Peahen: Fast and Precise Static Deadlock Detection via Context Reduction

Yuandao Cai
The Hong Kong University of Science and Technology
Hong Kong, China
ycaibb@cse.ust.hk

Chengfeng Ye
The Hong Kong University of Science and Technology
Hong Kong, China
cyeaa@cse.ust.hk

Qingkai Shi
Ant Group
Shenzhen, China
qingkai.sqk@antgroup.com

Charles Zhang
The Hong Kong University of Science and Technology
Hong Kong, China
charlesz@cse.ust.hk

ABSTRACT

Deadlocks still severely inflict reliability and security issues upon software systems of the modern age. Worse still, as we note, in prior static deadlock detectors, good precision does not go hand-in-hand with high scalability — their approaches are either context-insensitive, thereby engendering many false positives, or suffer from the calling context explosion to reach context-sensitive, thus compromising good efficiency. In this paper, we advocate Peahen, geared towards precise yet also scalable static deadlock detection. At its crux, Peahen decomposes the computational effort for embracing high precision into two cooperative analysis stages: (i) context-insensitive lock-graph construction, which selectively encodes the essential lock-acquisition information on each edge, and (ii) three precise yet lazy refinements, which incorporate such edge information into progressively refining the deadlock cycles in the lock graph only for a few interesting calling contexts.

Our extensive experiments yield promising results: Peahen dramatically out-performs the state-of-the-art tools on accuracy without losing scalability; it can efficiently check million-line systems at a low false positive rate; and it has uncovered many confirmed deadlocks in dozens of mature open-source systems.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

Concurrency, static analysis, deadlock detection, context-sensitivity

ACM Reference Format:
Yuandao Cai, Chengfeng Ye, Qingkai Shi, and Charles Zhang. 2022. 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 '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549110

1 INTRODUCTION

Deadlocks continue to scourge modern software systems with a denial of service or even malicious attacks [46]. Just in 2021, more than sixty open issues in Firefox and Linux Kernel were associated with deadlocks [18, 40]. Over the last decade, many famous systems (e.g., Apache, Android, TensorFlow) have also fallen victims; more than eighty CVE IDs have been assigned to the uncovered deadlock vulnerabilities therein [13]. Thus, it is still increasingly urgent to hunt deadlocks before the production runs. This paper focuses on *resource deadlocks*, arising when each thread in a set circularly waits for another thread in the set to release an acquired lock or a single thread attempts to reacquire a non-reentrant lock.

Static analysis [1, 7, 44, 55, 64] has the advantage of penetrating obscure program paths harboring deadlocks, which, however, are difficult for dynamic analysis to reach [4, 5, 11, 33, 34, 39, 61, 67]. Yet, despite the impressive strides for decades, the fast and precise static deadlock detection for million-line systems remains far from satisfactory. Specifically, past research [2, 16, 42, 49, 62] has strongly indicated that context-sensitive deadlock detectors can suppress false positives against the context-insensitive ones. On the downside, as we note, the context-sensitive detectors [16, 42] suffer considerably from the scalability penalties induced by the *calling context explosion*. For instance, the state-of-the-art context-sensitive detector [42] even failed to analyze a few thousand lines of code within half an hour or 24 GB of memory.

A crucial reason, as we observe, is that the traditional context-sensitive deadlock detection constructs a context-sensitive lock graph through a pre-computed context-sensitive lockset analysis [16, 36, 38, 42], computing the locks held at each program location under different calling contexts. A node in the graph represents a lock object while a directed edge, labeled by the string of calling contexts, indicates the order of lock acquisitions under certain

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3549110

calling contexts. To disclose deadlocks, the prior approaches traverse the lock graph to discover cycles that characterize cyclic lock-acquisition orders. Again, the results of lockset analysis are leveraged to refine the detected cycles to check if the related lock acquisitions under the specific calling contexts can be executed concurrently, i.e., if they are not protected by a common lock. Yet, we notice that the explosive calling contexts can exceedingly slow down the pre-computed lockset analysis and blow up the lock graph with acquired edges, thereby inevitably impeding the graph construction and deadlock-cycle discovery.

To address the problem, our key insight is that only a minority of the calling contexts are associated with deadlocks in practical programs. As a result, a conventional context-sensitive detector could cause a great deal of redundancy in the pre-computed context-sensitive lockset analysis, which is oblivious of the related contexts around the deadlock cycles. In this work, to reduce the irrelevant calling contexts, Peahen decomposes the cost of reaching high-fidelity by first efficiently constructing a context-insensitive and smaller lock graph. With the lock graph in hand, Peahen can focus on the context-sensitive refinement effort around only a few calling contexts that are related to the deadlock cycles, whereby saving much cost to deadlock-irrelevant computation.

More specifically, given the initially detected cycles in the context-insensitive lock graph, Peahen employs three precise yet lazy refinements that effectively approximate three different necessary conditions of a deadlock:

- A thread identification analysis to examine whether a deadlock is induced by a single thread or multiple threads.
- A context-sensitive non-concurrency analysis to identify whether a multi-threaded deadlock may concurrently occur.
- A constraint solving process to validate the path feasibility of a deadlock with a costly SMT solver.

Surprisingly, the refinement process itself is also efficient because the more precise refining results contribute to fewer deadlock cycles as well as less lazy computation.

One salient feature underpinning our context reduction is that our lock-graph construction infers and offers the lock-acquisition information to bootstrap the lazy refinements in close cooperation. First, our lock-graph construction algorithm is compositional and faster than the previous context-sensitive lockset analysis [16, 38, 42], because it does not enumerate any exponentially possible calling contexts spanning across the lock acquisitions. Second, our algorithm foreseeingly reasons and encodes the essential edge information on which threads acquire the locks and where the lock acquisitions take place, which is integrated into the lazy deadlock refinements to be efficiently leveraged.

Results. We have implemented Peahen for C/C++ systems, where the non-nested locks (e.g., Pthread APIs) are extensively exploited. Like many modern static bug-finding tools [55, 57], Peahen is soundy [45], sacrificing soundness in a few common cases for high fidelity. The experimental results are highlighted below:

- *Advancing the state-of-the-art*: we compared Peahen to two recent detectors, CBMC [42] (context-sensitive) and Infer [2, 22] (context-insensitive) under the benchmarks [42]. It is noted that we refer to the static deadlock detector [42] as CBMC throughout because it is maintained in a git branch of the CBMC project [41].
  - Peahen achieves 136× speed-up versus CBMC and, reasonably, is slightly slower than Infer.
  - Peahen can reduce around 73.45% and 90.0% false positives against CBMC and Infer, respectively. In addition, Peahen can detect all the intentionally planted deadlocks on their benchmarks [42].
  - Our context-insensitive lock graph can reduce about 95.5% acquired edges that are otherwise identified and added to a context-sensitive graph. Moreover, our lock-graph construction and cycle-detection process are about 231 × and 46 × faster than the ones in CBMC, respectively.
- *Effective on large systems*: we have used Peahen to capture deadlock issues on a diverse set of real-world, widely-used open-source C/C++ software systems.
  - Peahen can effectively finish the checking of multi-MLOC code bases (e.g., Firefox around 2 hours) at a relatively low false-positive rate (30.43% on average).
  - Peahen has detected more than eighty confirmed deadlocks from dozens of famous open systems, all told [50].

To sum up, our contributions are as follows:

- We advocate Peahen, a context-reduction technique for fast and precise deadlock detection in large software systems.
- We present a compositional algorithm to construct a context-insensitive lock graph for non-nested locks, which is also collaborative with lazy deadlock refinements.
- A substantial experiment that demonstrates Peahen's good scalability and precision versus the previous work.

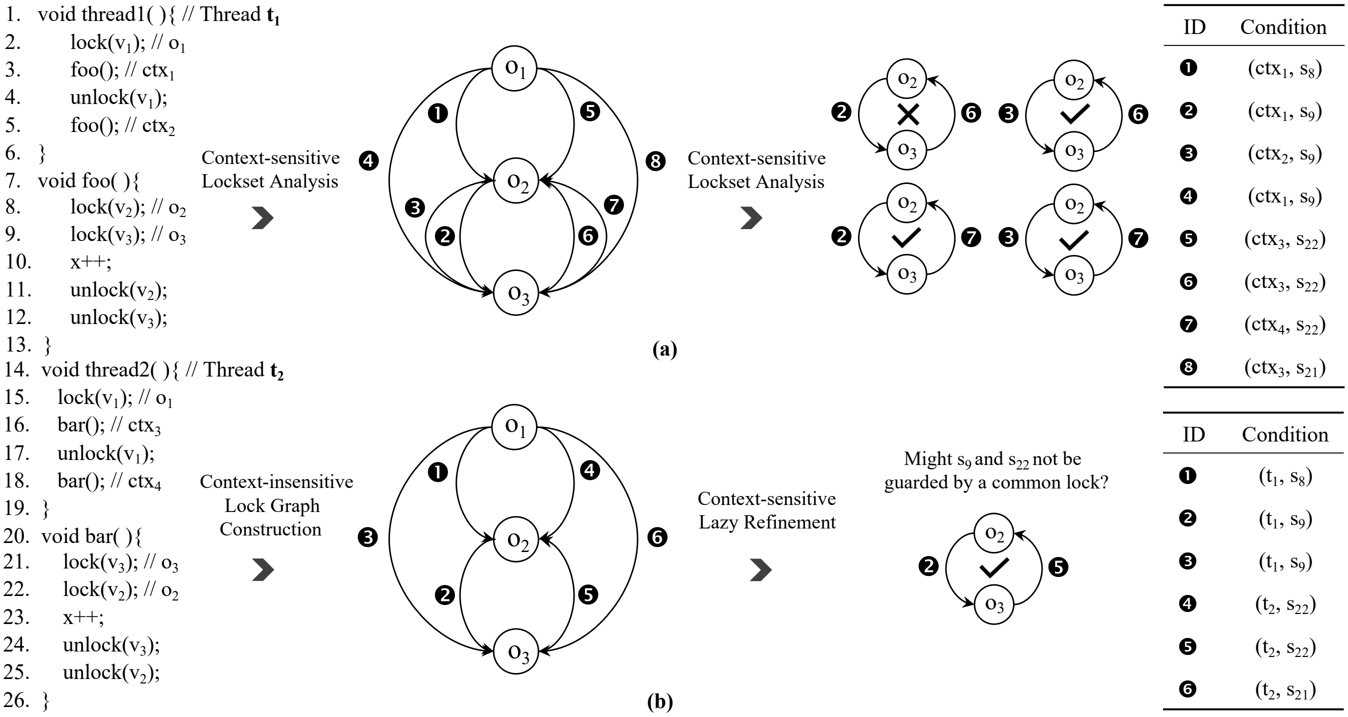
2 PEAHEN IN A NUTSHELL

We first utilize a buggy program in Figure 1 to clarify the concept of deadlock (§ 2.1). We then illustrate the shortcomings of the previous context-sensitive solutions (§ 2.2). Finally, we highlight the essence of our more effective context-reduction approach (§ 2.3).

