui){
560    if (status != SS$_NORMAL) {
561       ERR_raise_data(...,status);
563       return 0;
564    }
566    CRYPTO_THREAD_unlock(ui->lock);
368    return 1;
369 }
```

Figure 2: A missing lock release in OpenSSL.

```
 82 CEN64_THREAD_RETURN_TYPE gdb_thread(...) {
 85    pthread_mutex_lock(&gdb->client_mutex);
 88    if (gdb->flags & GDB_FLAGS_INITIAL) {
 89       pthread_cond_wait(..., &gdb>c_mutex);
 90    } else {
 91       pthread_mutex_lock(&gdb->client_mutex);
 92    }
 97    pthread_mutex_unlock(&gdb->client_mutex);
143 }
```

Figure 3: A double locking in Cen64.

2.2 A Taxonomy of Lock Misuses

Next, we illustrate and characterize the lock misuses.

Bug characterization. First, Figure 2 shows ①, a missing release of the lock `ui->lock` at Line 563. Note that the lock `ui->lock` does not need to be released at Line 376, because the path condition of reaching Line 376 indicates that the lock is not acquired. Thus, path-sensitive analysis that characterizes path conditions is essential to develop a highly precise lock-misuse detector.

Second, Figure 3 illustrates ②, double acquisitions of the lock `client_mutex` at Line 85 and Line 91. Some missing lock releases can lead to double locking, but they are conceptually different. Specifically, the bug in Fig-

```
330 ret_t cherokee_collector_rrd_new (...){
373    re = pthread_create (..., worker_func, n);
375    ...
379    re = pthread_mutex_init (&n->mutex, NULL);
380    if (re != 0) {
382       return ret_error;
383    }
389 }
```

Figure 4: Using uninitialized locks in Cherokee.

ure 3 is not induced by missing lock releases. Notably, once a lock is acquired, no matter whether the lock is not released ① or acquired twice ②, other threads holding the same lock should wait without interacting. As a result, both errors ① and ② occur sequentially.

Third, using uninitialized locks ③ can occur when locks are entirely not initialized or are initialized non-deterministically. The former is self-explanatory, while the latter can result in concurrency errors. Figure 4 shows that the lock `&n->mutex` is initialized after the child thread is created at Line 373. A concurrency error occurs when the child thread uses lock `&n->mutex` before the lock is initialized by the parent thread. From a typestate view [15], the state of `&n->mutex` is uninitialized at Line 373, indicating hints to find the concurrent bug.

Fourth, in Figure 5, when Line 467 is executed, the unheld lock `trace_lock` is erroneously released at Line 516 (④). The issue can further trigger a concurrency error if a thread reaching Line 516 attempts to release the lock `trace_lock` that is being held by other threads. From a typestate angle, the lock `trace_lock` is not acquired at Line 516 when `category < 0` satisfies, demonstrating the importance of being path-sensitive.

Fifth, acquisitions of different locks among concurrent threads should follow the same order, because cyclic ones ⑤ can become multi-threaded deadlocks. It is worth noting that, while cyclic lock acquisitions within a thread cannot result in deadlocks, such a situation can still be considered a lock discipline violation, as argued in some literature [71, 72], which can easily introduce deadlocks through software evolution. Figure 6 shows cyclic lock acquisitions in a method executed by two concurrent threads. From a typestate aspect, (i) lock `ctxt->mutex` is acquired by a thread at Line 385, waiting for lock `entry->mutex` to be released at Line 431; (ii) lock `entry->mutex` is acquired by another thread at Line 431, waiting for lock `ctxt->mutex` to be released at Line 443; and (iii) neither can make progress.

Implication I. First, in contrary to bugs ① and ②, lock misuses ③-⑤ can cause concurrency errors under thread interactions. Second, it is important for a static lock misuse detector to resolve pointer aliasing of related variables, path-sensitively track lock typestates (e.g., acquired, released), and characterize thread interleavings.

```
459 BIO *OSSL_trace_begin(int category){
465    category = ossl_trace_get_category(category);
466    if (category < 0)
467       return NULL;
473    if (!CRYPTO_THREAD_write_lock(trace_lock))
474       return NULL;
491 }
493 void OSSL_trace_end(int category, BIO * channel){
498    category = ossl_trace_get_category(category);
516    CRYPTO_THREAD_unlock(trace_lock);
519 }
```

Figure 5: Releasing unacquired locks in OpenSSL.

```
381 static void *extract_worker_thread_func(...){
385    pthread_mutex_lock(ctxt->mutex);
431    pthread_mutex_lock(&entry->mutex);
433    pthread_mutex_unlock(ctxt->mutex);
442    if (chunk.type == XB_CHUNK_TYPE_EOF) {
443       pthread_mutex_lock(ctxt>mutex);
444       pthread_mutex_unlock(&entry->mutex);
445       my_hash_delete(ctxt->filehash,...);
446       pthread_mutex_unlock(ctxt->mutex);
470    }
478 }
```

Figure 6: Cyclic lock acquisition orders in MariaDB.

2.3 Security Impacts of Lock Misuses

Many lock misuses could wreak severe havoc on security based on their program contexts. Next, we illustrate how these lock misuses can turn into security issues.

Security issues arising from misused locks. We begin by delving into security issues by utilizing misused locks. According to our manual inspections, the most common security impacts of lock misuses are *denial-of-service* and *memory corruption*. Concurrent cyclic lock acquisitions [56, 58] and double locking [64, 69], in particular, can cause a system to hang with deadlocks. Besides, missing lock releases [55, 62, 66], as a special kind of memory leak, can crash the entire system due to memory exhaustion. In addition, releasing unacquired locks and using uninitialized locks are the main culprits of rendering memory corruption and system crashes [57, 60, 61, 65, 67].

Moreover, we find that lock misuses could even result in *privilege escalation* and other unidentified issues (claimed by maintainers). For CVE-2010-4210, releasing an unheld lock in FreeBSD allowed local attackers to gain elevated privileges, overwrite arbitrary memory locations, and execute arbitrary code. According to CVE-2014-9748 in libuv, a thread attempted to release a lock held by other threads, possibly having other unspecified impacts by leveraging concurrency bugs.

Other security bugs arising from misused locks. It is surprising to find that lock misuses are often related to other security bugs, such as atomicity violations [65], use-after-free [63], and double free [61], indirectly leading to many exploitable security issues.

Figure 7 shows another example due to a program-

```
98  int search_makelist(search_t *results,...){
145     pthread_mutex_unlock(&conn->lock);
146     int tmp = conn_setup(conn);
147     pthread_mutex_unlock(&conn->lock);
203 }
```

Figure 7: Releasing unacquired locks in Axel.

ming typo: the unlock statement at Line 145 should be a lock statement. As a result, both unlock statements at Lines 145 and 147 attempt to release an unheld lock `&conn->lock`. Even worse, since the lock `&conn->lock` is not acquired, an atomicity violation occurs within the thread-unsafe method `conn_setup`. Similarly, many prior lock-misuse vulnerabilities lead to concurrency bugs, such as CVE-2014-9748 in Libuv, CVE-2014-8131 in libvirt, and CVE-2020-10573 in Janus.

Implication II. Such common and severe security issues make detecting lock misuses increasingly urgent. For utility, two requirements should be satisfied. First, a detector should be *efficient* for millions of lines of code (e.g., Linux Kernel). Second, the detector should be *precise*. Otherwise, it would disturb developers with excessive false reports and waste their precious time.

3 LOCKPICK in a Nutshell

This section first formulates the lock misuse problem (§ 3.1). We then outline the limitations of the previous work (§ 3.2) and the essence of LOCKPICK (§ 3.3).

3.1 Lock Misuses Formulation

Based on the implications presented in § 2, we define a lock-specific finite-state machine model to characterize the lock-state transitions and interactions.

Finite-state machine (FSM). Figure 8 shows the designed FSM. For simplicity, we utilize three methods, $init\_lock(v)$, $lock(v)$, and $unlock(v)$, respectively, to indicate the initialization, acquisition, and release of a lock object o referred to by a pointer variable v. In detail,

• $\Sigma = \{\mathbf{U}, \mathbf{I}, \mathbf{A}, \mathbf{E}, \mathbf{X}\}$ is a set of all possible states of a lock. More specifically, Uninitialized (**U**) is the beginning state, while Initialized (**I**), Acquired (**A**), Error(**E**), and Exit(**X**) are the final states.

• When $init\_lock(v)$, $lock(v)$, or $unlock(v)$ is invoked, the lock states would move according to the FSM. Specifically, following the locking disciplines, a lock o is initialized via $init\_lock(v)$, with its state transiting from **U** to **I**. Note that it is important to ensure that a lock is safely initialized before using it. Next, if the lock o is always released after being acquired until the threads exit, the state of the lock o would transit between states **I** and **A** correctly, finally resulting to the state **I**.

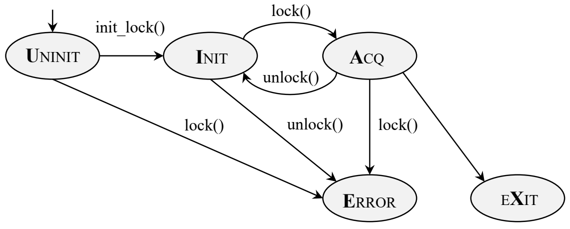


Figure 8: A finite-state machine for a single lock.

Lock misuse. However, based on the study in § 2, lock misuse errors ①-⑤ could happen when the disciplines mentioned before are violated. Formally, we characterize the lock misuses as follows:

• ① (Missing lock releases): at the exit points of a thread, the state of a lock o is acquired **A**, and then transits to state **X** because the lock is not correctly released.

• ② (Double locking): when a lock o is acquired again via $lock(v)$, the state of the lock o transits from **A** to **E**.

• ③ (Using uninitialized locks): a lock o is in the state **U**, and then transits to be **E** via using $lock(v)$. Notably, $init\_lock(v)$ may be concurrently executed by other threads, so the error ③ can occur non-deterministically.

