Deadlock Detection: Background, Techniques, and Future Improvements

LU Jiachen[1], NIU Zhi[2], CHEN Li[2], DONG Luming[2], SHEN Taoli[1]

(1. Zhejiang University, Hangzhou 310058, China;
2. ZTE Corporation, Xi'an 710114, China)

DOI: 10.12142/ZTECOM.202402009

https://kns.cnki.net/kcms/detail/34.1294.TN.20240604.1720.002.html, published online June 5, 2024

Manuscript received: 2023-12-19

Abstract: Deadlock detection is an essential aspect of concurrency control in parallel and distributed systems, as it ensures the efficient utilization of resources and prevents indefinite delays. This paper presents a comprehensive analysis of the various deadlock detection techniques, including static and dynamic approaches. We discuss the future improvements associated with deadlock detection and provide a comparative evaluation of these techniques in terms of their accuracy, complexity, and scalability. Furthermore, we outline potential future research directions to improve deadlock detection mechanisms and enhance system performance.

Keywords: deadlock detection; static analysis; dynamic analysis

Citation (Format 1): LU J C, NIU Z, CHEN L, et al. Deadlock detection: background, techniques, and future improvements [J]. ZTE Communications, 2024, 22(2): 71 – 79. DOI: 10.12142/ZTECOM.202402009
Citation (Format 2): J. C. Lu, Z. Niu, L. Chen, et al., "Deadlock detection: background, techniques, and future improvements," *ZTE Communications*, vol. 22, no. 2, pp. 71 – 79, Jun. 2024. doi: 10.12142/ZTECOM.202402009.

1 Introduction

Concurrency control is a critical aspect of parallel and distributed computing systems, as it ensures that multiple processes can access shared resources without conflicts or performance degradation. One of the major concerns in concurrency control is the occurrence of deadlocks, which can lead to indefinite delays and inefficient resource utilization. Deadlocks occur when a set of processes are blocked, each waiting for a resource held by another process in the set. This circular dependency prevents any of the processes from making progress, causing a significant impact on system performance.

A variety of deadlock detection techniques have been proposed in the literature, which can be broadly categorized into two categories. Static techniques analyze the system's source code, data structures, or control flow graphs to detect potential deadlocks without running the program. Dynamic techniques, on the other hand, monitor the system's execution at runtime to detect deadlock occurrences.

Moreover, each tool adopts a different set of strategies, with technical details not always fully documented or publicized. These factors have resulted in a knowledge gap that hinders users of these tools, particularly researchers conducting deadlock analysis. To narrow this gap, several questions must be addressed:

• Q1: How do current static deadlock detection techniques improve efficiency and reduce false positives?

• Q2: How do current dynamic deadlock detection techniques improve efficiency and reduce false positives?

• Q3: What is the future direction of development for deadlock detection techniques?

To answer these questions, we present a comprehensive analysis of existing deadlock detection techniques, through the study of five popular deadlock tools shown in Table 1. As the vast majority of deadlock detection tools are not open source, we can only discuss these techniques qualitatively and answer Q1 and Q2 accordingly. After the above discussion, we answer Q3 by providing an outlook and summary for the future development of deadlock detection.

By systematically dissecting and evaluating the tools, we can make new observations that amend or complement prior knowledge. Our major observations are as follows.

• Static deadlock detection typically improves efficiency

▼**Table 1. Groups of deadlock detectors that our study covers**

Type	Tools	Release Date
Static	D4[1]	Jun. 2018
	Peahen[2]	Nov. 2022
Dynamic	GoodLock[3]	Nov. 2005
	MagicLock[4 – 5]	Mar. 2014
	Sherlock[6]	Aug. 2014

through parallelization, pre-screening, and other methods.

• Dynamic deadlock detection typically improves efficiency by merging or deleting states in the lock graph.

• Despite dynamic and static methods generating lock traces using completely different approaches, there are commonalities in the subsequent deadlock detection process.

The remainder of this paper is structured as follows. Section 2 provides a brief background on deadlocks and deadlock detection. Sections 3 and 4 present a comprehensive analysis of current static and dynamic deadlock detection techniques, using a selection of tools as examples. Section 5 summarizes and compares the aforementioned tools. The future direction of development for deadlock detection techniques is discussed in Section 6. Section 7 concludes the paper.

2 Background: Deadlock and Defense

2.1 Deadlock

A deadlock is a state in which each member of a group is waiting for another member, including itself, to take action[7]. The occurrence of a deadlock requires the satisfaction of the following four necessary conditions[8].

• Mutual-exclusion: Each lock object can only be owned by one thread.

• Wait-for: A thread does not release the lock object it has acquired while waiting to acquire another lock object.

• No-preemption: A thread cannot seize the lock object of another thread.

• Circular-wait: Each thread holds one or more lock objects while simultaneously requesting lock objects that other threads have already acquired.

More specifically, for the abstract model of non-reentrant locks L_1 and L_2, and threads T_1 and T_2, the following two deadlock situations exist: When T_1 holds L_1 and is waiting for T_1 to release L_1 (e.g. T_1 locks L_1 twice). When T_1 is waiting for T_2 to release L_1, T_2 is also waiting for T_1 to release L_1.

In order to deal with deadlocks in concurrent systems, there are generally three preventive measures[9]:

• Deadlock prevention: breaking one of the four conditions mentioned earlier to prevent the occurrence of a deadlock;

• Deadlock avoidance: dynamically detecting the possibility of a deadlock and taking appropriate measures to avoid it;

• Deadlock detection: detecting the existence of a deadlock and taking appropriate measures to recover from it.