2.1 Example Illustration

Let the symbol ctx represent the calling context and s_i denote the statement at Line i. For the code snippet shown in Figure 1, there are two concurrent threads t_1 and t_2, running the functions $thread1()$ and $thread2()$ respectively. Specifically, the thread t_1 first executes the function $foo()$ holding a lock o_1 and then executes the function $foo()$ again with the lock o_1 released before. Similarly, the thread t_2 executes the function $bar()$ twice, one of which is protected by the common lock o_1.

If we take a close look at the functions $foo()$ and $bar()$, there is an order inversion between the locks o_2 and o_3. That is, a deadlock may be triggered when the thread t_1 acquires the lock o_2 at s_8, waiting at s_9 for o_3 to be released, but thread t_2 acquires the lock o_3 at s_{21}, waiting at s_{22} for o_2 to be released, and neither can progress. However, the function $foo()$ under the calling context ctx_1 and the function $bar()$ under the calling context ctx_3 cannot be executed concurrently, because they are protected by a common lock o_1 (also called a gate lock [23, 49]). Thus, the two functions $foo()$ and $bar()$ under other compositional calling contexts, such as $foo()$ at cxt_1



```
1.  void thread1( ){ // Thread t1
2.      lock(v1); // o1
3.      foo(); // ctx1
4.      unlock(v1);
5.      foo(); // ctx2
6.  }
7.  void foo( ){
8.      lock(v2); // o2
9.      lock(v3); // o3
10.     x++;
11.     unlock(v2);
12.     unlock(v3);
13. }
14. void thread2( ){ // Thread t2
15.     lock(v1); // o1
16.     bar(); // ctx3
17.     unlock(v1);
18.     bar(); // ctx4
19. }
20. void bar( ){
21.     lock(v3); // o3
22.     lock(v2); // o2
23.     x++;
24.     unlock(v3);
25.     unlock(v2);
26. }
```

Context-sensitive Lockset Analysis
o1
o2
o3
❶
❷
❸
❹
❺
❻
❼
❽
Context-sensitive Lockset Analysis
(a)
Context-insensitive Lock Graph Construction
Context-sensitive Lazy Refinement
Might s9 and s22 not be guarded by a common lock?
(b)

ID	Condition
❶	(ctx_1, s_8)
❷	(ctx_1, s_9)
❸	(ctx_2, s_9)
❹	(ctx_1, s_9)
❺	(ctx_3, s_{22})
❻	(ctx_3, s_{22})
❼	(ctx_4, s_{22})
❽	(ctx_3, s_{21})

ID	Condition
❶	(t_1, s_8)
❷	(t_1, s_9)
❸	(t_1, s_9)
❹	(t_2, s_{22})
❺	(t_2, s_{22})
❻	(t_2, s_{21})

Figure 1: Figure(a) shows the previous context-sensitive approach, which uses a dual-purpose, context-sensitive lockset analysis for both constructing a lock graph and identifying the gate lock. Figure(b) shows the context reduction approach, which first constructs a context-insensitive, smaller graph and detects fewer cycles, followed by the context-sensitive lazy refinements.

and $bar()$ at ctx_4, can be executed concurrently and a deadlock bug could surface to "bit", bringing the program to a grinding halt.

To sum up, there are two key steps for static deadlock detection. First, the lock-acquisition orders in each thread should be identified to derive cyclic orders among locks. Second, the initial results on reversed orders of lock acquisitions should be further refined such as context-sensitively identifying the gate-lock scenarios. Importantly, the solutions should both be precise and highly scalable, sieving through millions of lines of code in just a few hours.

2.2 The State-of-the-Art

However, to go for high precision, the prior approaches could not scale up to large systems. An essential reason is that, they commonly employ a flow-, context-sensitive lockset analysis, which computes the locks held at each program point [38, 42, 62]. The precise pre-computed lockset analysis can kill two birds with one stone; it is capable of context-sensitively identifying the lock acquisitions and the gate locks. On the downside, the explosive calling contexts in large programs can drastically slow down the lockset analysis and blow up the lock graph with acquired edges.

To illustrate, in Figure 1(a), the approach first constructs a context-sensitive lock graph using a context-sensitive lockset analysis. For instance, the lockset analysis can tell that the lock o_1 is held at the statement s_8 under the calling context ctx_1. Therefore, the thread t_1 can hold the lock o_1 and acquire the lock o_2 at s_8 under ctx_1, resulting in a context-sensitive acquired edge from o_1 to o_2. Every edge condition, shown in the right table, consists of the corresponding calling context ctx of the lock statement s that refers to the destination lock. For example, the condition on the edge ❶ is (ctx_1, s_8), indicating that a thread (t_1) holds lock o_1 and then acquires lock o_2 at s_8 under the calling context ctx_1.

Based on the context-sensitive lock graph, a cycle detection is performed to identify the context-sensitive cycles on the graph. As shown in Figure 1(a), four context-sensitive cycles are detected. Again, the technique uses the context-sensitive results of the lockset analysis to identify the gate-lock scenarios. For the detected cycle with the edges ❷ and ❻, the s_9 under the calling context ctx_1 and the s_{22} under ctx_3 are protected by a gate lock o_1, so the statements s_9 and s_{22} cannot be reached simultaneously under the corresponding calling contexts. On the other hand, the remaining three "ungated" cycles are reported to the developers, warning of the calling contexts of $foo()$ and $bar()$ to trigger a deadlock bug.

We notice that the context-sensitive acquired edges are not necessary for identifying the lock-acquisition orders in each thread. For example, the four detected context-sensitive cycles indicate the same case, i.e., there is a cyclic-waiting condition between lock o_2 at s_{22} in thread t_2 and lock o_3 at s_9 in thread t_1. On the other hand, the refinements only target and identify whether statements s_9 and s_{22} (related to a deadlock cycle) can be concurrently executed, which is oblivious to the pre-computed lockset analysis.

2.3 Our Context-Reduction Approach

To ameliorate the aforementioned problems, we introduce a context-reduction approach, breaking down the prohibitive cost of being context-sensitive for all sub-tasks. That is, PEAHEN uses a fast algorithm to construct a context-insensitive lock graph and the cycle detection to find the fewer context-insensitive deadlock cycles. The cycles are then successively and lazily refined for a few deadlock-related calling contexts.

To illustrate, in Figure 1(b), PEAHEN begins by constructing a smaller context-insensitive lock graph. Apparently, our lock graph saves two unnecessary edges. The edge conditions are shown in the right-hand table with the calling context removed (Extra edge information is introduced later). In other words, all the different calling contexts are reduced or merged into one. Armed with this smaller graph, PEAHEN efficiently performs a cycle-detection algorithm to detect only one deadlock cycle. The context-insensitive cycle with edges ❷ and ❺ indicates that there is a cyclic waiting-condition between lock o_3 at s_9 and lock o_2 at s_{22} without any known calling context information. Then, a context-sensitive gate-lock analysis is queried by the question, "Might s_9 and s_{22} not be guarded by a common lock?" The answer is "Yes", flagging they are not protected by lock o_1 under three compositional calling contexts, s_9 at ctx_1 and s_{22} at ctx_4, s_9 at ctx_2 and s_{22} at ctx_3, as well as s_9 at ctx_2 and s_{22} at ctx_4. The context-sensitive report is identical to the one in the previous approach by courtesy of the context-sensitive refinements.

It is worth emphasizing that, the final gate-lock analysis, in either the previous methods or PEAHEN, computes four conditions of calling contexts to identify the gate locks. However, we argue that the pre-computed, exhaustive lockset analysis, unaware of any potential program points that might lead to deadlocks, could miss many opportunities to avoid futile context-sensitive computation. Comparatively, our lazy refinements can work efficiently on the calling contexts around the identified deadlock cycles of interest.

This example briefly illustrates the distinction between the prior work and PEAHEN. In § 3, we further introduce the notion of our lock graph and the varieties of the deadlock cycles, with an emphasis on which lock-acquisition information is encoded in the lock graph cooperatively for the deadlock cycles' refinements.

3 PRELIMINARIES

In this section, we elucidate the basic terminologies and notations, and state the key technical challenges we aim to conquer.

Language and Abstract Domain. Let the symbol $\mathcal{V}$ ($v \in \mathcal{V}$) denote pointer variables while O ($o \in O$) indicate memory objects such as lock objects. A thread ID t ($t \in \mathcal{T}$) represents a thread, which corresponds to a context-sensitive fork site and, thus, denotes a unique runtime thread. Following the common practice, we use the method $fork(t, f)$ to uniformly represent the creation of the thread t executing a procedure f ($f \in \mathcal{F}$), and the method $join(t)$ to denote the destruction site of the thread t. Meanwhile, for a fork site allocated in a loop, we create two threads with identical attributes but different thread IDs. We assume the locks are non-nested, and are not necessarily released in the reverse order in which they are acquired. The implementation of pointer analysis is discussed in § 5. Henceforth, without losing generality, we use method $pt(v)$ to indicate the points-to set of the pointer variable v.

