Keshmesh: Bringing Advanced Static Analysis to Concurrency Bug Pattern Detectors

Mohsen Vakilian\textsuperscript{1}, Stas Negara\textsuperscript{G}, Samira Tasharofi\textsuperscript{M}, and Ralph E. Johnson\textsuperscript{1}

\textsuperscript{1}University of Illinois at Urbana-Champaign, \textsuperscript{G}Google, \textsuperscript{M}Microsoft
mvakili2@illinois.edu, snegara@google.com, samirat@microsoft.com, rjohnson@illinois.edu

Abstract. Bug patterns are coding idioms that may make the code less maintainable or turn into bugs in future. The state-of-the-art tools for detecting concurrency bug patterns (CBPs) perform simple, intraprocedural analyses. While this simplicity makes the analysis fast, it does not provide protection against CBPs that involve aliasing or multiple methods. This paper introduces a practical and extensible framework, Keshmesh, which employs advanced static analysis for detecting CBPs. Keshmesh builds upon the points-to analysis of WALA, a static analysis framework, and the user interface of FindBugs, a popular bug pattern detection tool. Keshmesh detects five CBPs using interprocedural analyses and fixes two of them. The challenges in automatic detection of these CBPs include reducing the rate of false positives, scaling to large projects, and accurate propagation of unsafe accesses along the call graph up to synchronization constructs. Keshmesh found 40 previously unknown CBP instances and only 12 false positives in six open-source projects. Programmers fixed 11 of the 20 issues we reported. These results show that Keshmesh is applicable to large projects, finds CBPs that programmers want to fix, and reports few false positives.

1 Introduction

Bug patterns are coding idioms that are likely to result in confusion or unexpected results. For instance, the use of the return value of \texttt{getClass()} as a synchronization object is a bug pattern, because it may yield unexpected synchronization behaviors in the presence of subclasses and inner classes (Sec. 2.2).

Though a bug pattern may not cause an actual bug, it is recommended that programmers avoid bug patterns for three major reasons. First, bug patterns may cause actual bugs in future as the software evolves. Second, it is often cheaper to fix bug patterns before they turn into actual bugs. Third, a bug pattern makes it difficult for programmers to infer the intended behavior of the code, reducing the maintainability of code.

Even experienced programmers introduce obvious bug patterns [10]. The results of a recent study at Google showed that a considerable number of simple bug patterns persist through tests and code reviews [4]. Error-prone [4], a bug pattern detection tool, found a total of 302 instances of six simple bug patterns,
such as “Empty if statement”, “Exception created but not thrown”, and “Objects.equal self assignment”, in Google’s code base. The desire to eradicate bug patterns has led to the popularity of FindBugs [1], an automatic bug pattern detector.

Concurrency bug patterns (CBPs) are a class of bug patterns specific to concurrent programs. Many CBPs are complex and their precise detection requires points-to or interprocedural analyses. The SEI CERT catalog [14] is a recent and comprehensive catalog of CBPs for Java.

Existing bug pattern detectors employ intraprocedural analyses and ignore aliasing relations. Such simple analyses lead to inaccurate results and leave the code unprotected against complex bug patterns that involve aliasing or multiple methods.

This paper introduces KESHMEH, a practical framework that brings advanced static analysis to CBP detectors. KESHMEH builds upon the strengths of WALA [3], a modern static analysis framework, and FindBugs, a popular bug pattern detector. The SEI CERT catalog has rated the severity, likelihood, and remediation cost of each bug pattern. We prioritized the CBPs based on these three attributes and selected five of the top ten CBPs. We built KESHMEH to detect a generalized form of these five CBPs and provide automated fixes for two of them. Along the way, we gradually factored the common machineries of the detectors and fixers into an extensible framework. This framework makes it easier to add support for other CBPs. Nonetheless, due to the complexity of CBPs, extending KESHMEH to support a new CBP is still a nontrivial programming task.

We leveraged the infrastructure provided by WALA for points-to and interprocedural analysis to implement custom analyses for detecting five CBPs. To overcome the scalability issues of points-to analysis, KESHMEH customizes the context-sensitivity of WALA and limits the analysis to parts of the program reachable from the methods annotated by the user as entry points. KESHMEH presents its findings to users through the user interface of FindBugs. This integration with FindBugs makes KESHMEH easily accessible to the large user base of FindBugs and compatible with existing continuous integration systems.

The KESHMEH framework provides two key features for detecting CBPs. First, it uses a hybrid notion of context-sensitivity (Sec. 4.2) for points-to analysis that strikes a good balance between scalability and accuracy. The hybrid context-sensitivity of KESHMEH uses type-sensitivity for Java Development Kit (JDK) classes, call-site sensitivity (kCFA) for static methods, and object-sensitivity for instance methods. Second, KESHMEH provides an interprocedural, synchronization-aware data flow framework for propagating thread-unsafe accesses along the call graph (Sec. 3.2).

KESHMEH found 51 instances of CBPs and only 12 false positives in six open-source projects (Apache Tomcat, Apache Commons Pool, OkHttp, PyDev, Derby, and Elevator), 40 of which were not previously known. We reported these findings in the form of 20 issues to the issue tracking systems of the corresponding projects. The programmers fixed 11 of the reported issues. These results show
that Keshmesh is applicable to real-world, large programs, has a low rate of false positives, and finds bug patterns that developers want to remove (Sec. 5).

This paper contributes to the detection of CBPs in several ways:

1. We introduce Keshmesh, a practical framework for detecting and fixing CBPs. Keshmesh is open source and available at http://keshmesh.cs.illinois.edu.
2. We developed novel algorithms for detecting two CBPs (Sec. 3.2) that required sophisticated interprocedural analysis.
3. We used the Keshmesh framework to implement five CBP detectors and two fixers. The variety of the supported bug patterns shows the extensibility of the framework.
4. Keshmesh found 40 new instances of CBPs in real-world open-source software.
5. We tuned our analysis to make Keshmesh accurate and applicable to large programs. The evaluation results show that the object sensitivity level of the points-to analysis has little impact on the running time and accuracy of Keshmesh.

2 Bug Patterns

This section presents real-world instances of five of the ten highly-rated CBPs of the SEI CERT catalog [14]. Keshmesh supports all of these five bug patterns.

2.1 BP1: Synchronizing on an object that may be reused

Objects such as interned strings and boxed primitive values may be reused. For example, boxings of a primitive int value may yield objects with equal or different identities, depending on the magnitude of the int value. Due to the intricacies of the conditions that determine whether such objects may be reused, locking the monitor associated with these objects may result in unexpected behavior. For synchronization, programmers should use objects whose identities can be compared easily, without the need to refer to the Java language specification. This bug pattern corresponds to bug pattern “LCK01-J: Do not synchronize on objects that may be reused” from the SEI CERT catalog [14]. Fig. 1a shows an instance of BP1 (Sec. 2.1) in PyDev.

