

Project

Language-based Security (TDA602/DIT101)

Group 11

- Himanshu Chuphal (guschuhi@student.gu.se)
- Raya Altarabulsi (rayyatar@gmail.com)

Table of Contents

Background	4
What is a Data Race?	4
What is a Race Condition?	4
Difference	4
Need for Analysis Tools?	4
Common Types of Races	5
Common Data Races in Java	5
Common Data Races in C/C++	6
Use Cases	7
Project Goal	7
Relevance to language-based security	8
Vulnerability Report 2018-19	8
Selected Tools	8
Project Scope	9
raceDetectionToolKit	10
ToolKit Information	10
Tools Comparison	11
Programs and Tools	12
1. Helgrind (from Valgrind ToolKit)	12
2. GCC+ThreadSanitiser	13
3. Rv-Predict	14
4. ThreadSafe	15
5. RoadRunner	16
6. Vmlens:	17
7. Java Pathfinder (JPF)	18
8. Coverity (Static Analysis Paid Tool)	19
Tools Limitations	19
Insight, Recommendations for Developers	19
Appendix	20
Contribution Details, Group 11	20
Code Documentation	20
References	33

List Of Figures

Image 1: Vulnerabilities Report
 Image 2: raceDetectionToolKit
 Image 3: ThreadSafe Results
 Image 4: ThreadSafe Results 2
 Image 5: VMLens Program1
 Image 6: VMLens Program2
 Image 7: VMLens Results1
 Image 8: VMLens Results2

List of Tables

Table 1: Tools Comparison

Deliverables

```

Group11_Final_Project_Submission/
├── Presentation
│   └── Project_LBS_2010_PPT.pdf
├── Project_Group11.pdf
├── Proposal
│   └── Project_Proposal_TDA602_DIT101_Group11.pdf
├── RaceDetectionToolKit
│   ├── C
│   │   ├── dot-prod.c
│   │   ├── filerace.c
│   │   └── simplrace.c
│   ├── C++
│   │   └── example.cc
│   ├── Java
│   │   └── Counter.java
│   ├── __init__.py
│   ├── bin
│   │   └── example
│   └── raceDetectionToolKit.py
├── UseCases_and_Benchmarking
│   ├── C_Benchmarking.xlsx
│   ├── C_Programs.zip
│   ├── Java_Benchmarking.xlsx
│   ├── Java_Programs.zip
│   └── Readme.txt
  
```

8 directories, 16 files

Analysis tools for race detection

Background

Data races and race conditions are easy to cause and hard to debug. We can't detect all data races. Detection of feasible races relies on detection of apparent data races. These issues can be prevented to an extent using static and dynamic race detector tools. Data race detection tools are either static or dynamic.

What is a Data Race?

A data race occurs when two concurrent threads access a shared variable and when:

- at least one access is a write and
- the threads use no explicit mechanism to prevent the accesses from being simultaneous.

What is a Race Condition?

A race condition is a property of an algorithm or a program, system, etc. that is manifested in displaying anomalous outcomes or behavior because of the unfortunate ordering or relative timing of events in the system.

Difference

Race conditions and data races are related but different concepts. Because they are related, they are often confused. A better comparison :

A **race condition** is a situation, in which the result of an operation depends on the interleaving of certain individual operations.

A **data race** is a situation, in which at least two threads access a shared variable at the same time. At least one thread tries to modify the variable.

A race condition can be the reason for a data race. In contrary, a data race is an undefined behaviour. ***We will be focusing on Data Race detection tools in this document.***

Need for Analysis Tools?

1. Data races are very hard to detect with traditional testing techniques. It requires occurrence of simultaneous access from multiple threads to a particular region which results in corrupted data that violates a particular user-provided assertion or test case.

2. Traditional software engineering testing methods are inadequate, because all tests passing most of the time with only rare, mysterious failures might create a false sense of reliability.

3. Concurrent programs are notoriously prone to defects caused by interference between threads. These are difficult errors to detect via traditional testing alone because scheduling nondeterminism leads to exponentially-many possible thread interleavings, each of which may induce an error. The recent advent of multi-core processors only exacerbates this problem by exposing greater concurrency.

4. Data races are notoriously difficult to find and reproduce because they often happen under very specific circumstances. Therefore, you could have a successful pass of the tests most of the time but some test fails once in a while with some mysterious error message far from the root cause of the data race.

Common Types of Races

Despite all the effort on solving data race problem, it remains a challenge in practice to detect data races effectively and efficiently.

The most popular kinds of data races can be summarized based on our experiments with some vulnerable programs in Java and C/C++.

All the use cases for each common data races types(vulnerable programs and with fixes, comparison sheet inside UseCases_and_Benchmarking Folder) are attached separately with the documents.

Common Data Races in Java

[References \[2\] \[3\]](#)

1. Simple race

- Occurs when at least two threads access a shared variable or a shared memory location in general like a file without any synchronization and at least one of the accesses is write.
- *Use cases attached:* CounterRaceCondition, FileRace, SimpleRace.

2. Non thread-safe Java class usage

- Using non-thread safe classes such that java.util.Array or java.util.HashMap without synchronization can lead to race condition.
- *Use cases attached:* RaceOnArrayList, RaceOnSynchronizedMap

3. Broken spinning loop (Data Race on a Condition)

- synchronize multiple threads based on whether some condition has been met

- A data race on that condition will occur here from multiple threads
- *Use case attached:* BrokenSpinningLoop

4. Write under reader lock

- usually happens when you use a `java.util.concurrent.locks.ReadWriteLock` to increase the level of concurrency but mistakenly write to the protected data under read mode
- *Use case attached:* WriteUnderReadLock

5. Double-checked locking

- pattern for implementing lazy initialization in a multithreaded environment.
- *Use case attached:* DoubleCheckedLocking

Common Data Races in C/C++

[References \[2\] \[3\]](#)

1. Concurrent Access to a Shared Variable/File

- problem description is straightforward: multiple threads are accessing a shared variable/file without any synchronization.
- *Use cases attached:* dot-product, fileRace, memcpy, raceCondition, simpleRace, squareSum, stolzQueue

2. Simple State Machine

- Each thread models some state machine transitions
- reasonable locking policy that appears to protect shared resources
- *Use case attached:* simple-state-machine

3. Double-checked Locking

- a shared resource that is expensive to construct, so it is only done when necessary
- the pointer is first read without acquiring the lock, and the lock is acquired only if the pointer is NULL. The pointer is then checked again once the lock has been acquired in case another thread has done the initialization between the first check and this thread acquiring a lock.
- *Use case attached:* double-checked-locking

4. Broken Spinning Loop

- When we want to synchronize multiple threads based on whether some condition has been met. And it is a common pattern to use a while loop that repeatedly checks that condition
- *Use case attached:* spinning-loop

Use Cases

We did experiments with total 50 programs.

-vulnerable programs:

We have at least one vulnerable program for each of the previously mentioned data races. Most of codes used are examples for RV-Predict tool [References\[9\]](#). We did experiment for each program with RV-Predict /Vmlens in Java and RV-Predict/Valgrind/ThreadSanitizer in C,C++.

-with fixes:

We fixed each program with multiple options when applicable to investigate false positive cases. For java programs we avoided data races with locks (synchronization/ semaphore) and with volatile field declaration. [References\[10\]](#). Similarly in C programs we used locks (mutex) and Atomic field declaration.

