LockPeeker: Detecting Latent Locks in Java APIs

Ziyi Lin1, Hao Zhong2; Yuting Chen2*, Jianjun Zhao3
1School of Software, Shanghai Jiao Tong University, China
2Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
3Department of Advanced Information Technology, Kyushu University, Japan
{linziyi, zhonghao, chenyt}@sjtu.edu.cn, zhao@ait.kyushu-u.ac.jp

ABSTRACT
Detecting lock-related defects has long been a hot research topic in software engineering. Many efforts have been spent on detecting such deadlocks in concurrent software systems. However, latent locks may be hidden in application programming interface (API) methods whose source code may not be accessible to developers. Many APIs have latent locks. For example, our study has shown that J2SE alone can have 2,000+ latent locks. As latent locks are less known by developers, they can cause deadlocks that are hard to perceive or diagnose. Meanwhile, the state-of-the-art tools mostly handle API methods as black boxes, and cannot detect deadlocks that involve such latent locks. In this paper, we propose a novel black-box testing approach, called LockPeeker, that reveals latent locks in Java APIs. The essential idea of LockPeeker is that latent locks of a given API method can be revealed by testing the method and summarizing the locking effects during testing execution. We have evaluated LockPeeker on ten real-world Java projects. Our evaluation results show that (1) LockPeeker detects 74.9% of latent locks in API methods, and (2) it enables state-of-the-art tools to detect deadlocks that otherwise cannot be detected.

CCS Concepts
•Software and its engineering → Software testing and debugging;

Keywords
Latent lock; deadlock detection; API method

1. INTRODUCTION
In multi-threaded programs, locks are used to exclusively access memories. Defects such as deadlocks and livelocks can be raised when locks are inappropriately enforced and/or released by threads. As such defects decrease code quality, many approaches [4, 5, 13, 23, 28] have been proposed to detect locking-related defects. With the support of these approaches, many concurrency bugs have been detected, and some are previously unknown [13, 28].

1Corresponding co-authors

In modern programming, APIs are widely used, but their locks may not be well documented. Latent locks can live in API sources. In this paper, we refer to a lock located inside an API whose source is unaccessible as a latent lock. We have conducted a preliminary study by analyzing the source files of J2SE1. The results show there are 2000+ locks behind J2SE APIs. These locks can be taken as latent locks if developers do not step into their sources.

Latent locks typically exist in two types of APIs: close-sourced APIs which are encrypted (e.g., commercial libraries), and APIs whose programming languages are different from the programming languages of client code (e.g., native C/C++/C#/Assembly APIs that are called through Java Native API). As developers may not be able to access the source code of encrypted or native APIs, it becomes impossible to find latent locks inside and detect defects related to these locks.

Motivating example and the state-of-the-art.
Latent locks can cause serious bugs such as deadlocks. Figure 1a shows a simplified version of Eclipse BIRT-287102 which contains a deadlock caused by latent locks. When a thread (thread1) of Thread1 starts, it invokes the J2SE method forName (line 6); when a thread (thread2) of Thread2 starts, it acquires a lock on obj (line 10). As shown in Figure 1b, a deadlock can happen, because the native method forName0 acquires a lock on its parameter, loader: thread1 can invoke loadClass and acquire a lock on obj (lines 14-15) when holding loader; at the same time, thread2 invokes forName and acquires loader (line 11) when holding obj. However, even the thread dump (as Figure 1c shows) does not state that locks are held and acquired in the native method. Without any knowledge that a thread can lock a classloader when invoking forName0, a program analysis tool cannot detect the deadlock.

To the best of our knowledge, previous deadlock detecting approaches are less effective in finding such bugs, since they omit analyzing APIs, and are often not comprehensive enough to analyze the cross language sources. For example, our previous work [20] shows that existing tools (e.g., JPF [32] and CheckMate [13]) cannot detect deadlocks that involve latent locks in native code. As surveyed by Hong and Kim [10], to detect lock-related defects, all the existing approaches construct execution models for sources under analysis. For example, Sorrentino et al. [29] dynamically record reads and writes to shared variables, and construct an execution model for the program in analysis. However, when sources are unavailable, it is difficult to determine which variables are shared, and thus difficult to detect the corresponding concurrency bugs.

Some approaches [14, 26, 33] can analyze API methods, if their

1http://docs.oracle.com/javase/8/docs/api/index.html
2https://bugs.eclipse.org/bugs/show_bug.cgi?id=287102
source code is available. Some other approaches generate replacements for API methods. For example, Shafiei and Breugel [27] manually write replacement code for native methods. As another example, Kalinovsky [15] introduces a technique that compiles Java bytecode. However, as a replacement is usually not precisely equivalent to the original method, a latent lock in the original API method may not appear in its replacement.

Our approach.

