TSVD4J: Thread-Safety Violation Detection for Java

Shanto Rahman, Chengpeng Li, August Shi
The University of Texas at Austin, USA
{shanto.rahman,chengpengli,august}@utexas.edu

***Abstract*—Concurrency bugs are difficult to detect and debug. One class of concurrency bugs are thread-safety violations, where multiple threads access thread-unsafe data structure at the same time, resulting in unexpected behavior. Prior work proposed an approach TSVD to detect thread-safety violations. TSVD injects delays at API calls that read/write to specific thread-unsafe data structures, tracking whether multiple threads can overlap in their accesses to the same data structure through the delays, showing potential thread-safety violations. We additionally enhance the TSVD approach to also consider read/write operations to object fields. We implement the TSVD approach in Java in our tool TSVD4J. TSVD4J can be integrated as a Maven plugin that can be included in any Maven-based application. Our evaluation on 12 applications shows that TSVD4J can detect 55 pairs of code locations accessing the same shared data structure across multiple threads, representing potential thread-safety violations. We find that the addition of tracking field accesses contributed the most to detecting these pairs. TSVD4J also detects more such pairs than existing tool RV-Predict. The demo video for TSVD4J is available at https://www.youtube.com/watch?v=-wSMzlj5cMY.**

I. Introduction

Concurrency bugs are difficult to detect and debug due to their non-deterministic characteristics. Concurrency bugs come in a number of forms, including data races, atomicity violations, and deadlocks [17]. As such, prior work has spent substantial effort to help developers to find concurrency bugs as well as to reproduce their failures [13], [15], [17], [21].

One class of concurrency bugs are thread-safety violations, which occur when multiple threads read and write to the same thread-unsafe data structure, e.g., one thread reading from a non-synchronized Map while another writes to it, resulting in unexpected behavior [15], [21]. Figure 1 illustrates an example thread-safety violation from log4j[1]. The appenderList is a Vector object that can be accessed by multiple threads, and multiple threads can execute the example code concurrently. One thread may read elements from appenderList (Line 7), but at the same time another thread modifies appenderList by removing all elements from the Vector (Line 10). The two code locations form a conflicting-pair representing a thread-safety violation, since one thread can read from the thread-unsafe Vector while another one writes to it, resulting in unexpected behavior (ArrayIndexOutOfBoundsException).

Li et al. recently proposed a technique called TSVD that specifically detects thread-safety violations on thread-unsafe data structures that are part of the .NET framework [15].

[1]https://bz.apache.org/bugzilla/show_bug.cgi?id=54325

```
1  public class AppenderAttachableImpl {
2    public void removeAllAppenders() {
3      if (appenderList != null) {
4        int len = appenderList.size();
5        for (int i = 0; i < len; i++) {
6          Appender a =
7              (Appender) appenderList.elementAt(i);
8          a.close();
9        }
10       appenderList.removeAllElements();
11       appenderList = null;
12     }
13   }
14 }
```

Fig. 1: Example thread-safety violation from application log4j.

Their approach tracks API calls that read/write to these data structures, injecting delays at these calls and tracking whether multiple threads access the same data structure within these delays. If there is an overlap, where one thread performs a write, then it reports a potential thread-safety violation.

We present TSVD4J, an implementation of the TSVD approach but for Java applications. Our implementation handles read/write API calls for common thread-unsafe Java library data structures, such as List and Set, that match what Li et al. implemented for TSVD in their evaluation [4], [15]. Further, we extend the ideas of TSVD to also handle multi-thread accesses to fields. If two threads read/write to the same field of the same object, we also inject delays around these field accesses and check whether there can be a potential thread-safety violation. TSVD4J reports the potential thread-safety violations as conflicting-pairs, which are the two code locations that read/write to the same data. We evaluate TSVD4J on 12 Java applications, including six applications from an existing benchmark of concurrency bugs, JaConTeBe [16]. TSVD4J detects 55 conflicting-pairs. We find that tracking field accesses contributed the most to detecting conflicting-pairs. We also run another concurrency-bug detection tool RV-Predict on the same applications, detecting only 17 conflicting-pairs. Our tool is publicly available [5].