All the programs and fixes are Included with the Shared deliverables inside **UseCases_and_Benchmarking** folder:

1. C_Benchmarking.xlsx , C_Programs(vulnerable programs & with fixes)
2. Java_Benchmarking.xlsx, Java_Programs(vulnerable programs & with fixes)

Project Goal

Track the program which are vulnerable to data races and races conditions and understand the behaviour and analyse different concurrency bugs by experimenting with various **static and dynamic** race detector tools in multi threaded programs. The use of multithreaded techniques in complex applications can help to both simplify code and improve responsiveness.

As is often the way, good tools can be helpful with detecting such issues, and on Linux we arguably have better tools for such things, and they're free. This document shows a simple example of a data race condition and shows how to use of multiple different tools to find the issues.

The project scope also includes analyzing types of races caught and not caught by these tools and compare the different tools on false positives and negatives using concurrency bugs benchmark for different program scenarios.

Relevance to language-based security

Race detector tools for modern programming languages give an ability to identify application- and language-level security threats and analyze and fix application-level attacks as race conditions, buffer overruns, and code injections.

Vulnerability Report 2018-19

2018 has exceeded the previous year's vulnerability influx, tacking on a 12-percent rise over 2017's total of number of vulnerabilities published. As seen in the chart below, 2018 saw 16,412 new CVEs published vs. 14,595 in 2017. It seems 2017's initial raising of the bar is here to stay, and 2019 expected to boast a similar tally. [References \[7\]](#)

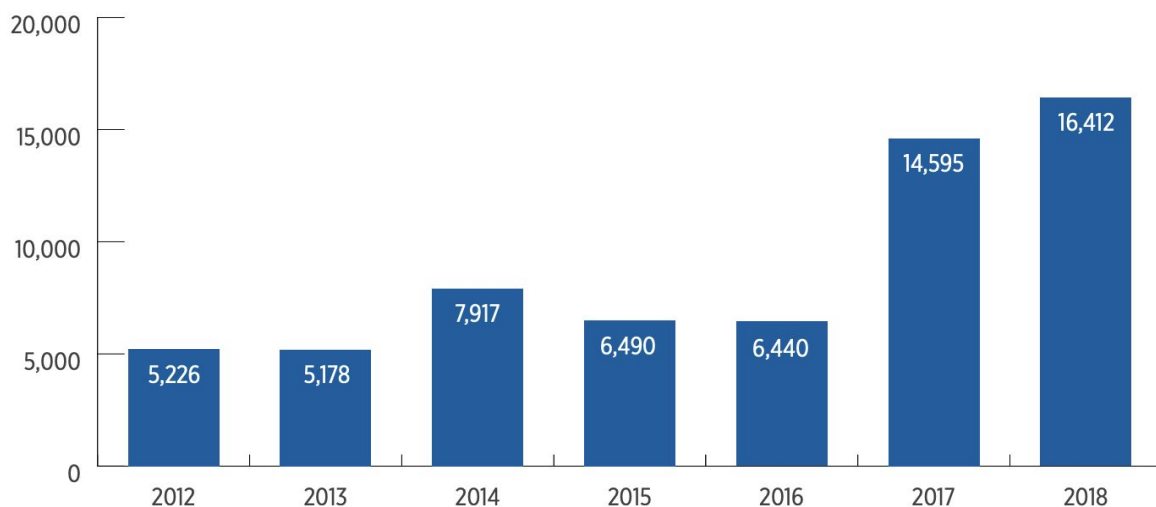


Image 1: Vulnerabilities Report

Selected Tools

As part of our initial research, we selected some popular tools to benchmark namely Race Detector & Healer for Java (using IBM ConTest) , FindBugs, Coverity, SonarQube, Checkmarx, ITS4, RATS, Flawfinder, Splint, Valgrind etc. However, as we started to learn more about the tools and actually started using the tools libraries in our local environments, we found that some of tools have been marked as obsolete and no longer supported with new version of programming languages compilers. At the same time, we encountered different new tools having good support and no false positives.

Based on our experiments, some of the interesting tools we selected are as follows :

1. Valgrind (helgrind)
2. GCC+ThreadSanitiser
3. Rv Predict - (C/Java)
4. Vmlens
5. Java PathFinder
6. ThreadSafe
7. RoadRunner
8. FindBugs/SpotBugs
9. Coverity- Static Tool

Based on the experiments and usage of these popular tools, we summarized the comparison of the tools on different aspects in a table (section Tools Comparison).

Project Scope

Goal	Evaluation
Experiment with tools for race detection in threaded programs	✓
Types of races by the tool	✓
Compare the different tools	✓
Usability and ease of Use	✓
Code Documentation	✓
Command Line Tools Kit	✓

raceDetectionToolKit

We tried to integrate some of the tools (*having support for command line executions*) under a menu based script to be able to compile and run different tools and publish the results.

Run it as :

`python raceDetectionToolKit.py`

```
+-----+
|          |
|  LBS, TDA602_DIT101 - 2019  |
|          |
|  RACE DETECTION ToolKit    |
|          |
+-----+

Welcome : Input options to use this Tool >>

1. Valgrind (Helgrind) >>
2. GCC+ThreadSanitiser
3. RV-Predict/C
4. RV-Predict/Java
5. ThreadSafe
6. Vmlens
7. Java PathFinder (JPF)
8. RoadRunner
9. EXIT

Enter an option to run the tool [ 1-8 ] = 1
```

Image 2: *raceDetectionToolKit*

ToolKit Information

- Programming Language - C/C++ and Java
- Interface - Command line
- Information about :
 - Installation steps
 - Man-Page, URL
 - Programming language support
 - Use Cases etc.

Installation Steps:

```
# refreshing the repositories
sudo apt update

# installing python 2.7 and pip for it
sudo apt install python2.7 python-pip

# installing python-pip
sudo apt install python3-pip

Dependencies :
  Pip install Texttable ; pip install bunch
```

Usage :: `python raceDetectionToolKit.py`

Tools Comparison

Based on our experiments with different selected tools for detecting race conditions in multiple vulnerable programs in C/C++ and Java programming languages, we managed to get a tools comparison table.

Tools/ Aspects	Valgrind	GCC+ Thread Sanitiser	ThreadSafe	Road Runner	Vmlens	JPF	Coverity	Rv Predict
Analyzes Java Code	✗	✗	✓	✓	✓	✓	✓	✓
Analyzes C/C++ Code	✓	✓	✗	✗	✗	✗	✓	✓
Run-Time Analysis	✓	✓	✓	✓	✓	✓	✗	✓
No False Positives	✓	✓	✗	??	✓	✓	??	✓
Robustness	✓	✓	✗	??	✓	✓	✓	✓
Results Consistency	✓	✓	✓	??	✓	✓	✓	✓
Interactive	✗	✗	✗	✗	✗	✗	✗	✗
Low Overhead	✓	✓	✓	✓	✓	✓	✗	✓
Support Availability	✓	✓	✓	✗	✓	✓	✓	✓
Open Source	✓	✓	✗	✓	✗	✓	✗	✗
Ease of Use	✓	✓	✗	✗	✓	✓	✗	✓

Table 1: Tools Comparisons

?? = Many useless Results, Not-Conclusive

Programs and Tools

1. Helgrind (from Valgrind ToolKit)

[Code Documentation : \[1\]](#)

Helgrind is a Valgrind tool for detecting synchronisation errors in C, C++ and Fortran programs that use the POSIX pthreads threading primitives. Helgrind can detect three classes of errors

- Misuses of the POSIX pthreads API
- Potential deadlocks arising from lock ordering problems'
- Data races -- accessing memory without adequate locking or synchronisation

Problems like these often result in unreproducible, timing-dependent crashes, deadlocks and other misbehaviour, and can be difficult to find by other means.