2.2 Deadlock Prevention

The objective of deadlock prevention is to ensure that deadlocks do not happen by breaking the necessary conditions for system deadlocks. As recommended by HAVENDER[10], the following approaches effectively negate each of the four remaining conditions in turn.

1) Mutual-exclusion condition denied. It allows multiple processes to access the same resource at the same time. This strategy can be applied to read-only data files[11], disks, and other software and hardware resources, but it is not always feasible because some resources may be inherently non-shareable.

2) Wait-for condition denied. All resources are allocated through a static approach. This means that a process must request all the necessary resources before execution and will not begin execution until all the required resources have been obtained. This approach is simple to implement but significantly reduces both resource utilization and language expressiveness. This is because the static allocation of resources cannot support runtime features such as recursion and polymorphism[12].

3) No-preemption condition denied. When a process requests a resource that is currently unavailable, it will be blocked until the resource is available. If the process cannot acquire the resource after a certain period of time, it will release all resources currently held and restart the request process. This approach is not always feasible as certain resources may inherently be non-preemptible. Currently, this strategy is only employed for the allocation of memory and processor resources.

4) Circular-wait condition denied. It assigns a unique number to each resource and requires processes to request resources in ascending order. This approach is more effective and widely used, such as Android[13], compared with the previous deadlock prevention methods.

2.3 Deadlock Avoidance

Deadlock avoidance takes a more proactive approach than deadlock prevention, striving to recognize and avoid potential deadlocks before they occur. This is typically achieved through careful resource allocation and monitoring of the system state.

The most well-known deadlock avoidance algorithm is the Banker's Algorithm[14], which requires processes to declare their maximum resource needs upfront. The algorithm then allocates resources in a manner that guarantees a safe state, ensuring that no deadlock can occur. However, this approach requires accurate resource estimates and may become computationally complex for large-scale systems.

Deadlock avoidance is generally considered more favorable than prevention in database systems, as these systems already can abort transactions. Although avoidance may result in the unnecessary aborting of transactions, it is still preferred over prevention.

2.4 Deadlock Detection

Deadlock detection is the process of identifying deadlocks in a computing system, either before or during execution. Compared with deadlock prevention and avoidance, deadlock detection minimizes the need for human intervention in the program and avoids affecting the program's performance and sharing capabilities. Due to its high practicality and versatility, deadlock detection technology has been widely researched in recent years, and significant progress has been made. Therefore, this article focuses on researching and summarizing deadlock detection technology.

Deadlock detection techniques can be classified into two categories based on whether the program needs to be executed: static deadlock detection discussed in Section 3 and dynamic deadlock detection discussed in Section 4.

3 Static Deadlock Detection

Static deadlock detection is a technique that can identify deadlocks without executing the program[15]. This technique involves analyzing the source code or program structure to identify potential deadlocks. In static analysis, detectors track the acquired lock objects and those being requested. When circular dependencies between locks result in a deadlock, a bug is detected. Techniques such as model checking[16], dataflow analysis[17], and control flow analysis[18] can be employed to detect deadlock-prone situations. Previous research[1–2, 19–20] used static analysis to detect deadlocks from source codes. While some promising results have been achieved as described in the following text, it is still a long way to achieve a complete solution to deadlock bugs. For instance, static analysis cannot account for dynamic program behavior and static detectors often produce many false positives[21].

In the remainder of this section, we discuss two representative static deadlock detectors to illustrate the recent development direction of static deadlock detection.

3.1 D4

D4 is a fast concurrency analysis framework based on concurrent and incremental pointer analysis. By redesigning the pointer analysis, correct conclusions can be obtained by only re-analyzing the incremental code, which avoids redundant computation in traditional whole-program analysis.

The pointer assignment graph (PAG) is a data structure used in pointer analysis algorithms to represent the assignments and relationships between pointers and objects in a program. Each program variable corresponds to a node within the PAG, and variable assignments are reflected through the creation of one or more edges. The PAG consists of two distinct node types: pointer nodes, representing pointer or reference variables, and object nodes, representing memory locations or objects. Each pointer node is associated with a points-to set denoted by pts, which contains the set of object nodes that the pointer may point to. Each edge represents a subset constraint between the points-to sets, i.e., $p \rightarrow q$ means $\mathrm{pts}(p) \subseteq \mathrm{pts}(q)$. We consider Fig. 1 as an illustrative example of a PAG, where p and q denote pointer nodes, and o_1 and o_2 represent object nodes. In this case, $\mathrm{pts}(p)$ consists of $\{o_1\}$, while $\mathrm{pts}(q)$ encompasses $\{o_1, o_2\}$.

The new parallel incremental pointer analysis is mainly based on the following properties of the acyclic PAG.

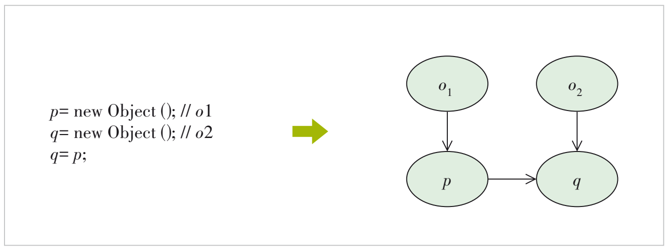
1) Deleting edges property. As shown in Fig. 2, for an object node o and two pointer nodes p and q in a PAG, if q has an incoming neighbor p (i.e., there exists an edge $p \rightarrow q$) and $o \in \mathrm{pts}(p)$, o can reach p without going through q.