• ④ (Releasing unacquired locks): a lock o is in the state **I** and incorrectly transits to **E** via using $unlock(v)$, indicating that the lock is released twice. When another concurrent thread tries to leverage the lock (e.g., executing a statement $lock(v)$), the typestate violation can cause a concurrency error, trying to release the lock that is being held by another concurrent thread.

• ⑤ (Cyclic lock acquisitions): identifying lock acquisitions should keep track of the lock-state interactions. To indicate that a thread acquires a lock o' while holding another lock o, we use the symbol $o \rightsquigarrow o'$. In the case of two threads, consider a statement $lock(v)$ (v points to o). If the state of another lock o' is acquired **A** at the statement $lock(v)$, a lock acquisition $o' \rightsquigarrow o$ is induced. Similarly, at a statement $lock(v')$ (v' points to o'), an acquisition $o \rightsquigarrow o'$ can be also induced, if the state of another lock o is acquired **A**. Due to the cyclic acquisition orders, a deadlock occurs when statements $lock(v')$ and $lock(v)$ can be run concurrently by two threads.

Notably, to reason about cyclic ones from all acquisition orders, it suffices to build a lock graph [93] with vertexes representing locks and edges denoting acquired relations between locks. A cycle in the graph represents cyclic acquisition orders, such as $o \rightsquigarrow o'$ and $o' \rightsquigarrow o$.

Remark. According to the FSM shown in Figure 8 and the implications in § 2, the lock misuses have three essential characteristics. First, misuses of a single lock occur when the lock's state transits to the wrong states **E** or **X**. Second, reasoning about cyclic lock acquisitions ⑤ should consider the typestate interactions on multiple locks. Third, as shown in § 2, identifying lock misuses ③-⑤ should take concurrency semantics into account by identifying the execution relations (e.g., being may-happen-in-parallel) between statements.

Example 3.1. We use $tcreate(t, f)$ and $tjoin(t)$ to denote the creation and destruction sites of a thread t. Figure 9 shows the control-flow graphs of two code snippets, where each graph node is denoted by the symbol n_i.

First, consider Figure 9(a), where the lock o_1 is uninitialized when creating a child thread t_1. Hence, a concurrency bug ③ can manifest when thread t_2 executes $lock(v_1)$ before thread t_1 executes $init\_lock(v)$. Besides, since $unlock(v_1)$ in t_1 can also be executed concurrently with $lock(v_1)$ in t_2, a concurrency error ④ can occur.

Second, consider Figure 9 (b). There are two cyclic lock acquisition orders (⑤) between o_1 and o_2 (pointed to by v_1 and v_2, respectively), which are illustrated by the lock graph shown in Figure 9 (c). However, the code is deadlock-free, because the statement $lock(v_1)$ in t_2 must happen before statement $lock(v_2)$ in thread t_1, which is enforced by the thread-destruction site $tjoin(t_1)$.

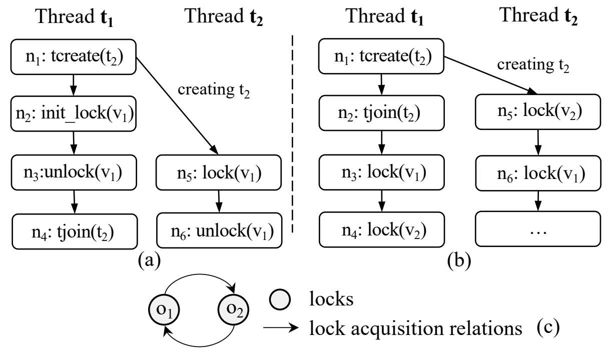


Figure 9: (a) and (b) show two code snippets. Irrelevant statements and function names are omitted. The variables v_1 and v_2 point to locks o_1 and o_2, respectively. (c) shows a lock graph to illustrate cyclic acquisition orders.

3.2 Goals and Challenges

We target the static detection of the sequential and concurrent lock misuses in Table 1, which is relatively less explored than data race detection [5,20,33,34,44,74,90]. Lock misuses, as shown in § 2, can become concurrency errors and security issues. However, precisely analyzing large concurrent codebases is stunningly challenging:

- First, as illustrated in § 2 and also witnessed by the previous work [14,19,81,96,99], path-sensitivity is critical to embracing high precision. However, being path-sensitive may result in unaffordable performance penalties caused by explosive program paths.
- Second, analyzing concurrent programs should characterize the possible thread interleavings for correctness [7, 10, 84], since ignoring the interleavings can result in missing concurrency bugs. However, it is prohibitively expensive to propagate data-flow facts across control flows induced by the exponentially possible thread interleavings [25].

Even worse, when path-sensitively analyzing concurrent programs, the scalability issue further deteriorates.

Limitations of the past work. One line of previous research focuses on *sequential typestate error detection* [37, 86, 88], most of which is path-insensitive and may result in excessive false positives. Besides, many sequential typestate analyses have other limitations (e.g., lacking pointer aliasing information [40], partial analysis of a single translation unit [11], and pattern-based analysis [1]), posing limitations of missing sequential bugs. More importantly, they do not characterize thread interleavings, thus missing the capability of finding concurrency-related lock misuses (§ 3.1).

On the other hand, existing *data-flow analysis for concurrent programs* [7, 10, 39, 83, 84] can be inefficient for lock misuse detection. Take the concurrency error ③ as an example. For each statement $init\_lock(v)$ in a thread, the previous approach needs to reason about whether other concurrent threads have been using the lock. Unfortunately, alternating reasoning over intra-thread and inter-thread semantics is notoriously expensive [22] due to the explosive thread interleavings. More importantly, the computation can be redundant because not all $init\_lock(v)$ is necessary for further inter-thread reasoning. For the example in Figure 4, we only need to reason about the lock `&n->mutex`, which is initialized after the creation of the child thread and could be used without initialization. Due to the redundancy in computation of lock misuse detection, we note that the previous approaches either suffer from prohibitive performance penalties [7, 83], or have to give up path-sensitivity, thereby sacrificing precision to achieve good scalability [10,39,84].

3.3 Our LOCKPICK Approach

To address the problems, we note that lock typestate violations can provide guidance to reason about thread interleavings of interest. Meanwhile, such violations can be captured without reasoning about concurrent semantics (illustrated in § 2). Thus, our key idea is to separate the sequential reasoning about typestate violations from the concurrent reasoning of misused locks. Such a separation has the following two salient advantages:

- Typestate violations are tracked along thread-local data flows. Note that the tracking is a one-time effort, thus reducing the notorious performance overhead caused by alternating between intra-thread and inter-thread reasoning [7,22,39,83,84].

- The typestate tracking produces a "slice" of the program states (e.g., the relevant statements, the related path conditions, and the interesting locks' states), which effectively guides the subsequent analysis in reasoning about thread interleavings of interest.

In this spirit, we present LOCKPICK, a novel static analysis framework with two collaborative stages:

- We propose a domain-dedicated typestate analysis to path-sensitively track the typestate transitions and interactions on locks subject to our FSM shown in Figure 8, by tracing sequential data flows.
- By using the typestate results as a guide, we identify various lock misuses by querying may-happen-in-parallel relations between related statements.

For example, suppose we need to detect the concurrency errors ⑤. First, we identify interesting typestate as hints, i.e., the cyclic lock acquisitions (e.g., $o \rightsquigarrow o'$ and $o' \rightsquigarrow o$). We then consider whether two locks (e.g., o and o') can be acquired by two concurrent threads. As a result, we only need to reason about the related thread interleavings between a few statements by foreseeingly and precisely identifying the lock typestates as a guide.

4 Algorithm Design

This section first describes the designed abstract domains (§ 4.1). We then elaborate our path-sensitive data-flow algorithm to track lock typestate transitions (§ 4.2). Finally, we present our demand-driven MHP analysis (§ 4.3) and how to detect the lock misuses by combining the computed data-flow facts and MHP queries (§ 4.4).

4.1 Preliminaries

Program abstraction. First, we introduce the basic notions to abstract concurrent programs.

Abstract thread. An abstract thread refers to a thread creation site, such as a call to `pthread_create()`. More specifically, we use $tcreate(t, f)$ to represent the creation of a thread t executing procedure f ($f \in \mathbb{F}$), and $tjoin(t)$ to denote the destruction site of the thread t. Abstract threads are modeled context-sensitively so that a thread, $t \in \mathbb{T}$, always refers to a context-sensitive fork site, i.e., a unique runtime thread. Following much previous work [7, 44], we create two different threads with identical attributes for a fork site allocated in a loop.

Inter-thread control-flow graph. A conventional control-flow graph can be tailored to an inter-thread control-flow graph $ICFG$ by accounting for thread creation sites. Following previous work [33, 34], thread creation edges are added from the creation sites to the entry nodes of functions executed by created threads, thereby characterizing the parent-child thread relationships. More specifically, $ICFG$ comprises CFG of each thread t, denoted as CFG_t. Each CFG_t is in a standard form, which is a directed graph denoted as $CFG_t = (N, E)$. The nodes N consist of statement nodes and the merge nodes of two predecessors. In addition, the directed edges E represent intra- or inter-procedural control flows between nodes.

$$\begin{array}{rlrl}
\text{Abstract thread} & t \in \mathbb{T} & \text{Statements} & s \in \mathbb{S} \\
\text{Memory objects} & o \in \mathbb{O} & \text{Program variables} & v \in \mathbb{V} \\
\text{Points-to results} & \mathbb{E} := \mathbb{V} \to 2^{\mathbb{O}} & & \\
\text{Lock typestates} & ty \in \Sigma := \{\mathbf{U}, \mathbf{I}, \mathbf{A}, \mathbf{E}, \mathbf{X}\} & & \\
\text{Data-flow facts} & d \in \mathbb{D} := 2^{\mathbb{O} \times \Sigma \times \Phi} & & \\
\text{MHP results} & \mathbb{M} := \mathbb{S} \to 2^{\mathbb{S}} & &
\end{array}$$

Figure 10: Abstract domain.