Official Documentation - <http://valgrind.org/docs/manual/hg-manual.html>

For program and tools results, Please see **Appendix Section(Code Documentation, 1)**

A sample program that shows a deliberate race condition. It is written in C++, and uses the [pthreads](#) library for basic threading functionality – to create new threads, and to wait for their completion.

- This program has a global data member named “zob” which is modified by the various threads that are created.
- the different threads can modify this data member at the same time which can cause unexpected behaviour.
- The data is a simple “int” data type, which is usually atomic. However, if the data type was something more complicated, problems will start to get nasty. Especially as they can be erratic and sensitive to timing.

Results

```
==3301== Possible data race during read of size 4 at 0x30C2A8 by thread #3
==3301== ERROR SUMMARY: 4 errors from 2 contexts (suppressed: 2 from 1)
```

For detailed Results in Appendix Section(Code Documentation, 1)

Helgrind works best when your application uses only the POSIX pthreads API. However, if you want to use custom threading primitives, you can describe their behaviour to Helgrind using the ANNOTATE_* macros defined in helgrind.h. This functionality was added in release 3.5.0 of Valgrind.

Usage :

```
g++ -pthread -o example example.cc
valgrind --tool=helgrind ./example
```

Conclusions

1. The above example shows one occurrence of a potential race condition that Valgrind/Helgrind has detected. Likewise, we can get multiple errors detected.
2. if we modify the test program to enable the mutex (by uncommenting the USEMUTEX definition and recompiling), and then re-run the test with Valgrind, you'll see that these race condition warnings disappear
3. With a C++ program one good approach is to have a mutex as being a "mutable" private member of a given class – lock it on entry to a public method and unlock it on exit of such a method. While this can be done using the operating system primitives such as pthreads or Win32 Threads, it is much easier to use a library such as [boost::thread](#) which provides things like scoped locks and read/write mutexes. Also, C++11 now has built-in support for threading which will make everyone's life easier.
4. One downside of these tools is that the program runs slower when the tool is in use, but that's a small price to pay.
5. 'This shows how easy it can be to detect such issues when you have decent tools at hand. But ideally you wouldn't rely on such tools. It's better to code multi-threaded software correctly in the first place. As a simplistic rule-of-thumb, if any resource needs to be used from more than one thread it should be protected by a mutex.

2. GCC+ThreadSanitiser

[Code Documentation : \[2\]](#)

Another option for detecting race conditions is to use GCC. Recently, GCC has added a "thread sanitiser" component which does a good job. It is based on this project contributed by Google: <https://code.google.com/p/thread-sanitizer/>
And GCC : <https://gcc.gnu.org/>

This facility is also available in LLVM, but my examples use GCC.

Again, build the test program with debugging enabled, but this time with a few extra compilation & link switches.

Program

The same C++ code as we used for Valgrind test is used again.

Usage:

For both the compilation switches and the linker switches add:

```
-fsanitize=thread -fPIE -pie
```

```
g++ -pthread -o example example.cc -fsanitize=thread -fPIE -pie
```

Run the test program as normal:

```
./example
```

Results

For detailed Results, Please check Appendix Section(Code Documentation, 2)

WARNING: ThreadSanitizer: data race (pid=3377)

SUMMARY: ThreadSanitizer: data race (/media/sf_project/tools/valgrind/program/example+0x1455) in threadMain(void*)

Conclusions

1. Using GCC with Thread Sanitiser seems to be faster than Valgrind/Helgrind, but I plan to do more comparative testing soon. Note that this tool is currently supported only on 64-bit platforms, not 32-bit.
2. Both Valgrind and GCC+ThreadSanitiser are very helpful tools for detecting threading issues, and either, or both, should be part of your development arsenal.

3. Rv-Predict

[Code Documentation : \[6\]](#)

Rv-Predict is one of Runtime Verification Company products. Rv-Predict can automatically and efficiently detect concurrency issues of data races in Java and C/C++ software with no false positives.

Official Website: <https://runtimeverification.com/predict/>

Program:

- The program contains the simplest form of a data race where a shared variable is accessed by multiple thread without synchronization;
- It is written in C and uses [pthreads](#) library for multi threading.
- DOTDATA is globally accessible structure and the method dotprod obtained all its input and output to a DOTDATA structure.
 - When a thread is created, the method dotprod is activated and all threads update a shared structure without synchronization and the main thread needs to wait for all threads to complete

For Program in Appendix Section(Code Documentation, 6)

Usage:

RV-Predict in C works in two steps:

- Create an instrumented version of a multithreaded C program :
rvpc dot-product.c
- Run the program and perform the data-race analysis:
./a.out

Results:

For detailed Results, Please check Appendix Section(Code Documentation, 6)

- Two data races were correctly predicted by Rv-Predict:
 - Concurrent read and write by 2 threads of the shared variable dotstr.sum in the method dotprod

- Concurrent write of dotstr.sum in the method dotdrop and concurrent read of the same variable inside the printf.

For another example to check false positives, Please check Appendix Section(Code Documentation, 6)

Conclusions:

- Comparing with Helgrind and ThreadSanitizer, Rv-Predict was the only tool that detected the second race (write dotstr.sum and read it inside printf)
- RV-Predict is free to use for academic projects and research and easy to run with no false positives.
- RV-Predict also leverages unique predictive capabilities to detect possible races even if they do not occur in the execution trace recorded by RV-Predict.

4. ThreadSafe

[Code Documentation : \[4\]](#)

The Java™ platform allows developers to build high performance concurrent software that exploits modern hardware. ThreadSafe is a static analysis tool for finding concurrency bugs and potential performance issues in Java™ programs. Concurrency is often the most natural way to design a software system, even when high performance is not critical. But concurrent software is difficult to get right, especially in a team that has many demands on it.

ThreadSafe looks for the most common concurrency bugs... and some rarer bugs too. It can check your application conforms to many of the recommendations for concurrent Java found in the CERT Oracle Secure Coding Standard for Java.

ThreadSafe analyses are available in three different packages to suit your environment:

- Eclipse plug-in: shows problems in the IDE where bugs are easiest to investigate and fix.
- SonarQube plug-in: provides a shared view of ThreadSafe results for your team.
- Command Line: builds HTML reports of problems cross-referenced with source code.

Official Documentation - <http://www.contemplatetd.com/>
Program

We used Eclipse plugin Free Trial to run some buggy test examples.

Results

For detailed Results, Please check Appendix Section(Code Documentation, 3)

More Examples:

1. <http://download.contemplateld.com/projects/examples/index.html#by-type/5/5/17>
2. <http://download.contemplateld.com/projects/jmeter/index.html>

Conclusions

1. Easy installation: Install ThreadSafe from an Update Site
2. No configuration required: uses existing project sources, class files and libraries
3. Prioritize and fix bugs according to their severity
4. Access to all IDE features helps you investigate bugs efficiently
5. Rich information about findings including relevant guards and locks
6. Rule documentation via the Eclipse help system including suggested fixes'
7. One downside is it is not an open source and only gives 14 days trials which we used for the experimentation.