Having observed that API methods are not easy to analyze and believing that proper replacements can facilitate program analysis, we propose in this paper an approach named LockPeeker that leverages locking behavior’s effect in Java to construct a locking model for each API method. Such effect is reserved in any Java API, no matter whether it is encrypted or native. The model clearly presents whether some latent locks can be enforced in an API method, and thus aids program analysis tools in detecting deadlocks intertwined by explicit and latent locks.

However, there remain many difficulties of generating such locking behavior models, since locked resources, locking structures, and locking conditions are not explicitly defined, even in their API documents. Many related questions are raised: Are there locks in an API method? Which object(s) will be locked within the API method? What are the relations among locks? Are they nested? Is there any condition for locking? Without answering these questions, it is not possible to construct a precise locking behavior model for an API method, since its details are still blind to analysis.

LockPeeker has two major steps to detect latent locks in Java API methods: (1) it repeatedly executes a target method and collects the locks triggered during execution, through which a lock tree structure is synthesized; and (2) locking conditions are inferred by observing which test inputs trigger the locks. Synthesized lock trees can be leveraged by existing tools for deadlock analysis.

This paper makes the following contributions:

- **Problem statement.** To the best of our knowledge, we are the first to state the research problem of detecting latent locks in API methods. The research problem can motivate follow-up work that leads to more practical approaches and tools.

- **Approach.** We propose a novel approach, called LockPeeker, that detects latent locks in Java API methods. The essential idea of LockPeeker is that, given an API method, to perform an extensive unit testing of this method, through which (1) a tree of latent locks within the method can be derived by observing whether any locks can be enforced, and (2) the locking condition(s) can be inferred by learning which lock(s) can be acquired by which test input(s).

- **Implementation and evaluation.** We have implemented a tool to support LockPeeker and evaluated LockPeeker on ten real-world Java projects. Our results show that LockPeeker detects 74.9% of locks in API methods. Furthermore, LockPeeker allows locking trees and locking conditions to be generated for Java API methods, which helps find deadlocks that otherwise cannot be detected from such methods.

The rest is organized as follows. Section 2 introduce our preliminaries. Section 3 presents our approach. Section 4 evaluates LockPeeker. Section 5 surveys related work. Section 6 discusses relevant issues and the future work. Section 7 concludes this paper.

2. **PRELIMINARIES**

In this paper, we advocate the idea of inferring the locking structure and the associated conditions for a Java API method. In Java, a thread can use the synchronized keyword to acquire an object’s intrinsic lock that enforces exclusive access to the object, allowing a block of statements be exclusively executed. More sophisticated locking idioms (e.g., using a Lock object) are supported in the java.util.concurrent.locks package. For simplifying our discussion, we focus on intrinsic locks in this paper, because a Lock object can be interpreted as one or more intrinsic locks.

In a method, locks can be sequentially or nestedly acquired and
The locks can be conditionally enforced. We thus use condition lock tree (CLT) to represent such a locking structure and the locking conditions in a method:

**Definition 2 (Condition Lock Tree).** For a method, its CLT is an SLT whose nodes are supplemented with conditions.

- The root (con) denotes the default condition for all nodes.
- A node (v, con) denotes the variable on which the lock is placed and the condition to trigger the lock.

A child can be associated with a condition stronger than its parent’s condition. The condition on root is by default true, indicating that the lock is triggered unconditionally. Any condition on a node is by default the same as the condition on its parent.

For example, Figure 2 shows an SLT and a corresponding CLT derived from the `foo` method. The root of the SLT contains a test instance consists of receiver, method parameters and class fields. Node `v3` is `foo`’s third parameter and Nodes `f1` and `f2` are class fields, where `v3` and `f1` are nested locks represented by the left subtree, and `f2` is a lock separately represented by the right subtree. The locking condition is shown on the CLT. Node `f1` has a specified condition, indicating the variable `f1` is locked under the condition `(v1 <= 0 || v2)`, and other nodes’ conditions are by default true, indicating that they are locked unconditionally. We explain more details of SLTs and CLTs in Section 3.2.

3. APPROACH

This section first presents an overview of LockPeeker (Section 3.1), and then presents how to synthesize the structure of a CLT (Section 3.2) and how to infer the conditions of the CLT (Section 3.3).

### 3.1 Overview

LockPeeker constructs a CLT describing the locks for a target method \( m \). As shown in Figure 3, LockPeeker takes two major steps in constructing a CLT:

**Step 1: Constructing SLTs.** LockPeeker iteratively generates test instances and executes \( m \). At each iteration, LockPeeker checks whether locks are triggered in \( m \): If a test instance can trigger locks, an SLT is constructed to represent the observed locks. In this step, we can generate many SLTs for a given API method, each corresponding to one test instance.

**Step 2: Merging SLTs into a CLT.** LockPeeker merges the SLTs into a CLT whose nodes’ conditions are inferred by learning the test instances. The CLT represents the observable locks and locking conditions in \( m \), allowing a program replacement to be created. The program replacement allows existing tools to step into \( m \) during locking analysis.