Based on the properties mentioned above, supposing an edge $q \rightarrow p$ is deleted from PAG and other edges remain unchanged, we only need to check the incoming neighbors of p (i.e., the deleted edge's destination), which is much faster than traversing the whole PAG for checking the path reachability.

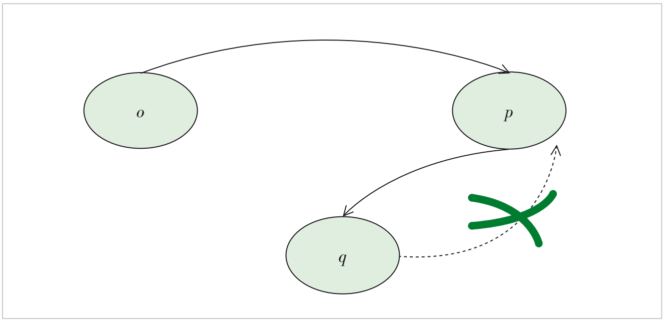
2) Propagating changes property. To propagate a change to a node, it is sufficient to check the other incoming neighbors of the node. If the points-to set of any incoming neighbor contains the change, the node can be skipped. Otherwise, the change should be applied to the node and propagated further to all its outgoing neighbors.

As shown in Fig. 3, for two object nodes o_1 and o_2, and four pointer nodes x, y, z, and w in a PAG, supposing that an edge $q \rightarrow p$ is deleted from PAG and other edges remain unchanged, we only need to check z and w, which are the outgoing neighbors of y.

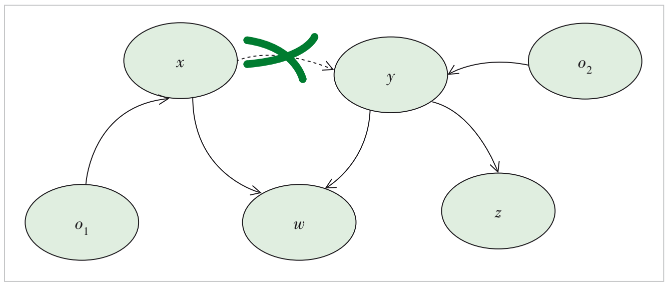
The two theorems above ensure that when a statement is de-



▲Figure 1. An example of pointer assignment graph (PAG)



▲Figure 2. Incoming neighbors property



▲Figure 3. Outgoing neighbors property

leted, it is only necessary to check the local neighbors of the affected nodes in the PAG to determine the changes in points-to sets and perform change propagation. This greatly reduces the amount of computation required for recomputing points-to sets or traversing the entire graph.

3.2 Peahen

Peahen[2] explores a context-reduction approach for fast and precise deadlock detection in real-world programs. Traditional static deadlock detection techniques first construct a context-sensitive lock graph based on lockset analysis, and then analyze the lock graph to discover precise deadlock cycles. However, it has been observed that large-scale context-sensitive lock calling contexts can cause state space explosion and make it difficult to eliminate false positive deadlock cycles. To address that problem, Peahen splits the static deadlock detection technique into two stages: the context-insensitive lock graph construction and three lazy deadlock cycles refinements.

1) Context-insensitive lock graph construction. Refs. [22 – 24] build context-sensitive lock graphs including a large scale of unnecessarily acquired edges. A context-insensitive lock graph with selected acquired edges cloning can simplify deadlock analysis. Peahen presents an inter-procedural algorithm that constructs a context-insensitive lock graph without requiring any context analysis. The algorithm then proceeds to clone selected multi-thread edges. For example, Fig. 4 shows a program using nested locks. Thread 1 and thread 2 both run functions foo() and bar(). If thread 1 is running at line 09 waiting for lock o_1 and thread 2 is running at line 16 waiting for lock o_2, a deadlock problem will occur. Fig. 5 is its lock graph. Peahen first adds edges in every function and then builds an intra-procedural lock graph using the bottom-up dataflow analysis. If an acquired edge represents different threads' lock dependencies, it must be cloned and distinguished as different thread IDs.

2) Deadlock cycle refinements. To identify deadlock cycles precisely and efficiently, Peahen performs the three following steps to refine lock graph cycles lazily.

```
T1:                              T2:
01: void thread1(){              11: void thread2(){
02:    fork(t2, thread2);        12:    foo();
03:    foo();                    13: }
04:    join(t2);                 14: void foo(){
05: }                            15:    lock(v1);
06: void bar(){                  16:    lock(v2); // o2
07:    unlock(v1); // o1         17:    bar();
08:    x++;                      18:    unlock(v2);
09:    lock(v1);                 19:    unlock(v1);
10: }                            20: }
```

▲Figure 4. Code using non-nested locks

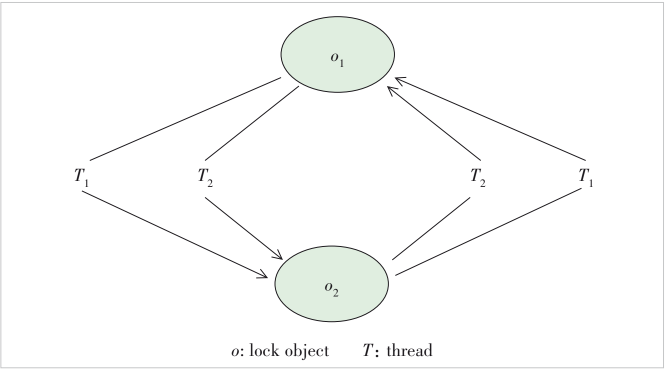


Figure 5. Context-insensitive lock graph

• Single- and multi-threaded cycle computation. Peahen divides lock cycles into single-threaded cycles and multi-threaded cycles. Single-threaded cycles indicate that a lock node owns an edge pointing towards itself. Multi-threaded cycles are defined in that every cycle edge is owned by different threads.