5. RoadRunner

[Code Documentation : \[5\]](#)

ROADRUNNER is a dynamic analysis framework designed to facilitate rapid prototyping and experimentation with dynamic analyses for concurrent Java programs. It provides a clean API for communicating an event stream to back-end analyses, where each event describes some operation of interest performed by the target program, such as accessing memory, synchronizing on a lock, forking a new thread, and so on. This API enables the developer to focus on the essential algorithmic issues of the dynamic analysis, rather than on orthogonal infrastructure complexities.

Each back-end analysis tool is expressed as a filter over the event stream, allowing easy composition of analyses into tool chains. This tool-chain architecture permits complex analyses to be described and implemented as a sequence of more simple, modular steps, and it facilitates experimentation with different tool compositions. Moreover, the ability to insert various monitoring tools into the tool chain facilitates debugging and performance tuning.

Despite ROADRUNNER's flexibility, careful implementation and optimization choices enable ROADRUNNER-based analyses to offer comparable performance to traditional, monolithic analysis prototypes, while being up to an order of magnitude smaller in code size. We have used ROADRUNNER to develop several dozen tools and have successfully applied them.

The ROADRUNNER framework is available for download from <http://www.cs.williams.edu/~freund/rr>.

Usage:

```
> javac test/Test.java
> rrrun test.Test
```



```
> rrrun -tool=TL:RS:LS test.Test
You should get many race errors on field Test.y.
Test with FastTrack:
```

```
> rrrun -tool=FT2 test.Test
```

Program

For detailed Results, Please check Appendix Section(Code Documentation, 4)

rrrun test.Test

rrrun -tool=TL:RS:LS test.Test

Results

```
<threadMaxActive> 3 </threadMaxActive>
<errorTotal> 3 </errorTotal>
  <name> test/Test.y_I </name>
  <error> <name> FastTrack </name> <count> 3 </count> </error>
```

Conclusions

1. ROADRUNNER has proven invaluable for building and evaluating analyses. Implementing a first prototype of a new analysis often takes a few days rather than the weeks or months required to build a system from scratch. Much of this improvement comes from the ROADRUNNER event stream model, which matches the level of abstraction with which we formulate analyses.
2. In addition, initial versions of ROADRUNNER analyses can often be applied to large-scale systems such as Eclipse, because ROADRUNNER deals with and hides many of the complexities involved in scaling to such systems. This scalability and easy experimentation is critical for evaluating and refining new analyses.
3. Tool composition in ROADRUNNER has also played a major role in how we design, implement, test, and validate dynamic analyses.
4. It enables us to express complex algorithms as the composition of simpler, modular ones; to reuse analysis components unchanged; and to insert monitoring and debugging filters into the tool chain without cluttering the analysis code.

6. Vmlens:

[Code Documentation : \[6\]](#)

It is a tool that dynamically traces multithreaded java software to detect race conditions and deadlocks with no false positives. It provides a shortcut or a new way for junit test and it analyzes and fixes all detected bugs inside Eclipse. Junit is a unit testing framework that automatically tests different classes and methods in java.

Official Website: <http://vmlens.com/>

Program:

We used Eclipse plugin Free Trial to run some buggy test examples.

Results

For detailed Results, Please check Appendix Section(Code Documentation, 5)

Usage:

We run the application as JUNIT Test traced with vmlens.

Conclusion

- Vmlens is easy to install and run and it is also easy to follow the fields in which race detection is accessed.
- Vmlens is not a free tool, it only provides a 14 days trial as an Eclipse plugin.
- All thread interleavings are also tested as Vmlens insert wait, notify instructions during the test.

7. Java Pathfinder (JPF)

[Code Documentation : \[7\]](#)

JPF is a Verification framework for Java programs that is developed by NASA and it became an open source since 2005, and it is available on Github since 2018. (

<https://github.com/javapathfinder>)

JPF was primarily used as a model checking software for concurrent programs to detect flaws such that data races and deadlocks. But nowadays it has been used as a runtime system with many different execution modes besides defects detecting like deep runtime monitoring and low level program inspection so you can consider it as a special Java virtual machine that can be used for model checking. However, model checking software has a scalability problem.

Program:

For Program in Appendix Section(Code Documentation, 7)

Usage:

```
java -jar build/RunJPF.jar src/examples/Racer.jpf
```

Results:

```
===== error 1
gov.nasa.jpf.listener.PreciseRaceDetector
race for field Racer@15f.d
```

For detailed Results, Please check Appendix Section(Code Documentation, 7)

The race was successfully detected by JPF and it provides a “trace” that shows the execution history to find this flaw.

Conclusions:

- JPF provides a trace to the data race found.
- It only works for a (.jpf) file i.e: jpf test file to work on, which is not very easy to obtain.
- Due to its scalability problem, checking different programs is not available.[References\[8\]](#)

8. Coverity (Static Analysis Paid Tool)

Coverity Prevent static code analysis tool for C/C++ and Java. This new technology introduces what the company claims is the first static defect detection of race conditions, one of the most difficult to find concurrency errors that occurs in multi-threaded applications. Coverity Prevent helps control the complexity of multi-threaded applications by automatically identifying these hard-to-find, often crash-causing concurrency defects.

Race Condition. Multiple threads access the same shared data without the appropriate locks to protect access points. When this defect occurs, one thread may inadvertently overwrite data used by another thread, leading to both loss of information and data corruption.

Deadlock. Two or more threads wait for a lock in a circular chain such that the locks can never be acquired. When this defect occurs, the entire software system may halt, as none of the threads can either proceed along their current execution paths or exit.

Thread Block. A thread calls a long-running operation while holding a lock thereby preventing the progress of other threads. When this defect occurs, application performance can drop dramatically due to a single bottleneck for all threads.

Tools Limitations

- Predefined known vulnerability database detection
- Predictions, No in depth analysis
- Might generate false positives
- Not an Open source
- No active tools community
- Lack of support
- Dynamic Tools can't detect all data races

Insight, Recommendations for Developers

- Tools are simple to use
- Use Multiple reliable tools (top 3 worked for us) to be sure
- Good starting point to do manual security audits
- Beware of false positives!
- Command Line Interfaces
- Link with Continuous Integration (CI) of the Project
- Publish reports for developers
- Check Compiler warnings
- Contribute to the community
- Share Knowledge!

Appendix

Contribution Details, *Group 11*

Team Members	Contributions
Himanshu Chuphal (<i>guschuhi@student.gu.se</i>)	<ul style="list-style-type: none"> • Project Proposal • Selections of the tools • Background and Project outline • Tools Kit in python for command line tools • Tools comparison table • Presentation • Programs executions and execution • Tools Usage documentation • Code Documentation
Raya Altarabulsi	<ul style="list-style-type: none"> • Project Proposal • Selections of the tools • Types of races • Tools comparison table • Presentation • Programs executions and execution • Tools Usage documentation • Code Documentation • Use Cases • Comparison sheet

Code Documentation

Here we have code documentation explaining programs, executions and usage for all 7 tools (Program and results for each tool including snapshot).

Please note all the use cases with the tools are attached separately with this documentation.