II. Implementation

We implement TSVD4J based on the TSVD approach proposed by Li et al. [4], [15]. The TSVD approach dynamically instruments the underlying code to track read/write API calls on specified thread-unsafe data structures, such as thread-unsafe lists and maps. The intuition is that if two threads

979-8-3503-2263-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-Companion58688.2023.00029

```java
  public class ArrayListDemo {
    public static void main(String[] args) {
      ArrayList<Integer> arr = new ArrayList();
-     arr.add(20);
+     Proxy.add(arr, 20, loc);
  }
```

Fig. 2: Instrumentation changes.

```java
1  public class Proxy {
2    public static void add(ArrayList arr, Object obj,
3        String loc) {
4      long threadId = Thread.currentThread().getId();
5      int objId = System.identityHashCode(arr);
6      Op op = OpCode.WRITE;
7      Runtime.onCall(threadId, objId, loc, op);
8      arr.add(obj);
9    }
10 }
```

Fig. 3: Proxy method for ArrayList.add.

```java
public class Runtime {
  public static void onCall(long threadId, int objId,
      String loc, Op op) {
    Trap t1 = new Trap(threadId, objId, loc, op);
    if (traps.containsKey(objId)) {
      for (Trap t2 : traps.get(objId)) {
        if (t2.getThreadId() != threadId) {
          if (t2.getOp() == WRITE || op == WRITE) {
            reportConflict(loc, t2.getLoc());
          }
        }
      }
    } else {
      traps.put(objId, new LinkedList<>());
    }

    traps.get(objId).add(t1);
    delay();
    traps.get(objId).remove(t1);
  }
}
```

Fig. 4: Entry-point for backend runtime analyzer.

perform read/write operations on the same data structure, and the threads perform these operations relatively close to each other in time, then it is likely the two threads can interleave in a different way as to result in a different order of operations and different behavior involving the data structure. The code locations for these two calls then form a conflicting-pair that should be reported to the user.

TSVD4J consists of an *instrumenter* to modify application's bytecode and a *runtime analyzer* to track specific API calls.

Instrumenter. The instrumenter leverages ASM [1] to modify the application's bytecode at runtime. For each method in a loaded class, the instrumenter looks for method invocations (invokevirtual) of API methods that we aim to track. We currently configure TSVD4J to track the read/write operations for 10 thread-unsafe classes from the Java standard library collections, e.g., List, Map, Set implementations that are not synchronized. The instrumenter replaces each such method invocation to instead invoke the corresponding proxy method that we implement in the backend. Figure 2 shows an example of how the instrumenter changes a call to ArrayList.add to instead invoke Proxy.add (the example shows the change in Java source code for ease of presentation, but all changes are done on Java bytecode). The proxy method takes as input the data structure (e.g., the ArrayList arr), the arguments to the method, and finally the code location of the call, represented as the class and line number on which the call occurs.

Figure 3 shows the proxy method Proxy.add that the instrumenter invokes instead of ArrayList.add. In general, all proxy methods follow a similar structure. The proxy method obtains the currently running thread ID and the object ID for the data structure. Each proxy method is aware of whether the method it is proxying is a read or write operation (e.g., add is a write operation). The proxy method finally passes all the relevant information to the runtime analyzer (Line 7) before invoking the actual method (Line 8) as to ensure the same functionality as before instrumentation.

Runtime Analyzer. The runtime analyzer injects delays and tracks whether read/write operations occur on the same data structure across multiple threads at similar times such that there may be a thread-safety violation. The entry-point to the runtime analyzer is the Runtime.onCall method (Line 7 of Figure 3).

Figure 4 shows the general logic of Runtime.onCall. Runtime.onCall creates a Trap object corresponding to the API call. This Trap object stores all the relevant information (thread ID, object ID, code location, and the type of operation) to be saved and tracked. Runtime.onCall first checks through the global mapping from object ID to Traps to see whether there is a previous Trap corresponding to the same object. If another thread accessed the same object, and one of these accesses (the previous one or this one) is a write operation, then there is a potential conflict. The runtime analyzer reports this code location along with that previous Trap's code location to be a conflicting-pair, representing a potential thread-safety violation. Note that a Trap exists in the mapping if its corresponding thread is still delaying, meaning the accesses may happen out of order on the thread-unsafe data structure within that delay time amount.

Runtime.onCall adds the new Trap into the global mapping of Traps and injects a delay (100ms). The current thread therefore pauses while its Trap remains in the traps mapping. Other threads continue executing and potentially invoke Runtime.onCall to access the same data structure, leading to conflicting-pairs. After the delay, Runtime.onCall removes the Trap as to not be paired with later accesses, which are unlikely to form thread-safety violations.

Aside from the main logic for the runtime analyzer, the TSVD approach relies on other heuristics to adjust delay time, remove unlikely conflicting-pairs, etc., that are described in the original TSVD work [15] and implemented for .NET [4]. We also implement these parts for Java as well.