• Concurrent cycle computation. At this step, Peahen tries to refine multi-threaded cycles to concurrent cycles. Concurrent cycles must rule out two cases: A thread in the multi-threaded cycle has been destroyed, or the multi-threaded cycle has been guarded by a lock.

• Path-feasible cycle computation. Finally, Peahen performs path feasibility analysis on the concurrent cycles using the satisfiability modulo theory (SMT) solver. Peahen is the first one to introduce path feasibility analysis into deadlock cycle refinements.

4 Dynamic Deadlock Detection

Dynamic deadlock detection is a technique that detects deadlocks in a multi-threaded program based on the execution trace[3]. Most of the dynamic deadlock detection approaches map an execution trace to data structures such as lock-order graphs, and then the running detection algorithms to identify deadlocks. In recent years, dynamic deadlock detection has been widely used in the field of software testing (e.g., GoodLock[3, 25], DeadlockFuzzer[26], MagicLock[4 – 5], and Sherlock[6]). Compared with static deadlock detection, dynamic deadlock detection can better obtain happens-before relationships and other runtime information in a program. However, these approaches are limited in their ability to detect deadlocks, such as the failure to cover all possible program states and the possibility of false positives.

In the remainder of this section, we discuss three representative dynamic deadlock detectors to illustrate the recent development direction of dynamic deadlock detection.

4.1 GoodLock

GoodLock[3] is a dynamic deadlock detection algorithm analyzing a trace generated from the execution of the program. It consists of two main components: trace generation and detection. First, the program under test is instrumented to record synchronization events when executed. The detection algorithm analyzes the execution trace and constructs a lock graph that identifies potential deadlocks through the presence of cycles. Although GoodLock is not sound nor complete, it is an improvement on the basic lock graph algorithm[27], which reduces false positives in the presence of gate locks (a common lock taken first by involved threads). The main strategy for reducing false positives is described as follows.

• Extended lock graph. Traditional lock graphs can only represent partial information about the ordering of lock acquisitions by threads. For example, in an abstract model of thread T , locks L_1 and L_2, if there exists a state where thread T holds L_1 and acquires L_2 during execution, then a directed edge from L_1 to L_2 is added to the lock graph, written as $L_1 \rightarrow L_2$. Therefore, a cycle can be created in the lock graph through any cyclic acquisition of locks, even if the acquisition cannot happen parallelly, leading to false positives. To address this problem, GoodLock[3] introduced the concept of an extended lock graph. The extended lock graph is an extension of the traditional lock graph that includes more information about which thread causes the addition of the edge and which gate locks are held by that thread when the target lock is taken. Based on this extended information, false positives caused by single-threaded and guarded cycles can be eliminated during the detection phase.

• Segments. As the example in Fig. 6, the algorithm on the extended lock graph reports a cycle between threads T_1 (lines 05 – 06) and T_2 (lines 08 – 09) on locks L_1 and L_2. However, a deadlock is impossible since thread T_2 is joined on by the main thread in line 03. Therefore, the two code segments, lines 05 – 06 and lines 08 – 09, can never run in parallel. The algorithm to be presented will prevent such cycles from being reported by formally introducing such a notion of segments that cannot execute in parallel. A new directed segmentation graph will record which segments execute before others. The lock graph is then extended with extra-label information, which specifies what segment locks are acquired in, and the validity of a cycle now incorporates a check that the lock acquisitions occur in parallel executing segments. Based on this extended information, false positives caused by segmented cycles can be eliminated during the detection phase.

Apart from the strategies mentioned above, various optimization strategies have also been added to GoodLock's variants, as described in later sections.