2.2 BP2: Synchronizing on the class object returned by getClass()

Though any object can be used for synchronization, the use of the return value of the method getClass as a lock object is misleading. The method call e.getClass() returns the Class object of the object that the expression e evaluates to. An interaction of using the return value of method getClass as a lock object with other features such as subclasses or inner classes can lead to improper synchronizations. Fig. 2a illustrates an instance of BP2 (Sec. 2.2) in Tomcat.
Fig. 1: Real examples of BP₁ and BP₃.

(a) An instance of BP₁ in PyDev, which was fixed in Git commit 927add5. Line 5 uses an object of type Integer as a lock object.

```java
public class PyOutlinePage ...
{
    public ... class SourceChannel ...
    {
        public final ReentrantLock lock;

        final InputStream asInputStream() {
            synchronized (this.lock) {
                if (stream == null)
                    this.stream = stream = new ...
                return stream;
            }
        }
    }
}
```

(b) An instance of BP₃ in JaXLib (Subversion revision 1427). Line 6 uses ReentrantLock as the lock object of a synchronized block.

```java
private volatile Integer linkLevel= 1;
void unlinkAll() {
    synchronized (linkLevel) {
        linkLevel--;
        if (linkLevel == 0) {
            removeSelectionListener(...);
            if (linkWithEditor != null) {
                linkWithEditor.unlink();
            }
        }
    }
}
```

Fig. 2: An example of how an instance of BP₂ can turn into a bug as the code evolves.

(a) An instance of BP₂ in Tomcat. Line 4 uses the return value of getClass() as a lock object. This instance was fixed in Git commit 06b8609.

```java
class javaURLContextFactory ... {
    Context getInitialContext(...) ... {
        synchronized (getClass()) {
            if (initialContext == null) {
                initialContext = ...;
            }
            return initialContext;
        }
    }
}
```

(b) The methods getInitialContext in clojureURLContextFactory and javaURLContextFactory (Fig. 2a) do not synchronize their accesses to the shared variable initialContext properly, because getClass() returns different objects in the two classes.

```java
class clojureURLContextFactory extends javaURLContextFactory {
    javaURLContextFactory {
        synchronized (getClass()) {
            initialContext = ...;
            return initialContext;
        }
    }
}
```

Fig. 2b is a hypothetical example illustrating how an interaction of getClass and sublassing can result in unexpected results. If programmers add a new class, called clojureURLContextFactory, which subclasses javaURLContextFactory (Fig. 2a), they may mistakenly assume that it is safe to protect the shared variable initialContext in the subclass by mimicking the behavior of the superclass and locking the return value of getClass. However, since getClass returns different objects when called in a class and subclass, concurrent accesses
to the shared variable `initialContext` through `javaURLContextFactory` and its subclass may conflict.

Similarly, using the return value of `getClass` as a lock object in a pair of outer and inner classes can lead to improper synchronization. Bug pattern BP2 is based on bug pattern “LCK02-J: Do not synchronize on the class object returned by `getClass()`” from the SEI CERT catalog [14].

### 2.3 BP3: Synchronizing on a high-level concurrency object

We consider a class that implements the `Lock` or `Condition` interfaces of the `java.util.concurrent.locks` package a high-level concurrency object. Although the monitor associated with any object can be locked by a `synchronized` block, the use of a high-level concurrency object in such a way is misleading. This is because a high-level concurrency object is meant to be used as a lock through its specific API methods. This bug pattern is based on bug pattern “LCK03-J: Do not synchronize on the intrinsic locks of high-level concurrency objects” from the SEI CERT catalog [14]. Possible fixes are to lock the monitor associated with an instance of `Object` or to lock and unlock the high-level concurrency object by calling its `lock()` and `unlock()` methods, if any. Fig. 1b (line 6) shows an instance of BP3 (Sec. 2.3) in JaXLib.

### 2.4 BP4: Using an instance lock to protect a shared static variable

A single object should be used to protect a shared static variable. If the object used as the lock is a non-static field, then multiple instances of the class may access the shared static variable through different lock objects, leading to conflicting accesses to the shared variable. Thus, it is safer to protect accesses to a shared static variable by a static variable.

This bug pattern is based on bug pattern “LCK06-J: Do not use an instance lock to protect shared static data” from the SEI CERT catalog [14]. BP4 generalizes this bug pattern by considering any object reachable from a static field as shared static data not just static fields. We made this generalization because objects that are reachable from a static field are likely to be shared across multiple instances of the class. Fig. 3a contains an instance of BP4, because it uses an instance lock (line 4) to protect accesses to static field `id` (line 6). Note that the `synchronized` modifier of a method is equivalent to a `synchronized` block on object `this` (an instance variable) that wraps the whole method body.

### 2.5 BP5: Unprotected accesses to shared variables

A non-volatile variable whose accesses are not properly protected by synchronization and locking may yield a data race. The CERT catalog [14] contains two bug patterns for unprotected accesses to shared variables: VNA00-J and VNA01-J. We have generalized VNA00-J and VNA01-J by not restricting the shared data to primitive variables or references to immutable objects. BP5 considers unprotected accesses to any shared data, including primitive variables, references to immutable, and references to mutable objects as bug patterns.
(a) An instance of BP4 in a Tomcat module (Subversion revision 1435416). Method `getSelector` uses an instance lock (this) to protect an access to the shared static field `id` on line 6.

(b) An instance of BP5 in Tomcat. Field `enabled` (line 2) is shared data, but, its accesses (lines 5, 13, and 14) are not mutually exclusive.

Fig. 3: Real examples of BP4 and BP5.

VNA00-J: Ensure visibility when accessing shared primitive variables
Fig. 3b illustrates an instance of VNA00-J. In this example, accesses to the shared primitive variable `enabled` are not protected (lines 2, 5, 13, and 14).

VNA01-J: Ensure visibility of shared references to immutable objects
Like primitive variables, accesses to shared references have to be protected, even if the references refer to immutable objects.

Fixes In general, there are three possible solutions to unprotected accesses to shared variables.

1. Make the shared variable `volatile`.
2. Use classes of `java.util.concurrent.atomic`, which support lock-free thread-safe programming on single variables, e.g., `AtomicBoolean` and `AtomicReference`.
3. Protect accesses to the shared variable using a lock or monitor.

All of the above fixes are applicable to the example in Fig. 3b.
3 Algorithms

This section presents the algorithm for detecting each bug pattern accompanied by pseudocodes. The helper functions used in the pseudocodes are defined in Tab. 1.