Lock Graph. Against the prior context-sensitive lock graph [42], we design a context-insensitive lock graph to characterize the lock-acquisition orders and to facilitate the cooperation with the lazy refinements. Notably, an identified cycle in the lock graph exposes the cyclic lock acquisitions, manifesting a potential deadlock.

Definition 1. A context-insensitive lock graph (LG) is a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{C})$, where

- $\mathcal{N}$ is a set of nodes, each of which, n, corresponds to a lock object o, $o \in O$. Thus, we also use o to denote a node.
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is a set of edges. The direction of an edge indicates the lock-acquisition order. An edge (o_1, o_2) means that a thread holds a lock o_1, the source-lock node, and then acquires another lock o_2, the destination-lock node.
- $\mathcal{C}$ maps each edge, $e \in \mathcal{E}$, to a tuple (t, s_1, s_2), the edge condition, indicating that the thread t holds the source lock o_1 at s_1 and then acquires the destination lock o_2 at s_2.

There are two noteworthy points. First, the above lock graph is context-insensitive in the sense that each edge is no longer encoded with the corresponding call strings. Second, to provide the crucial lock-acquisition information for lazily refining the deadlock cycles, for each edge (o_1, o_2), two types of information are memorized. Particularly, each edge condition (t, s_1, s_2) includes (i) *the thread information*, the thread ID t, about which thread acquires the locks o_1 and o_2 and (ii) *the program location information*, statements s_1 and s_2, where the acquisitions of locks o_1 and o_2 take place. We show later how to leverage this information for refining deadlock cycles discovered in the lock graph.

Deadlock Cycle. With the lock graph, we define four varieties of deadlock cycles. In a single-threaded deadlock due to reacquiring a non-reentrant lock, a thread t_1 acquires a lock o_1 at s_1 and proceeds to acquire the same lock o_1 at s_2. Thus, it suffices to identify whether there is a detected cycle with only one edge and the number of involved nodes equal to one (reacquiring the same lock). Such cycles are called *single-threaded cycles.*

In a multi-threaded deadlock, in the case of two threads, a thread t_1 first acquires a lock o_1 at s_1 and waits for another lock o_2 to be released at s_2 while a thread t' first acquires the lock o_2 at s'_1 and waits for the lock o_1 to be released at s'_2. More complex cases involving more than two threads resemble the above. Thus, it suffices to identify the cycles with all the acquired edges labeled with different thread IDs, which we call *multi-threaded cycles.*

Definition 2. A deadlock cycle discovered in an LG is termed as a single-threaded cycle if the cycle has only one edge and one node. Meanwhile, a cycle is termed as a multi-threaded cycle if the cycle has multiple edges labeled with distinctive thread IDs.

In addition, for the above example of a multi-threaded cycle, whether the lock statements s_2 and s'_2 can be concurrently executed by the threads t_1 and t_2 should be precisely identified to eschew any bogus deadlock warnings. Such a multi-threaded cycle is called *a concurrent cycle.* Note that a single-thread cycle has no use for accounting for the concurrency semantics.

Definition 3. A multi-threaded cycle is termed as a concurrent cycle if the related statements referring to the destination-lock nodes in different threads can be concurrently executed.

Finally, for both single-threaded cycles and multi-threaded cycles, they should be path-feasible, i.e., the related statements should be path-reachable simultaneously, which we call *path-feasible cycles.*

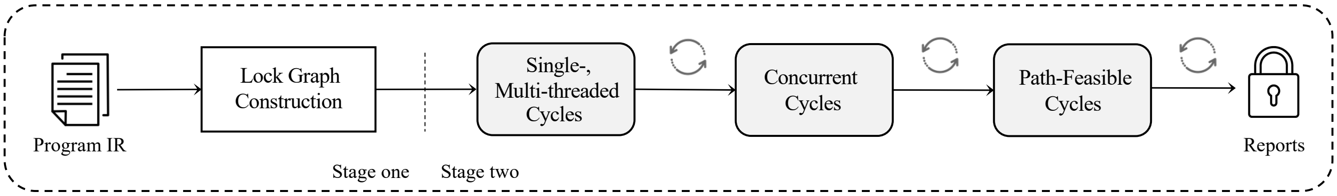


Figure 2: The workflow of Peahen.

Definition 4. A single-threaded cycle or a multi-threaded cycle is termed as a path-feasible cycle if the related statements involving the cycle are path-reachable with the feasible path conditions.

There are three points to note. First, concurrent cycles and path-feasible cycles are the stronger conditions than single-threaded cycles or multi-threaded cycles to be genuine deadlocks. Second, identifying path-feasible cycles is generally more expensive than identifying concurrent cycles, because the former should validate the path conditions via constraint solving. Third, to our knowledge, thus far, no prior deadlock detector has taken the non-nested locks, statements' concurrent execution relations, and statements' path-feasibility into account at once [2, 41, 49].

Problem Statement. After introducing the basic notions, we next state two technical challenges we aim to address.

(1) How to design an efficient algorithm for constructing the context-insensitive lock graph and, in the meantime, encoding the lock-acquisition information on related edges.
(2) How to synergize multiple lazy analyzes to incorporate such edge information into progressively refining the discovered deadlock cycles to regain high precision.

Before introducing our algorithms in § 4, we give an outline of our solutions against the above two challenges.

(1) We present an inter-procedural compositional algorithm for constructing the context-insensitive lock graph. It is sensitive to threads and encodes the lock-acquisition information on edges for the coming refinements. (§ 4.1)
(2) We present three precise yet lazy refinements to progressively improve the precision of the initially detected deadlock cycles by successively identifying single-, multi-threaded cycles, concurrent cycles, and path-feasible cycles. (§ 4.2)

4 PEAHEN IN DETAIL

In this section, we follow the Peahen's workflow as illustrated in Figure 2 to unravel our algorithms. Peahen has two symbiotic stages subsuming the context-insensitive lock-graph construction (§ 4.1) and three precise yet lazy deadlock-cycle refinements (§ 4.2).

4.1 Context-Insensitive LG Construction

This part delves into our inter-procedural algorithm to construct the context-insensitive lock graph defined in § 3.

Constructing the context-insensitive lock graph is non-trivial. First, in the non-nested locks, the loose correlation between the lock statements and unlock statements across complex, deep calling contexts makes inter-procedural analysis hard. Many detectors for Java programs are limited to the nested locks [2, 49]. Second, despite no need to be sensitive to the different call sites, encoding the thread information on edges should characterize different threads so as to distinguish which threads acquire the locks.

To conquer the first challenge, a context-independent function summary is devised particularly for the decoupling non-nested locks, which enables compositional analysis and avoids repetitive functional computation. Second, to distinguish which threads acquire the locks, we create clones of the lock acquisitions in the related functions that are invoked by different threads, and encode the related edges with the different thread identification.

Lock Summary. Our idea of a function summary is to capture the additional locks at the exit relative to the entry that are acquired and released therein in sequence. While analyzing any of its callers for identifying the lock acquisitions, the reusable summary can thus tell which locks can be additionally acquired and which locks can be released therein in the control-flow orders.

Example 4.1. Figure 3 shows a buggy program. There are two concurrent threads t_1 and t_2, both running the function $foo()$ and $bar()$. Intuitively, a function summary for $bar()$ should indicate that, at the exit relative to the entry, the lock o_1 is first released at s_7 and, subsequently, the lock o_1 is acquired at s_{10}. When analyzing any callers of $bar()$, we do not need to analyze $bar()$ any more.

Formally, a summary for a function f, called a lock summary, denoted as $LS(f)$, is a finite sequence, i.e.,

$$LS(f) := (L_0, \cdots, L_k),$$

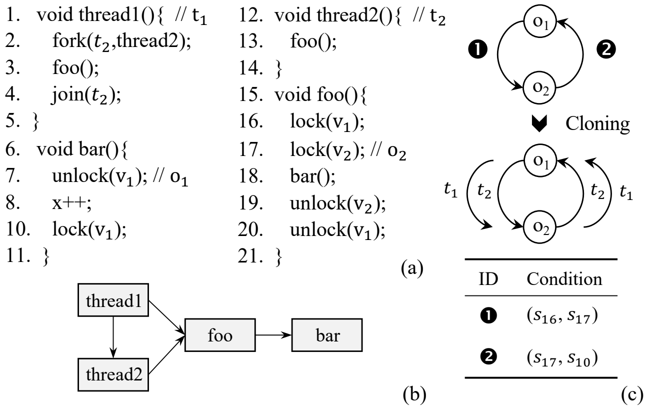
each of which is a set L_i ($i \in [0, k]$). All the lock summaries can be denoted as a set $\mathbb{LS}$, i.e., $\forall f \in \mathcal{F} : LS(f) \in \mathbb{LS}$.

Notably, first, each element L_i in the sequence $LS(f)$ is a set, because each statement, $lock(v)$, may refer to more than one lock and the lock behaviours on different branches are joined flow-sensitively. In other words, the lock summaries are path-insensitive. Second, the order between L_{i-1} and L_i indicates the control-flow orders, which matters for analyzing the non-nested locks.