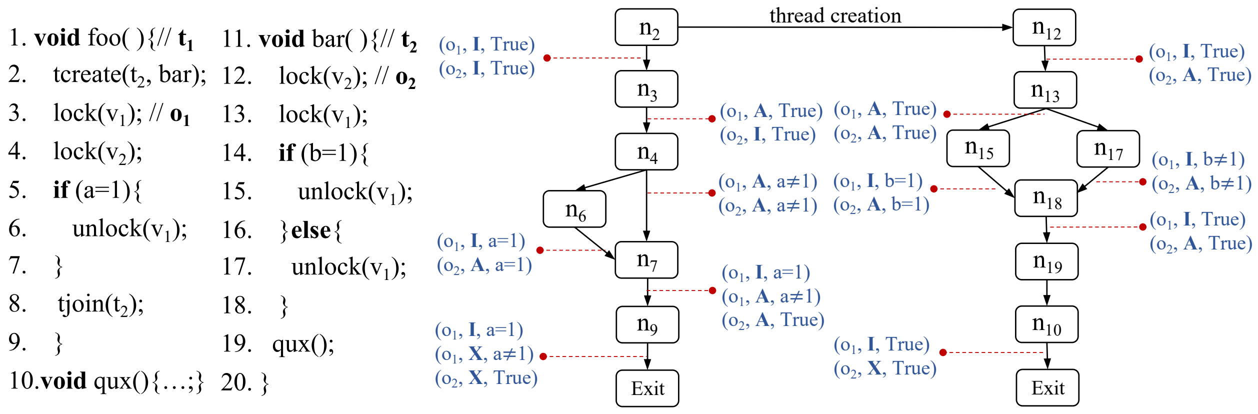
Abstract domain. Following the formulation of lock misuse in § 3, we present a dedicated abstract domain to precisely capture lock typestate transitions and interactions, and effectively reason about the execution relationships between related statements.

As shown in Figure 10, $v \in \mathbb{V}$ is a program variable, while $o \in \mathbb{O}$ is a memory object, such as a lock object. We use a demand-driven pointer analysis (§ 5), which can lazily resolve the points-to set $\mathbb{E}(v)$ of a given variable v. By using $\mathbb{E}$, the subsequent analysis can query the set of accessed lock objects at a related statement, e.g., $lock(v)$. At a high level, to effectively detect lock misuses, we keep track of the following facts for the specific control-flow nodes in CFG_t of each thread t:

• *Typestates* of lock objects: data-flow facts of a lock object are a set of tuples, which are denoted as $d \in \mathbb{D} := 2^{\mathbb{O} \times \Sigma \times \Phi}$. For example, the set $\{(o, \mathbf{A}, \varphi_1), (o, \mathbf{E}, \varphi_2)\}$ at the node of CFG_t indicates that the lock o has two different typestates $\mathbf{A}$ and $\mathbf{E}$ qualified by the path conditions φ_1 and φ_2, respectively.

• *May-happen-parallel (MHP) relations* between statements with specific lock typestates: we resolve the MHP relations $\mathbb{M}(s)$ between different statements using an on-demand MHP analysis [43]. For example, $\mathbb{M}(lock(v))$ denotes the set of statements that may be executed concurrently with the statement $lock(v)$.

Example 4.1. In Figure 9 (a), by tracking the typestates of the lock o_1 at the node n_1, the data-flow fact $(o_1, \mathbf{U}, True)$ is produced as the lock o_1 is uninitialized $\mathbf{U}$ at n_1. The data-flow fact at the node n_5 then becomes $(o_1, \mathbf{E}, True)$, because no initialization statements are reachable from the node n_5. After finding such typestate violations, we can use the MHP analysis to identify the statement $init\_lock(v)$ in thread t_1, i.e.,



```
void foo( ){// t1
    tcreate(t2, bar);
    lock(v1); // o1
    lock(v2);
    if (a=1){
      unlock(v1);
    }
    tjoin(t2);
}
void qux(){...;}
```

```
void bar( ){// t2
    lock(v2); // o2
    lock(v1);
    if (b=1){
      unlock(v1);
    }else{
      unlock(v1);
    }
    qux();
}
```

thread creation
n2
n3
n4
n6
n7
n9
Exit
n12
n13
n15
n17
n18
n19
n10
Exit
(o1, I, True) (o2, I, True)
(o1, A, True) (o2, I, True)
(o1, A, a≠1) (o2, A, a≠1)
(o1, I, a=1) (o2, A, a=1)
(o1, I, a=1) (o1, A, a≠1) (o2, A, True)
(o1, I, a=1) (o1, X, a≠1) (o2, X, True)
(o1, I, True) (o2, A, True)
(o1, A, True) (o2, A, True)
(o1, I, b=1) (o2, A, b=1)
(o1, I, b≠1) (o2, A, b≠1)
(o1, I, True) (o2, A, True)
(o1, I, True) (o2, X, True)

Figure 11: the buggy code and its ICFG with the computed data-flow facts. Irrelevant nodes in ICFG are removed.

$init\_lock(v) \in \mathbb{M}(lock(v_1))$. By doing so, we can discover a concurrency error ③ caused by the lock o_1 that is initialized non-deterministically.

The abstract domain has two salient advantages, respecting the bug characteristics discussed in § 2. First, the data-flow facts distinguish the locks' typestates along different paths, enabling precise typestate violation detection. Note that the path conditions can be solved lazily until the stage of bug detection. Second, using the typestate results as a guide, we can query MHP relations among statements with typestate violations, thus reducing redundant thread interleaving reasoning. In particular, we do not need an independent and prohibitive thread-escape analysis like in the prior work [7, 34, 44], because locks are commonly shared among threads.

Next, we deliberate on two specific challenges:

1. How to design efficient data-flow analysis to track the lock typestates (§ 4.2) and MHP analysis to identify MHP relations among statements (§ 4.3).
2. How to synergize the lock typestates and MHP results for effective lock misuse detection (§ 4.4).

4.2 Typestates Analysis for Locks

Prior to detecting lock misuses, we perform a path-sensitive data-flow analysis to track the typestates of locks, which facilitates the subsequent analyses. We orchestrate the key steps in our typestate algorithm, including the intra- and inter-procedural processes.

At a high level, Algorithm 1 traverses each CFG_t ($t \in \mathbb{T}$) and computes the data-flow facts (from the domain $2^{\mathbb{O}\times\Sigma\times\Phi}$) on the related CFG nodes. On each node, we iteratively update a map from the incoming data-flow fact to their corresponding outgoing fact, propagating the fact to the next nodes. Without losing generality, we use IN_n and OUT_n to denote the incoming and outgoing data-flow facts on a node n, respectively. Besides, we use $prev(n)$ and $cond(n)$ to denote the predecessors and branch condition of a node n, respectively.

Algorithm 1: Path-sensitive lock typestate analysis.

```
Procedure TypeStateAnalysis():
    $Threadlist := Threadlist \cup \{t_0\}$ // main thread
    $IN_{n_o} := \{(o, \mathbf{U}, True) | o \in \mathbb{O} \wedge o$ is a lock object$\}$;
    while $Threadlist \neq \emptyset$ do
        retrieve a thread $t$ from $Threadlist$;
        DataFlowAnalysisForThread ($CFG_t$);
Function DataFlowAnalysisForThread($CFG_t$):
    $Worklist \hookleftarrow$ add the entry node $n_0$ of $CFG_t$;
    while $Worklist \neq \emptyset$ do
        if $n$ is a thread creation $tcreate(t', f)$ then
            $Threadlist := Threadlist \cup \{t'\}$;
        else if $n$ is $init\_lock(v)$, $lock(v)$, $unlock(v)$ then
            $OUT_n := F_{stmt}(n, IN_n)$;
        else if $n$ is a merge node, $n_1, n_2 \in prev(n)$ then
            $OUT_n := F_{seletive}(n, IN_{n_1}, IN_{n_2})$;
        else if $n$ is a callsite $cs$ invoking method $f$ then
            label each $d$ with $(_{cs}$ and enter $f$;
        else if $n$ is a return node of method $f$ then
            label each $d$ with $)_{cs}$ and return to callers;
        propagate data-flow facts $OUT_n$ to next nodes;
```

Handling of locking statements. In detail, we first set the data-flow facts of all locks as uninitialized **U** at the entry node of $ICFG$. When passing a node n that can change the states of locks, i.e., $init\_lock(v)$, $lock(v)$, or $unlock(v)$, we generate the corresponding outgoing ones OUT_n based on the incoming data-flow facts IN_n by using the rules shown in Figure 12. Specifically,

• First, we enumerate each data-flow fact in IN_n and identify the states of each lock o ($o \in \mathbb{E}(v)$) by querying the points-to set information.

• Second, we update the state of lock o regarding

$$F_{stmt}(n, IN_n)\frac{\begin{array}{c} n \in \{init\_lock(v), lock(v), unlock(v)\} \\ \forall(o,ty,\varphi) \in IN_n, o \in \mathbb{E}(v) \\ \varphi_1 := cond(n) \end{array}}{OUT_n \cup \{F_{tran}(n,d)\}}$$

$$F_{tran}(init\_lock(v),(o,ty,\varphi)) = \begin{cases} (o,\mathbf{I},\varphi\wedge\varphi_1) & \texttt{if } ty=\mathbf{U} \\ (o,\mathbf{E},\varphi\wedge\varphi_1) & \texttt{if } ty=\mathbf{A} \\ (o,\mathbf{E},\varphi\wedge\varphi_1) & \texttt{if } ty=\mathbf{I}. \end{cases}$$

$$F_{tran}(lock(v),(o,ty,\varphi)) = \begin{cases} (o,\mathbf{E},\varphi\wedge\varphi_1) & \texttt{if } ty=\mathbf{U} \\ (o,\mathbf{E},\varphi\wedge\varphi_1) & \texttt{if } ty=\mathbf{A} \\ (o,\mathbf{A},\varphi\wedge\varphi_1) & \texttt{if } ty=\mathbf{I}. \end{cases}$$

$$F_{tran}(unlock(v),(o,ty,\varphi)) = \begin{cases} o,\mathbf{I},\varphi\wedge\varphi_1) & \texttt{if } ty=\mathbf{A} \\ (o,\mathbf{E},\varphi\wedge\varphi_1) & \texttt{if } ty=\mathbf{I}. \end{cases}$$

Figure 12: Rules for handling the statements in Alg. 1.