To simplify the presentation of the algorithms for detecting BP4 and BP5, we assume a normalized form of the program. However, the actual implementation of KESHMESH supports the original non-normalized form. The normalization replaces every synchronized method by a non-synchronized method with a synchronized block wrapping its whole body. For static methods, the synchronized block uses the corresponding class literal as its lock expression, and for instance methods, the synchronized block uses this as its lock expression.

Table 1: Descriptions of the helper functions used in the pieces of pseudocode. Functions whose names are self-explanatory are omitted.

<table>
<thead>
<tr>
<th>Function</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>callGraph</td>
<td>$E$, a set of entry methods</td>
<td>a context-sensitive call graph created by WALA and rooted at the methods in $E$</td>
</tr>
<tr>
<td>nodes</td>
<td>$G$, a graph</td>
<td>the nodes of $G$</td>
</tr>
<tr>
<td>instructions</td>
<td>$N$, a call-graph node</td>
<td>the bytecode instructions in the WALA’s intermediate representation (IR) of $N$</td>
</tr>
<tr>
<td>pointsToSet</td>
<td>$V$, an IR variable</td>
<td>the set of objects that $V$ may point to</td>
</tr>
<tr>
<td>pointedBySet</td>
<td>$O$, a set of objects</td>
<td>the set of variables that may point to a subset of $O$</td>
</tr>
<tr>
<td>type</td>
<td>$O$, an object</td>
<td>the type of $O$</td>
</tr>
<tr>
<td>callGraphNode</td>
<td>$p$, a variable</td>
<td>the call graph node containing $p$</td>
</tr>
<tr>
<td>method</td>
<td>$N$, a call graph node</td>
<td>the method corresponding to $N$</td>
</tr>
<tr>
<td>isThreadSafe</td>
<td>$T$, a type</td>
<td>Are the fields of the class likely to be accessed concurrently?</td>
</tr>
<tr>
<td>declaringClass</td>
<td>$F$, a field</td>
<td>the class declaring the field</td>
</tr>
<tr>
<td>possibleCallSites</td>
<td>$N_1, N_2$ call graph nodes</td>
<td>the set of possible call sites in $N_1$ to $N_2$</td>
</tr>
</tbody>
</table>

3.1 BP1–3: Synchronizing on objects not suitable for locking

To detect BP1–3, KESHMESH examines the lock object of every synchronized block. WALA’s IR contains a monitorenter instruction for each synchronized block in the source code. If the variable of the monitorenter instruction may point to an object unsuitable for locking, KESHMESH reports the synchronized block as containing an instance of BP1–3 (Fig. 4).
input : $E$, a set of entry methods
output : the set of instances of BP$_1$–$3$

1 $B ← ∅$
2 foreach $N ∈$ nodes(callGraph($E$)) do
3   foreach $I ∈$ instructions($N$) do
4     if $I = $monitorenter $V$ then
5       foreach $O ∈$ pointsToSet($V$) do
6         if isUnsuitableForLocking($O$) then
7           $B ← B ∪ \{ I \}$
8 return $B$

Fig. 4: The template of the algorithm for detecting BP$_1$–$3$. Function isUnsuitableForLocking has to be customized for each of the bug patterns BP$_1$–$3$.

BP$_1$: Synchronizing on an object that may be reused Keshmesh detects reusable objects by examining their allocation sites. More precisely, Keshmesh defines the function isUnsuitableForLocking in Fig. 4 such that it returns true if and only if the allocation site of $O$ is in one of the following locations: String.intern, Integer.valueOf, Long.valueOf, Long.LongCache.<clinit>, Short.valueOf, Short.ShortCache.<clinit>, Byte.ByteCache.<clinit>, Boolean.<clinit>, Float.valueOf, Double.valueOf.

Fig. 5a shows two synchronized blocks. Since the lock expressions of both blocks point to an object allocated inside String.intern, Keshmesh reports both as instances of BP$_1$. Note that an intraprocedural analysis, like that of FindBugs, can identify the first synchronized block (line 3) as an instance of BP$_1$, but not the second one (line 8). This is because the declared return type of method m2 is Object.

void m1() {
  String lock = "LOCK";
  synchronized (lock) {
    ...
  }
}
void m2() {
  synchronized (getLock()) {
    ...
  }
}
Object getLock() {
  return new String("LOCK").intern();
}

(b) Keshmesh marks the synchronized block (line 2) as an instance of BP$_2$.

(a) Keshmesh marks both synchronized blocks (lines 3 and 8) as instances of BP$_1$.

Fig. 5: Keshmesh can accurately detect complex variants of BP$_1$ and BP$_2$ that require interprocedural and points-to analyses.
BP$_2$: Synchronizing on the class object returned by getClass() KEMESH inspects the allocation site of an object to determine if the object is returned by getClass(). That is, KEMESH defines the function isUnsuitableForLocking in Fig. 4 such that it returns true if and only if the type of O is Class and the allocation site of O is method Object.getClass. Relying on an interprocedural points-to analysis empowers KEMESH to detect indirect uses of getClass() as the lock expression of a synchronized block (Fig. 5b). Find-Bugs, as an example of a tool that is limited to intraprocedural analysis, cannot detect the instance of BP$_2$ in Fig. 5b.

Fixer If the objects that the lock expression points to are of the same class, say C, KEMESH provides an automated fix to replace the lock expression by C.class.

BP$_3$: Synchronizing on a high-level concurrency object KEMESH considers an object that implements either the Lock or Condition interface a high-level concurrency object. Thus, it defines the function isUnsuitableForLocking in Fig. 4 such that it returns true if and only if O is a high-level concurrency object.

Fixer In addition, if the high-level concurrency object, say O, implements the Lock interface, KEMESH provides the user with an automated fixer. If the user applies the fixer, it will replace the synchronized block by a try-finally block, which invokes O.lock() at the beginning of the try block and O.unlock() in the finally block.

3.2 BP$_4$–5: Unprotected access to shared data

The rest of this section describes our algorithms for detecting BP$_4$ and BP$_5$. First, we introduce a synchronization-aware framework that both detectors use to propagate unsafe accesses along the call graph. Then, we explain how the two detectors build upon this framework.

Synchronization-aware propagation of unsafe accesses A common interprocedural part of our algorithms for detecting BP$_4$ and BP$_5$ is to propagate unsafe accesses along the call graph up to safe synchronized blocks. The detectors for BP$_4$ and BP$_5$ use this framework by defining functions unsafeAccesses, isSafeSynchronizedBlock, and mayEscapeUnsafeAccesses. Function unsafeAccesses(N) returns those accesses in call graph node N that the detector considers unsafe and would like to propagate along the call graph. isSafeSynchronizedBlock(B) returns true if and only if synchronized block B can protect the unsafe accesses inside it and make them safe. Safe synchronized blocks prevent further propagation of unsafe accesses. Function mayEscapeUnsafeAccesses(C) returns true if and only if the call site C may escape the unsafe accesses of the callee to the caller. If the unsafe accesses cannot escape
from the callee, the framework does not propagate them to the caller. Fig. 6 shows the formulation of the propagation framework as an interprocedural data-flow framework. The framework first computes the unsafe accesses within each call graph node. Then, for each call site \( C \) corresponding to call graph edge \( N_1 \rightarrow N_2 \), if \( C \) may escape unsafe accesses and it is not enclosed by a safe synchronized block, the framework iteratively propagates the unsafe accesses of \( N_2 \) to \( N_1 \). This propagation terminates when the map \( P \) reaches a fixed-point.