4.2 MagicLock

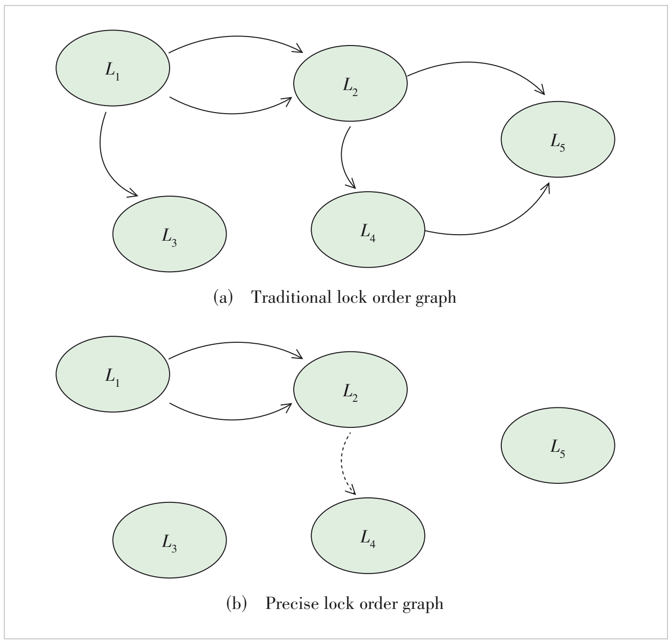
MagicLock[4 – 5] is a more efficient variant of GoodLock[3]. The two tools share a similar technological approach to deadlock detection, both consisting of two phases: trace generation and detection. In fact, directly checking on the lock order graph of GoodLock[3] for a large-scale program is impractical due to the huge cost of time. For example, in the ITCAM application, The authors in Ref. [28] reported a lock order graph with over 300 000 nodes and 600 000 edges. The nodes in the graph represent the procedures in the program, and the edges represent the call relationships between the procedures. GoodLock spent 48 h and 13.6 GByte memory to traverse it to find cycles if they exist[28]. So MagicLock[4] proposed some strategies to reduce the size of the lock order graph in the trace generation phase and the time spent on detecting deadlocks. The main strategies are described as follows.

```
Main:
01: t1 = new T1();
02: t1.start();
03: t1.join();
04: new T2().start();

T1:
05: synchronized(L1) {
06: synchronized(L2) {}
07: }

T2:
08: synchronized(L2) {
09: synchronized(L1) {}
10: }
```

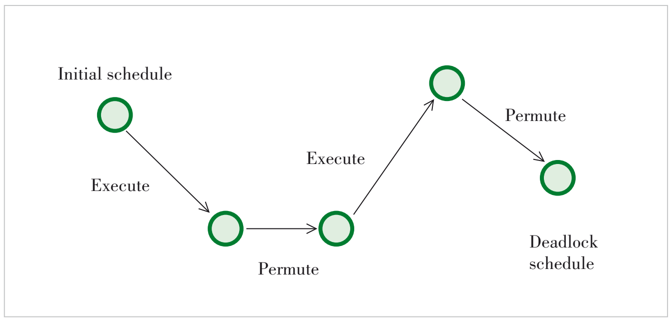
Figur 6. Example program of false positives

• Graph pruning. Previous work has proposed several strategies for simplifying states, such as merging the state of locks[28]. Although merging locks can reduce the search space, all locks that cannot lead to deadlocks are still retained in the lock graph, leading to redundant traversal. MagicLock[4] iteratively removes the lockset and their edges, resulting in a more precise lock order graph. The strategy of iteratively removing edges is based on the following observation: for a node participating in a potential deadlock cycle, the node must have both incoming and outgoing edges. Therefore, during each iteration, the edges of nodes that possess solely incoming and outgoing edges will be eliminated. As the traditional lock order graph example in Fig. 7(a), the algorithm's first iteration will remove the edges pointing to L_3 and L_5, and the second iteration will remove the edge pointing to L_4, as indicated by the dotted line in Fig. 7(b). Based on this additional information, the states that would not cause potential deadlocks are deleted, which improves the efficiency of the algorithm.

• Thread-specific lock dependency. MagicLock uses a thread-specific lock dependency relation denoted by thread-specific triple $D_i = \langle t, m, Lt \rangle$ for each thread. Here, t represents the thread number, m denotes the lock that is being acquired by the current statement, and Lt represents the set of locks that are currently held by the thread t. This triple captures the dependencies between the locks that a thread holds and the locks that it attempts to acquire. Based on the afore-


L_1
L_2
L_3
L_4
L_5
(a) Traditional lock order graph
L_1
L_2
L_3
L_4
L_5
(b) Precise lock order graph

▲**Figure 7. Lock order graph example**



▲**Figure 8. Sherlock deadlock search**

mentioned strategy, MagicLock employs a new depth-first-search algorithm to traverse D_i for each thread t_i. This approach differs from the iGoodLock algorithm in Deadlock-Fuzzer[26]. As iGoodLock employs transitive closure for iterative cycle detection, a noticeable limitation is that iGoodLock has to store all intermediate results, which consumes a lot of memory[26].

4.3 Sherlock

Sherlock[6] is a more effective variant of GoodLock[3]. Leveraging the power of concolic execution, Sherlock exhibits remarkable proficiency in identifying deadlocks after an extensive computation of one million steps, a feat beyond the reach of conventional technologies, which other existing technologies cannot discover. Sherlock also consists of two phases: producing deadlock candidates with GoodLock and concolic execution to drive an execution toward a deadlock candidate. As the algorithm for generating candidate deadlocks in GoodLock has already been introduced earlier, this section mainly focuses on the concolic execution in Sherlock[6].

Sherlock first produces a set of deadlock candidates using GoodLock. Then, it uses concolic execution to search for each of the deadlock candidates. The key idea is to turn each search for a deadlock into a search for an event sequence (schedule) that leads to the deadlock. As a deadlock search example in Fig. 8, for each schedule that leads to a deadlock, Sherlock alternates between the "execute" and "permute" steps. The "execute function" attempts to execute a given schedule and determine whether it leads to a deadlock. The "permute function" permutes a given schedule. The search begins with an initial schedule found simply by "InitialRun function". The search fails if "execute" cannot execute a given schedule, "permute" cannot find a better permutation, or the search times out. In the following paragraphs, we will briefly discuss each of these phases: "InitialRun," "Execute," and "Permute".