1.Valgrind

```
<code>
#include <cstdlib>
#include <iostream>
#include <vector>
```

```

#include <stdexcept>
#include <unistd.h>
#include <pthread.h>
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static int zob = 0;
void *threadMain(void *)
{
    sleep(1);

    for (int i = 0; i < 100; i++)
    {
#ifdef USEMUTEX
        pthread_mutex_lock(&mutex);
#endif
        zob++;
#ifdef USEMUTEX
        pthread_mutex_unlock(&mutex);
#endif
    }
    return 0;
}

static void doTest()
{
    const size_t NumThreads = 3;
    std::vector<pthread_t> threads;

    // Create threads
    for (size_t i = 0; i < NumThreads; ++i)
    {
        std::cout << "Creating thread #" << i << std::endl;
        pthread_t tid;
        if (pthread_create(&tid, 0, threadMain, 0) != 0)
            throw std::runtime_error("Failed to create thread");
        threads.push_back(tid);
    }

    // Small pause to keep the output easier to follow
    std::cout << "pausing..." << std::endl;
    sleep(3);
    std::cout << "...paused." << std::endl;

    // Wait for threads to stop before returning
    for (size_t i = 0; i < NumThreads; ++i)
    {
        // block until thread i completes
        std::cout << "Stopping thread #" << i << std::endl;
        if (pthread_join(threads[i], NULL) != 0)
            throw std::runtime_error("Failed to join thread");
    }
}

int main(int argc, char *argv[])
{
    try
    {
        doTest();
    }
    catch (std::exception& e)
    {
        std::cerr << "Exception: " << e.what() << std::endl;
        return EXIT_FAILURE;
    }
    pthread_mutex_destroy(&mutex);
    std::cout << "zob=" << zob << std::endl;
}

```

```

    return EXIT_SUCCESS;}
</code>

```

Results:

valgrind --tool=helgrind ./example

```

==3301== Helgrind, a thread error detector
==3301== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==3301== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==3301== Command: ./example
==3301==
Creating thread #0
Creating thread #1
Creating thread #2
pausing...
==3301== Possible data race during read of size 4 at 0x30C2A8 by thread #3
==3301== Locks held: none
==3301==    at 0x10915D: threadMain(void*) (in
/media/sf_project/tools/valgrind/program/example)
==3301==    by 0x4C36C26: ??? (in
/usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==3301==    by 0x53EA6DA: start_thread (pthread_create.c:463)
==3301==    by 0x572388E: clone (clone.S:95)
==3301==
==3301== This conflicts with a previous write of size 4 by thread #2
==3301== Locks held: none
==3301==    at 0x109166: threadMain(void*) (in
/media/sf_project/tools/valgrind/program/example)
==3301==    by 0x4C36C26: ??? (in
/usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==3301==    by 0x53EA6DA: start_thread (pthread_create.c:463)
==3301==    by 0x572388E: clone (clone.S:95)
==3301== Address 0x30c2a8 is 0 bytes inside data symbol "_ZL3zob"
==3301== -----
==3301==
==3301== Possible data race during write of size 4 at 0x30C2A8 by thread #3
==3301== Locks held: none
==3301==    at 0x109166: threadMain(void*) (in
/media/sf_project/tools/valgrind/program/example)
==3301==    by 0x4C36C26: ??? (in
/usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==3301==    by 0x53EA6DA: start_thread (pthread_create.c:463)
==3301==    by 0x572388E: clone (clone.S:95)
==3301==
==3301== This conflicts with a previous write of size 4 by thread #2
==3301== Locks held: none
==3301==    at 0x109166: threadMain(void*) (in
/media/sf_project/tools/valgrind/program/example)
==3301==    by 0x4C36C26: ??? (in
/usr/lib/valgrind/vgpreload_helgrind-amd64-linux.so)
==3301==    by 0x53EA6DA: start_thread (pthread_create.c:463)
==3301==    by 0x572388E: clone (clone.S:95)
==3301== Address 0x30c2a8 is 0 bytes inside data symbol "_ZL3zob"
==3301==
...paused.
Stopping thread #0
Stopping thread #1
Stopping thread #2
zob=300
==3301==
==3301== For counts of detected and suppressed errors, rerun with: -v

```

```

==3301== Use --history-level=approx or =none to gain increased speed, at
==3301== the cost of reduced accuracy of conflicting-access information
==3301== ERROR SUMMARY: 4 errors from 2 contexts (suppressed: 2 from 1)

```

It's best to build the test program with debugging enabled. This will help Valgrind give us more accurate information, like in the following excerpt:

2. GCC++ThreadSanitiser

Results

WARNING: ThreadSanitizer: data race (pid=3377)

Read of size 4 at 0x55d2d4aa82a8 by thread T2:

#0 threadMain(void*) <null> (example+0x1455)

#1 <null> <null> (libtsan.so.0+0x296ad)

Previous write of size 4 at 0x55d2d4aa82a8 by thread T1:

#0 threadMain(void*) <null> (example+0x146a)

#1 <null> <null> (libtsan.so.0+0x296ad)

As if synchronized via sleep:

#0 sleep <null> (libtsan.so.0+0x4d77b)

#1 threadMain(void*) <null> (example+0x143c)

#2 <null> <null> (libtsan.so.0+0x296ad)

Location is global 'zob' of size 4 at 0x55d2d4aa82a8 (example+0x0000002042a8)

Thread T2 (tid=3380, running) created by main thread at:

#0 pthread_create <null> (libtsan.so.0+0x2bcee)

#1 doTest() <null> (example+0x1543)

#2 main <null> (example+0x174e)

Thread T1 (tid=3379, finished) created by main thread at:

#0 pthread_create <null> (libtsan.so.0+0x2bcee)

#1 doTest() <null> (example+0x1543)

#2 main <null> (example+0x174e)

SUMMARY: ThreadSanitizer: data race (/media/sf_project/tools/valgrind/program/example+0x1455) in threadMain(void*)

=====

...paused.

Stopping thread #0

Stopping thread #1

Stopping thread #2

zob=300

ThreadSanitizer: reported 1 warnings

3. ThreadSafe

Program :

```

|— badpractice
|   |— ConcurrentModificationCatch.java
|— collections
|   |— GetCheckPutRatherThanPutIfAbsent.java
|   |— IterationOverViewBothLocks.java
|   |— IterationOverViewNoLock.java
|   |— IterationOverViewWrongLock.java
|   |— NonAtomicCheckPut.java
|   |— NonAtomicGetCheckPut.java
|   |— SharedNonThreadSafeContent.java
|   |— SynchronizationOnView.java
|   |— ThreadSafeCollectionConsistentlyGuarded.java
|   |— UnsafeIteration.java

```

- ├─ UnsafeReplacement.java
- ├─ guardedby
 - ├─ GuardedByIsNotValid.java
 - ├─ GuardedByViolated.java
 - └─ NonFinalGuard.java
- ├─ immutability
 - └─ FinalVolatile.java
- ├─ locking
 - ├─ BlockingMethodCalledWithLockHeld.java
 - ├─ Deadlock.java
 - ├─ InconsistentSyncOnCollection.java
 - ├─ InconsistentSyncOnField.java
 - ├─ IsLockedThenLock.java
 - ├─ LockFieldReassigned.java
 - ├─ MixedSyncOnCollection.java
 - ├─ MixedSyncOnField.java
 - ├─ SyncOnReusableObject.java
 - ├─ UnreleasedLock.java
 - └─ UnsyncAccessFromAsyncMethod.java