**input**: \( G \), the call graph  
**output**: \( P \), a map from each call graph node \( N \) to the set of unsafe accesses propagated to \( N \) \( (P_N) \).

**function**: propagateAccesses\((G)\)

1. \( \text{MeetOperator} \leftarrow \cup \)  
   // Initialize the map \( P \).
2. foreach \( N \in G \) do
3. \( P_N \leftarrow \text{unsafeAccesses}(N) \)  
   // Define the edge transfer function.
4. \( \Phi_{E} \leftarrow \text{function} N_1, N_2 \) do
5. foreach \( C \in \text{possibleCallSites}(N_1, N_2) \) do
6. if \( \text{mayEscapeUnsafeAccesses}(C) \) and \( \exists_B B \in \text{synchronizedBlocks}(N_1) \) and \( \text{doesEnclose}(B, C) \) and \( \text{isSafeSynchronizedBlock}(B) \) then
7. return \( P_{N_2} \)
8. else
9. return \( \emptyset \)
10. solveDataFlowProblem\((G, \text{MeetOperator}, P, \Phi_{E})\)
11. return \( P \)

Fig. 6: BP\(_4\)-5: This function propagates the unsafe instructions of each method along the call graph up to safe synchronized blocks. The propagation is formulated as an interprocedural data-flow problem, which is solved through a fixed-point iteration.

**BP\(_4\)**: Using an instance lock to protect a shared static variable Keshmesh detects instances of BP\(_4\) in several steps:

1. Compute the set \( SF \) of static fields of the program.
2. Override function \( \text{isSafeSynchronizedBlock} \) (Fig. 6) such that \( \text{isSafeSynchronizedBlock}(B) \) returns true if and only if \( B = \text{monitorenter} R \) and the points-to set of \( R \) is a nonempty subset of that of \( SF \).
3. Override function \( \text{mayEscapeUnsafeAccesses} \) to always return true.
4. Override function \( \text{unsafeAccesses} \) (Fig. 6) such that it returns the instructions that may access an object pointed to by a static field. More concretely, let \( \text{unsafeAccesses}(N) \) return all instructions \( I \in N \), where \( \text{may-} \)
ModifyStaticField(I, SF) (Fig. 8) is true and I is not enclosed by any safe synchronized block, as defined by function isSafeSynchronizedBlock.

5. Instantiate the synchronization-aware data-flow framework (Fig. 6) to propagate unsafe accesses along the call graph up to safe synchronized blocks.

6. For each call graph node $N$, collect the set of instructions that may directly or indirectly (interprocedurally) access an object reachable from $SF$ by holding instance lock. In other words, for each call graph node $N$, take the union of the set $P_N$ computed by the data-flow framework (Fig. 6) and the set of method calls in $N$ that have propagated some unsafe accesses to $N$ while solving the data-flow framework.

The propagation of unsafe accesses, which is an interprocedural step of the analysis, is necessary to handle cases such as the one illustrated in Fig. 7.

```java
public class SocketConnector {
    static int id = 0;
    static Object staticLock = new Object();
    void getSelectors() {
        synchronized (staticLock) {
            getSelector();
        }
        getSelector();
    }
    NioThread getSelector() {
        return new NioThread(id++);
    }
}
```

Fig. 7: This hypothetical example shows that the interprocedural analysis of BP$_4$ is necessary to distinguish the safe (line 6) and unsafe (line 8) indirect accesses to field id (line 2).

input : $I$, an instruction  
$SF$, a set of static fields  
output : A boolean value indicating whether the instruction $I$ may modify a static field in $SF$

function: mayModifyStaticField($I$, $SF$)

1. if $I = \text{putstatic} \ V F$ then  
2. return not isFinal($F$)  
3. else if $I = \text{putfield} \ R V F$ then  
4. return pointsToSet($R$) $\cap$ pointsToSet($SF$) $\neq \emptyset$

Fig. 8: BP$_4$: This function checks if an instruction may modify a static field.
**BP₅:** Unprotected accesses to shared variables  To detect BP₅, KESHMESH needs to identify the instructions that may unsafely access some shared data. KESHMESH assumes that a class that implements Runnable, extends Thread, or contains a synchronized block is meant to be thread-safe. It then treats all objects reachable from the fields of thread-safe classes as shared data. An instruction may access shared data indirectly by calling another method. Finding such unsafe accesses requires an interprocedural reasoning. KESHMESH instantiates the synchronization-aware data-flow framework (Fig. 6) to propagate unsafe accesses along the call graph. The data-flow framework transfers unsafe accesses from a callee to its caller if at least one argument of the method invocation may be shared data and the call site is not enclosed by a synchronized block. KESHMESH detects instances of BP₅ in several steps:

1. Find the set of classes that are meant to be thread-safe. In Fig. 9, KESHMESH considers class A as a class meant to be thread-safe, because it contains a synchronized block.

2. Find the instructions in each call graph node that may access some shared data (Fig. 10). KESHMESH considers accesses to final and volatile fields as safe (Fig. 10, lines 2–3). For the remaining accessed fields, if the field is static and belongs to a thread-safe class, KESHMESH considers it shared.

---

```java
public class A {
    int counter = 0;
    @EntryPoint
    public static void main(String args[]) {
        A a = new A();
        a.m();
    }

    void m() {
        /* [VNA00J,01 */
        counter++;
        /* ] */
        /* [VNA00J,02 */
        increment();
        /* ] */
        synchronized (new Object()) {
            counter++;
            increment();
        }
    }

    void increment() {
        /* [VNA00J,03 */
        counter++;
        /* ] */
    }
}
```

Fig. 9: A test input from KESHMESH suite of tests for BP₅. Lines 9, 10, and 17 are marked as expected instances of BP₅. The KESHMESH testing framework ensures that the actual instances of BP₅ found by KESHMESH match the expected ones marked in the input file. KESHMESH does not report any instances of BP₅ on lines 12 and 13, because they are inside a synchronized block. It also does not report an instance of BP₅ on line 6 because the only argument of the method invocation may be shared data and the call site is not enclosed by a synchronized block.
**Fig. 10: BP5:** This function determines if an instruction may access some shared field.