### 3.2 Synthesizing Structure

#### 3.2.1 Constructing SLTs

For a target API method \( m \), LockPeeker constructs an SLT for \( m \) through performing a unit testing of the method. We formalize a detection process as follow:

\[
slt = par(thread_1.acquire(i), thread_2.call(m, I))
\]

(1)

where `par` requires that the two threads shall run in parallel; \( I \) denotes a test instance; \( i \) denotes a variable of \( I \); and `slt` denotes a SLT that describes the observed locks in the execution. A simple construction involves two threads: `thread_1` is a thread acquiring a lock on \( i \), `thread_2` another calling \( m \). Once `thread_2` is blocked to wait for locking \( i \), \( i \) is the object locked in \( m \). Therefore, LockPeeker firstly starts `thread_1` and then `thread_2`, and observes whether `thread_2` is blocked while invoking \( m \), i.e., whether \( i \) is locked within \( m \).
The two threads are observed (by the main thread) started to acquire a lock on the test instance I. After the lock is acquired, \( \text{thread}_1 \) suspends and keeps holding it (lines 3-5). \( \text{thread}_2 \) is then started to invoke \( m \) with I as its input values by reflection (lines 6-7). The two threads are observed (by the main thread). Line 8 checks whether \( \text{thread}_2 \) is blocked, and more importantly, whether \( \text{thread}_2 \) happens to be blocked by the intrinsic lock that is being held by \( \text{thread}_1 \), as \( \text{thread}_2 \) may also be blocked for the other reasons (e.g., an invocation of the \( \text{wait}() \) method in \( m \)). If so, an empty SLT \( \text{tree} \) is created (line 9) and \( \text{checkNesting} \) is called to check whether there exist nested locks (line 10). If there are nested locks, they are all added into \( \text{tree} \) (lines 11-12). Only the current checking object \( i \) is added otherwise (line 14).

After that, \( \text{tree} \) is merged into \( \text{ret} \) (line 16). The merging of SLTs can be performed by checking whether an SLT can be equivalent to another or contained by another:

Relation 1 (SLT EQUIVALENT). Two SLTs are structurally equivalent, regardless of the test instance, denoted as \( \text{slt}_1 \equiv \text{slt}_2 \), as shown in Figure 4a.

Relation 2 (SLT CONTAINING). One SLT \( \text{slt}_1 \) contains another \( \text{slt}_2 \), denoted as \( \text{slt}_2 \subseteq \text{slt}_1 \), if there exists an \( \text{slt}_1 \)'s subtree which can be added to a certain node of \( \text{slt}_2 \)'s such that \( \text{slt}_1 \equiv \text{slt}_2 \), as shown in Figure 4b.

Algorithm 2 shows how function \( \text{merge}() \) works. It firstly checks if the two candidate SLTs to merge satisfy Relation 1 or Relation 2, and returns \( \text{cur} \) for Relation 1 or the containing SLT for Relation 2 (lines 1-2). Otherwise, it moves on to traverse target's children nodes and compares each one (say \( \text{targetChild} \)) with each of \( \text{cur} \)'s child node (say \( \text{curChild} \)). If \( \text{targetChild} \) and \( \text{curChild} \) are same (nodes \( \text{node}_1(v_1) \) and \( \text{node}_2(v_2) \) are same when \( v_1 = v_2 \)), their subtrees are merged recursively (lines 7-10). If no same node can be found, \( \text{targetChild} \) is added as a new child (lines 11-18).

When two non-nested locks are observed, we cannot determine which lock is ahead of another, as we do not have the source of \( m \). However, for the purpose of deadlock detection, only the nested locking sequence matters.

Algorithm 1 shows an iterative process of constructing for a method an SLT w.r.t. a test instance. It creates the SLT’s root (ret) based on the test instance \( I \) (line 1). For each variable \( i \) in \( I \), \( \text{thread}_i \) is started to acquire a lock on \( i \). After the lock is acquired, \( \text{thread}_1 \) suspends and keeps holding it (lines 3-5). \( \text{thread}_2 \) is then started to invoke \( m \) with \( I \) as its input values by reflection (lines 6-7). The two threads are observed (by the main thread). Line 8 checks whether \( \text{thread}_2 \) is blocked, and more importantly, whether \( \text{thread}_2 \) happens to be blocked by the intrinsic lock that is being held by \( \text{thread}_1 \), as \( \text{thread}_2 \) may also be blocked for the other reasons (e.g., an invocation of the \( \text{wait}() \) method in \( m \)).
between
is an integer whose value is
example, if an value is of a
for exploring branches. LockPeeker reuses the mutation operators
are. The variable values of a test instance are randomly mutated
branches; meanwhile, it is expensive to use a number of mutants to
tants are sufficient: too few mutants may not explore all necessary
Based on the new test instance $I_3$, LockPeeker builds a new SLT ($S_3$). If $S_3$
equalss $S_1$, LockPeeker searches between $I_2$ and $I_3$; if $S_3$
equalss $S_2$, LockPeeker searches between $I_3$ and $I_1$. The search process
continues until $\Delta I$ is close to zero.