$$F_{selective}(IN_{n_1}, IN_{n_2})\frac{\begin{array}{c} \forall d_1 := (o,ty_1,\varphi_3) \in IN_{n_1} \\ \forall d_2 := (o,ty_2,\varphi_4) \in IN_{n_2} \end{array}}{\begin{cases} OUT_n \cup \{d_1,d_2\} \texttt{ if } ty_1 \neq ty_2 \\ OUT_n \cup \{F_{merge}(d_1,d_2)\} \texttt{ else} \end{cases}}$$

$$F_{merge}(d_1,d_2) = (o,ty_1,\varphi_3\vee\varphi_4) \hookleftarrow (o,ty_1,\varphi_3),(o,ty_2,\varphi_4).$$

Figure 13: Rules for handling merge nodes in Alg. 1.

the kinds of statements, following the FSM's transitions shown in Figure 8. For example, after analyzing $unlock(v)$, a lock o with state **A** becomes **I**. In addition, we encode the corresponding branch condition in the data-flow facts for high precision. When the lock's state changes in the branch, a branch condition is encoded. Specifically, we update the path conditions in the data-flow facts by taking the conjunction of the branch condition φ_1 and the previous path conditions φ, as $\varphi \wedge \varphi_1$.

Example 4.2. In Figure 11, for simplicity, we assume that all locks are initialized. When computing data-flow facts for $lock(v_1)$ at n_3, the state of lock o_1 becomes acquired **A**, i.e., $(o_1, \mathbf{A}, True)$. For $lock(v_2)$ at n_4, the state of lock o_2 becomes acquired **A**, i.e., $(o_2, \mathbf{A}, True)$. For $unlock(v_1)$ at n_6, the fact $(o_1, \mathbf{A}, True)$ becomes $(o_1, \mathbf{I}, a = 1)$, because lock o_1 is released at the branch.

Handling of merging nodes. We proceed to discuss how to handle the merge nodes precisely and efficiently. Consider two incoming data-flow facts at a merge node. On the one hand, indiscriminately merging them into one (e.g., using the join operation in conventional data-flow analysis) can result in precision loss. For example, in Figure 11, at node n_7, merging states of the lock o_1 coming from different branches yields $(o_1, \mathbf{A} \text{ or } \mathbf{I}, True)$, which cannot precisely figure out in which branches the lock o_1 is released or acquired. On the other hand, always distinguishing the facts from different branches without merging can gradually lead to explosive data-flow facts, as in disjunctive abstract domains [24, 77].

Instead, as shown in Figure 13, we merge the data-flow facts d_1 and d_2 selectively, when the facts of the same lock have the same typestate. Note that these data-flow facts are redundant, indicating the same state of an identical lock. Specifically, when merging two data-flow facts into one, we take the disjunction of the path conditions to qualify the incoming facts. Thus, our data-flow analysis is precise without indiscriminately joining and is efficient with the selective merging.

Example 4.3. Consider the merge node n_7 in Figure 11. The data-flow facts of o_1 at the two predecessors n_4 and n_6 are $(o_1, \mathbf{A}, a \neq 1)$ and $(o_1, \mathbf{I}, a = 1)$, respectively. We do not merge the two facts of o_1 because the lock typestates differ. In constrast, the data-flow facts of o_2 at n_4 and n_6 are $(o_2, \mathbf{A}, a \neq 1)$ and $(o_2, \mathbf{A}, a = 1)$, respectively. We merge the two facts to one data-flow fact $(o_2, \mathbf{A}, True)$ since the lock's state remains the same.

In our data-flow analysis, we handle loops, based on the practical observation that most double locking or unlocking errors manifest themselves by the second iteration. Thus, loops are unrolled into two iterations.

Inter-procedural analysis. CFL-reachability [76] is used to reach context-sensitive, which is orthogonal to our contributions. Thus, we briefly describe the process. In Algorithm 1, each data-flow fact is assigned a string to validate the calling contexts. When propagating a fact along a call edge at a call site cs, we append a left parenthesis $(_{cs}$ to the string. When propagating a fact back to a call site cs along a return edge, we append a right parenthesis $)_{cs}$ to the string. The data-flow fact propagation is valid (in terms of context sensitivity) if strings between calls and returns have matched parentheses.

4.3 Demand-Driven MHP Analysis

After computing the sequential data-flow facts, we need to infer the execution relations between statements of interest for the lock misuse detection (§ 4.4). To this end, we tailor the prior MHP analysis [43, 44], which uses a static happens-before graph (HBG) to characterize thread synchronizations. In particular, given a statement s, an MHP analysis computes the set of statements, $\mathbb{M}(s)$, that may be executed concurrently with s.

First, we use Figure 14 to illustrate the HBG construction. Intuitively, a directed edge in an HBG indicates the happens-before (HB) relation between statements. Thus, we can build the HBG as follows. For each thread creation statement $tcreate(t, f)$, we add an HB edge from the statement to the entry point of t. For each thread destruction statement $tjoin(t)$, we add an HB edge from

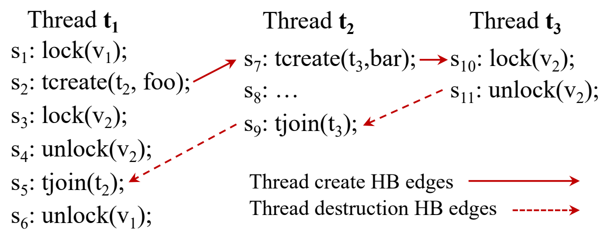


Figure 14: An happens-before graph for MHP relations.

the exit point of t to the statement. Besides, HB relations within a thread characterize their control-flow orders.

Based on the HBG, we can query the MHP relations between different statements. Specifically, if two vertexes (statements) in an HBG are not reachable, they share the MHP relation, and vice versa. For example, in Figure 14, statements s_1 and s_{11} are not MHP (s_1 happens before s_{11}), because they are reachable in the HBG. Also, statements s_4 and s_{11} are MHP because they are unreachable. As a result, given a statement s to collect $\mathbb{M}(s)$, it suffices to collect the set of statements, each of which is unreachable to s in the HBG. More technical details can be found in the previous work [44, 102].

Moreover, we tailor the approach [44] by removing unrelated statements from an HBG since, for lock misuse detection, it suffices to reason about the MHP relations for the related statements (i.e., $init\_lock(v)$, $lock(v)$, or $unlock(v)$). By using the simplified graph, we can avoid traversing irrelevant vertexes for computing MHP relations, thereby improving the performance.

Specifically, recall the implications in § 2. Armed with the HBG, we can combine typestate results and MHP relations for efficiently detecting lock misuses, reducing the unnecessary concurrency computation.

Example 4.4. In Figure 9 (b), the data-flow fact at n_4 is $(o_1, \mathbf{A}, True)$, denoting the lock o_2 is acquired at n_4 with holding lock o_1. Similarly, the data-flow fact at n_6 is $(o_2, \mathbf{A}, True)$. Thus, LOCKPICK identifies the acquisition orders $o_1 \rightsquigarrow o_2$ in t_1 and $o_2 \rightsquigarrow o_1$ in t_2, forming a cyclic order as shown in Figure 9 (c). Next, we query the MHP relations to identify that $lock(v_2)$ in thread t_1 cannot be MHP with $lock(v_1)$ in t_2, i.e., $lock(v_2) \notin \mathbb{M}(lock(v_1))$.

4.4 Lock Misuse Bug Detection

This section describes how to detect lock misuses within a thread (§ 4.4.1) and between threads (§ 4.4.2). Essentially, our basic idea is to examine data-flow facts on the related CFG nodes, identify the statements with typestate violations of interest, and query the MHP relations between the statements to discover concurrency bugs.

4.4.1 Detecting Lock Misuses within a Thread

First, as specified in § 2 and § 3.1, lock misuses ① and ② only occur thread-locally. As a result, we only need to examine sequential typestate violations by leveraging the results of the data-flow analysis (§ 4.2).

Missing lock releases ①. We examine each outgoing data-flow fact at the exit node n of CFG_t ($t \in \mathbb{T}$) to figure out the data-flow fact $(o, \mathbf{X}, \varphi)$ that transits from Acquired ($\mathbf{A}$) to Exit($\mathbf{X}$), disclosing a missing lock release. More importantly, we need to solve the corresponding path condition φ to achieve path sensitivity, which is similar to checking the following errors.

Double locking ②. For each statement $lock(v)$, we examine each outgoing data-flow fact of a lock o (pointed to by v) whose state is Error($\mathbf{E}$). Note that the state $\mathbf{E}$ of the lock o at the statement $lock(v)$ can indicate errors either ② or ③. To identify the double locking, it suffices to examine the incoming data-flow facts to examine the transition history of FSM from Acquired ($\mathbf{A}$) to $\mathbf{E}$.

Example 4.5. In Figure 11, we examine the data-flow facts at the exit nodes n_9 and n_{10} to detect the missing lock release ①. For instance, the data-flow fact $(o_2, \mathbf{X}, True)$ at n_9 indicates a lock-leak bug ①.

Overall, the lock misuses ① and ② are easier to detect without taking thread interactions into account.

4.4.2 Detecting Lock Misuses between Threads

Detecting lock misuses ③-⑤, on the other hand, should characterize thread interleavings between statements and memory interactions on lock objects. For instance,

- A lock can be initialized after creating a thread that may use the uninitialized lock;
- A thread may hold a lock that is released by another concurrent thread without acquiring the lock first;
- Concurrent threads can acquire different locks in an interdependent order, waiting forever for each other.