**input**: $I$, an instruction

$N$, a call graph node

**output**: A boolean flag indicating whether instruction $I$ in $N$ may access some shared field

**function**: `mayAccessSharedField(I, N)`

1. if $I = \text{getfield Result, Reference, Field}$ or $I = \text{putfield Reference, Value, Field}$ or $I = \text{getstatic Result, Field}$ or $I = \text{putstatic Value, Field}$ then
2.   if `isVolatile(Field)` or `isFinal(Field)` then
3.     return false
4.   if $I = \text{getstatic Result, Field}$ or $I = \text{putstatic Value, Field}$ then
5.     if `isThreadSafe(declaringClass(Field))` then
6.       return true
7.   else
8.     $P \leftarrow \text{pointedBySet(pointsToSet(Reference))}$
9.     `hasExternalPointer \leftarrow \exists_{p \in P} \text{isLocalVariable(p) and callGraphNode(p) \neq N}$ and not `isInitialization(method(callGraphNode(p)))`
10.    `hasThreadSafePointer \leftarrow \exists_{p \in P} \text{isThreadSafe(type(p))}`
11.    `hasThreadSafeFieldPointer \leftarrow \exists_{p \in P} \text{isField(p) and isThreadSafe(type(container(p)))}`
12.    return `hasExternalPointer` and (`hasThreadSafePointer` or `hasThreadSafeFieldPointer`)

**Fig. 11: BP5:** This function determines if a local variable of a method may get the value of a method parameter or static field. It recursively follows the def-use chain until it reaches a local variable, method parameter, or static field.

**input**: $\text{Var}$, a program variable

$N$, a call graph node

**output**: A boolean flag indicating whether the variable may get the value of a method parameter or static field.

**function**: `isReachableFromMethParamOrStaticField(Var, N)`

1. if $\text{Var} \in \text{parameters(method(N))}$ then
2.   return true
3. else if $I = \text{def(Var)}$
4.   return true
5. else if $I = \text{getstatic Result, Field}$ then
6.   return true
7. else if $I = \text{getfield Result, Reference, Field}$ then
8.   return `isReachableFromMethParamOrStaticField(Reference, N)`
9. else
10.  return false

**data** (Fig. 10, lines 4–6). If the accessed field is non-static and does not correspond to a field of a local variable, KESHMEH considers it shared data.
public class A {
    @EntryPoint
    public static void main(String args[]) {
        new B().accessNonLocally();
        new B().accessLocally();
        new C().increment();
    }
}

class B {
    C c1 = new C();
    void accessNonLocally() {
        /* [VNA00J,01 */
        c1.counter = 1;
        /* ] */
        if (c1.finalCounter == 0) {
            c1.staticVolatileCounter = 1;
        }
    }
    void accessLocally() {
        C c2 = new C();
        c2.counter = 1;
        /* [VNA00J,02 */
        c2.staticCounter = 1;
        /* ] */
    }
}

class C implements Runnable {
    int counter = 0;
    static volatile int staticVolatileCounter = 0;
    final int finalCounter = 0;
    static int staticCounter = 0;
    void increment() {
        /* [VNA00J,03 */
        counter++;
        /* ] */
    }
    public void run() {
        /* .. */
    }
}

Fig. 12: A test input from the KESHMESH suite of tests for BP5 that exercises function MayAccessSharedField (Fig. 10). KESHMESH marks line 12 as containing an access to shared data, because field B.c1 is an external pointer to the object accessed on this line, i.e., counter. Lines 13 and 14 are not marked because they involve final and volatile fields. KESHMESH does not mark line 19 because there is no external pointer to the object accessed on this line.

More precisely, KESHMESH looks for the existence of an external pointer, i.e., a pointer to the accessed object from a different call graph node (Fig. 10, lines 8–9). If there exists an external pointer that has a thread-safe type or is the field of a thread-safe type, KESHMESH considers the accessed field shared data (Fig. 10, lines 10–12). Fig. 12 shows examples of instructions that KESHMESH considers as ones accessing shared data. In Fig. 9, KESHMESH concludes that the instructions corresponding to lines 9, 12, and 17 may access shared data.

3. Override function unsafeAccesses(N) such that it returns any instruction \( I \in N \) that may access some shared data (i.e., \( \text{mayAccessSharedField}(I, N) = \text{true} \)) but is not enclosed by a synchronized block in call graph node \( N \). In Fig. 9, KESHMESH treats the instructions on lines 9 and 17 as unsafe accesses.
4. Override function `isSafeSynchronizedBlock(B)` in Fig. 6 to always return `true`, because this bug pattern considers accesses protected by any `synchronized` block as safe accesses.

5. Override function `mayEscapeUnsafeAccesses(C)` in Fig. 6 to return `true` if the method call `C` has some method argument `a` such that `isReachableFromMethParamOrStaticField(a) = true` (Fig. 11).

6. Instantiate the synchronization-aware data-flow framework (Fig. 6) to propagate unsafe accesses across the call graph. In Fig. 9, `Keshmesh` propagates the instruction on line 17 to line 10, but, it does not propagate the instruction to line 13.

7. Collect the instructions of each call graph node to report to the user. For every call graph node `N`, `Keshmesh` reports any instruction of `N` that may access some shared field unsafely (i.e., `P_N`, where `P` is the map returned by the data-flow framework). In addition, it reports any method calls in `N` that has propagated some instructions to `N` while solving the data-flow framework problem. For the example in Fig. 9, `Keshmesh` reports instances of `BP_5` on lines 9, 10, and 17.

4 Framework

`Keshmesh` is an extensible framework, which provides common functionalities for detecting and fixing CBPs. Nonetheless, adding a new bug pattern detector to `Keshmesh` is still a nontrivial programming task. However, our experience shows that the framework does make it easier to add new detectors and fixers. For example, one of the authors joined the team after two bug patterns were implemented. By reusing the framework, this author was able to implement two bug patterns much faster than what it took the other two authors to implement bug patterns of comparable complexity. This anecdotal evidence indicates the productivity gained by reusing the `Keshmesh` framework. The rest of this section gives an overview of several reusable design elements of `Keshmesh`.

4.1 User-specified entry methods

Users can specify which parts of the program they want `Keshmesh` to analyze. This feature allows the user to focus the analysis on the concurrent parts of the code. Users can use the `@EntryPoint` annotation to specify certain methods as entry points. `Keshmesh` collects the set of methods annotated as such and configures WALA to build a call graph rooted at these methods. Fig. 9 illustrates an example in which the `main` method is annotated with `@EntryPoint`.