Results

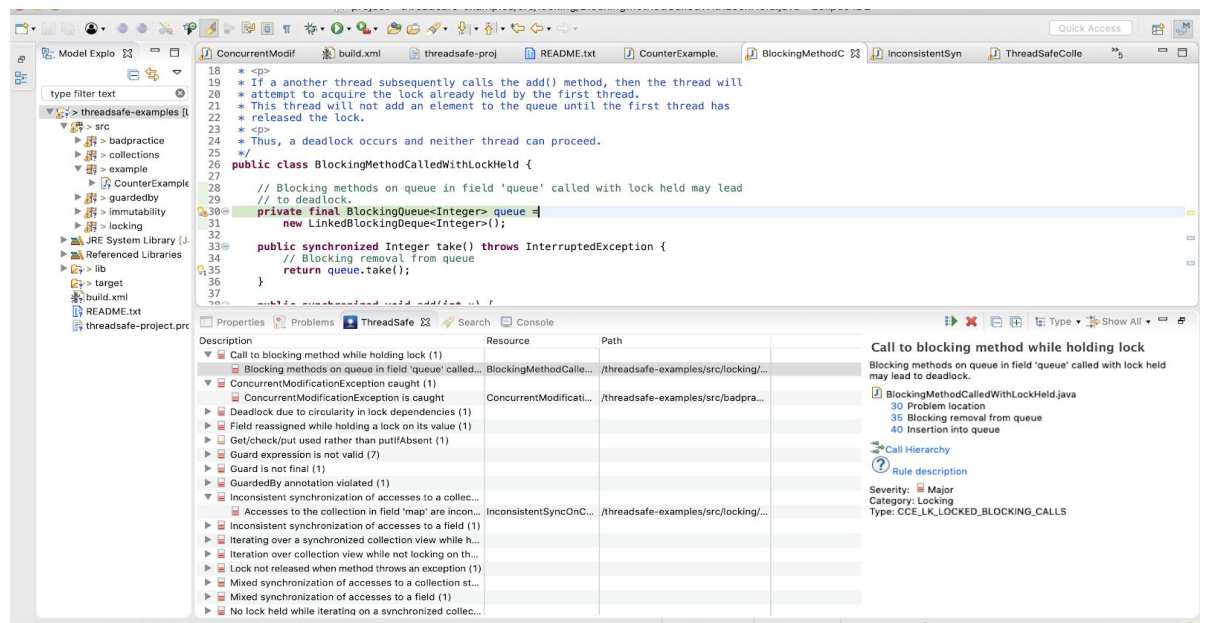


Image 3: ThreadSafe Results

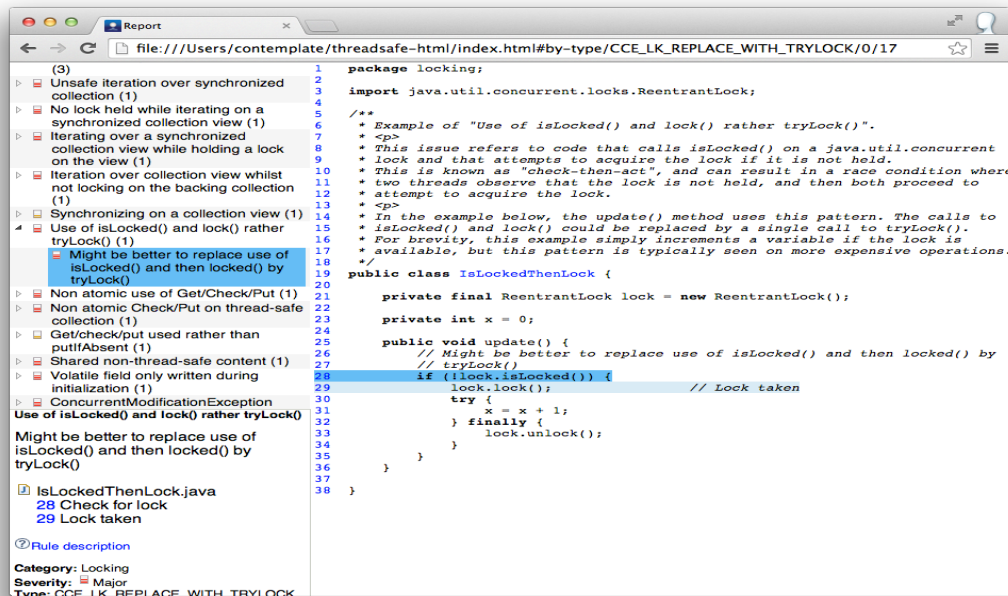


Image 4: ThreadSafe Results 2

4. RoadRunner Program

```
package test;

public class Test extends Thread{
    static final int ITERS = 100;
    static int y;

    public void inc() {
        y++;
    }

    @Override
    public void run() {
        for (int i = 0; i < ITERS; i++) {
            inc();
        }
    }

    public static void main(String args[]) throws Exception {
        final Test t1 = new Test();
        final Test t2 = new Test();
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Is it " + (ITERS * 2) + "? " + y);
    }
}
```

Result:

```
<threadCount> 3 </threadCount>
<threadMaxActive> 3 </threadMaxActive>
<errorTotal> 3 </errorTotal>
<distinctErrorTotal> 1 </distinctErrorTotal>
<methods>
</methods>
<fields>
```

```

    <field>
      <name> test/Test.y_I </name>
      <error> <name> FastTrack </name> <count> 3 </count> </error>
    </field>
  </fields>
</arrays>
</locks>
<fieldAccesses>
</fieldAccesses>
<errorCountPerErrorType>
  <errorType> <name> FastTrack </name> <count> 3 </count> </errorType>
</errorCountPerErrorType>

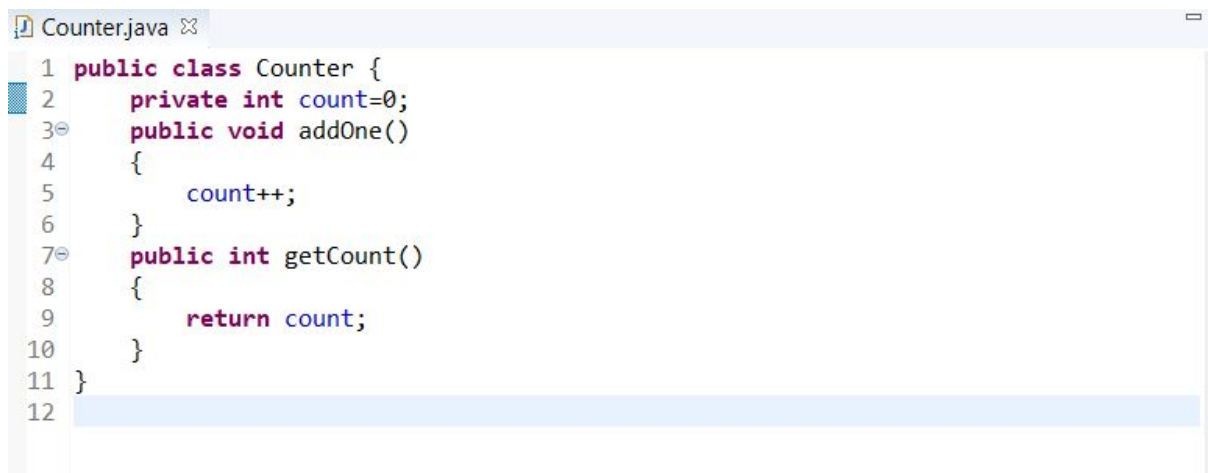
```

5. VMLens:

Program:

We will test a multithreaded java code with JUnit:

Counter class:



```

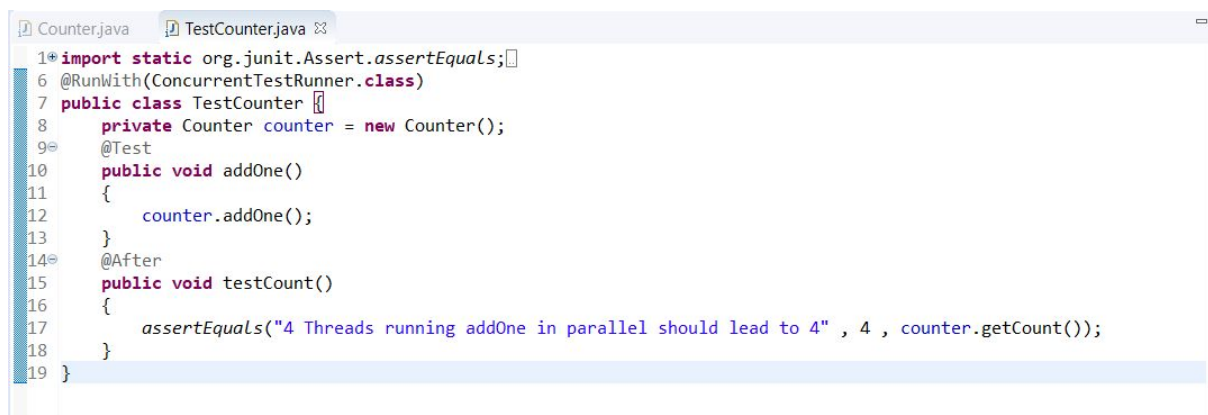
Counter.java
1 public class Counter {
2     private int count=0;
3     public void addOne()
4     {
5         count++;
6     }
7     public int getCount()
8     {
9         return count;
10    }
11 }
12

```

Image 5: VMLens Program1

And the Junit test is:

Test Counter:



```

Counter.java  TestCounter.java
1* import static org.junit.Assert.assertEquals;
2 @RunWith(ConcurrentTestRunner.class)
3 public class TestCounter {
4     private Counter counter = new Counter();
5     @Test
6     public void addOne()
7     {
8         counter.addOne();
9     }
10    @After
11    public void testCount()
12    {
13        assertEquals("4 Threads running addOne in parallel should lead to 4" , 4 , counter.getCount());
14    }
15 }

```

Image 6: VMLens Program2

Results:

ConcurrentTestRunner will run the method addOne() in 4 parallel threads. And then it will run the testCount(). Vmlens could detect the race condition for Counter.count in this case.

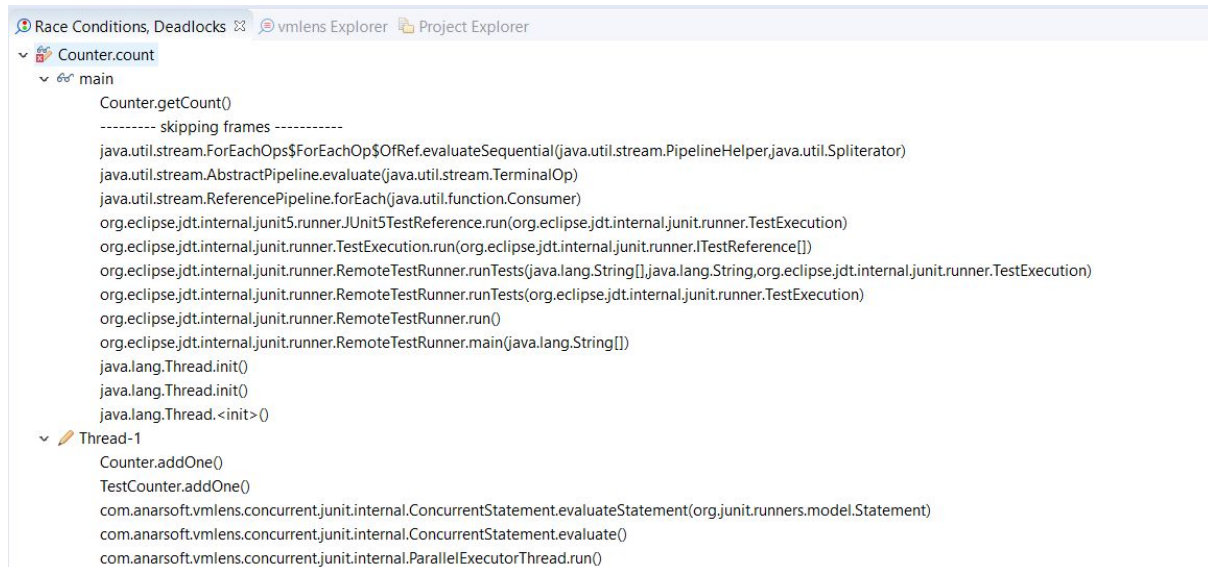



Image 7: Vmlens Results1



refers to the stack trace for reading threads of race condition and  for writing ones.

In the previous simple example we can avoid this race condition case by declaring the counter variable as a volatile so the counter variable will be written back to the main memory immediately.

And Vmlens did not detect any false positives for this case:

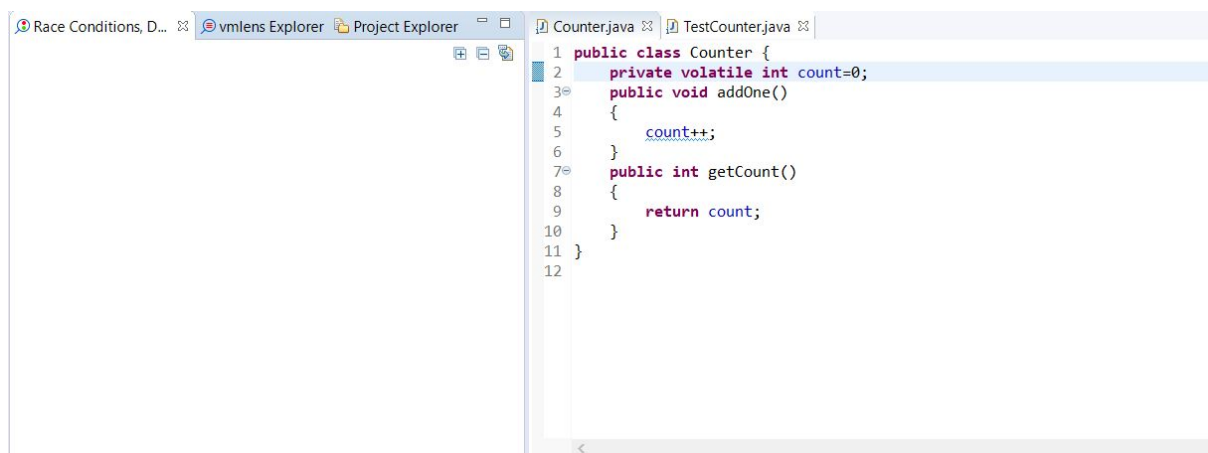


Image 8: VMLens Results2

6. RV-Predict:

Program:

```

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#include "nbcompat.h"

typedef struct {
    float *a, *b;
    float sum;
    int veclen;
} DOTDATA;

/* Define globally accessible variables */

#define NUMTHRDS 3
#define VECLEN 10
DOTDATA dotstr;
pthread_t callThd[NUMTHRDS];

static void *
dotprod(void *arg)
{
    int i, start, end, len;
    long offset;
    float mysum, *x, *y;
    offset = (long)arg;

    len = dotstr.veclen;
    start = offset * len;
    end = start + len;
    x = dotstr.a;
    y = dotstr.b;

    mysum = 0;
    for (i = start; i < end; i++)
        mysum += x[i] * y[i];

    dotstr.sum += mysum;
    printf("Thread %ld did %2d to %2d: mysum=%f global sum=%f\n",
        offset, start, end, mysum, dotstr.sum);

    return NULL;
}

int main(void)
{
    long i;
    float *a, *b;
    void *status;