Formally, for each element L_i,

$$L_i := \mathcal{R} \times \mathcal{O} \times \mathcal{S} \times \mathcal{B},$$

where $L_i := \{\langle r, o, s, b\rangle | r \in \{+, -\}, b \in \{0, 1\}\}$. First, every "plus" tuple $\langle +, o, s, b\rangle$ in L_i denotes that lock o may be acquired via executing a statement s, $lock(v)$. Whether the lock o has been released in the current function f is indicated by a boolean value b. Second, every "minus" tuple $\langle -, o, s, b\rangle$ in L_i indicates that the lock o would definitely be released via executing a statement s, $unlock(v)$. Whether the unlock statement s has released the lock o, previously acquired in the current function f, is also indicated by b. Below is an example for readers to comprehend the domain $\mathcal{B}$. Note that a lock *may be* acquired at statement s while a lock *must be* released at s, which are resolved by the pointer information. This manner is



```
1.  void thread1(){ // t1
2.    fork(t2,thread2);
3.    foo();
4.    join(t2);
5. }
6.  void bar(){
7.    unlock(v1); // o1
8.    x++;
10.   lock(v1);
11. }
12. void thread2(){ // t2
13.   foo();
14. }
15. void foo(){
16.   lock(v1);
17.   lock(v2); // o2
18.   bar();
19.   unlock(v2);
20.   unlock(v1);
21. }
```

ID	Condition
❶	(s_{16}, s_{17})
❷	(s_{17}, s_{10})

Figure 3: Figure (a) is a buggy code using non-nested locks, (b) is its call graph, and (c) illustrates the cloning process.

identical to the prior work for identifying all the lock acquisitions in common practice [36, 38, 41].

Example 4.2. Back to Figure 3, the lock summary $LS(bar)$ is computed as $(\{\langle -, o_1, s_7, 0\rangle\}, \{\langle +, o_1, s_{10}, 0\rangle\})$. First, the tuple $\langle -, o_1, s_7, 0\rangle$ indicates that the lock o_1 must be released at s_7, and its value of b is 0, implying that the lock o_1 is acquired from callers not therein. Second, the tuple $\langle +, o_1, s_{10}, 0\rangle$ shows that the lock o_1 may be acquired at s_{10} (in fact in this case, it is definitely acquired) and has not been released in the function $bar()$ indicated by $b = 0$.

Compositional Algorithm. At a high level, Algorithm 1 identifies the acquisitions and releases of locks in a bottom-up fashion by tracking and updating the lock summary for each function. During the process, the lock graph is incrementally constructed.

The intra-procedural analysis is fairly standard dataflow analysis that explores every statement of the intra-procedural control-flow graph (CFG) in the reversed post-order. At callsites, the callees' summaries are used to answer which locks are acquired and released therein. The data-flow result, the lock summary, at the exit of the function is recorded and other intermediate results are removed. For handling loops, we devise join operators to handle merge nodes in CFG to guarantee termination. Recursive functions are iteratively recomputed to reach fixed-point results. There are three key intra-procedural steps for computing $LS(f)$ and updating LG.

Handling a lock statement $s : lock(v)$ (Lines 12 - 16):

- First, the produced acquired edges at s should be identified. Peahen connects an edge (o', o) to LG from each previously acquired yet not released lock, i.e., $\langle +, o', s', b = 0\rangle$ in $LS(f)$, to each lock that may be acquired at s.
- Second, the possible acquired locks at s should be tracked in the summary $LS(f)$. Thus, Peahen adds each lock o that may be acquired at $lock(v)$, i.e., $\langle +, o, s, 0\rangle$, to $LS(f)$.

Handling an unlock statement $s : unlock(v)$ (Lines 17 - 23):

- First, Peahen obtains the definitely released lock o by identifying the variable v from the pointer information. If there is no must information, it skips the next steps.
- Second, whether the lock, previously acquired in f, is released at s should be tracked in $LS(f)$. That is, if there is a lock $\langle +, o', s', 0\rangle$ in $LS(f)$ $(o' = o)$, that is released at s, we update the tuple $\langle +, o', s', 0\rangle$ to $\langle +, o', s', 1\rangle$.
- Third, the summary $LS(f)$ should track the released lock at s. That is, if s has released the lock o previously acquired in the f on the last step, $\langle -, o, s, 1\rangle$ is added to $LS(f)$. Otherwise, $\langle -, o, s, 0\rangle$ is added to $LS(f)$.

Example 4.3. We have $LS(bar) = (\{\langle -, o_1, s_7, 0\rangle\})$ at s_7. At s_{10}, because there is no previously acquired lock o_1 indicated in $LS(bar)$, we directly have $LS(bar) = (\{\langle -, o_1, s_7, 0\rangle\}, \{\langle +, o_1, s_{10}, 0\rangle\})$. When analyzing $foo()$ at s_{16}, we have $LS(foo) = (\{\langle +, o_1, s_{16}, 0\rangle\})$. Next, at s_{17}, we have $LS(foo) = (\{\langle +, o_1, s_{16}, 0\rangle\}, \{\langle +, o_2, s_{17}, 0\rangle\})$. There is a previously acquired lock o_1 in $LS(foo)$, so we need to add an edge $e : (o_1, o_2)$,❶, to LG, which is initially encoded with $c : (s_{16}, s_{17})$.

Handling a call site $s : y = call foo(x)$ (Lines 24 - 32)

- First, the inter-procedural acquired edges produced should be identified. It connects an edge (o'', o') in LG from each currently acquired yet not released lock, $\langle +, o'', s'', 0\rangle$ in $LS(f)$, to each acquired lock, $\langle r', o', s', b\rangle$ in $LS(foo)$.
- Second, the summary $LS(f)$ should track whether some locks would be released in foo. Therefore, it updates the value b of the locks, acquired in f but released in foo, from 0 to 1. Finally, it updates $LS(f)$ by including the callee's $LS(foo)$ in the control-flow order.

Other kinds of statements do not affect the acquisitions and releases of locks and, thus, are not analyzed.

Example 4.4. In Figure 3, recall that the computed $LS(foo)$ at s_{17} is $(\{\langle +, o_1, s_{16}, 0\rangle\}, \{\langle +, o_2, s_{17}, 0\rangle\})$ and $LS(bar)$ is computed as $(\{\langle -, o_1, s_7, 0\rangle\}, \{\langle +, o_1, s_{10}, 0\rangle\})$. When analyzing the callsite $bar()$ at s_{18}, the analysis enumerates the $LS(bar)$ to check each tuple. When it is a released lock like $\langle -, o_1, s_7, 0\rangle$ and the lock o_1 also exists in $LS(foo)$, the analysis sets its boolean value to 1, i.e., $\langle +, o_1, s_{16}, 0\rangle$ becomes $\langle +, o_1, s_{16}, 1\rangle$ to indicate that the lock o_1 has been released in $bar()$. When it is an acquired lock like $\langle +, o_1, s_{10}, 0\rangle$, the analysis adds an edge to the lock graph from the acquired but unreleased lock in the current $LS(foo)$, i.e., from o_2 in $LS(foo)$ to o_1 in $LS(bar)$. We thus succeed in identifying an inter-procedural acquired edge (o_2, o_1) (❷). Finally, we add the summary $LS(bar)$ to $LS(foo)$, so $LS(foo)$ at s_{18} becomes $(\{\langle +, o_1, s_{16}, 1\rangle\}, \{\langle +, o_2, s_{17}, 0\rangle\}, \{\langle -, o_1, s_7, 1\rangle\}, \{\langle +, o_1, s_{10}, 0\rangle\})$. Note that locks o_2 acquired at s_{17} and o_1 acquired at s_{10} would be released at s_{19} and s_{20}, respectively.

Selective Cloning. After analyzing each function f, if the function f is invoked by different threads and is a root function, Peahen clones the acquired edges created in f and distinguishes which threads acquire the locks by encoding the thread identification on edge conditions. (Lines 7 - 9) For example, in Figure 3, the two threads t_1 and t_2 both run the function $foo()$ that calls $bar()$, and $foo()$ is the root shared function. We only need to clone the edges in the root function like $foo()$ to avoid repeated edges, because the edges of $bar()$ are included in the ones of $foo()$. For the non-sharing functions, we encode the edges with a unique thread ID.

Example 4.5. Following the bottom-up order of the call graph in Figure 3(b), our algorithm first analyzes the $bar()$ as well as $foo()$

Algorithm 1: Context-Insensitive LG Construction

```
$CG \longleftarrow$ construct a call graph for a multi-threaded program $P$;
foreach unvisited $f$ in reverse topological order of $CG$ do
    $(\emptyset) \longleftarrow$ initialize $LS(f)$;
    foreach $s$ in reverse post-order of CFG of $f$ do
        HandleEachInstAndPropagate($s$);
    Add $LS(f)$ at the exit program point of $f$ to $\mathbb{LS}$;
    foreach $t$, the thread $t$ invoking $f$, $t \in \mathcal{T}$ do
        foreach $e$, the edge $e$ created in $f$ do
            (clone $e$) update its condition $c$ with thread ID $t$;
return A context-insensitive lock graph $LG$;
Procedure HandleEachInstAndPropagate($s$):
    if $s : lock(v)$ then
        foreach $\langle +, o', s', b = 0 \rangle \in L_i, i \in [0, |LS(f)|)$ do
            foreach $o \in pt(v)$ do
                Add an edge $e : (o', o)$ with $c : (s', s)$ to $LG$;
        Add$\{\langle +, o, s, 0 \rangle | o \in pt(v)\}$ to $LS(f)$;
    else if $s : unlock(v)$ then
        $o \longleftarrow pt(v)$;
        if $o = o', \langle +, o', s', b' = 0 \rangle \in L_i, i \in [0, |LS(f)|)$ then
            $\langle +, o', s', 1 \rangle \longleftarrow \langle +, o', s', 0 \rangle$;
            Add $\{\langle -, o, s, 1 \rangle\}$ to $LS(f)$;
        else
            Add $\{\langle -, o, s, 0 \rangle\}$ to $LS(f)$;
    else if $s : y = call\ foo(x)$ then
        Retrieve $LS(foo) := (L'_0, \dots)$ from $\mathbb{LS}$;
        foreach $\langle +, o'', s'', 0 \rangle \in L_i, i \in [0, |LS(f)|)$ do
            foreach $\langle r', o', s', b \rangle \in L'_j, j \in [0, |LS(foo)|)$ do
                if $r' = +$ then
                    Add edge $e : (o'', o'), c : (s'', s')$ to $LG$;
                else if $r' = - \wedge o'' = o'$ then
                    $\langle +, o'', s'', 1 \rangle \longleftarrow \langle +, o'', s'', 0 \rangle$;
        Add $LS(foo)$ to $LS(f)$;
```

and updates the lock graph with two corresponding acquired edges, (o_1, o_2) and (o_2, o_1). Then, since the $foo()$ is the root function invoked by two different threads, t_1 and t_2, the analysis clones the two edges with two different thread identifiers t_1 and t_2 to distinguish which threads acquire these locks. Armed with the lock graph, Peahen can start by detecting the deadlock cycle.