Test Listener. TSVD4J relies on executions to detect thread-safety violations. A user can either attach the TSVD4J agent to the JVM and execute a main method or existing JUnit tests. We implement a JUnit test listener to track when tests start and end. When a test finishes running, the listener extracts the newly detected conflicting-pairs during that test run and saves them in files associated with that test. As such, TSVD4J intermittently outputs conflicting-pairs per test instead of waiting for the entire test suite to finish running, in case the user has to stop TSVD4J early. Further, we want to associate conflicting-pairs to individual tests in case the user wants to debug further, so they know which test(s) to rerun.

Tracking Field Accesses. In our preliminary study into known concurrency bugs from the JaConTeBe dataset [2], [16], we find that many threads not just access thread-unsafe data structures but also shared object fields. We modify the instrumenter to also track field accesses similarly as for API calls. The instrumenter also modifies getfield/getstatic and putfield/putstatic bytecode instructions, corresponding to read and write operations on fields, respectively. The instrumenter replaces these instructions with the corresponding proxy methods that similarly pass relevant information to the runtime analyzer while also performing the actual operation. Instead of using the object ID, these proxy methods create hashcodes corresponding to the instance object ID (or class name for static fields) along with the field name as to uniquely identify the object/field pair. The remaining logic involving the runtime analyzer remains the same as it delays and tracks the accesses to the same object/field pair across multiple threads.

III. Usage

We implement TSVD4J as a Java agent that can be attached when starting a JVM process by passing in the TSVD4J JAR as an argument through the -javaagent flag. TSVD4J can also be included as a Maven plugin by modifying the pom.xml:

```
<plugin>
  <groupId>edu.utexas.ece</groupId>
  <artifactId>tsvd4j-maven-plugin</artifactId>
  <version>0.1-SNAPSHOT</version>
</plugin>
```

A user invokes the TSVD4J Maven plugin through the command mvn tsvd4j:tsvd4j. The plugin executes the underlying unit tests through Surefire, attaching a Java agent and test listener to instrument code to interact with the backend runtime analyzer. In addition, a user can configure TSVD4J with flags to control whether they want to track API calls or field accesses only (both are tracked by default) [5].

TSVD4J writes all detected conflicting-pairs under the .tsvd4j/ output directory. The output format per file has a conflicting-pair on every line, represented as two code locations (class name and line number), separated by ":".

IV. Evaluation

A. Setup

We evaluate TSVD4J on six applications from JaConTeBe [2], [16] (a benchmark dataset of Java concurrency bugs) that can build using Java 8 and up. In addition, we obtain six popular open-source Java Maven GitHub applications. These applications use multiple threads, meaning they can possibly have thread-safety violations. We run TSVD4J on these applications' unit tests. We use the latest commit for each application so we can report detected potential thread-safety violations to developers.

TABLE I: Application statistics.

ID	Project	SHA	# kLOC	# Tests
J1	JaConTeBe-dbcp	-	9	4
J2	JaConTeBe-derby	-	572	5
J3	JaConTeBe-groovy	-	105	6
J4	JaConTeBe-log4j	-	35	5
J5	JaConTeBe-lucene	-	68	2
J6	JaConTeBe-pool	-	11	5
P1	TooTallNate/Java-WebSocket	aad6654	16	641
P2	davidmoten/rxjava2-extras	d0315b6	19	390
P3	fluent/fluent-logger-java	da14ec3	2	18
P4	javadelight/delight-... [3]	da35edc	2	79
P5	ktuukkan/marine-api	af00038	16	926
P6	openpojo/openpojo	00ed7a0	24	1204
Total			**879**	**3285**

TABLE II: Conflicting-pairs reported by each technique.

ID	RV-Predict	TSVD4J with API Only	TSVD4J with Field Only	TSVD4J
J1	4	0	1	1
J2	1	0	1	1
J3	1	0	1	1
J4	0	0	8	8
J5	0	0	0	0
J6	1	0	1	1
Total	**7**	**0**	**12**	**12**
P1	2	0	25	25
P2	0	3	1	4
P3	4	3	1	4
P4	2	0	3	3
P5	2	4	1	5
P6	0	2	0	2
Total	**10**	**12**	**31**	**43**
Total	**17**	**12**	**43**	**55**

Table I shows the full set of applications on which we evaluate. The top six applications are from JaConTeBe while the remaining applications are the open-source GitHub applications. For each application, we also show the commit SHA on which we evaluate (simply "-" for JaConTeBe) and the size of the application in kLOC and number of tests.