4.2 Call graph construction and points-to analysis

`Keshmesh` customizes WALA in two ways. First, it configures WALA’s target method selectors to bypass the methods in third-party JAR files, and thus, limit the size of the call graph.
Second, Keshmesh uses a hybrid context-sensitivity for building the call graph and points-to analysis. The hybrid context-sensitivity consists of $k$-object-sensitivity or type-sensitivity for instance methods, 1-call-site sensitivity for non-instance methods, and type-sensitivity for Java Development Kit (JDK) classes. Users of Keshmesh can provide a configuration file to choose between type-sensitivity or $k$-object-sensitivity for instance methods.

Call-site sensitivity differentiates object allocations and methods by call-site labels. With 1-call-site sensitivity, a method is conceptually cloned for each of its call-sites and each clone is analyzed separately. In general, $k$-call-site sensitivity uses a chain of up to $k$ call sites that lead to the method invocation to differentiate the method and its object allocations.

We chose object-sensitivity because of the accumulating evidence [7, 12, 13, 17, 23] in the literature that object-sensitivity yields better precision and cost for object-oriented programs. $k$-object-sensitivity uses the labels of allocation sites to distinguish object allocations and methods that would have otherwise been collapsed into the same entity. For example, 1-object-sensitivity conceptually clones a method for each of its different receivers. In general, $k$-object-sensitivity uses a chain of up to $k$ receivers to distinguish methods and object allocations.

### 4.3 Synchronization-aware propagation of unsafe accesses

Keshmesh builds upon the data-flow framework of WALA to provide a framework for propagating unsafe accesses along the call graph (Fig. 6). BP\textsubscript{4} and BP\textsubscript{5} use this framework for computing indirect (through method calls) unsafe accesses to shared data.

### 4.4 Reasoning about the structure of concurrent Java programs

Keshmesh provides utilities to inspect the concurrency constructs. For instance, it provides methods to obtain positions of WALA’s instructions in source code, check if an instruction is within a given synchronized block, and an intraprocedural escape analysis.

### 4.5 Integration with FindBugs

Keshmesh integrates with the user interface of the FindBugs plug-in for Eclipse. Keshmesh integrates both the detectors and fixers with FindBugs. FindBugs can be extended by third-party detectors. We have used this feature of FindBugs to integrate the detectors of Keshmesh with it. This allows the user to get the results of Keshmesh detectors by running FindBugs as before.

FindBugs fixers are not as extensible as its detectors. We had to patch the source code of the FindBugs plug-in for Eclipse to make the fixers of Keshmesh available in FindBugs. Keshmesh fixers use the AST Rewrite API of Eclipse JDT to fix instances of the bug patterns. The main AST rewriting part of the fixers are independent of FindBugs.
4.6 Unit testing detectors and fixers

We implemented a testing framework for Keshmesh to test both its detectors and fixers. This framework enables the programmer to write test Java programs augmented with inline comments that specify the expected instances of bug patterns and fixers. The testing framework automatically runs Keshmesh on the test programs and compares the outcomes of the detectors and fixers with the ones specified in the comments. Fig. 9 shows an example input file for testing the detection of BP5.

5 Evaluation

This section reports the results of the experiments that we carried out to answer the following research questions.

RQ1 Does Keshmesh find problems in real-world applications?
RQ2 Does Keshmesh find problems that programmers want to fix?
RQ3 Is Keshmesh applicable to large projects?
RQ4 What is the false positive rate of Keshmesh?
RQ5 How does the degree of context sensitivity affect the accuracy and scalability of Keshmesh?
RQ6 How does Keshmesh compare to other bug and bug pattern detectors?

5.1 RQ1: Does Keshmesh find problems in real-world applications?

We found true instances of all CBPs (BP1–5) in open-source projects. The majority of these instances belong to BP5. Perhaps the complexity of BP5 is the reason for its high frequency in real-world applications.

BP1–3 Keshmesh found one previously unknown instance of bug pattern BP2 (in Tomcat) and no such instances for bug patterns BP1 and BP3. We manually inspected all synchronized blocks in PyDev, and Tomcat, and did not find any instances of these bug patterns (except the one detected by Keshmesh). This manual inspection confirmed that the lack of reports were not due to the potential false negatives of Keshmesh.

Since Keshmesh found few previously unknown instances of BP1–3, we searched bug reports and commit histories of several projects for known instances of BP1–3 in older revisions.

We found five known instances of BP1: two in Jetty, two in PyDev, and one in Toolbox for Java/JTOpen. We found three known instances of BP2: two in Google App Engine and one in SQuirreL SQL Client. Keshmesh detected a previously unknown instance of BP2 in Tomcat. We also found one previously known instance of BP3 in JaXLib. Keshmesh could have detected all of these known bug pattern instances. When we ran Keshmesh on PyDev, we confirmed that it finds the two known instances of BP1.
Searching the older revisions, we manually identified four known instances of BP_4: one in each of Tomcat Modules, XalanJ2, Harmony, and Derby. Keshmesh could have detected all these known instance of BP_4. When we ran Keshmesh on Derby, we confirmed that Keshmesh finds Derby’s known instance of BP_4.

Overall, Keshmesh detected 39 previously unknown instances of BP_5 (Tab. 4). We also found a total of 11 known instances of BP_5, and confirmed that Keshmesh can detect four of them, which appear in the applications we ran Keshmesh on (Tab. 4).

5.2 RQ_2: Does Keshmesh find problems that programmers want to fix?

We ran Keshmesh on six projects (Tab. 2). Keshmesh found 51 instances of CBPs, 40 of which were previously unknown true CBPs. We reported 20 (1 BP_2 and 19 BP_5) issues based on the findings of Keshmesh. The programmers fixed 11 (1 BP_2 and 10 BP_5) of these issues. This indicates that Keshmesh finds problems that programmers want to fix. For example, one programmer wrote to us:

Nice find on the [...] bug. I have no idea how you found it, but it’s a real problem and I got a fix out. The threading code [...] is pretty subtle, especially [...] where there’s a lot of threads to coordinate. I’d love to do more static analysis on it to figure out where the other problems are.