Remark. In Algorithm 1, a key step that is omitted to reach context-insensitive is aggressively cloning the edges of lock acquisitions in a function on all its call sites and labeling the edges with the respective calling contexts. Instead, we selectively clone for a function invoked by different threads and encode the lock-acquisition information on each edge. This is because Peahen is only concerned with the cyclic lock acquisitions in each thread. In this way, the lazy refinement can incorporate such information into lazily refining deadlock cycles around a few calling contexts of interest, thereby bypassing premature calling context explosion.

Summary. The effectiveness of our LG construction is twofold. First, it is compositional, efficiently analyzing each function independently of its callers and bypassing the enumeration of all the calling contexts. The context-insensitive lock graph is smaller yet characterizes threads, which is judiciously encoded with the essential lock-acquisition information for lazy refinements. Second, the smaller graph empowers fast cycle detection and less memory space to store. In the subsequent phase, our refinements can focus on the computation effort around the fewer calling contexts related to the context-insensitive cycles to regain high precision.

4.2 Collaborative Lazy Deadlock Refinements

Once the lock graph is in hand, Peahen conducts a cycle-detection algorithm [31] to identify an initial over-approximation set of deadlock cycles. Then, it performs the following three analyses that approximate the different necessary conditions of being deadlocks, dedicatedly refining these initial cycles. Readers can review the cycle definitions in § 3.

(1) Single-, Multi-threaded Cycle Computation (§ 4.2.1)
(2) Concurrent Cycle Computation (§ 4.2.2)
(3) Path-Feasible Cycle Computation (§ 4.2.3)

Note that every stage exploits the edge information on each cycle, refines the cycles from the preceding one in a lazy fashion, and avoids any futile computation to acyclic edges.

4.2.1 *Single-, Multi-threaded Cycle Computation.*

At this stage, the set of the initial cycles, denoted as $\mathbb{C}$ ($\circ \in \mathbb{C}$), is refined as the set $\mathbb{C}^1$, the union of single-threaded cycles $\mathbb{C}^1_1$ and multi-threaded cycles $\mathbb{C}^1_2$. Let the function $T(\circ)$ return the set of all thread IDs on edges in a cycle $\circ$, $N(\circ)$ return the set of all nodes involved in a cycle $\circ$, and $E(\circ)$ return all the edges in a cycle $\circ$.

$$\mathbb{C}^1_1 = \{\circ | \exists \circ \in \mathbb{C} : |E(\circ)| = 1 \wedge |N(\circ)| = 1\}$$
$$\mathbb{C}^1_2 = \{\circ | \exists \circ \in \mathbb{C}, \forall t_1, t_2 \in T(\circ) : t_1 \neq t_2\}$$
$$\mathbb{C}^1 = \mathbb{C}^1_1 \cup \mathbb{C}^1_2$$

Indeed, owing to the encoded thread information on edges foreseeingly by the LG construction, this first refinement is simple by efficiently checking the edge conditions (and the number of nodes).

4.2.2 *Concurrent Cycle Computation.*

At this stage, the multi-threaded cycles, $\mathbb{C}^1_2$, are refined by identifying and retaining concurrent cycles, which, together with $\mathbb{C}^1_1$, are denoted as $\mathbb{C}^2$.

We perform non-concurrency analysis to reveal, given a multi-thread cycle, whether the statements that refer to the destination-lock nodes can be executed concurrently, arising from the fork/join and lock/unlock synchronizations. At a high level, our basic idea is to lazily collect the thread IDs referred by the join statements and the unreleased locks referred by the lock statements *before* an interesting statement s under the related calling contexts.

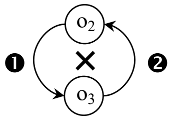
Specifically, each join statement, $join(t)$, witnesses the destruction of thread t before s, and, thus, any other statements in the thread t cannot be concurrent with s. Similarly, an unreleased lock o referred by a lock statement, $lock(v)$, reveals that the lock o protects the statement s. Therefore, any other statements also protected by the common lock o cannot be concurrent with s.

Peahen performs a backward search from any interesting statement s, following the inter-procedural context-sensitive control-flow graph to collect the set of joined thread IDs, $JT(s, ctx)$, and the

```
1.  void thread1(){ // t1
2.     fork(t2, thread2);
3.     lock(v1); // o1
4.     lock(v2); // o2
5.     if (θ1){
6.        lock(v3); // o3
7.        x++;
8.        unlock(v3);}
9.     unlock(v2);
10.    unlock(v1);
11.    join(t2);
12. }
```

```
13. void thread2(){ // t2
14.    lock(v1);
15.    lock(v3);
16.    if (¬θ1){
17.       lock(v2);
18.       x++;
19.       unlock(v2);
20.    }
21.    unlock(v3);
22.    unlock(v1);
23. }
```

(a)



ID	Condition
❶	(t_1, s_4, s_6)
❷	(t_2, s_{15}, s_{17})

(b)

Figure 4: An example for the three lazy refinements.

set of unreleased locks, $UL(s, ctx)$, reachable from the related calling contexts ctx. The search and collection processes themselves are similar to the previous lockset analysis [38, 48, 49], but are performed lazily. This is because we are aware of the related statements, where the destination locks are acquired, owing to the edge information. Formally, let the $D(\circ)$ return all the lock statements that refer to the destination locks in a cycle $\circ$.

$$\mathbb{C}_1^2 = \{\circ \mid \exists s_1 \in D(\circ), \forall ctx_1, \exists t_1 \in JT(s_1, ctx_1) : t_1 \in T(\circ)\}$$

$$\mathbb{C}_2^2 = \{\circ \mid \forall s_1, s_2 \in D(\circ), \forall ctx_1, ctx_2 : UL(s_1, ctx_1) \cap UL(s_2, ctx_2) \neq \emptyset\}$$

$$\mathbb{C}^2 = \{\circ \mid \circ \in \mathbb{C}_2^1 \land \circ \notin \mathbb{C}_1^2 \land \circ \notin \mathbb{C}_2^2\} \cup \mathbb{C}_1^1$$

Set $\mathbb{C}_1^2$ indicates that at least a thread t_1 involved in a multi-threaded cycle is destroyed. The set $\mathbb{C}_2^2$ indicates that at least two statements s_1 and s_2 (refer to the destination locks) involved in a multi-threaded cycle are guarded by a common lock. A multi-threaded cycle with either of these circumstances is then removed.

4.2.3 *Path-Feasible Cycle Computation*.

Finally, Peahen performs path validation to identify path-feasible deadlock cycles. Our basic idea is to validate whether the related statements involved in a cycle are path-feasible simultaneously. Let $\mathcal{P}(s)$ ($p \in \mathcal{P}$) be all the intra-procedural paths leading to a statement s and $\Phi(p)$ indicate the path conditions on p. Let function $S(\circ)$ return the encoded lock statements, referring to the source lock and the destination lock, on acquired edges in an interesting cycle $\circ$.

$$\mathbb{C}^3 = \{\circ \mid \exists \circ \in \mathbb{C}^2, \forall s \in S(\circ), \exists p \in \mathcal{P}(s) : \Phi(p) = True\}$$

While validating path feasibility is expensive in general due to the need for SMT solving [65], we observe that, for deadlock detection, it can be efficient because of the following observations. First, the number of deadlock cycles to be proved path-feasible are much fewer, owing to the successive sieving by the previous two stages. Second, we observe that expensive solving for the full program paths is not necessary for practical deadlock detection, so Peahen only solves the intra-procedural paths for high efficiency (also discussed in the evaluation § 5).

Example 4.6. Figure 4 shows an initially detected cycle with edges ❶ and ❷, because there is a reversed order between the locks o_2 and o_3. It is a multi-threaded cycle since it is induced by different threads t_1 and t_2 through analyzing the conditions on edges. In non-concurrency analysis, Peahen starts to collect the join statements before statements s_6 and s_{17} to find the joined threads, which are empty. Then, Peahen collects the common unreleased lock before statements s_6 and s_{17} that is o_1. Thus, the cycle is not a concurrent cycle. It is also not a path-feasible cycle due to the contradicted path conditions between s_6 and s_{17}. However, we do not solve the constraints, thanks to the preceding non-concurrency analysis.

Summary. Our refinements confer two salient advantages to our deadlock detection. First, our refinements are lazy, which are aware of the interesting calling contexts around the initially detected cycles and thus avoid performing pre-computed whole-program context-sensitive computation. Note that it is the result of the synergy between the LG construction and the lazy refinements that ensures the effectiveness of Peahen. Second, our refinements are precise, regaining the high precision for the initial context-insensitive deadlock cycles. More importantly, as the refining deadlock cycles become more and more precise, fewer cycles lead to less lazy computation. Previous work [16, 42, 43, 49, 62] does not consider the path-feasible deadlocks, which is taken into account by Peahen.

5 EVALUATION

To evaluate Peahen, we consider the following research questions:

Q1 How efficient and effective is Peahen compared to the previous static deadlock detectors? (§ 5.1)

Q2 Can Peahen scale up to million-line systems? Can Peahen detect genuine deadlocks? (§ 5.2)

All the experiments are performed on a computer with two 20 core Intel (R) Xeon (R) CPU E5-2698 v4 @ 2.20GHz and 256GB physical memory running Ubuntu-16.04.