For each JaConTeBe application, we run the provided driver test code that forces the failure. For the other applications, we include the TSVD4J Maven plugin and run on their tests. For comparison purposes, we also run RV-Predict, a tool for detecting Java data races [13], for each application. RV-Predict reports data races similarly as conflicting-pairs, so we can compare the results of both tools. For both TSVD4J and RV-Predict, we use a timeout of 6 hours per application.

B. Results

Table II shows the results of running RV-Predict and TSVD4J. We show the results for RV-Predict and variants of running TSVD4J when tracking only API calls, only

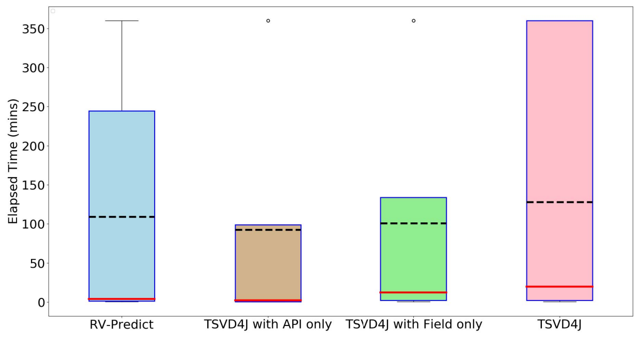

Fig. 5: Runtime of all techniques per application.

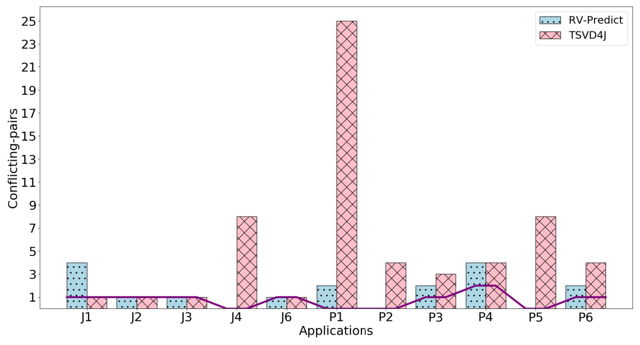

Fig. 6: Distribution of conflicting-pairs per application.

field accesses, and both. We present the total number of conflicting-pairs detected by each technique, divided between just JaConTeBe applications, just GitHub applications, and for all applications. Figure 5 shows the time to run RV-Predict, TSVD4J, and its two variants i.e., tracking API calls only and tracking fields only, in boxplot form. The boxes depict for each technique the time it took to run each one across all applications (solid line is median, dashed line is mean).

Overall, TSVD4J can detect more conflicting-pairs than RV-Predict, at 55 to 17 conflicting-pairs, respectively. The runtimes are similar, though TSVD4J has a slightly higher mean (127.7 versus 108.8 minutes, respectively). We see that tracking field accesses was the most beneficial to detecting conflicting-pairs, detecting 43 conflicting-pairs to the 12 from tracking just API calls, though at a higher cost. Using just one or the other greatly reduces the overall runtime compared with using both, though we still see all techniques reaching the timeout for some applications.

Figure 6 shows the distribution of conflicting-pairs reported by TSVD4J and RV-Predict across all applications (J5 is not shown as both techniques do not detect any conflicting-pair). The figure also shows a line that represents the amount of overlap in detected conflicting-pairs between each technique. Overall, there are 10 conflicting-pairs in common between the two techniques. RV-Predict detects seven conflicting-pairs that TSVD4J does not. Besides missing conflicting-pairs due to inherent nondeterminism, some other reasons for why TSVD4J did not detect them are that they involve API calls that we currently do not support, or the execution with TSVD4J did not reach the relevant parts within the 6 hours timeout. We plan to extend TSVD4J's tracking and improve its efficiency.

For JaConTeBe applications with known concurrency bugs, we examine the conflicting-pairs reported by TSVD4J for each application to understand whether it detects all the bugs. We find that TSVD4J is able to identify the reads and writes as possible conflicting-pairs for all bugs marked as data race bugs from JaConTeBe. TSVD4J does not detect remaining bugs such as deadlock bugs, which is expected. For the open-source applications, we manually inspected the conflicting-pairs reported by TSVD4J, and we classified six conflicting-pairs to be false positives. Some reasons for false positives involve the actual runtime type of the object being a concurrent data structure (our instrumentation only used the statically-declared type to determine what to track) or the conflicting-pair involves two write operations to a field where both operations write the same constant value. In the future, we plan to improve TSVD4J to extract further runtime information to better filter out such cases, potentially at a higher runtime cost.

For the cases we determined not to be false positives via inspection, we further attempted to manifest a failure by injecting delays around the reported code locations. We could manifest at least one failure for each application. Further, we reported an issue request per application.