3.3 Synthesizing Condition
After exploring boundary values by mutating inputs, LockPeeker
generates many discrete SLTs. It infers conditions and deduces a
CLT out of those generated SLTs to represent the locks of an API
method. For simplicity, we assume that an input is compared with
only an operator (e.g., $>$, $<$, and $==$), and the values to compare
are constants.

Infering a decision tree LockPeeker uses a decision tree [25] to
classify SLTs. In a decision tree, a leaf represents a tree structure
that is shared by a number of SLTs; the path from the root to a leaf
represents a classification rule for the tree structure. LockPeeker
infers the decision tree with the C4.5 algorithm implemented in an
open source machine learning tool Weka [7]. The algorithm takes
encoded SLTs as its inputs, and infers a decision tree. An SLT is
encoded by its tree structure and inputs, where the tree structure is
used for classifying and the input values for inferring rules for each
classification.

Construting CLTs Based on the decision tree, LockPeeker
constructs CLTs. A tree structure can be restored from a leaf of a
decision tree (DT), and the condition for such tree structure be deduced
from the path from DT’s root to the leaf. The CLT is constructed
by putting the condition to the root of the tree structure. Multiple
leaves can lead to the same tree structure, indicating the condition
for such a CLT is a union of all the paths.

Merging CLTs Algorithm 3 describes how CLTs are merged. Most
of the algorithm is similar to Algorithm 2 except that: (1) no checks
for Relation 1 or Relation 2 are performed at the beginning, be-
cause nodes conditions are still required to merge even if any of the
relations is satisfied; (2) conditions are combined with an $or$
operator (line 6) when two nodes are same (nodes $node_1(v_1, con_1)$
and $node_2(v_2, con_2)$ are same, if $v_1 == v_2$).

Figure 5 shows an example of the whole process of revealing la-
tent locks in the $foo$ method in Figure 2. It starts from an initial
test instance which is then iteratively mutated to generate SLTs. Two
SLTs ($tree_1$ and $tree_4$) have two nodes, and the other two
SLTs ($tree_2$ and $tree_3$) have three nodes, more SLTs with simi-
lar structures are not shown. After they are encoded, $tree_1$ and
$tree_4$ are grouped into $locktree_1$, and $tree_2$ and $tree_3$ are grouped
into $locktree_2$. LockPeeker infers a decision tree with three leaves
from encoded SLTs. The right two leaves refer to $locktree_2$, and
the left leaf refers to $locktree_1$. Based on the decision tree, Lock-
Peeker constructs two CLTs with conditions specified on roots.
$locktree_1$’s condition is parsed from the decision tree’s leftmost
path, $locktree_2$’s condition is parsed from the other two paths. At
last, two CTLs are merged into one and merged conditions are simi-
lified (i.e., $f1$’s condition is simplified to $v_1 < 0 \lor v_2 == TRUE$, and the other nodes’ conditions $true$).
Algorithm 3: cur.cMerge(target)

**Input:** Current CLT, cur

**Output:** Current CLT merged with the objective CLT

1: for each targetChild in target.children do
2:     node2Add ← null
3:     for each curChild in cur.children do
4:         if targetChild.isSame(curChild) then
5:             node2Add ← null
6:             curChild.condition ← curChild.condition ∨
7:             targetChild.condition
8:         break
9:     end if
10: end for
11: if node2Add ≠ null then
12:     cur.add(node2Add)
13: end if
14: return cur

4. EVALUATION

We have developed a tool for LockPeeker, and conducted evaluations on ten open source projects. The evaluation is to answer the following four research questions:

- **RQ1:** How effective is LockPeeker in revealing locks in Java API methods (Section 4.1)?
- **RQ2:** What kinds of deadlocks can be detected, if our detected latent locks are integrated (Section 4.2)?
- **RQ3:** What is the significance of LockPeeker’s threshold (Section 4.3)?
- **RQ4:** What are the essential test instance variables that may trigger locks (Section 4.4)?

Correspondingly, our evaluation results, which will be explained in this section, indicate that (1) LockPeeker detects 74.9% locks from real-world project methods; (2) with our detected latent locks, existing tools are able to detect real deadlocks that involve latent locks in native methods; (3) when the threshold is 100, we have 90.8% of confidence that the conditions and structures of locks can be fully revealed; and (4) a majority of locks are placed on method parameters, fields, and receivers.

4.1 RQ1. Detected Locks

Table 1: Subjects. In this table, M and EM represent the total number of methods and the number of evaluated methods, respectively; L and EL represent the total number of locks and the number of evaluated locks, respectively.