- "InitialRun function". The InitialRun function executes the program with some particular input and records the schedule. For each program, Sherlock uses the predetermined inputs to execute the program because those inputs are useful enough.
- "Execute function". The arguments of "execute function" are a program, a schedule, and a deadlock candidate. The "execute function" will attempt to execute the given schedule, determine whether it leads to a deadlock, and return the input to the program that is used to execute the schedule. The implementation of the "execute function" uses concolic execution[29–32] collecting operational information (e.g. assignments and conditions), and generate inputs that are more likely to reach the specified state for the next running.
- "Permute function". The arguments of "permute function" are a schedule and a deadlock candidate. The "permute function" will attempt to better the given schedule and primarily improves SAID et al.'s "permute function"[33]. The "permute function" in Sherlock encapsulates the encoding of lock-order graphs and alias information into constraints. These constraints are subsequently solved using an SMT solver, while satisfying conditions such as happens-before relationships. Finally, the feasible solutions are traversed to discover an optimal schedule.

5 Comparative Evaluation

Many existing deadlock detection methods lack open-source code, and a significant portion of open-source projects have low usability due to poor maintenance. Therefore, we provide a qualitative evaluation of these techniques. We evaluate these tools from two aspects: scalability and effectiveness. Scalability is the property of a system to handle a growing amount of work. One definition for software systems specifies that this may be done by adding resources to the system. The effectiveness is measured from three aspects: false positives, false negatives, and detection of new vulnerabilities, as shown in Table 2. Finally, we summarize our observations.

5.1 Static Deadlock Detection

• Scalability. Previous work[19] focused on whole-program analysis and it was difficult to efficiently detect deadlocks. D4 reduces redundant calculations during the analysis process through incremental analysis and further accelerates deadlock detection through parallelization. Peahen reduces the overhead of subsequent deadlock detection stages through a fast pre-processing phase.

• Effectiveness. D4[1] and Peahen[2] are both unsound and incomplete. Peahen is almost sound, aside from a few well-identified reasonable unsound choices for achieving higher precision. There are two sources of unsoundness in our implementations. First, the pointer analysis shares the same unsound sources as Peahen uses. For instance, it does not correctly handle pointer arithmetic, array accesses, containers, etc. Second, the lock graph construction ignores the locks that are inside blocks of the assembly code as the prior deadlock detectors. The unsoundness of D4 also mainly comes from two aspects. Firstly, the imprecision of pointer analysis that results in D4 cannot handle the situation where a lock variable may point to multiple objects. Secondly, D4 ignores reflection and library functions during the analysis process, leading to false negatives. Since Peahen has discovered new deadlock issues while D4 only demonstrates the efficiency of a new algorithm on the Dacapo benchmark[34] without finding new deadlock issues, we consider Peahen more effective than D4.

In summary, traditional static program analysis has become increasingly difficult to complete within an acceptable time frame for most existing programs. Therefore, much of the current research focuses on improving deadlock detection efficiency through parallelization, pre-screening, and other methods.

5.2 Dynamic Deadlock Detection

• Scalability. Traditional dynamic deadlock detection has only limited scalability. To relax the dynamic deadlock detection overhead, many seminal approaches have been proposed. The MulticoreSDK[27] firstly groups the locks being held by different threads at the same code location in the same group and then merges multiple groups into the same group whenever they have at least one shared lock to reduce the size of the lock order graph. The MagicLock[5] employs an iterative approach to eliminating locks that cannot cause deadlocks from the lock order graph. The latest one, AirLock[35], speeds up the online cycle discovery by first finding "simple cycles" without considering any execution information (e.g., threads) and then constructing deadlock cycles by taking full execution information into account.

• Effectiveness. Most dynamic deadlock detection techniques are also unsound and incomplete. Unlike typical fuzzing, dynamic deadlock detection usually only relies on lock-traces generated during runtime since it is difficult to produce inputs that can reach the deadlock state and verify the actual occurrence of deadlocks. An incomplete dynamic deadlock may induce false positives due to, for instance, ignoring happens-before relations. Thus, Sherlock[6] is integrated with other techniques via scheduling a real deadlock and identifying and solving execution constraints. DeadlockFuzzer[26] uses fuzzing to confirm whether the cycle of locks is a real deadlock. Similar to static deadlock detection, dynamic deadlock detection techniques like MagicLock[5] and AirLock[29] only demonstrate their speed and efficiency through evaluation. Only Sherlock discovers new deadlock problems through concolic execution. Therefore, we consider Sherlock more effective than MagicLock and AirLock.

In summary, traditional dynamic program analysis has also become increasingly difficult to complete within an acceptable time frame for most existing programs. Therefore, much of the current research focuses on improving deadlock detection efficiency by merging or eliminating states in the lock graph.

▼Table 2. Evaluation results across all vulnerability discovery techniques

Type	Tools	Scalability	False Positives	False Negatives	New Bugs
Static	D4	High	True	True	False
	Peahen	High	True	Almost false	True
Dynamic	GoodLock	Low	True	True	False
	MagicLock	Medium	True	True	False
	Sherlock	Medium	True	True	True

6 Future Works

Although significant progress has been made in both static and dynamic deadlock detection over the past years to address this security challenge, there are still many open and unsolved issues.

• Scalability. Due to the rapid expansion of software codes, existing deadlock detection tools still struggle to handle projects at the level of millions of lines of code, such as the Linux kernel[36] and Firefox[37], within an acceptable range. Therefore, future works should still focus on improving the scalability of deadlock detection tools.