However, it is stunningly challenging to detect misused locks between concurrent threads. Our key idea is that the computed typestate violations allow us to avoid checking all possible thread interleavings; we only need to query MHP relations between the statements where the data-flow facts encode lock typestates of interest. The detection of errors ③-⑤ are detailed below.

Using uninitialized locks ③. For each statement $lock(v)$ at the node n of CFG_t, we examine the outgoing data-flow facts in OUT_n to identify a lock o, which (i) is pointed to by variable v and (ii) has an erroneous typestate $\mathbf{E}$ transiting from Uninitialized ($\mathbf{U}$). More specifically, there could be two causes: (i) the lock o is completely uninitialized; (ii) the lock o is initialized after creating a child thread t, causing a non-deterministic error.

To uncover the concurrency errors, we disclose a statement $init\_lock(v')$ such that (i) variable v' points to the lock o, and (ii) $init\_lock(v')$ can be MHP with $lock(v)$.

Thus, LOCKPICK identifies both causes of using uninitialized locks and generates the corresponding reports for developers to understand the errors under concurrency.

Releasing unacquired locks ④. For each statement $unlock(v)$, we examine each outgoing data-flow fact of lock o (pointed to by v) whose state is **E**. Either sequential or concurrent errors could be caused, depending on whether there could be another concurrent thread holding the same lock o. To discover the concurrency errors, we identify a statement $lock(v')$ so that (i) the variable v' points to the lock o, i.e., $o \in \mathbb{E}(v')$ and (ii) the statement is MHP with $unlock(v)$, i.e., $lock(v') \in \mathbb{M}(unlock(v))$.

Cyclic lock acquisitions ⑤. We identify typestate interactions on locks to capture each lock-acquisition order. Specifically, for $lock(v)$, we identify each acquisition order $o' \rightsquigarrow o$ between other locks o' (not pointed to by v) and the lock o (pointed to by v). To this end, we examine the data-flow facts of other locks o' whose state is **A**, i.e., $(o', \mathbf{A}, \varphi)$, to identify each lock-acquisition order $o' \rightsquigarrow o$.

Next, to derive the cyclic ones ⑤ from all lock acquisition orders, we construct a lock graph [8, 93], where a cycle indicates the cyclic lock acquisitions.

Finally, we query the MHP results to identify whether the cyclic acquisitions can occur among different concurrent threads. Take a common two-thread deadlock as an example (more than two threads are similar). We examine whether two statements $lock(v)$ and $lock(v')$ (where the two acquisitions, $o \rightsquigarrow o'$ and $o' \rightsquigarrow o$, are induced, respectively), can be run concurrently, i.e., $lock(v') \in \mathbb{M}(lock(v))$, thereby causing a concurrency error.

Example 4.6. In Figure 11, the data-flow fact at n_4 is $(o_1, \mathbf{A}, True)$, revealing an acquisition order $o_1 \rightsquigarrow o_2$. Also, the data-flow fact at n_{13} is $(o_2, \mathbf{A}, True)$, deriving an acquisition order $o_2 \rightsquigarrow o_1$. In a constructed lock graph, the acquisitions $o_1 \rightsquigarrow o_2$ and $o_2 \rightsquigarrow o_1$ form a cycle. Next, we query MHP relations to decide whether statements at n_4 and n_{13} can be MHP. If so, the cyclic acquisitions can cause a concurrent deadlock.

Summary. For effective lock misuse detection, LOCKPICK is armed with a synergetic combination of points-to facts, lock typestates, and MHP relations. Compared to the previous work [7, 22, 39, 83, 84], LOCKPICK has two noteworthy benefits. First, our path-sensitive typestate analysis is easier to scale up without initially considering thread interleavings. Second, LOCKPICK instead defers the thread interleaving reasoning until statements with lock typestate violations are discovered, enabling our MHP analysis to focus on the related statements.

5 Implementation

We have built LOCKPICK as a static bug-finding tool on top of the LLVM infrastructure and the Z3 SMT solver [101]. LOCKPICK can analyze many decoupled locks commonly used in C/C++ programs, such as Pthread APIs, spinlocks, and std::mutex. Next, we will delve into some interesting implementation details.

Pointer analysis. Our approach requires pointer aliasing information (e.g., identifying the points-to set of lock variables and thread pointers, and encoding path constraints). In practice, the locks could act as fields in structures, be passed in functions as parameters, and be reassigned at different program locations. To provide the precise points-to sets of pointers, we use the flow-, context-, and field-sensitive value-flow-based pointer analysis [81]. More specifically, we use may-information to identify lock acquisitions and must-information to identify lock releases [8, 33, 34].

Path conditions. The path conditions are encoded as first-order logic formulae over bit-vectors, which are solved by the Z3 SMT solver. A variable is modeled as a bit vector, of which the length is the bit width (e.g., 32) of the variable type (e.g., int). Note that our bit-vector constraints can handle many language constructs by utilizing pointer aliasing information (e.g., memory accesses, arithmetic computation, method calls) [3, 95].

Soundiness. Since our tool aims to detect bugs rather than perform rigorous verification, we make a few reasonably unsound choices (i.e., soundy [47]), following much work [7, 52, 81, 85]. To do so, we can thus overcome many of the inherent limitations of static analysis, ensuring our tool is both scalable and precise. Next, we summarize the unsound sources. First, loops are unrolled twice. However, we find that the loop handling is effective because the double locking or unlocking can still be captured by LOCKPICK. For example, we have found 53 real lock misuses in loops. Second, our pointer analysis does not handle inline assembly or pointer arithmetic. The analysis handles loops by analyzing two iterations. As stated in the work [2], "A C pointer alias analysis cannot be strictly sound, or else it would be very imprecise." To improve precision, we manually model some standard C libraries like memset and memcpy, which are important for the points-to analysis, but we have not yet modeled standard template libraries like std::vector and std::map. In § 6, we will investigate the effectiveness of these design choices by comprehensively evaluating precision, performance, and recall.

6 Evaluation

This section investigates two research questions:

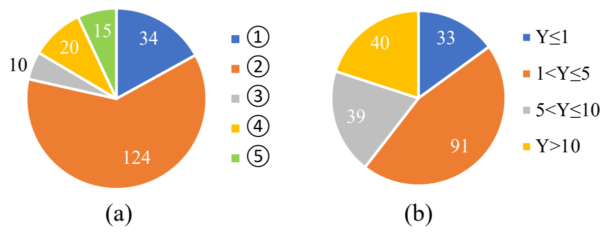


Figure 15: (a) and (b) show the distributions of bug type and hidden time (Year) for the 203 confirmed bugs, resp.

- **RQ1:** How effective and practical is LOCKPICK at uncovering lock misuses in mature open-source software systems (§ 6.1)?
- **RQ2**: How does LOCKPICK perform compared to the state-of-the-art tools (§ 6.2 and § 6.3)?

All the experiments were finished on a computer with two 20-core Intel(R) Xeon CPU@2.20GHz and 256GB physical memory running Ubuntu-16.04.

6.1 Effectiveness on Bug Hunting

From July 2021 to December 2021, we ran LOCKPICK extensively to uncover lock misuse bugs and vulnerabilities in many open-source systems. The programs we evaluated are from the benchmarks analyzed by much previous static and dynamic bug-finding work [7, 21, 28, 38, 44, 81, 94, 99]. Thus, these systems are regarded as prevalent, essential, and thoroughly examined.

Confirmed bugs. At the time of writing, we had used LOCKPICK to find 203 developer-confirmed lock misuses in over eighty software systems. Figure 15 shows the distribution of different lock misuses. We make two observations. First, all the varieties of lock misuses identified in our empirical study are prevalent, as LOCKPICK can uncover at least ten errors for each variety. Second, through our manual inspection, we find that 30 lock misuses of the 203 confirmed lock misuse bugs can result in concurrency errors. The links to bug issues and discussions are listed online [49].

Moreover, we emphasize that the found lock misuses are difficult to detect, as they have remained hidden for an average of 7.4 years. Moreover, more than 40 bugs were introduced more than a decade ago. For example, LOCKPICK detected a *21-year-old* bug in FreeBSD [23], as shown in Figure 16. Notably, many scanned programs (e.g., OpenSSL, MariaDB, and Apache HTTPd) have been regularly checked by some commercial static analyzers such as Coverity, Fortify, etc. None of these bugs were detected by those industrial tools, providing strong evidence of LOCKPICK's effectiveness.

In addition to actively reporting bugs and potential vulnerabilities, we also prepared patches for developers to ease their fixes. At the time of writing, 184 lock misuses had been fixed. The developers highly appreciated our bug-finding and fixing efforts with numerous comments like “nice catch!”, “excellent find!”, “thank you!”.

```
static CLIENT * clnt_com_create(...){
    mutex_unlock(&rpcsoc_lock);
    sport = pmap_getport(raddr, (u_long)prog,...);
    if (sport == 0) {
        goto err;
    }
    mutex_lock(&rpcsoc_lock);
err:
    mutex_unlock(&rpcsoc_lock);
}
```

Figure 16: A 21-year-old bug in FreeBSD.

```
629 static pj_status_t  codec_open(...){
638     pj_mutex_lock (opus_data->mutex);
769     if (err != OPUS_OK) {
771         return PJMEDIA_CODEC_EFAILED;
772     }
790     pj_mutex_unlock (opus_data->mutex);
791     return PJ_SUCCESS;
792 }
```

Figure 17: A lock misuse vulnerability in PJSIP.

Security impacts. We not only report bugs but also investigate their security impacts with the developers. We had received 16 CVE IDs at the time of writing. Next, we will conduct case studies on two vulnerabilities.