V. Related Work

There have been much prior work in detecting concurrency bugs. One class of techniques focus on detecting data races or atomicity violations, either through observing and/or manipulating existing executions [7], [9], [13], [15], [18], [19], generating new test inputs that can expose such bugs [8], [21], [24], model checking [12], [25], or static analysis [11]. TSVD works with existing executions, such as test executions, and injects delays at key points to detect thread-safety violations [15]; we build TSVD4J as an implementation of TSVD for Java applications. We compare against RV-Predict, another tool that can detect data races for Java [13]. Another class of techniques focus specifically on deadlock concurrency bugs [6], [10], [14], [20], [22], [23], [26]. TSVD4J is not designed to specifically detect deadlocks.

VI. Conclusions

We present TSVD4J, a tool for detecting thread-safety violations in Java applications. We implement TSVD4J based on the prior TSVD approach, with adjustments to also detect thread-safety violations on shared object fields. Our implementation is available as a Maven plugin that can be integrated into any Maven-based application, and TSVD4J runs the application's tests to detect thread-safety violations based on their execution. Our evaluation of TSVD4J on 12 applications shows that TSVD4J can detect 55 conflicting-pairs that represent thread-safety violations in the code. Compared against existing tool RV-Predict, TSVD4J detects more thread-safety violations in the similar runtime, largely due to tracking field accesses.

Acknowledgements

This research was partially supported by National Science Foundation grant no. CCF-2145774.

References

[1] ASM. https://asm.ow2.io/.

[2] JaConTeBe. https://sir.csc.ncsu.edu/portal/bios/JaConTeBe.php.

[3] Nashorn sandbox. https://github.com/javadelight/delight-nashorn-sandbox.

[4] TSVD. https://github.com/microsoft/TSVD.

[5] TSVD4J. https://github.com/UT-SE-Research/TSVD4J.

[6] Y. Cai and W. K. Chan. MagicFuzzer: Scalable deadlock detection for large-scale applications. In *International Conference on Software Engineering*, pages 606–616, 2012.

[7] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *International Conference on Programming Language Design and Implementation*, pages 258–269, 2002.

[8] A. Choudhary, S. Lu, and M. Pradel. Efficient detection of thread safety violations via coverage-guided generation of concurrent tests. In *International Conference on Software Engineering*, pages 266–277, 2017.

[9] D. Deng, W. Zhang, and S. Lu. Efficient concurrency-bug detection across inputs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 785–802, 2013.

[10] J. Deshmukh, E. A. Emerson, and S. Sankaranarayanan. Symbolic deadlock analysis in concurrent libraries and their clients. In *International Conference on Automated Software Engineering*, pages 480–491, 2009.

[11] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles*, pages 237–252, 2003.

[12] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Symposium on Principles of Programming Languages*, pages 256–267, 2004.

[13] J. Huang, P. O. Meredith, and G. Rosu. Maximal sound predictive race detection with control flow abstraction. In *International Conference on Programming Language Design and Implementation*, pages 337–348, 2014.

[14] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *International Symposium on Foundations of Software Engineering*, pages 327–336, 2010.

[15] G. Li, S. Lu, M. Musuvathi, S. Nath, and R. Padhye. Efficient scalable thread-safety-violation detection: Finding thousands of concurrency bugs during testing. In *Symposium on Operating Systems Principles*, pages 162–180, 2019.

[16] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao. JaConTeBe: A benchmark suite of real-world Java concurrency bugs. In *International Conference on Automated Software Engineering*, pages 178–189, 2015.

[17] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[18] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 37–48, 2006.

[19] N. Machado, B. Lucia, and L. Rodrigues. Production-guided concurrency debugging. In *Symposium on Principles and Practice of Parallel Programming*, pages 29:1–29:12, 2016.

[20] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering*, pages 386–396, 2009.

[21] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *International Conference on Programming Language Design and Implementation*, pages 521–530, 2012.

[22] M. Samak and M. K. Ramanathan. Multithreaded test synthesis for deadlock detection. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 473–489, 2014.

[23] M. Samak and M. K. Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *Symposium on Principles and Practice of Parallel Programming*, pages 29–42, 2014.

[24] V. Terragni and M. Pezzè. Effectiveness and challenges in generating concurrent tests for thread-safe classes. In *International Conference on Automated Software Engineering*, pages 64–75, 2018.

[25] W. Visser, C. S. Pundefinedsundefinedreanu, and S. Khurshid. Test input generation with Java PathFinder. In *International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[26] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *European Conference on Object-Oriented Programming*, pages 602–629, 2005.