Implementation. Peahen is implemented for C/C++ programs on top of the LLVM compiler infrastructure and the Z3 SMT solver. For precision, we implement on-demand field-, flow-, and context-sensitive pointer analysis [55]. Specifically, Peahen can leverage context-sensitive pointer information in the refinement stage, where the related calling contexts of lock objects are inferred.

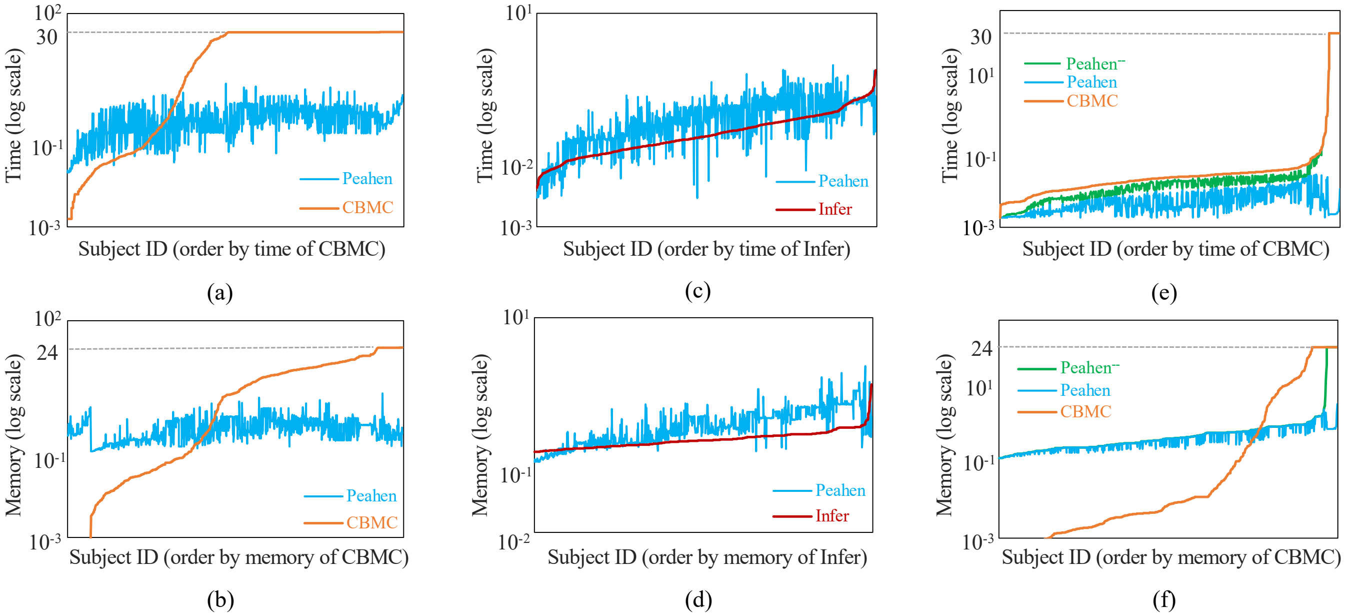
Following many well-known bug finding tools [34, 48, 55, 57, 64], our implementation is soundy [45], which means that our tool is mostly sound, aside from a few well-identified reasonable unsound choices for achieving higher precision. Particularly, there are two sources of the unsoundness in our implementations. First, our pointer analysis shares the same unsoundness sources as the one we use. For instance, it does not correctly handle the pointer arithmetic, array accesses, containers, and so on. Second, our lock graph construction ignores the locks that are inside blocks of the assembly code as the prior deadlock detectors [2, 41, 49].

5.1 Q1: Comparing with Previous Techniques

The goal of this experiment is to show that the prior context-sensitive approach is unscalable while the context-insensitive one is imprecise. Comparatively, Peahen provides a better sweet spot in the trade-off between precision and performance.

We compared Peahen against two recent and relevant tools for the non-nested, non-reentrant locks: CBMC [42] and Infer.

- CBMC is context-sensitive, constructing a context-sensitive lock graph through a pre-computed context-sensitive lockset analysis and using the results of lockset analysis again to refine the deadlock cycles discovered in the lock graph.



(a) (c) (e)

(b) (d) (f)

Figure 5: (a)(b)(c)(d) shows the time (min) and memory (GB) comparison in entire deadlock detection among Peahen, CBMC, and Infer. (e)(f) shows the time (min) and memory (GB) comparison in LG construction among Peahen, CBMC, and Peahen$^-$.

- Infer is an abstract-interpretation-based context-insensitive deadlock detector, reasoning locks by syntactic expressions. We use this version [22, 26] for supporting Pthread APIs.

Benchmark. To reduce the subjectivity, the chosen benchmarks are collected from CBMC [41]. The benchmarks consist of 1005 diverse programs from the Debian GNU/Linux distribution (e.g., Fuse, Glfw, Libmicrohttpd), ranging in size from a few KLoC to 52 KLoC with a total of *11.3 MLOC*. Additionally, they intentionally planted 8 programs with deadlocks [42].

Following their evaluation [42], memory and time are restricted to 24 GB and 30 minutes per program, respectively.

5.1.1 *Scalability.*

We check whether each tool can successfully analyze all programs within the time and memory budget. First, CBMC spent about 16297.27 minutes (around 272 hours) with 526 programs out of time and 78 programs out of memory. Comparatively, Peahen succeeded in checking within 119.36 minutes (around 2 hours), achieving 136 × speed up. Second, Infer finished checking all these programs within 73.11 minutes (around 1.2 hours), which is slightly faster than Peahen.

In detail, shown in Figure 5 (a)(b), for the programs analyzed by CBMC with more than one minute and one GB, Peahen significantly outperforms CBMC. It demonstrates our approach is efficient for large programs. Shown in Figure 5 (c)(d), compared to Infer, Peahen is also competitive with many projects finished at slightly more memory and time cost. This cost is reasonable since Peahen is more precise with context-sensitivity as shown below.

5.1.2 *Precision.*

The experimental results show that Infer, CBMC, and Peahen report 301, 113, and 30 issues, respectively. To sum up, Peahen can reduce around 73.45% and 90.0% of false positives compared to CBMC and Infer, respectively.

In detail, we manually checked each generated report and made three conclusions. First, we found that the bugs flagged by Peahen are also detected by CBMC and Infer. Second, CBMC and Infer generate excessive spurious bugs. Specifically, after checking the bugs generated by CBMC and Infer that are not detected by Peahen, we found that they are false positives. For 30 bugs generated by Peahen, only 8 of them are deemed to be false after manual confirmation. Third, Peahen can detect the eight intentionally-introduced deadlocks, which suggests that the precision of Peahen is achieved without sacrificing the recall.

Peahen is more precise owing to the two reasons. First, Peahen conducts more precise refinements against Infer and CBMC. For example, Figure 6 shows a false positive in PDSH detected by Infer and CBMC, where the cyclic lock acquisitions between the locks *src* and *dst* are on the contradictory paths and cannot be simultaneously path-feasible. Hence, when multiple threads execute the function *cbuf_copy*, a deadlock could not occur. Second, in practice, the lock objects could be derived from structures as fields, passed in as parameters, and reassigned at different locations of control flow. The pointer analysis we implemented is field-, flow- and context-sensitive, which can aid in obviating certain false positives.

5.1.3 *Effectiveness on Two Collaborative Stages.*

Compared to CBMC, we study the effectiveness of Peahen to reach context-sensitive through two collaborative stages - the context-insensitive lock graph construction and the precise yet lazy refinements.

Lock Graph Construction. First, we compare the time and memory cost of each program to construct the lock graph. To evaluate our context-reduction technique and exclude the side effects from different implementations of pointer analyses, we implemented Peahen$^-$, which constructs a context-sensitive lock graph

```
int cbuf_copy (cbuf_t src, cbuf_t dst, int len, int *ndropped) {
// Lock cbufs in order of lowest memory address to prevent deadlock.
        if (src < dst) {
                cbuf_mutex_lock(src);
                cbuf_mutex_lock(dst);
        } else {
                cbuf_mutex_lock(dst);
                cbuf_mutex_lock(src);
        }
}
```

Figure 6: A false positive detected by other tools.

by cloning the acquired edges for all the calling contexts during our compositional algorithm (Algorithm 1).

The results are illustrated in Figure 5 (e)(f), with the programs running out of time during the pointer analysis of CBMC removed. Figure 5 (e) shows that the proposed algorithm in Peahen is consistently faster than Peahen$^-$ and CBMC, respectively achieving about 230 × and 231 × speed-up, on average. For this reason, Peahen does not enumerate any exponentially possible calling contexts spanning across the lock acquisitions.

Figure 5 (f) illustrates that, dealing with the programs analyzed by CBMC and Peahen$^-$ with more than one GB, Peahen significantly outperforms Peahen$^-$ and CBMC in terms of memory cost, saving about 23 GB memory at most. We manually checked these programs and found they intensively use locks so that a context-sensitive LG is expensive to construct.

Finally, we studied the number of acquired edges on the buggy programs. This reveals that our context-insensitive lock graph can merge about 95.5% edges on average that would otherwise be identified and added to the context-sensitive ones of Peahen$^-$ and CBMC. Moreover, the results also reveal that the cycle-detection algorithm of Peahen working on the context-insensitive lock graph achieves about 46× speed-up on average.

Lazy Refinements. Next, we study the effectiveness of the refinements. It is disclosed that our refinements can prune away 43 deadlock cycles in about 65 minutes, reducing about 58.9% of false warnings. Three refinements can progressively prune away about 35%, 40%, and 25% false warnings, respectively.

Answer to Q1: Armed with our context-reduction approach, Peahen achieves higher precision without sacrificing good scalability versus the state-of-the-art approaches.

5.2 Q2: Checking Large-Scale Projects

We evaluate the effectiveness of Peahen for large programs. We set a high bar and selected four additional real-world projects based on two principles. First, the project should be million-line and widely-used in practice. Second, the projects should be using mutex locks intensively. As a result, the projects we chose comprise four real-life open-source C/C++ projects shown in Table 1, ranging in size from around 1.75 MLoC to nearly 9 MLoC.