<table>
<thead>
<tr>
<th>Project</th>
<th>LOC</th>
<th>M</th>
<th>EM</th>
<th>L</th>
<th>EL</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBCP</td>
<td>5,792</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>1.2</td>
</tr>
<tr>
<td>Derby</td>
<td>357,575</td>
<td>535</td>
<td>34</td>
<td>581</td>
<td>37</td>
<td>10.5.1.1</td>
</tr>
<tr>
<td>FtpServer</td>
<td>12,039</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>1.0.6</td>
</tr>
<tr>
<td>Groovy</td>
<td>119,586</td>
<td>42</td>
<td>22</td>
<td>44</td>
<td>23</td>
<td>1.7.9</td>
</tr>
<tr>
<td>HsqlDB</td>
<td>165,787</td>
<td>97</td>
<td>30</td>
<td>105</td>
<td>32</td>
<td>2.3.3</td>
</tr>
<tr>
<td>Log4j</td>
<td>15,615</td>
<td>39</td>
<td>13</td>
<td>43</td>
<td>15</td>
<td>1.2.15</td>
</tr>
<tr>
<td>Lucene</td>
<td>45,842</td>
<td>104</td>
<td>21</td>
<td>126</td>
<td>24</td>
<td>2.9.3</td>
</tr>
<tr>
<td>Pool</td>
<td>1,891</td>
<td>13</td>
<td>10</td>
<td>13</td>
<td>10</td>
<td>1.2</td>
</tr>
<tr>
<td>Tomcat</td>
<td>218,882</td>
<td>417</td>
<td>33</td>
<td>464</td>
<td>35</td>
<td>8.0.29</td>
</tr>
<tr>
<td>Xalan</td>
<td>12,039</td>
<td>23</td>
<td>13</td>
<td>24</td>
<td>13</td>
<td>2.7.2</td>
</tr>
<tr>
<td>Total</td>
<td>955,084</td>
<td>1,283</td>
<td>189</td>
<td>1,414</td>
<td>203</td>
<td></td>
</tr>
</tbody>
</table>

4.1.1 Setup

Subjects: Ten popular open source projects used in concurrency testing researches [20] are employed as our subjects (Table 1). LockPeeker takes their methods as API methods, and does not analyze their code. The ground truth is established by a code reading of these projects. We firstly searched in the ten projects the synchronized keywords that are enforced within methods, and in total, 1,414 synchronized keywords were found in 1,283 methods. The results were analyzed by three software engineering graduate students to identify the sources of locking objects, the locking structures and locking conditions. This work is manually completed, as few tools can just meet our requirements.

As discussed in Section 3.2.1, we focus on locking candidates from method parameters, fields, and receivers. To guarantee the representativeness, we selected locks from all of the three types, which include 189 methods with 203 locks: All of the 47 methods that acquire 49 locks on method parameters were selected, and for fields and receivers, we randomly selected ten methods from each project. We thus obtained 77 methods that acquire 86 locks on fields, and 67 methods that acquire 68 locks on receivers. There are two overlapping methods for parameters and fields.

Table 1 shows our subjects. Columns “Project”, “Version”, “LOC”, “M”, “EM”, “L”, “EL”, and “Version” list their project names, versions, project sizes, the numbers of methods that have locks, and the number of methods used in the evaluation respectively. Columns “L” and “EL” list the numbers of synchronized keywords inside methods and those of evaluated locks, respectively.

Metrics: We compared our detected locks with the ground truth at three levels: (1) whether a lock is detected; (2) whether the relations (sibling, nesting) among locks are detected, where the sibling sequence is ignored; and (3) whether the branching specifications
(e.g., condition expressions, catch clause, and loops) are detected. When all tree levels are reached, we say that the lock is strictly detected. When the first two levels are reached, the lock is loosely detected. Meanwhile, even if a detected lock is loosely correct, it helps reveal deadlocks.

The definitions of our recall are:

\[
RC_1 = \frac{\text{strictly detected}}{\text{total}}
\]

\[
RC_2 = \frac{\text{loosely detected}}{\text{total}}
\]

where strictly detected (or loosely detected) denotes the number of the strictly (or loosely) detected locks, and total denotes the total number of locks.

**Default call sequence and input values** To reveal locks, a method shall be invoked with appropriate call sequences and values. Typically, tools analyze sources to obtain call sequences and values. For example, GRT [22] can generate correct call sequences. But such techniques are not ready for API methods whose source are inaccessible. We thus use the default values for primitive objects, instantiate complicated objects by their default constructors or mocked objects, but omit call sequences.

### 4.1.2 Result

LockPeeker does not report any false positives in the evaluation. We manually compare the reported locks with the ground truth. All reported locks reside in the subject programs.

Table 2 shows the recalls of our detected locks. Columns “Parameter”, “Field”, and “Receiver” list detected locks on method parameters, fields, and receivers, respectively. Column “Total” lists total detected locks. Sub-columns “RC1”, “RC2”, and “#Locks” list the RC1s, RC2s, and the number of locks, respectively.