5.3 RQ_3: Is Keshmesh applicable to large projects?

Tab. 2 shows the number of lines of code for each project. All projects except Elevator are medium to large real-world applications, which shows that Keshmesh scales well. Since we are interested in CBPs, we employed several heuristics to select entry points. First, we detected concurrency constructs like synchronized and volatile keywords as well as Thread and Runnable types. Then, we marked as entry points those methods whose call graph would contain the detected concurrency constructs. Our experience shows that establishing entry points is straightforward and does not require a deep knowledge of the underlying code. As an anecdotal evidence, three undergraduate students who worked in two independent groups were able to select appropriate entry points. The students found instances of CBPs in Tomcat and OkHttp. Tab. 3 shows that Keshmesh finished in under a minute for the applications that we ran it on (except OkHttp with unbounded object-sensitivity). We ran Keshmesh while allocating up to 3G of RAM to Eclipse. The bottleneck of Keshmesh is in the preprocessing step, which computes the call graph and points-to information. In each run, Keshmesh performs the preprocessing step once and shares the results with the detectors.
Table 2: The analyzed applications

<table>
<thead>
<tr>
<th>Project</th>
<th>Description</th>
<th>Size (LOC)</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Derby</td>
<td>a relational database management system</td>
<td>616,780</td>
<td>github.com/-apache/derby</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>an implementation of the Java Servlet and JavaServer Pages technologies</td>
<td>589,526</td>
<td>github.com/-apache/tomcat</td>
</tr>
<tr>
<td>PyDev</td>
<td>an Eclipse-based IDE for Python</td>
<td>461,325</td>
<td>github.com/-aptana/Pydev</td>
</tr>
<tr>
<td>OkHttp</td>
<td>an HTTP+SPDY client for Android and Java applications developed at Square, Inc.</td>
<td>13,139</td>
<td>github.com/-square/okhttp</td>
</tr>
<tr>
<td>Apache Commons Pool</td>
<td>an object-pooling library</td>
<td>10,029</td>
<td>github.com/-apache/-commons-pool</td>
</tr>
<tr>
<td>Elevator</td>
<td>a benchmark program for evaluating tools on parallel programs</td>
<td>350</td>
<td>pjbench.-googlecode.com</td>
</tr>
</tbody>
</table>

5.4 RQ4: What is the false positive rate of Keshmesh?

If the code can be made safer in response to a report generated by Keshmesh, we consider the report a true positive, otherwise, a false positive. Keshmesh reported a low number of false positives (12).

Keshmesh did not report any false positives for BP1–3. It reported one false positive for BP4 in a recent revision of Tomcat. Tab. 4 shows the rate of false positives for BP5.

5.5 RQ5: How does the degree of context sensitivity affect the accuracy and scalability of Keshmesh?

We ran Keshmesh with different levels of $k$-object-sensitivity (Tab. 3). When $k = 0$, Keshmesh uses type-sensitivity instead of object-sensitivity for instance methods. When $k > 0$, Keshmesh uses $k$-object-sensitivity. When $k = \infty$, object-sensitivity does not bound the number of receivers for characterizing the contexts.

Tab. 3 shows that the level of context-sensitivity had little impact on construction time and size of the call graph. Moreover, Keshmesh reported the same CBP instances regardless of the level of context-sensitivity. The only significant effect was that Keshmesh did not terminate in 15 minutes when run on OkHttp with unbounded object-sensitivity (Tab. 3). Overall, the degree of context sensitivity for instance methods of the application code did not affect the scalability and accuracy of Keshmesh in practice. However, in theory, higher levels of context-sensitivity lead to higher accuracy. Fig. 13 is a contrived example showing that 1-object-sensitivity produces a false positive that 2-object-
Table 3: Call graph size and running time of Keshmesh for different object-sensitivities. The times are reported in milliseconds. **Entry** stands for entry point. **Context** denotes the level of object-sensitivity. CG stands for call graph.

<table>
<thead>
<tr>
<th>Application</th>
<th>Commit</th>
<th>Entry</th>
<th>Context</th>
<th>CG nodes</th>
<th>CG time</th>
<th>BP_1 time</th>
<th>BP_2 time</th>
<th>BP_3 time</th>
<th>BP_4 time</th>
<th>BP_5 time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Derby</td>
<td>5a93780</td>
<td>1</td>
<td>7</td>
<td>4280</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>455</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>7</td>
<td>4124</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>588</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>7</td>
<td>3879</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>699</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>∞</td>
<td>7</td>
<td>3710</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>516</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>f0d9854</td>
<td>1</td>
<td>2102</td>
<td>10515</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1330</td>
<td>533</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2102</td>
<td>10644</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1205</td>
<td>528</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2102</td>
<td>10697</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1165</td>
<td>550</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>∞</td>
<td>2102</td>
<td>8946</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>1114</td>
<td>585</td>
<td></td>
</tr>
<tr>
<td>PyDev</td>
<td>7332bcf</td>
<td>1</td>
<td>1675</td>
<td>19277</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1848</td>
<td>695</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1675</td>
<td>22585</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1943</td>
<td>487</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1675</td>
<td>18845</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1850</td>
<td>826</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>∞</td>
<td>1676</td>
<td>17785</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2333</td>
<td>865</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20ee5bd</td>
<td>1</td>
<td>1707</td>
<td>17514</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>938</td>
<td>1052</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1707</td>
<td>20154</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>846</td>
<td>911</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1707</td>
<td>19570</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>814</td>
<td>898</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>∞</td>
<td>1707</td>
<td>18533</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>750</td>
<td>884</td>
<td></td>
</tr>
<tr>
<td>OkHttp</td>
<td>9bed92f</td>
<td>1</td>
<td>3832</td>
<td>12054</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2082</td>
<td>1344</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>3864</td>
<td>11150</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>1990</td>
<td>1674</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3873</td>
<td>11777</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2188</td>
<td>1768</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>∞</td>
<td>3880</td>
<td>10957</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>2035</td>
<td>1726</td>
<td></td>
</tr>
<tr>
<td>Apache Commons Pool</td>
<td>b0294a0</td>
<td>1</td>
<td>1817</td>
<td>8048</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>766</td>
<td>755</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1841</td>
<td>7916</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>892</td>
<td>646</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1853</td>
<td>8174</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>889</td>
<td>463</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>∞</td>
<td>1857</td>
<td>8291</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>874</td>
<td>579</td>
<td></td>
</tr>
<tr>
<td>Elevator</td>
<td>213</td>
<td>1</td>
<td>1895</td>
<td>10128</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>885</td>
<td>763</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>1895</td>
<td>9028</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>958</td>
<td>565</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>1895</td>
<td>9690</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>728</td>
<td>663</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>∞</td>
<td>1896</td>
<td>9782</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>765</td>
<td>716</td>
<td></td>
</tr>
</tbody>
</table>
Table 4: Statistics about the instances of BP5. Known stands for the number of known true positives that Keshmesh found. Unknown stands for the number of previously unknown true positives that Keshmesh found. Reported stands for the number of true positives that Keshmesh found and we reported. Fixed stands for the number of true positives that we reported and the developers fixed. False Positives stands for the number of false positive that Keshmesh reported.

<table>
<thead>
<tr>
<th>Application</th>
<th>Known</th>
<th>Unknown</th>
<th>Reported</th>
<th>Fixed</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache Derby</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Apache Tomcat</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>PyDev</td>
<td>2</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>OkHttp</td>
<td>0</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Apache Commons Pool</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Elevator</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td><strong>Overall</strong></td>
<td>4</td>
<td>39</td>
<td>19</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

sensitivity does not. The obscurity of this example explains why additional levels of object-sensitivity do not affect the accuracy of Keshmesh in practice.