5.2.1 ***Scalability and Precision.*** For evaluating the scalability, we checked how much time and memory is spent conducting the entire deadlock detection. The results are shown in Table 1, suggesting that Peahen succeeded in checking Firefox (with nearly 9 MLoC) in around two hours and within 50 GB memory.

ID	Project Name	Size (KLoC)	Deadlock Detection		Bug Report	
			Memory (GB)	Time (min)	#All	#TP
1	MariaDB	1,753	27.97	34.85	11	8
2	FFmpeg	2,065	37.41	39.75	5	4
3	MySQL	3,185	31.17	34.17	3	2
4	Firefox	8,910	45.90	120.30	4	2

Table 1: Q2: Experimental results on large programs.

For precision, we manually checked each error generated by Peahen and then identified its false positive rate. The results show that Peahen generated 23 deadlock warnings with 7 false positives, resulting in a 30.43% false-positive rate. This process may be subjective and pose threats to the validity of its precision. Therefore, we also sought the confirmations from the original developers and sent the patches to ease their work. At the time of writing, seven of the bugs have been confirmed by them.

Thanks to its precise refinements, Peahen can prune away numerous false warnings. For false positives, a portion of them are induced by the imprecise pointer alias relations (e.g., infeasible paths, function pointers), inducing the imprecise lock acquisitions. Another source comes from ignoring other uninterpreted synchronizations (e.g., waits/notifies, ad-hoc synchronizations), inducing the imprecise concurrent execution relations. To overcome these limitations, Peahen needs to adopt more precise pointer analysis and characterize more synchronizations.

5.2.2 ***Confirmed Real-World Deadlocks.*** We have been using Peahen to scan open-source projects extensively and continuously. At the time of writing, there were already more than eighty confirmed deadlock bugs from dozens of the famous systems [50], despite many of which are consistently and intensively undergoing bug checking. Most of the bugs have been fixed in the recent release versions of the software, making them more reliable and robust. Next, we study two real deadlocks in Figure 7.

A deadlock is detected in FreeIPA, the upstream project for Red Hat Identify Management. If we take a closer look, the bug pattern is quite simple, where the statement at Line 2199 should be an unlock statement. However, this bug has been missed by the developers, users as well as dynamic testing and has existed for about eight months. The developers were very concerned with this bug, noting that "This code is used quite frequently by FreeIPA, why have we not seen a problem with it until now." It demonstrates that static analysis matters, and, of course, scaling up to this large program is also critical, which is addressed in this paper.

A deadlock is uncovered in AML, a memory management library for high-performance computing applications. A deadlock occurs when a thread t_1 reaches Line 93 while another thread t_2 reaches Line 264, relating to the path conditions of lock statements. With the path-sensitive analysis, Peahen can precisely uncover it.

Answer to Q2: Peahen is efficient for million-line systems, flagging deadlocks at a low false positive rate. In terms of real-world impacts, it has found many confirmed deadlocks.

```
2149    static void check_replicas_are_done_cleaning(cleanruv_data *data) {
2193            if (!slapi_is_shutting_down())
2197                    pthread_mutex_lock(&notify_lock);
. . .
2199                    pthread_mutex_lock(&notify_lock);
2208    }
```

```
82.     int aml_active_sched_num_tasks (struct aml_sched *sched){
91              if (Condition1){
92                      pthread_mutex_lock(&s->workq_lock);
93                      pthread_mutex_lock(&s->doneq_lock);
105             }
106     }
253     struct aml_task *aml_active_sched_wa(struct aml_sched_data *data){
258             pthread_mutex_lock(&(sched->doneq_lock));
263             if (Condition2){
264                     pthread_mutex_lock(&(sched->workq_lock));
270             }
272     }
```

Figure 7: Real deadlocks in FreeIPA and AML respectively.

6 RELATED WORK

Ensuring the reliability of concurrent programs is an important problem [9, 24, 27, 43, 51, 52, 63]. As integral parts, deadlock detection, prevention [20, 29, 30, 35, 60], and fixing [3] are diverse directions, among which we focus on surveying detection.

6.1 Static Deadlock Detection

Detectors for Java code typically rely on balanced locking. This work [62] targets Java libraries and constructs a global context-sensitive lock graph. In general, their tool can report the false positives due to the non-concurrency and path-infeasible deadlocks. JADE [49] is object-sensitive, which breaks down the problem into several context-sensitive sub-tasks, including reachability, may-happen-in-parallel, non-reentrant, and non-guarded analyses. JADE focuses on two-thread deadlocks and could report spurious path-infeasible deadlocks. Brotherston et al. [2] propose a context-insensitive incremental detector for Android Java, detecting deadlocks in an abstract-interpretation style and analyzing code changes with respect to the unchanged portions. Their tool may induce excessive false positives without context-insensitive analyses.

Detectors for C code do not expect balanced locking, which significantly complicates static deadlock detection [14, 36, 37, 56]. LockLint [12] is not fully-automatic such that it requires annotations from users to specify the lock acquisition orders. RacerX [16] reaches context-sensitive by caching lockset results at the function level, which may lead to ill scalability in large systems. In addition, compared to Peahen, RacerX uses the syntactic and type information to reason pointer aliasing, require annotations, and employs some unsound heuristics to reduce false positives. Kroening et al. [42] proposes a detector for Pthread APIs, which conducts the context- and thread-sensitive lockset analysis for the lock-graph construction and the cycle refinements. However, as mentioned before, context-sensitive exhaustive lockset analysis and lock-graph construction are expensive.

Some work [47] targets for the programs using data-centric synchronizations [15, 21] and requires the manual annotations to identify the ordering between atomic-sets.

6.2 Dynamic Deadlock Detection

Many testing approaches and model checkers can manifest deadlocks [10, 11, 19, 28, 59, 61]. We focus on investigating recent predictive analysis, which has been applied to detect a variety of concurrency bugs such as data races [25, 53, 58] and deadlocks. Specifically, deadlock prediction can be categorized into two classes in terms of soundness (i.e. the absence of false positives).

Unsound deadlock prediction may induce false positives due to, for instance, ignoring happens-before relations. Thus, they commonly need to be integrated with other techniques via scheduling a real deadlock [32, 34, 54] or identifying and solving execution constraints [6, 17]. At the core of the prediction, they construct a lock graph by tracking the lock-acquisition history from just a single execution, and identify the cycles. To relax the deadlock-prediction overhead, many seminal approaches [4, 5, 66] have been proposed. The latest one, AirLock [5], speeds up the online cycle discovery via first finding “simple cycles” without considering any execution information (e.g, threads) and then constructing deadlock cycles by taking full execution information into account.

Note that, Peahen and AirLock have similar flavors in the sense of reducing lock graphs by simplifying the edges and efficiently discovering cycles. Nevertheless, Peahen addresses many challenges that are unique to static deadlock detection, thereby differing in many aspects. For example, instead of constructing the lock graph from one execution, Peahen addresses the calling context explosion induced by over-approximating all possible executions. In addition, unlike AirLock, Peahen is armed with the refinement process to validate the feasibility of deadlocks (e.g., considering the happens-before relation), whereby embracing high fidelity.

Another category of sound work never induces false positives with the theoretical guarantees. Specifically, Dirk [39] uses request events and a new form of execution constraints to disclose real deadlocks via solving all these constraints. SeqCheck [8] models the program branches in the execution traces and predicts the feasibility of event sequences for sound deadlock detection.

7 CONCLUSION

We have introduced Peahen, a context-reduction technique making static deadlock detection more scalable and precise. Peahen is quite promising, having already pinpointed many previously-unknown deadlocks on a dozen of well-known software systems. We believe that context reduction is a big step forward in deadlock detection, which ameliorates the major current pain point in practical deployment of deadlock detectors. We expect that Peahen can provide interesting insights into aiding in other static analyses to become more scalable with high precision for industrial-scale software.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work was partially funded by the RGC16206517 and ITS/440/18FP grants from the Hong Kong Research Grant Council, Donations from Microsoft and Huawei, and the Ant Research Program from Ant Group. Qingkai Shi is the corresponding author of this paper. This work was finished when Qingkai Shi was with Ant Group. He is currently with Purdue University and is available via email at shi553@purdue.edu.

REFERENCES

[1] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 259–269. https://doi.org/10.1145/2594291.2594299

[2] James Brotherston, Paul Brunet, Nikos Gorogiannis, and Max Kanovich. 2021. A Compositional Deadlock Detector for Android Java. In *Proceedings of ASE-36*. ACM.

[3] Yan Cai and Lingwei Cao. 2016. Fixing deadlocks via lock pre-acquisitions. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 1109–1120. https://doi.org/10.1145/2884781.2884819

[4] Yan Cai and W. K. Chan. 2012. MagicFuzzer: Scalable deadlock detection for large-scale applications. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, Martin Glinz, Gail C. Murphy, and Mauro Pezzè (Eds.). IEEE Computer Society, 606–616. https://doi.org/10.1109/ICSE.2012.6227156

[5] Yan Cai, Ruijie Meng, and Jens Palsberg. 2020. Low-overhead deadlock prediction. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1298–1309. https://doi.org/10.1145/3377811.3380367

[6] Yan Cai, Shangru Wu, and W. K. Chan. 2014. ConLock: a constraint-based approach to dynamic checking on deadlocks in multithreaded programs. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 491–502. https://doi.org/10.1145/2568225.2568312

[7] Yuandao Cai, Peisen Yao, and Charles Zhang. 2021. 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, 20211*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 1126–1140. https://doi.org/10.1145/3453483.3454099

[8] Yan Cai, Hao Yun, Jinqiu Wang, Lei Qiao, and Jens Palsberg. 2021. Sound and efficient concurrency bug prediction. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, Diomidis Spinellis, Georgios Gousios, Marsha Chechik, and Massimiliano Di Penta (Eds.). ACM, 255–267. https://doi.org/10.1145/3468264.3468549

[9] Milind Chabbi and Murali Krishna Ramanathan. 2022. A Study of Real-World Data Races in Golang. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 474–489. https://doi.org/10.1145/3519939.3523720

[10] Dongjie Chen, Yanyan Jiang, Chang Xu, Xiaoxing Ma, and Jian Lu. 2018. Testing multithreaded programs via thread speed control. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 15–25. https://doi.org/10.1145/3236024.3236077

[11] Hongxu Chen, Shengjian Guo, Yinxing Xue, Yulei Sui, Cen Zhang, Yuekang Li, Haijun Wang, and Yang Liu. 2020. 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*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 2325–2342. https://www.usenix.org/conference/usenixsecurity20/presentation/chen-hongxu

[12] Oracle Corporation. [n.d.]. LockLint Overview. https://docs.oracle.com/cd/E19059-01/wrkshp50/805-4947/6j4m8jrng/index.html.