The results show that (1) LockPeeker detects 74.9% of locks in total, and (2) it effectively detects all of the three types of locks. We inspected the inferred locks, and found that some inferred locks are complicated. For example, the `forceFlush` method in Derby has a nested lock when a boolean field is `false`. Below shows the relevant code:

```java
if (stopShipping) return;
synchronized (forceFlushSemaphore) {
    synchronized (objLSTSlock) {...}
}
```

LockPeeker successfully detects structures and conditions of many such locks. However, it fails in detecting 49 locks due to two main reasons, the details shall be discussed in Section 6.

- Incomplete call sequences and input values. 42 out of 49 locks are not detected, since special call sequences and input values are needed to trigger these locks. As API methods are in black boxes, it is infeasible to explore call sequences and input values.
- Complicated code structures. 7 out of 49 locks are not detected, since their structures and conditions are rather complicated.

In addition, as discussed in section 3.2.1, LockPeeker omits searching for the locks on variables of primitive types. It is worth noticing that programmers can enforce locks on such variables in practice, which violates our initial assumption. For example, the programmers of the Xalan project acquire locks on variables of the Boolean types.

In summary, our results lead to the first observation: LockPeeker is able to detect more than two thirds of locks in real-world APIs, even if their sources are inaccessible. In addition, the variety of the subjects in Table 2 is high. Although LockPeeker is less effective in detecting locks from complicated methods, these complicated methods are not evenly distributed in the selected subjects.

### 4.2 RQ2. Detected Deadlocks

#### 4.2.1 Setup

We integrate LockPeeker with a deadlock detection tool named CheckMate whose idea is originally proposed by Joshi et al. [13] and reimplemented in our previous work [20]. CheckMate instruments source code of a program under analysis to collect its concurrency behaviors (including synchronizing, starting, joining, waiting, and notifying threads). After executing the instrumented program, CheckMate records a trace program, which is an execution model of the original program. CheckMate then employs JPF to detect deadlocks from the execution model. We compared the capabilities of CheckMate in detecting deadlocks, before and after it is integrated with LockPeeker.

**Subjects.** Table 3 shows the descriptions and the repair time of our selected subjects. The following call sequences trigger deadlocks with latent locks:

1. The New Relic bug of Jboss.\(^6\) The native method `forName0` has a latent lock.
   - Thread1: `lock(forName0's parameter, ModuleClassLoader) → lock(Verifier);`
   - Thread2: `lock(Verifier) → lock(ModuleClassLoader).`

2. IBM IV-30066.\(^7\) The native method `forNameImpl` has a latent lock.

\(^6\)https://discuss.newrelic.com/t/jboss-7-1-1-crashs-with-deadlock/
\(^7\)http://www-01.ibm.com/support/docview.wss?uid=swg1IV30066
A deadlock happens when a data import thread opens a lot of files in a loop and another thread attempts to initialize a BIRT report engine in the same time. New Relic does not start up correctly because of a deadlock in 3 threads when New Relic is enabled.

<table>
<thead>
<tr>
<th>Bug</th>
<th>Bug Description</th>
<th>Repair Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>New Relic with JBoss</td>
<td>JBoss does not start up correctly because of a deadlock in 3 threads when New Relic is enabled.</td>
<td>Unknown</td>
</tr>
<tr>
<td>IBM IV-30066</td>
<td>In JVM class loading, there is a validation whether a class is already in loading process or not. But it may deadlock during this validation.</td>
<td>7 days</td>
</tr>
<tr>
<td>Eclipse BIRT-287102</td>
<td>A deadlock happens when a data import thread opens a lot of files in a loop and another thread attempts to initialize a BIRT report engine in the same time.</td>
<td>7 days</td>
</tr>
<tr>
<td>Groovy-4736</td>
<td>When multiple threads use GroovyClassLoader to get Groovy classes or just work with them and an other(s) threads change the sources, then deadlock can happen.</td>
<td>3 years</td>
</tr>
</tbody>
</table>

Table 3: Found real-world deadlock bugs that involve latent locks.

<table>
<thead>
<tr>
<th>Found locks in the method</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Class.forName0(Ljava/lang/String;ZLjava/lang/ClassLoader;) synchronized (p3)</td>
</tr>
<tr>
<td>p3: the third method parameter</td>
</tr>
</tbody>
</table>

Figure 6: An example of latent lock reported by LockPeeker.

- Thread1: lock(forNameImpl's parameter, ExtClassLoader) → waiting Thread2 terminates;
- Thread2: lock(ExtClassLoader) → terminate).

3. Eclipse BIRT-287102. The native method forName0 has a latent lock.
- Thread1: lock(forName0's parameter, ChildFirstURLClassLoader) → lock(JarFile);
- Thread2: lock(JarFile) → lock(ChildFirstURLClassLoader).

4. Groovy-4736. The two native methods forName0 and getDeclaredFields0 both have latent locks.
- Thread1: lock(HashMap) → lock(GroovyClassLoader$InnerLoader in getDeclaredFields0);
- Thread2: lock(forName0's parameter GroovyClassLoader$InnerLoader) → lock(HashMap).