• Recall. The false negatives of reports are the most serious problem of deadlock detection tools. The false negatives of reports can cause serious problems, because our ultimate goal is to eliminate deadlocks. Existing dynamic deadlock detection tools suffer from a significant number of false negatives due to their reliance on program execution traces, which makes it challenging to achieve high coverage. While static deadlock detection tools have fewer false negatives compared with their dynamic counterparts, they still have their own limitations in terms of false negatives. Therefore, enhancing the coverage of both static and dynamic deadlock detection tools remains an urgent and unresolved issue.

• Precision. The false positives of reports confuse developers and waste their time. Currently, besides DeadlockFuzzer[26] and other tools that report only the deadlocks confirmed by fuzzing to avoid false positives, all other deadlock detection tools suffer from false positives. However, reporting only the deadlocks confirmed by fuzzing can lead to a large number of false negatives, which is not a sweet spot. Therefore, future work should focus on improving the accuracy of deadlock detection tools.

• Communication deadlocks. Communication deadlock[38] is another kind of deadlock. Traditional deadlock detection only models locks and focuses on whether there is a circular "wait-for" in acquiring the locks. However, communication deadlock occurs when one or more threads are waiting for certain messages/signals from other threads, which are suspended and unable to send the required messages/signals or have already sent the messages/signals before a waiting thread starts to wait for the messages/signals. Due to the diverse and complex characteristics of communication deadlocks, compared with resource deadlocks analysis, there are few achievements in static and dynamic detection of communication deadlocks. Program analysis for resource deadlocks is still in its early stages.

7 Conclusions

In this paper, we present a comprehensive study of deadlock detection techniques from two perspectives: static and dynamic. Our research summarizes the different strategies of existing deadlock detection works and qualitatively compares their differences. Throughout the study, we derive a group of new observations that can complement previous understandings and also inspire future directions of deadlock detection.

Acknowledgement:

We would like to express our heartfelt gratitude to Professor ZHAO Yongwang for his invaluable guidance, support, and encouragement throughout his research.

References

[1] LIU B Z, HUANG J. D4: fast concurrency debugging with parallel differential analysis [J]. ACM SIGPLAN notices, 2018, 53(4): 359 – 373. DOI: 10.1145/3296979.3192390
[2] CAI Y D, YE C F, SHI Q K, et al. Peahen: fast and precise static deadlock detection via context reduction [C]//The 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM, 2022: 784 – 796. DOI: 10.1145/3540250.3549110
[3] BENSALEM S, HAVELUND K. Dynamic deadlock analysis of multi-threaded programs [EB/OL]. [2023-06-01]. https://www.havelund.com/Publications/padtad05.pdf
[4] CAI Y, CHAN W K. MagicFuzzer: scalable deadlock detection for large-scale applications [C]//Proceedings of 34th International Conference on Software Engineering (ICSE). IEEE, 2012: 606 – 616. DOI: 10.1109/ICSE.2012.6227156
[5] CAI Y, CHAN W K. Magiclock: scalable detection of potential deadlocks in large-scale multithreaded programs [J]. IEEE transactions on software engineering, 2014, 40(3): 266 – 281. DOI: 10.1109/TSE.2014.2301725
[6] ESLAMIMEHR M, PALSBERG J. Sherlock: scalable deadlock detection for concurrent programs [C]//The 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014: 353 – 365. DOI: 10.1145/2635868.2635918
[7] ARPACI-DUSSEAU R H, ARPACI-DUSSEAU A C. Operating systems: three easy pieces [M]. Raleigh, USA: Lulu Press, 2018
[8] COFFMAN E G, ELPHICK M, SHOSHANI A. System deadlocks [J]. ACM computing surveys, 1971, 3(2): 67 – 78. DOI: 10.1145/356586.356588
[9] ELMAGARMID A K. A survey of distributed deadlock detection algorithms [J]. ACM SIGMOD record, 1986, 15(3): 37 – 45. DOI: 10.1145/15833.15837
[10] HAVENDER J W. Avoiding deadlock in multitasking systems [J]. IBM systems journal, 1968, 7(2): 74 – 84. DOI: 10.1147/sj.72.0074
[11] Alvinashcraft. CreateFileW function (fileapi. h) [EB/OL]. [2023-05-17]. https://learn. microsoft. com/en-us/windows/win32/api/fileapi/nf-fileapi-createfilew#parameters
[12] BERLIZOV A N, ZHMUDSKY A A. The recursive adaptive quadrature in MS Fortran-77 [EB/OL]. [2023-05-17]. https://arxiv. org/abs/physics/9905035
[13] Google. mm/rmap. c-kernel/common-Git at Google-android. googlesource. com [EB/OL]. [2023-05-17]. https://android. googlesource. com/kernel/common/+/refs/heads/android13-5.15/mm/rmap.c
[14] IJKSTRA E W. Een algorithme ter voorkoming van de dodelijke omarming [EB/OL]. [2023-05-17]. http://www. cs. utexas. edu/users/EWD/ewd01xx/EWD108.PDF
[15] AYEWAH N, PUGH W, HOVEMEYER D, et al. Using static analysis to find bugs [J]. IEEE software, 2008, 25(5): 22 – 29. DOI: 10.1109/ms.2008.130
[16] CLARKE E M. Model checking [C]//The 17th International Conference on Foundations of Software Technology and Theoretical Computer Science. FSTTCS, 1997: 54 – 56
[17] KHEDKER U, SANYAL A, SATHE B. Data flow analysis: theory and practice [M]. Carrollton, UAS: CRC Press, 2017
[18] ALLEN F E. Control flow analysis [J]. ACM SIGPLAN notices, 1970, 5(7): 1 – 19. DOI: 10.1145/390013.808479
[19] NAIK M, PARK C S, SEN K, et al. Effective static deadlock detection [C]//The 31st International Conference on Software Engineering. IEEE, 2009: 386 – 396. DOI: 10.1109/ICSE.2009.5070538
[20] BROTHERSTON J, BRUNET P, GOROGIANNIS N, et al. A compositional deadlock detector for android java [C]//The 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021: 955 – 966
[21] DELIGIANNIS P, DONALDSON A F, RAKAMARIC Z. Fast and precise symbolic analysis of concurrency bugs in device drivers (T) [C]//The 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2015: 166 – 177. DOI: 10.1109/ASE.2015.30
[22] WILLIAMS A, THIES B, ERNST M D. Static deadlock detection for java libraries [EB/OL]. [2023-05-17]. https://people.csail.mit.edu/amy/papers/deadlock-ecoop05.pdf
[23] KAHLON V, YANG Y, SANKARANARAYANAN S, et al. Fast and accurate static data-race detection for concurrent programs [C]//International Conference on Computer Aided Verification. CAV: 226 – 239.10.1007/978-3-540-73368-3_26
[24] KROENING D, POETZL D, SCHRAMMEL P, et al. Sound static deadlock analysis for C/Pthreads [C]//Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, 2016: 379 – 390. DOI: 10.1145/2970276.2970309
[25] HAVELUND K. Using runtime analysis to guide model checking of java programs [C]//The 7th International SPIN Workshop on Model Checking of Software. SPIN, 2000: 245 – 264