[13] CVE. [n.d.]. CVE List. https://cve.mitre.org/cve/search_cve_list.html.

[14] Peng Di, Yulei Sui, Ding Ye, and Jingling Xue. 2015. 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*. IEEE Computer Society, 889–898. https://doi.org/10.1109/ICPP.2015.98

[15] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. 2012. A data-centric approach to synchronization. *ACM Trans. Program. Lang. Syst.* 34, 1 (2012), 4:1–4:48. https://doi.org/10.1145/2160910.2160913

[16] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[17] Mahdi Eslamimehr and Jens Palsberg. 2014. Sherlock: scalable deadlock detection for concurrent programs. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 353–365. https://doi.org/10.1145/2635868.2635918

[18] Firefox. [n.d.]. Bugzilla. https://bugzilla.mozilla.org/home.

[19] Patrice Godefroid. 1997. Model Checking for Programming Languages using Verisoft. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, Peter Lee, Fritz Henglein, and Neil D. Jones (Eds.). ACM Press, 174–186. https://doi.org/10.1145/263699.263717

[20] Mark Grechanik, B. M. Mainul Hossain, Ugo Buy, and Haisheng Wang. 2013. Preventing database deadlocks in applications. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 356–366. https://doi.org/10.1145/2491411.2491412

[21] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. 2008. Dynamic detection of atomic-set-serializability violations. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 231–240. https://doi.org/10.1145/1368088.1368120

[22] Dominik Harmim, Vladimir Marcin, and Ondrej Pavela. 2019. Scalable Static Analysis Using Facebook Infer.

[23] Klaus Havelund. 2000. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification, 7th International SPIN Workshop, Stanford, CA, USA, August 30 - September 1, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1885)*, Klaus Havelund, John Penix, and Willem Visser (Eds.). Springer, 245–264. https://doi.org/10.1007/10722468_15

[24] Jeff Huang. 2018. UFO: predictive concurrency use-after-free detection. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 609–619. https://doi.org/10.1145/3180155.3180225

[25] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 337–348. https://doi.org/10.1145/2594291.2594315

[26] Infer. [n.d.]. Scalable Static Analysis Using Facebook Infer. https://github.com/vmarcin/L2D2.

[27] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding Kernel Race Bugs through Fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP.2019.00017

[28] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. 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*. IEEE, 754–768. https://doi.org/10.1109/SP.2019.00017

[29] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. 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*, Mary W. Hall and David A. Padua (Eds.). ACM, 389–400. https://doi.org/10.1145/1993498.1993544

[30] Guoliang Jin, Wei Zhang, and Dongdong Deng. 2012. Automated Concurrency-Bug Fixing. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 221–236. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/jin

[31] Donald B. Johnson. 1975. Finding All the Elementary Circuits of a Directed Graph. *SIAM J. Comput.* 4, 1 (1975), 77–84. https://doi.org/10.1137/0204007

[32] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5643)*, Ahmed Bouajjani and Oded Maler (Eds.). Springer, 675–681. https://doi.org/10.1007/978-3-642-02658-4_54

[33] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. 2010. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and André van der Hoek (Eds.). ACM, 327–336. https://doi.org/10.1145/1882291.1882339

[34] Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. 2009. A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 110–120. https://doi.org/10.1145/1542476.1542489

[35] Horatiu Jula, Daniel M. Tralamazza, Cristian Zamfir, and George Candea. 2008. Deadlock Immunity: Enabling Systems to Defend Against Deadlocks. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 295–308. http://www.usenix.org/events/osdi08/tech/full_papers/jula/jula.pdf

[36] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. 2009. 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*, Hans van Vliet and Valérie Issarny (Eds.). ACM, 13–22. https://doi.org/10.1145/1595696.1595701

[37] Vineet Kahlon and Chao Wang. 2010. Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6174)*, Tayssir Touili, Byron Cook, and Paul B. Jackson (Eds.). Springer, 434–449. https://doi.org/10.1007/978-3-642-14295-6_39

[38] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. 2007. 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 (Lecture Notes in Computer Science, Vol. 4590)*, Werner Damm and Holger Hermanns (Eds.). Springer, 226–239. https://doi.org/10.1007/978-3-540-73368-3_26

[39] Christian Gram Kalhauge and Jens Palsberg. 2018. Sound deadlock prediction. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 146:1–146:29. https://doi.org/10.1145/3276516

[40] Linux Kernel. [n.d.]. Bugzilla. https://bugzilla.kernel.org/.

[41] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. [n.d.]. *Sound Static Deadlock Analysis for C/Pthreads.* http://www.cprover.org/deadlock-detection/

[42] Daniel Kroening, Daniel Poetzl, Peter Schrammel, and Björn Wachter. 2016. Sound Static Deadlock Analysis for C/Pthreads. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 379–390. https://doi.org/10.1145/2970276.2970309

[43] Bozhen Liu and Jeff Huang. 2018. 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*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 359–373. https://doi.org/10.1145/3192366.3192390

[44] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. 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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 725–739. https://doi.org/10.1145/3453483.3454073

[45] Benjamin Livshits, Dimitrios Vardoulakis, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, José Amaral, Bor-Yuh Chang, Samuel Guyer, Uday Khedker, and Anders Møller. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58 (01 2015), 44–46. https://doi.org/10.1145/2644805

[46] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. 2008. 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*, Susan J. Eggers and James R. Larus (Eds.). ACM, 329–339. https://doi.org/10.1145/1346281.1346323

[47] Daniel Marino, Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, and Jan Vitek. 2013. Detecting deadlock in programs with data-centric synchronization. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 322–331. https://doi.org/10.1109/ICSE.2013.6606578

[48] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 308–319. https://doi.org/10.1145/1133981.1134018

[49] Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. 2009. Effective static deadlock detection. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 386–396. https://doi.org/10.1109/ICSE.2009.5070538

[50] Pinpoint Platform. [n.d.]. Confirmed Bug List. https://whichbug.github.io/.

[51] Boqin Qin, Yilun Chen, Zeming Yu, Linhai Song, and Yiying Zhang. 2020. Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 763–779. https://doi.org/10.1145/3385412.3386036

[52] Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 747–762. https://doi.org/10.1145/3385412.3385993

[53] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem A. Sakallah. 2011. Generating Data Race Witnesses by an SMT-Based Analysis. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 313–327. https://doi.org/10.1007/978-3-642-20398-5_23

[54] Koushik Sen. 2008. 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*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 11–21. https://doi.org/10.1145/1375581.1375584

[55] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. 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*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 693–706. https://doi.org/10.1145/3192366.3192418

[56] Yulei Sui, Peng Di, and Jingling Xue. 2016. 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*, Björn Franke, Youfeng Wu, and Fabrice Rastello (Eds.). ACM, 160–170. https://doi.org/10.1145/2854038.2854043

[57] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, Mats Per Erik Heimdahl and Zhendong Su (Eds.). ACM, 254–264. https://doi.org/10.1145/2338965.2336784

[58] Chao Wang, Sudipta Kundu, Rhishikesh Limaye, Malay K. Ganai, and Aarti Gupta. 2011. Symbolic predictive analysis for concurrent programs. *Formal Aspects Comput.* 23, 6 (2011), 781–805. https://doi.org/10.1007/s00165-011-0179-2

[59] Chao Wang, Mahmoud Said, and Aarti Gupta. 2011. Coverage guided systematic concurrency testing. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic (Eds.). ACM, 221–230. https://doi.org/10.1145/1985793.1985824

[60] Yin Wang, Terence Kelly, Manjunath Kudlur, Stéphane Lafortune, and Scott A. Mahlke. 2008. Gadara: Dynamic Deadlock Avoidance for Multithreaded Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 281–294. http://www.usenix.org/events/osdi08/tech/full_papers/wang/wang.pdf

[61] Cheng Wen, Mengda He, Bohao Wu, Zhiwu Xu, and Shengchao Qin. 2022. Controlled Concurrency Testing via Periodical Scheduling. In *Proceedings of the ACM/IEEE 4th International Conference on Software Engineering* (Pittsburgh, USA) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3510003.3510178

[62] Amy L. Williams, William Thies, and Michael D. Ernst. 2005. Static Deadlock Detection for Java Libraries. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3586)*, Andrew P. Black (Ed.). Springer, 602–629. https://doi.org/10.1007/11531142_26

[63] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. *2020 IEEE Symposium on Security and Privacy (SP)* (2020), 1643–1660.

[64] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. 2018. 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*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 327–337. https://doi.org/10.1145/3180155.3180178

[65] Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast bit-vector satisfiability. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 38–50. https://doi.org/10.1145/3395363.3397378

[66] Jinpeng Zhou, Sam Silvestro, Hongyu Liu, Yan Cai, and Tongping Liu. 2017. 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*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 729–740. https://doi.org/10.1109/ASE.2017.8115684

[67] Jinpeng Zhou, Hanmei Yang, John Lange, and Tongping Liu. 2022. Deadlock Prediction via Generalized Dependency. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 455–466. https://doi.org/10.1145/3533767.3534377