4.2.2 Result

CheckMate originally detects none of the four deadlocks. After it is integrated with LockPeeker, CheckMate detects all the four deadlocks, since LockPeeker supplements CheckMate with the latent locks inside the unanalyzable API methods.

Example We use BIRT-287102 in Figure 1 to illustrate how our detected locks improve CheckMate.

CheckMate alone does not report the deadlock in BIRT-287102, as it is not aware of the existence of latent locks inside the native method forName0. The trace program (Figure 7a) generated from the buggy code (Figure 1a) shows that the two threads both request the lock on obj1. CheckMate alone does not report any deadlock for this because there is actually no cyclic locking behaviors in the trace program.

LockPeeker supports CheckMate with support in finding potential deadlock in two steps:
- Detecting latent locks. LockPeeker scans all the API methods that are called in the buggy code, and finds that the forName0 method has a latent lock on its third method parameter which is named as p3. For the native method, LockPeeker generated a CLT that has only one node, p3, indicating that the method has a latent lock on its third parameter with condition true. Figure 6 shows the reported latent locks.

CheckMate executes the resulting program, and generates a new trace program (Figure 7b) which has the latent lock. As highlighted by the gray background, a new lock on obj2 and relevant locking statements are newly generated. JPF thus can report the deadlock in BIRT-287102 by checking the new trace program.

Since the combination of LockPeeker and CheckMate advocates an idea of modifying the objective code, rather than modifying CheckMate itself, the resulting program with supplemented method replacements can also be used by other tools for detecting deadlocks. However, it still requires human efforts in matching locking scopes and callbacks (i.e., determining into which locations the method replacements should be instrumented), we plan to automate the matching process in future.

4.3 RQ3. Significance of Threshold

As Section 3.2 explains, the blind exploration phase requires a threshold to terminate its exploration. Here we use different threshold values to evaluate the impacts of the threshold on different types of conditions.

4.3.1 Setup
Subjects. The impacts of the threshold highly depend on locking condition’s structures.

According to our analysis on the locks in Table 1, conditions on locks can be atomic conditions or compound ones. An atom condition can be a nominal condition that determines whether values equalize to constants, or a numeric condition in which numeric values are compared. A compound condition is composed of two or more atomic conditions. It has two types of information: (1) a condition structure for organizing two or more atomic conditions using logical operators, and (2) constant values indicating options of nominal conditions and boundary values of numeric ones.

We found that 334 out of 1,414 locks have conditions. Most conditions are simple, since 306 out of 334 have two or less atomic conditions. As compound locking conditions consisting of four or more atomic conditions are rare, we analyzed those consisting of at most three atomic conditions.

From the ten projects in Table 1, we investigated the significance of the threshold by selecting six methods. Each method has locking conditions. Thus we collected two atomic conditions, three compound conditions with two atoms, and a compound condition with three atoms. We simplified each method such that the method only keeps the locking conditions and locking actions. The representativeness of the samples can be calculated by

\[
\text{representativeness} = \frac{\text{condition type}}{\text{total}}
\]  

(5)

where \(\text{condition type}\) is the number of locks in a type, and \(\text{total}\) the number of total locks with conditions (334 in our evaluation).

Metrics. We randomly searched the test instances. Following the guideline of Arcuri and Briand [2], for each threshold value, we executed our approach for one hundred times. We define two criteria to measure our results:

\[
C_1 = \frac{\text{structure detected}}{\text{total run}}
\]  

(6)

\[
C_2 = \frac{\text{detected}}{\text{total run}}
\]  

(7)

where \(\text{structure detected}\) is the times of correctly detecting the condition structures of locks, \(\text{detected}\) the times of correctly detecting both the condition structures and their boundary values, and \(\text{total run}\) is one hundred.

4.3.2 Results.

Table 4 shows the results. Columns “Project”, “Method”, and “Condition Type” list the sampled projects, methods, and condition types. Column “\(P_i\)” lists the representativeness values that are calculated by Equation 5. Columns “20” to “2500” list \(C_1\) and \(C_2\) values calculated by Equations 6 and 7, when threshold values are set from 20 to 2500.

The results indicate that in blind exploration phase 100 is sufficient for detecting both structures and values for most condition types (i.e., the top five types). However, blind exploration alone cannot handle more complicated types, as it fails in detecting conditions for three numeric conditions.

To reveal the overall impact of a threshold, for \(C_1\) and \(C_2\), their total values are calculated as:

\[
total_{C_1} = \sum P_i \times C_1
\]  

(8)

\[
total_{C_2} = \sum P_i \times C_2
\]  

(9)

When the threshold value is 50, the total values of \(C_1\) and \(C_2\) are 81.8% and 66.2%, respectively. When we chose some larger threshold values, the total values of \(C_1\) and \(C_2\) did not increase significantly. Thereafter, we selected 100 as the default threshold value, when the total values of \(C_1\) and \(C_2\) are 90.8% and 80.4%. The total value of \(C_2\) only increases to 80.6% when the threshold is 2,500.