5.6 RQ6: How does Keshmesh compare to other bug and bug pattern detectors?

We compared Keshmesh with FindBugs [1], a popular bug pattern detector, and Chord [17], a well-known static data race detector.

FindBugs can detect those instances of bug patterns BP1–3 that do not require interprocedural and alias analyses. This includes the instances of BP1–3 mentioned in Sec. 5.1. However, FindBugs cannot protect against more complex variants of BP1–3 (Fig. 5). Moreover, limited analysis capabilities of FindBugs do not allow it to detect any instances of bug patterns BP4 and BP5. Consequently, FindBugs would not detect 39 out of 40 unknown CBP instances that Keshmesh detected.

Keshmesh and Chord have different focuses and complement each other. Keshmesh detects CBPs while Chord detects data races. To compare Keshmesh and Chord, we ran both tools on Tomcat using the same entry points. Keshmesh detected eight true positives for BP5 and one false positive for BP4, while Chord reported only a false positive and a true positive related to BP5. This result shows that Keshmesh is more effective at finding bug patterns than Chord. Besides, as we showed above, developers are interested in fixing not only
public class A {
    @EntryPoint
    public static void main(String[] args) {
        B b1 = new B();
        synchronized (new Object()) {
            b1.setD(new ThreadSafeD());
        }
        B b2 = new B();
        b2.setD(new D());
        b2.d.value = 0;
    }
}

class B {
    public D d;
    void setD(D d) {
        C.instance.setD(this, d);
    }
}

class C {
    public static C instance = new C();
    void setD(B b, D d) {
        b.d = d;
    }
}

class D {
    int value;
}

class ThreadSafeD extends D implements Runnable {
    public void run() {
    }
}

Fig. 13: A contrived example that illustrates the impact of context sensitivity on the accuracy of Keshmesh. With 1-object-sensitivity, Keshmesh reports an instance of BP5 on line 10. An object-sensitivity level of at least two is needed to distinguish between the object allocations on lines 6 and 9 and avoid reporting a false positive.

bugs but also bug patterns—11 of the bug patterns detected by Keshmesh have already been fixed. At the same time, Chord detected four data races that do not belong to any bug pattern supported by Keshmesh. Thus, bug pattern detectors like Keshmesh and data race detectors like Chord complement each other well.

6 Related Work

There are numerous studies that try to improve the quality of concurrent programs. Two core techniques for detecting concurrency related issues are dynamic analysis and static analysis.

6.1 Dynamic Analysis

Dynamic analysis techniques instrument the programs and analyze them at runtime [6, 11, 16, 19, 20, 22, 25]. The key advantage of dynamic approaches is the
absence of false alarms since they work on real execution of the programs. However, since they only exercise a limited number of inputs and only the paths executed at run time, they suffer from poor coverage. Moreover, dynamic analysis approaches require the program to be executed, which is a burden especially if the program needs complex environmental setup. Another drawback of dynamic approaches is the runtime overhead.

6.2 Static Analysis

Unlike dynamic analysis, static analysis techniques do not require program execution since they work on the abstract model of the programs. The abstract model allows static analysis to have better coverage than the one achieved by dynamic analysis. However, the imprecise nature of the abstract model makes static analysis prone to a large number of false alarms.

Most of the static analysis tools aim for detecting real concurrency bugs, while a few of them target detecting CBPs, which is useful not only for detecting real bugs, but also for advocating best practices in concurrent programming.

**Bug Detectors** Static bug detectors aim for detecting various concurrency bugs. JLint [5] is one of the early tools that uses a limited interprocedural analysis to find deadlocks and data races in concurrent Java programs. Chord [17] is a well-known tool for detecting data races in concurrent Java programs. The race detection algorithm combines different kinds of static analysis to detect the pair of accesses involved in the race. RaceFreeJava [9] is a formal race-free type system for Java that statically detects data races.

Relay [24], and RacerX [8] are static race detectors for C programs that use variants of the lockset algorithm to make the tool scalable. KISS [21] is a tool that detects data races by transforming a concurrent C program to a sequential program which represents a subset of program behavior.

While all of these tools are developed for detecting concurrency bugs, Keshmesh focuses on detecting CBPs.

**Bug Pattern Detectors** Concurrency bug pattern detectors check for the misuses of concurrency constructs that may lead to real concurrency bugs at run time.

The most closely related work to Keshmesh is RSAR [15]. RSAR is a static analysis tool that combines syntactic pattern matching and WALA’s inter-procedural analysis to detect seven bug patterns. However, only the detector of one bug pattern uses WALA’s inter-procedural analysis. All other detectors employ simple syntactic pattern matching. Keshmesh has two key advantages over RSAR. First, Keshmesh leverages much of the power of WALA, including inter-procedural analysis, to detect the CBPs more accurately. Second, Keshmesh detects CBPs listed in a systematically developed catalog of bug patterns, namely, the SEI CERT catalog [14].

SyncChecker [18] is a bug pattern detector that tries to reduce false alarms. It uses the Java framework Soot [2] and combines it with points-to and may-happen-in-parallel (MHP) information to report fewer false alarms. One of the
differences between KESHMESH and SyncChecker is that the bug patterns detected by SyncChecker are not based on any standard catalog and hence they are different from the bug patterns detected by KESHMESH. Moreover, SyncChecker was not executed on large applications, which makes the scalability of the tool opaque.

FindBugs [10] is one of the most popular tools for detecting bug patterns in Java programs. The tool uses syntactic and intra-procedural data flow analysis to detect various bug patterns. Although the simple analysis makes this tool fast, it limits the accuracy and power of the tool in detecting bug patterns. KESHMESH employs interprocedural and alias analysis built on top of the WALA framework to address this limitation of FindBugs.

7 Conclusions

Unlike concurrency bugs, concurrency bug patterns (CBPs) may not cause a failure, but make the code less maintainable and are likely to turn into bugs as the code evolves. Thus, it is recommended to avoid bug patterns.

Existing CBP detectors support an ad-hoc set of bug patterns, do not scale to large programs, or cannot detect CBPs that cross multiple methods or involve aliasing.

We introduce a new framework, KESHMESH, which combines the power of a modern static analysis framework, WALA, with the ease of use of FindBugs, a popular bug pattern detector. KESHMESH provides detectors for five of the bug patterns of the SEI CERT catalog and fixers for two of them. In addition, KESHMESH provides a hybrid context-sensitivity tuned for its detectors and a synchronization-aware data-flow framework for interprocedural propagation of thread-unsafe accesses. The evaluation results show that advanced static analysis is applicable to CBP detection, leading to an effective and scalable CBP detector.

References

1. FindBugs, http://findbugs.sf.net/
3. T.J. Watson Libraries for Analysis (WALA), http://wala.sf.net/