LOCKPICK detected 12 double locking bugs in various modules of PJSIP [68], a well-known multimedia communication library. Figure 17 illustrates one of the bugs induced by the missing lock releases at Line 771. When the method `codec_open` is invoked twice, the lock is double acquired. The vulnerabilities can cause system deadlocks and be exploited to launch a DoS attack, affecting all users of PJSIP. Unfortunately, these vulnerabilities are long-standing, so all versions up to and including 2.11.1 are affected. Initially, we opened an issue to report these bugs, which were quickly removed by the developers. They reminded us that these vulnerabilities were so severe that we should report them via encrypted emails [73]. Finally, these vulnerabilities across different modules are assigned a unified CVE ID [68].

We detected a deadlock of cyclic lock acquisitions spreading across seven versions of MariaDB [53] (one of the most popular open source relational databases), shown in Figure 6. Developers claim that the flaw can be exploited to deadlock the receiving process via two steps: (i) streaming malformed data to xbstream and (ii) using the same pathname multiple times in a stream with timing delays. They were extremely concerned about this concurrency vulnerability, so five developers gradually joined the conversation. Due to the effectiveness of LOCKPICK in vulnerability hunting, two developers were curious about our tool and asked, ”Which scanner?”

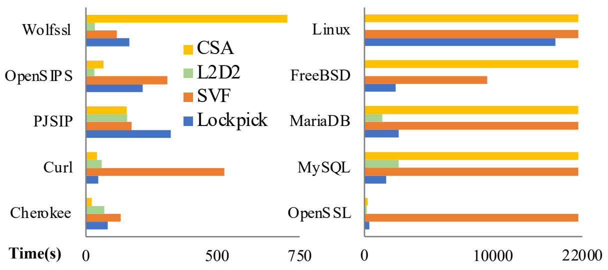


Figure 18: Time costs of LOCKPICK versus other tools.

Table 2: Results on bug hunting among tools.

Project	KLoC	SVF		L2D2		CSA		LOCKPICK	
		#FP	#R	#FP	#R	#FP	#R	#FP	#R
Cherokee	55	3	6	99%†	483	22	26	1	5
Curl	135	3	4	0	0	0	0	0	2
PJSIP	434	32	43	98%†	505	0	2	2	15
OpenSIPS	477	25	55	0	0	0	0	5	40
OpenSSL	490	66	68⋆	0	0	0	0	2	6
WolfSSL	944	16	20	3	3	0	1	3	11
MySQL	4,152	0	0⋆	100%†	1,157	95	99⋆	3	10
MariaDB	4,697	96%†	141⋆	100%†	4,993	100%†	229⋆	9	27
FreeBSD	8,457	66	81	NA	NA	0	0	12	31
Linux	15,987	88%†	328⋆	NA	NA	0	0	19	57
FPR	—	85.1%		99.2%		93.7%		27.5%	

#R denotes reports; ⋆ means the analyzer runs out of the time budget (six hours).
NA means the analyzer fails to compile the projects with its old compiler.
† means we compute FPR based on 100 reports among the excessive reports.

6.2 Comparison with Previous Tools

We compared LOCKPICK against three open-source, popular, and actively-maintained bug-finding tools.

- SVF [86] detects source-sink bugs by tracking value flows of objects along def-use chains. We revised its path-insensitive memory leak detector, SABER [85], to enable the sequential lock misuse detection of ① and ②.
- L2D2 [40] is a context- and path-insensitive tool that improves the industrial-strength bug finding tool INFER [27]. In addition, L2D2 handles pointer aliasing in analyzed programs based on syntactic expressions [6]. According to INFER developers, INFER only supports C++ programs while L2D2 extends INFER to detect errors ②, ④, and ⑤ in both C/C++ programs.
- CSA [11] is a symbolic executor with the limited function inlining [99] based on a single-file analysis. We used its Pthread-based API misuse checker (i.e., alpha.unix.PthreadLock) to detect errors ① and ②.

Benchmarks. To assess the time costs and precision, we chose ten popular and security-sensitive programs with a total of 35.8 MLoC, covering a wide spectrum of applications, such as operating systems, servers, databases, SSL/TLS libraries, and data transfer/multimedia communication tools. To assess the recall, we chose 26 vulnerabilities out of the 32 CVEs used in our study. Note that we excluded six vulnerabilities in the software written in other languages (e.g., Java, Rust).

Time cost. Figure 18 shows the comparison of time costs. We observe that SVF and CSA cannot finish analyzing large programs (e.g., Linux Kernel) in less than six hours. On the other hand, LOCKPICK's time costs are comparable to that of L2D2; LOCKPICK is slower than L2D2 only in six software projects. In conclusion, LOCKPICK can complete its analysis of the Linux Kernel (with approximately 16 MLoC) in 5.36 hours, which is efficient and practical.

Precision. Table 2 displays the results of bug reports. We manually validated whether each warning was genuine. However, we were unable to inspect more than 200 reports, so we selected 100 reports at random to review. To sum up, the FPR of SVF, L2D2, CSA, and LOCKPICK is approximately 85.1%, 99.2%, 93.7%, and 27.5%, respectively. The FPR results indicate that LOCKPICK is the most precise. The reasons for the high precision of LOCKPICK are discussed later. At the time of writing, 86 bugs in the ten systems have been confirmed by developers. In conclusion, compared to L2D2, SVF, and CSA, LOCKPICK can uncover various lock misuses in large-scale systems in a reasonable amount of time and with a low rate of false positives.

There are two representative false-positive cases reported by the other tools (i.e., SVF, L2D2, and CSA). First, in Figure 19, when ignoring the path conditions at Lines 195 and 325, the other tools reported a missing lock release at Line 326, which is spurious because the lock `ipt->start_lock` is not acquired under the condition `ipc.opt==IDLE_PROF_OPT_NONE`. Second, they reported a missing lock release at Line 558, which is spurious since the lock `&g_Hwlock` is not acquired when $ret \neq 0$. By virtue of its path-sensitive typestate analysis, LOCKPICK is resistant to these false positives.

Hunting known vulnerabilities. To examine the recall, we also assess whether LOCKPICK can detect the known 26 CVE vulnerabilities in C/C++ programs. According to our results, LOCKPICK can find all the CVEs in their corresponding historical versions. Overall, 153 unique reports were generated by LOCKPICK, with only 46 false positives induced. We observed that, in addition to the known CVEs, many found bugs have been fixed by developers in the newest version. Compared to SVF and CSA, 11 and 20 vulnerabilities, respectively, were missed, inducing approximately 90% and 98% FPRs. Excluding 20 vulnerabilities in software (i.e., Linux kernel, FreeBSD, and Xen) that L2D2 failed to run due to the compiling problems, L2D2 only detected three known ones out of six vulnerabilities, resulting in about 98% FPR. The reasons for their limited bug-finding capability are discussed later. Based on these impressive results, we believe that LOCKPICK is effective and practical in vulnerability hunting before production runs.

Cross-checking bug reports. In addition to comparing precision and recall, we cross-checked their bug reports to investigate why LOCKPICK is effective. We make four

```
void fio_idle_prof_init(void){
    if (ipc.opt == IDLE_PROF_OPT_NONE)
        return;
    pthread_mutex_lock(&ipt->start_lock);
}
void fio_idle_prof_start(void){
    if (ipc.opt == IDLE_PROF_OPT_NONE)
        return;
    pthread_mutex_unlock(&ipt->start_lock);
}
...
int IntelQaInit(void* threadId){
    ret = pthread_mutex_lock(&g_Hwlock);
    if (ret != 0) {
        return BAD_MUTEX_E;
    }
}
```

Figure 19: False positives reported by other tools.

observations. First, we find that LOCKPICK can detect all real bugs reported by other tools, reaching high precision without sacrificing bug finding capability. Second, a key reason for the spurious reports in other tools is infeasible program paths, illustrated previously by using Figure 19. We observe that, in practice, developers have a zero-tolerance policy for excessive false positives, creating a significant obstacle to finding real bugs. Third, CSA can miss many real bugs due to the limited inlining by only analyzing a single file and without modeling many locks (e.g., the spinlocks in Linux Kernel, FreeBSD), and L2D2 can miss many real bugs due to its reliance on syntactic expressions to reason pointer aliasing. Fourth, they are incapable of detecting various types of lock misuses since they do not characterize concurrency. For example, in Table 2, 52 lock misuse bugs are related to concurrency, which are missed by SVF. In conclusion, LOCKPICK has good precision and bug-finding capabilities, effectively mitigating the performance issue caused by characterizing concurrency semantics.

Reasons for misuses. By inspecting the confirmed bugs, we have identified several common causes of lock misuses, which could be helpful for avoiding misusing locks. First, missing lock releases ① often occurs in error-handling branches. Specifically, a buggy function often acquires a lock, encounters error conditions, and returns to callers without releasing the holding lock. Second, double locking bugs ② are often introduced by typographical errors, in which the corresponding unlock statements are written as lock statements. Third, using uninitialized locks ③ is caused by initializing locks in a non-deterministic manner (e.g., after creating a child thread using the locks). Fourth, releasing unheld locks ④ is commonly induced by (i) the inconsistent branch conditions between lock acquisitions and releases and (ii) inconsistent acquisitions and releases of locks in loops. Finally, cyclic lock acquisitions ⑤ are caused by using locks in a nested way, in which the nested locks are acquired and released in a disorganized manner.

6.3 Ablation Study

Finally, we conduct an ablation study based on the benchmarks shown in Table 2 to shed more light on the inner workings of the LOCKPICK framework. More specifically, we performed two experiments to dig into the actual details and analyze the intermediate artifacts of the lock-misuse analysis. First, we investigate the importance of being path-sensitive. Compared to disabling its SMT solving, LOCKPICK can significantly reduce an additional 40.4% of false positives with only a 19.2 % increase in time costs. Second, we examine the importance of reasoning concurrency. Our results show that LOCKPICK can detect 22.2 % more concurrency-related lock misuse bugs at 12.6% higher time costs than when the demand-driven concurrency analysis is disabled. Consider the promising bug-finding results of LOCKPICK shown in § 6.1 and § 6.2. The additional time costs are reasonable and acceptable in exchange for very high precision and effective bug-finding capability.

7 Related Work

We have introduced much related work in § 1 and § 3, respectively. Next, we summarize the previous work.