4.4 RQ4. Essential Variables

We next investigate which variables can trigger locks in API methods.

4.4.1 Setup

To answer which variables in test instances can cause locks inside API methods, we asked the three graduate students in Section 4.1 to classify locks into categories according to their origins. They analyzed all the locks in Table 1.

4.4.2 Result

Figure 8 shows the results. The 1,414 locks fall into the four categories:

- \(m\)'s parameters, denoted as \(S_{mp}\).
- An instance of class \(C\) (usually referred by \textit{this}), or the class object of \(C\) (i.e., \(C.class\), denoted as \(S_i\).
- Class \(C\)’s fields, denoted as \(S_f\).
- Instances, returned values from other classes, considered as environment variables, denoted as \(S_e\).

Our study shows that the three types of locks such as \(S_i, S_f, \) and \(S_{mp}\) cover most locks: nearly two thirds (904 out 1,414) of
the locks are acquired on fields \( S_i \); more than a quarter (385 out of 1,414) of locks are acquired on their receivers \( S_i \); the remaining locks are acquired on environment variables \( S_e \) and method parameters \( S_{mp} \).

4.5 Threats to Validity

The threat to external validity includes the representativeness of our selected subjects. Our evaluations were conducted on the ten open-source projects. Although the ten projects have nearly one million lines of code and are widely used, our analyzed code is limited, and the data from other projects can be different. The threat could be further reduced by introducing more projects as our subject in future work, which shall cover more locking idioms.

The internal threat comes from the false negatives. In our evaluation, some locking candidates are omitted. Although rare, locks on \( S_e \) can still exist, leading to some false negatives. We plan to find a solution to exploring the environment variables for API methods and revealing latent locks on \( S_e \) in our future work.

5. RELATED WORK

Deadlock detection in APIs Researchers have tackled the problem of detecting deadlocks caused by third party libraries. Williams et al. [31] summarize API code to speed up code analysis. Henhet al. studies the define-use constrains for multilingual programs. Tan on the debugging environment for multilingual programs. Tan [30] work on exception checking for JNI.

Lee and Onodera [16] statically analyze JNI programs to detect operators in native code to locate faults for multilingual bugs. Kondoh and Onodera [18] statically extract memory access models for both client code and native code, and inserts locks to guarantee atomicity.

Comparatively, our approach synthesizes locking code for API methods, complementing the preceding approaches.

6. DISCUSSION AND FUTURE WORK

As the first work on detecting latent locks in APIs, there are still adequate places for improvements. To provide insights for follow-up researchers, we carefully analyze the failures in Table 2, and we identify the following directions that need further exploration:

Call sequences and inputs When a target method requires sophisticated call sequences and input values to activate a lock acquisition, LockPeeker may not provide enough information to support such method invocations. Therefore, the target method execution may be ended too early to reach the locks, causing no false negatives. However, it is possible to support such situation by analyzing APIs’ documents or searching code base for existing client code that calls such APIs to find necessary preconditions to assure method execution.

Repeated locks A thread can lock a resource multiple times in a method, sequentially or nestedly, but Algorithm 1 can detect only the first locking. Suppose a lock, \( L \), is acquired twice in \( thread_1 \), and to reveal the second locking activity, it requires \( thread_2 \) to hold \( L \) just between the two acquisitions. In addition, for a nested situation, the acquisition in \( thread_2 \) will be blocked, if the object is already locked. Our current implementation still cannot handle the situations, but it may be feasible with the support of JDI\(^{10}\) (Java Debugging Interface).

Complicated condition branch LockPeeker cannot recognize complicated condition branches and conditions on complicated objects. For example, LockPeeker can discover locks inside \( catch \) clauses when the target method step into it, but cannot infer the corresponding \( try-catch \) clauses. As another example, it is common in Java to check conditions relevant to the returned value of a method invocation, but LockPeeker fails in recognizing such conditions.

7. CONCLUSION

Locks can be latent in API methods, which are not rare, but difficult to be detected. As existing approaches typically treat API methods as empty boxes, they are insufficient to detect deadlocks that involve latent locks in API methods. In this paper, we propose a novel approach, called LockPeeker, that detects latent locks in Java API methods by extensively testing a method, observing the locking behaviors, and inferring the locking structure and conditions. Our evaluation results have demonstrated that LockPeeker is capable of derive a clear locking tree for an API method with a small set of test instances. We believe that developers can use LockPeeker to identify the latent locks in API methods and improve the robustness of Java applications.

8. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their constructive comments. We would also thank Weizhao Yuan, Yingyi Wang and Xuliang Liu for their help in analyzing subject projects. This work is sponsored by the 973 Program in China (No. 2015CB-352203), the National Nature Science Foundation of China (No. 61572312, No. 61572313, and No. 61272102), and the grant of Science and Technology Commission of Shanghai Municipal (No. 15DZ1100305).

\(^{10}\)http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/architecture.html
9. REFERENCES