[26] JOSHI P, PARK C S, SEN K, et al. A randomized dynamic program analysis technique for detecting real deadlocks [J]. ACM SIGPLAN notices, 2009, 44(6): 110 – 120. DOI: 10.1145/1543135.1542489
[27] HARROW J J. Runtime checking of multithreaded applications with visual threads [C]//The 7th International SPIN Workshop in Model Checking and Software Verification. SPIN, 2000: 331 – 342
[28] LUO Z D, DAS R, QI Y. Multicore SDK: a practical and efficient deadlock detector for real-world applications [C]//The 4th IEEE International Conference on Software Testing, Verification and Validation. IEEE, 2011: 309 – 318. DOI: 10.1109/ICST.2011.22
[29] MAJUMDAR R, XU R G. Directed test generation using symbolic grammars [C]//The 22nd IEEE/ACM international conference on Automated software engineering. IEEE, 2007: 134 – 143
[30] GODEFROID P, KLARLUND P, SEN K. Dart: directed automated random testing [C]//The ACM SIGPLAN conference on Programming language design and implementation. ACM, 2005: 213 – 223
[31] SEN K, AGHA G. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools [C]//International Conference on Computer Aided Verification. CAV, 2006: 419 – 423. DOI: 10.1007/11817963_38
[32] SEN K. Concolic testing [C]//The 22nd IEEE/ACM international conference on automated software engineering. IEEE, 2007: 571 – 572
[33] SAID M, WANG C, YANG Z, et al. Generating data race witnesses by an SMT-based analysis [C]//The Third International Conference on NASA Formal Methods. NASA Formal Methods Symposium, 2011: 313 – 327
[34] BLACKBURN S M, GARNER R, HOFFMANN C, et al. The DaCapo benchmarks: java benchmarking development and analysis [C]//The 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. ACM, 2006: 169 – 190. DOI: 10.1145/1167473.1167488
[35] CAI Y, MENG R, PALSBERG J. Low-overhead deadlock prediction [C]//The 42nd International Conference on Software Engineering. IEEE, 2020: 1298 – 1309
[36] Bugzilla. Bugzilla main page: bugzilla.kernel.org [EB/OL]. [2023-06-01]. https://bugzilla.kernel.org/
[37] Bugzilla. Bugzilla main page: bugzilla.mozilla.org [EB/OL]. [2023-06-01]. https://bugzilla.mozilla.org/home
[38] JOSHI P, NAIK M, SEN K, et al. An effective dynamic analysis for detecting generalized deadlocks [C]//The 18th ACM SIGSOFT international symposium on Foundations of software engineering. ACM, 2010: 327 – 336. DOI: 10.1145/1882291.1882339

Biographies

LU Jiachen (lujc@zju.edu.cn) is a master student in cybersecurity at Zhejiang University, China. His research interests include program analysis and formal methods.

NIU Zhi received his master degree in control engineering from Chongqing University, China. He is currently working at ZTE Corporation. His research interests include distributed system, formal verification and software reliability.

CHEN Li received his bachelor degree in computer science from Northeastern University, China. He is currently working at ZTE Corporation. His research interests include software reliability, open source software risk analysis and network security.

DONG Luming received his master degree in control theory and control engineering from Huazhong University of Science and Technology, China. He is currently working at ZTE Corporation. His research interests include distributed system, formal verification, software reliability and innovative security technology for wireless communication.

SHEN Taoli is a master student in cybersecurity at Zhejiang University, China. His research interests include push-button verification and formal methods.