Locking study. Much work focuses on investigating lock APIs. In particular, the past work [46] has studied Android-specific misuses of the *wake lock*. In contrast, we investigate and detect the bugs and vulnerabilities resulting from the misuses of general locks in C/C++ programs. LOCKDOC [48] infers the locking rules for the data structures in the Linux kernel. In addition, much work focuses on the performance of locking algorithms [17, 35, 36] and synchronization inferences [42].

Static analysis. LOCKPICK can detect diverse lock misuses and be used as a supplement to existing data race detectors [33, 34, 44, 74, 75], which generally focus on thread-shared memory accesses. In § 2, we have revealed how lock misuses can result in security issues and trigger concurrency errors. Our approaches have a flavor of some past work [7, 12, 32], in the sense of identifying redundant thread interleavings. In contrast, our idea of reducing concurrency reasoning for lock misuse detection is to capture typestate violations as hints. Some work on detecting improper error handling [28, 94], cross-checking inconsistent bugs [45, 50, 54, 87, 105], and tracing source-sink bugs [21, 80–82] can uncover limited lock misuses (e.g., missing lock releases). Comparatively, LOCKPICK is dedicated and effective in path-sensitively detecting lock misuses, taking thread interactions into account.

Dynamic analysis. Dynamic tools like VALGRIND [89] and THREADSANITIZER [79] can instrument lock APIs and monitor the runtime program behaviors to detect lock misuses [9, 29, 70, 78, 79, 89, 91]. Notably, dynamic and static approaches have been separate realms for bug finding, having different merits. For thoroughly detecting lock misuses with higher code coverage, we believe that a static bug detector is superior since there is no need to provide specific inputs and configurations. As demonstrated, LOCKPICK has found many long-latent errors in popular large-scale software (e.g., MariaDB, OpenSSL, PJSIP), which were missed by users, regression testing, and other testing techniques.

8 Conclusion and Future Work

We have presented LOCKPICK, a practical static analysis framework for detecting lock misuses. The evaluation demonstrates that LOCKPICK is precise and efficient compared to the state-of-the-art tools. Moreover, LOCKPICK is quite promising, having uncovered 203 previously-unknown confirmed lock misuse errors on dozens of popular and well-checked open-source software with 16 assigned CVE IDs.

As stated in § 2, LOCKPICK cannot (in fact, is not designed to) detect lock misuses that are (i) induced by incorrect interactions between locks and other primitives (e.g., events and sockets) and (ii) caused by some ad-hoc system rules (which require user-supplied or learned specifications [42]). In the future, we will investigate these directions to enhance LOCKPICK.

Acknowledgments

We appreciate the anonymous reviewers for their insightful comments. We are also grateful to the software maintainers for their prompt feedback on the bug reports. This work was partially funded by PRP/004/21FX, ITS/440/18FP grants, and the donations from Microsoft and Huawei. Peisen Yao is the corresponding author.

References

[1] AMANN, S., NGUYEN, H. A., NADI, S., NGUYEN, T. N., AND MEZINI, M. A systematic evaluation of static api-misuse detectors. *IEEE Trans. Software Eng. 45*, 12 (2019), 1170–1188.

[2] AVOTS, D., DALTON, M., LIVSHITS, V. B., AND LAM, M. S. Improving software security with a C pointer analysis. In *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, G. Roman, W. G. Griswold, and B. Nuseibeh, Eds.

[3] BABIC, D., AND HU, A. J. Calysto: scalable and precise extended static checking. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008* (2008), W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds.

[4] BAI, J., LAWALL, J., CHEN, Q., AND HU, S. Effective static analysis of concurrency use-after-free bugs in linux device drivers. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, D. Malkhi and D. Tsafrir, Eds.

[5] BLACKSHEAR, S., GOROGIANNIS, N., O'HEARN, P. W., AND SERGEY, I. Racerd: compositional static race detection. *Proc. ACM Program. Lang. 2*, OOPSLA.

[6] BROTHERSTON, J., BRUNET, P., GOROGIANNIS, N., AND KANOVICH, M. A compositional deadlock detector for android java. In *Proceedings of ASE-36* (2021), ACM.

[7] CAI, Y., YAO, P., AND ZHANG, C. Canary: practical static detection of inter-thread value-flow bugs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, S. N. Freund and E. Yahav, Eds.

[8] CAI, Y., YE, C., SHI, Q., AND ZHANG, C. Peahen: fast and precise static deadlock detection via context reduction. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022* (2022), A. Roychoudhury, C. Cadar, and M. Kim, Eds., ACM, pp. 784–796.

[9] CHEN, H., GUO, S., XUE, Y., SUI, Y., ZHANG, C., LI, Y., WANG, H., AND LIU, Y. MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds.

[10] CHUGH, R., VOUNG, J. W., JHALA, R., AND LERNER, S. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2008), PLDI '08, Association for Computing Machinery, p. 316–326.

[11] CLANG STATIC ANALYZER. https://clang-analyzer.llvm.org/.

[12] CLARKE, E. M., GRUMBERG, O., MINEA, M., AND PELED, D. A. State space reduction using partial order techniques. *Int. J. Softw. Tools Technol. Transf. 2*, 3 (1999), 279–287.

[13] CVE. CVE Benchmarks for Studying Lock Misuses. https://docs.google.com/spreadsheets/d/1R8cNSb8i_Yk0AaysUrBhfTX_xN92437eNzDCxFPu8t4/edit#gid=0.

[14] DAS, M., LERNER, S., AND SEIGLE, M. ESP: path-sensitive program verification in polynomial time. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, J. Knoop and L. J. Hendren, Eds.

[15] DELINE, R., AND FÄHNDRICH, M. Typestates for objects. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, June 14-18, 2004, Proceedings*, M. Odersky, Ed., Lecture Notes in Computer Science.

[16] DI, P., SUI, Y., YE, D., AND XUE, J. Region-based may-happen-in-parallel analysis for C programs. In *44th International Conference on Parallel Processing, ICPP 2015, Beijing, China, September 1-4, 2015*.

[17] DICE, D., MARATHE, V. J., AND SHAVIT, N. Lock cohorting: a general technique for designing NUMA locks. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, J. Ramanujam and P. Sadayappan, Eds.

[18] DIRTYCOW. https://dirtycow.ninja/.

[19] Dor, N., Adams, S., Das, M., and Yang, Z. Software validation via scalable path-sensitive value flow analysis. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, G. S. Avrunin and G. Rothermel, Eds.

[20] Engler, D., and Ashcraft, K. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, Association for Computing Machinery, p. 237–252.

[21] Fan, G., Wu, R., Shi, Q., Xiao, X., Zhou, J., and Zhang, C. Smoke: scalable path-sensitive memory leak detection for millions of lines of code. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*.

[22] Farzan, A., and Madhusudan, P. Causal dataflow analysis for concurrent programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, O. Grumberg and M. Huth, Eds., vol. 4424 of *Lecture Notes in Computer Science*.

[23] FreeBSD Bugzilla. Bug #261051. https://bugs.freebsd.org/bugzilla/show_bug.cgi?id=261051.

[24] Giacobazzi, R., and Ranzato, F. Optimal domains for disjunctive abstract interpretation. *Science of Computer Programming 32*, 1-3 (1998), 177–210.

[25] Goodstein, M. L., Vlachos, E., Chen, S., Gibbons, P. B., Kozuch, M. A., and Mowry, T. C. Butterfly analysis: adapting dataflow analysis to dynamic parallel monitoring. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2010, Pittsburgh, Pennsylvania, USA, March 13-17, 2010* (2010), J. C. Hoe and V. S. Adve, Eds., ACM, pp. 257–270.

[26] Gu, R., Jin, G., Song, L., Zhu, L., and Lu, S. What change history tells us about thread synchronization. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, Association for Computing Machinery, p. 426–438.

[27] Infer. https://github.com/facebook/infer.

[28] Jana, S., Kang, Y. J., Roth, S., and Ray, B. Automatically detecting error handling bugs using error specifications. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, T. Holz and S. Savage, Eds.

[29] Jeong, D. R., Kim, K., Shivakumar, B., Lee, B., and Shin, I. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*.

[30] Jin, G., Song, L., Zhang, W., Lu, S., and Liblit, B. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, M. W. Hall and D. A. Padua, Eds.

[31] Jin, G., Zhang, W., and Deng, D. Automated concurrency-bug fixing. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, C. Thekkath and A. Vahdat, Eds.

[32] Kahlon, V., Gupta, A., and Sinha, N. Symbolic model checking of concurrent programs using partial orders and on-the-fly transactions. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, T. Ball and R. B. Jones, Eds., vol. 4144 of *Lecture Notes in Computer Science*.

[33] Kahlon, V., Sinha, N., Kruus, E., and Zhang, Y. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, H. van Vliet and V. Issarny, Eds.

[34] Kahlon, V., Yang, Y., Sankaranarayanan, S., and Gupta, A. Fast and accurate static data-race detection for concurrent programs. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, W. Damm and H. Hermanns, Eds., vol. 4590 of *Lecture Notes in Computer Science*.

[35] Kashyap, S., Calciu, I., Cheng, X., Min, C., and Kim, T. Scalable and practical locking with shuffling. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2019), SOSP '19, Association for Computing Machinery, p. 586–599.

[36] Kashyap, S., Min, C., and Kim, T. Scalable numa-aware blocking synchronization primitives. In *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017*, D. D. Silva and B. Ford, Eds.

[37] Kellogg, M., Shadab, N., Sridharan, M., and Ernst, M. D. Lightweight and modular resource leak verification. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2021), ESEC/FSE 2021, Association for Computing Machinery, p. 181–192.

[38] Kroening, D., Poetzl, D., Schrammel, P., and Wachter, B. Sound static deadlock analysis for c/pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2016), ASE 2016, Association for Computing Machinery, p. 379–390.

[39] Kusano, M., and Wang, C. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2016), FSE 2016, Association for Computing Machinery, pp. 799–809.

[40] L2D2. https://pajda.fit.vutbr.cz/xmarci10/fbinfer_concurrency.

[41] Leveson, N. G., and Turner, C. S. Investigation of the therac-25 accidents. *Computer 26*, 7.

[42] Li, G., Chen, D., Lu, S., Musuvathi, M., and Nath, S. Sherlock: unsupervised synchronization-operation inference. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds.

[43] Liu, B., and Huang, J. D4: fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds.

[44] Liu, B., Liu, P., Li, Y., Tsai, C.-C., Da Silva, D., and Huang, J. When threads meet events: Efficient and precise static race detection with origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (New York, NY, USA, 2021), PLDI 2021, Association for Computing Machinery, p. 725–739.

[45] Liu, D., Wu, Q., Ji, S., Lu, K., Liu, Z., Chen, J., and He, Q. Detecting missed security operations through differential checking of object-based similar paths. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021* (2021), Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds., ACM, pp. 1627–1644.

[46] Liu, Y., Xu, C., Cheung, S., and Terragni, V. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds.

[47] Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J. N., Chang, B. E., Guyer, S. Z., Khedker, U. P., Møller, A., and Vardoulakis, D. In defense of soundiness: a manifesto. *Commun. ACM 58*, 2 (2015), 44–46.

[48] Lochmann, A., Schirmeier, H., Borghorst, H., and Spinczyk, O. Lockdoc: Trace-based analysis of locking in the linux kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, G. Candea, R. van Renesse, and C. Fetzer, Eds.

[49] Lockpick. Confirmed and fixed bugs. https://whichbug.github.io/.

[50] Lu, K., Pakki, A., and Wu, Q. Detecting missing-check bugs via semantic- and context-aware criticalness and constraints inferences. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019* (2019), N. Heninger and P. Traynor, Eds., USENIX Association, pp. 1769–1786.

[51] Lu, S., Park, S., Seo, E., and Zhou, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, S. J. Eggers and J. R. Larus, Eds.

[52] Machiry, A., Spensky, C., Corina, J., Stephens, N., Kruegel, C., and Vigna, G. DR. CHECKER: A soundy analysis for linux kernel drivers. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds.

[53] MariaDB. https://github.com/MariaDB/server/pull/1948.

[54] Min, C., Kashyap, S., Lee, B., Song, C., and Kim, T. Cross-checking semantic correctness: the case of finding file system bugs. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, E. L. Miller and S. Hand, Eds.

[55] MITRE-CVE. CVE-2004-2650. https://nvd.nist.gov/vuln/detail/CVE-2004-2650.

[56] MITRE-CVE. CVE-2013-4553. https://nvd.nist.gov/vuln/detail/CVE-2013-4553.

[57] MITRE-CVE. CVE-2014-1453. https://nvd.nist.gov/vuln/detail/CVE-2014-1453.

[58] MITRE-CVE. CVE-2014-8131. https://nvd.nist.gov/vuln/detail/CVE-2014-8131.

[59] MITRE-CVE. CVE-2014-9748. https://nvd.nist.gov/vuln/detail/CVE-2014-9748.

[60] MITRE-CVE. CVE-2015-8767. https://nvd.nist.gov/vuln/detail/CVE-2015-8767.

[61] MITRE-CVE. CVE-2017-6353. https://nvd.nist.gov/vuln/detail/CVE-2017-6353.

[62] MITRE-CVE. CVE-2018-14660. https://nvd.nist.gov/vuln/detail/CVE-2018-14660.

[63] MITRE-CVE. CVE-2019-14034. https://nvd.nist.gov/vuln/detail/CVE-2019-14034.

[64] MITRE-CVE. CVE-2019-14763. https://nvd.nist.gov/vuln/detail/CVE-2019-14763.

[65] MITRE-CVE. CVE-2020-10573. https://nvd.nist.gov/vuln/detail/CVE-2020-10573.

[66] MITRE-CVE. CVE-2020-12658. https://nvd.nist.gov/vuln/detail/CVE-2020-12658.

[67] MITRE-CVE. CVE-2020-25604. https://nvd.nist.gov/vuln/detail/CVE-2020-25604.

[68] MITRE-CVE. CVE-2021-41141. https://nvd.nist.gov/vuln/detail/CVE-2021-41141.

[69] MITRE-CVE. CVE-2021-41213. https://nvd.nist.gov/vuln/detail/CVE-2021-41213.

[70] Mukherjee, S., Deligiannis, P., Biswas, A., and Lal, A. Learning-based controlled concurrency testing. *Proc. ACM Program. Lang. 4*, OOPSLA (2020), 230:1–230:31.

[71] Naik, M., Park, C., Sen, K., and Gay, D. Effective static deadlock detection. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*.

[72] Oracle. Oracle programming style. https://docs.oracle.com/cd/E18752_01/html/816-5137/guide-35930.html.

[73] PJSIP. Issue #2845. https://github.com/pjsip/pjproject/issues/2845.

[74] Pratikakis, P., Foster, J. S., and Hicks, M. LOCKSMITH: practical static race detection for C. *ACM Trans. Program. Lang. Syst. 33*, 1.

[75] Pratikakis, P., Foster, J. S., and Hicks, M. W. LOCKSMITH: context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, M. I. Schwartzbach and T. Ball, Eds.

[76] Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1995), POPL '95, Association for Computing Machinery, p. 49–61.

[77] Sankaranarayanan, S., Ivančić, F., Shlyakhter, I., and Gupta, A. Static analysis in disjunctive numerical domains. In *International Static Analysis Symposium* (2006), Springer, pp. 3–17.

[78] Sen, K. Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, R. Gupta and S. P. Amarasinghe, Eds.

[79] Serebryany, K., and Iskhodzhanov, T. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (New York, NY, USA, 2009), WBIA '09, Association for Computing Machinery, p. 62–71.

[80] Shi, Q., Wu, R., Fan, G., and Zhang, C. Conquering the extensional scalability problem for value-flow analysis frameworks. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020* (2020), G. Rothermel and D. Bae, Eds., ACM, pp. 812–823.

[81] Shi, Q., Xiao, X., Wu, R., Zhou, J., Fan, G., and Zhang, C. 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 2018, Philadelphia, PA, USA, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds.

[82] Shi, Q., Yao, P., Wu, R., and Zhang, C. Path-sensitive sparse analysis without path conditions. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021* (2021), S. N. Freund and E. Yahav, Eds., ACM, pp. 930–943.

[83] Sinha, N., and Wang, C. Staged concurrent program analysis. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, G. Roman and A. van der Hoek, Eds.

[84] Sui, Y., Di, P., and Xue, J. Sparse flow-sensitive pointer analysis for multithreaded programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, B. Franke, Y. Wu, and F. Rastello, Eds.

[85] Sui, Y., Ye, D., and Xue, J. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2012), ISSTA 2012, Association for Computing Machinery, p. 254–264.

[86] SVF. https://github.com/SVF-tools/SVF.

[87] Tan, X., Zhang, Y., Yang, X., Lu, K., and Yang, M. Detecting kernel refcount bugs with two-dimensional consistency checking. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (2021), M. Bailey and R. Greenstadt, Eds., USENIX Association, pp. 2471–2488.

[88] Torlak, E., and Chandra, S. Effective interprocedural resource leak detection. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, Association for Computing Machinery, p. 535–544.

[89] Valgrind. https://pmem.io/valgrind/generated/index.html.

[90] Voung, J. W., Jhala, R., and Lerner, S. RELAY: static race detection on millions of lines of code. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, I. Crnkovic and A. Bertolino, Eds.

[91] Wen, C., He, M., Wu, B., Xu, Z., and Qin, S. Controlled concurrency testing via periodical scheduling. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022* (2022), ACM, pp. 474–486.

[92] Wiki. The 2003 northeast blackout. https://en.wikipedia.org/wiki/Northeast_blackout_of_2003.

[93] Williams, A. L., Thies, W., and Ernst, M. D. Static deadlock detection for java libraries. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, A. P. Black, Ed., vol. 3586 of *Lecture Notes in Computer Science*.

[94] Wu, Q., Pakki, A., Emamdoost, N., McCamant, S., and Lu, K. Understanding and detecting disordered error handling with precise function pairing. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, M. Bailey and R. Greenstadt, Eds.

[95] Xie, Y., and Aiken, A. Context- and path-sensitive memory leak detection. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds.

[96] Xie, Y., and Aiken, A. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, J. Palsberg and M. Abadi, Eds.

[97] Xu, M., Kashyap, S., Zhao, H., and Kim, T. Krace: Data race fuzzing for kernel file systems. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*.

[98] Xu, M., Qian, C., Lu, K., Backes, M., and Kim, T. Precise and scalable detection of double-fetch bugs in OS kernels. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*.

[99] Yan, H., Sui, Y., Chen, S., and Xue, J. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds.

[100] Yang, J., Cui, A., Stolfo, S. J., and Sethumadhavan, S. Concurrency attacks. In *4th USENIX Workshop on Hot Topics in Parallelism, HotPar'12, Berkeley, CA, USA, June 7-8, 2012* (2012), H. Boehm and L. Ceze, Eds., USENIX Association.

[101] Z3. https://github.com/Z3Prover/z3.

[102] Zhan, S., and Huang, J. ECHO: instantaneous in situ race detection in the IDE. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds.

[103] Zhao, S., Gu, R., Qiu, H., Li, T. O., Wang, Y., Cui, H., and Yang, J. OWL: understanding and detecting concurrency attacks. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*.

[104] Zhou, J., Silvestro, S., Liu, H., Cai, Y., and Liu, T. UNDEAD: detecting and preventing deadlocks in production software. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds.

[105] Zhou, Q., Wu, Q., Liu, D., Ji, S., and Lu, K. Non-distinguishable inconsistencies as a deterministic oracle for detecting security bugs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022* (2022), H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds., ACM, pp. 3253–3267.