```

```

/* Assign storage and initialize values */
a = (float *)malloc(NUMTHRDS * VECLen * sizeof(float));
b = (float *)malloc(NUMTHRDS * VECLen * sizeof(float));

for (i = 0; i < VECLen * NUMTHRDS; i++) {
    a[i] = 1.0;
    b[i] = a[i];
}

dotstr.veclen = VECLen;
dotstr.a = a;
dotstr.b = b;
dotstr.sum = 0;

/* Create threads to perform the dotproduct */
for(i = 0; i < NUMTHRDS; i++) {
    /*
     * Each thread works on a different set of data.
     * The offset is specified by 'i'. The size of
     * the data for each thread is indicated by VECLen.
     */
    pthread_create(&callThd[i], NULL, dotprod, (void *)i);
}

/* Wait on the other threads */
for(i = 0; i < NUMTHRDS; i++)
    pthread_join(callThd[i], &status);

/* After joining, print out the results and cleanup */
printf("Sum = %f \n", dotstr.sum);
free(a);
free(b);
return 0;
}

```

Results:

Thread 2 did 20 to 30: mysum=10.000000 global sum=10.000000

Thread 1 did 10 to 20: mysum=10.000000 global sum=20.000000

Thread 0 did 0 to 10: mysum=10.000000 global sum=30.000000

Sum = 30.000000

Data race on `dotstr.sum` at dot-product.c:

Read in thread 2

> in dotprod at dot-product.c:65

Thread 2 created by thread 1

> in main at dot-product.c:112

Write in thread 3

> in dotprod at dot-product.c:65

Thread 3 created by thread 1

> in main at dot-product.c:112

Undefined behavior (UB-CEER4):

see C11 section 5.1.2.4:25 <http://rvidoc.org/C11/5.1.2.4>
 see C11 section J.2:1 item 5 <http://rvidoc.org/C11/J.2>
 see CERT-C section MSC15-C <http://rvidoc.org/CERT-C/MS15-C>
 see MISRA-C section 8.1:3 <http://rvidoc.org/MISRA-C/8.1>

Data race on `dotstr.sum` at dot-product.c:

Write in thread 2

> in dotprod at dot-product.c:65

Thread 2 created by thread 1

> in main at dot-product.c:112

Read in thread 3

> in dotprod at dot-product.c:67

Thread 3 created by thread 1

> in main at dot-product.c:112

Undefined behavior (UB-CEER4):

see C11 section 5.1.2.4:25 <http://rvidoc.org/C11/5.1.2.4>

see C11 section J.2:1 item 5 <http://rvidoc.org/C11/J.2>

see CERT-C section MSC15-C <http://rvidoc.org/CERT-C/MS15-C>

see MISRA-C section 8.1:3 <http://rvidoc.org/MISRA-C/8.1>

Program2:

Another Simple example to check Rv-Predict for false positives is:

```
#include<stdio.h>
#include<pthread.h>

int shared_var=1;

static void * thread1(void * arg){
    shared_var++;
    return NULL;
}

static void * thread2(void * arg){
    shared_var++;
    return NULL;
}

int main(int argc,char ** argv) {

    pthread_t t1,t2;

    pthread_create(&t1,NULL,thread1,NULL);
    pthread_create(&t2,NULL,thread2,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    printf("shared_var= %d\n",shared_var);
}
```

We will get the following data race:

Data race on `shared_var` at simplerace.c:

Read in thread 2
 > in thread1 at simplerace.c:7
 Thread 2 created by thread 1
 > in main at simplerace.c

Write in thread 3
 > in thread2 at simplerace.c:12
 Thread 3 created by thread 1
 > in main at simplerace.c:21

And to fix this very simple race condition we just define the shared variable as Atomic
`_Atomic int shared_var=1;`

And No races was predicted by PV-Predict in this case.

7. JPF

Program:

The program is Racer.jpf that we got from Racer.java:

```
public class Racer implements Runnable {

    int d = 42;

    public void run () {
        doSomething(1000);           // (1)
        d = 0;                       // (2)
    }

    public static void main (String[] args){
        Racer racer = new Racer();
        Thread t = new Thread(racer);
        t.start();

        doSomething(1000);           // (3)
        int c = 420 / racer.d;       // (4)
        System.out.println(c);
    }

    static void doSomething (int n) {
        // not very interesting..
        try { Thread.sleep(n); } catch (InterruptedException ix) {}
    }
}
```

The object (racer) is shared between two threads;

```
d = 0; in the run() method
Int c= 420 / racer.d; in the main method
```

And since we could have different output depending on thread scheduling we have a data race.

Results

JavaPathfinder core system v8.0 (rev 1c956392f5f716dc916efb28a2b6b979e1bbd01c) - (C) 2005-2014 United States Government. All rights reserved.

```
===== system under test
Racer.main()
```

```
===== search started: 5/9/19 2:37 PM
10
10
```

```
===== error 1
gov.nasa.jpf.listener.PreciseRaceDetector
race for field Racer@d15f.d
  main at Racer.main(Racer.java:35)
        "int c = 420 / racer.d;          // (4)" READ: getfield Racer.d
Thread-1 at Racer.run(Racer.java:26)
        "d = 0;                          // (2)" WRITE: putfield Racer.d
```

```
===== trace #1
----- transition #0 thread: 0
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"ROOT",1/1,isCascaded:false}
  [3168 insns w/o sources]
Racer.java:30      : Racer racer = new Racer();
Racer.java:19      : public class Racer implements Runnable {
  [1 insns w/o sources]
Racer.java:21      : int d = 42;
Racer.java:30      : Racer racer = new Racer();
Racer.java:31      : Thread t = new Thread(racer);
  [145 insns w/o sources]
Racer.java:31      : Thread t = new Thread(racer);
Racer.java:32      : t.start();
  [1 insns w/o sources]
----- transition #1 thread: 0
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"START",1/2,isCascaded:false}
  [2 insns w/o sources]
Racer.java:34      : doSomething(1000);          // (3)
Racer.java:41      : try { Thread.sleep(n); } catch (InterruptedException ix) {}
  [4 insns w/o sources]
----- transition #2 thread: 1
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"SLEEP",2/2,isCascaded:false}
  [1 insns w/o sources]
Racer.java:1       : /*
Racer.java:25      : doSomething(1001);          // (1)
Racer.java:41      : try { Thread.sleep(n); } catch (InterruptedException ix) {}
  [4 insns w/o sources]
----- transition #3 thread: 1
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"SLEEP",2/2,isCascaded:false}
```



```

[3 insns w/o sources]
Racer.java:41      : try { Thread.sleep(n); } catch (InterruptedException ix) {}
Racer.java:42      : }
Racer.java:26      : d = 0;                                // (2)
----- transition #4 thread: 0
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"SHARED_OBJECT", 1/2,isCascaded:false}
[3 insns w/o sources]
Racer.java:41      : try { Thread.sleep(n); } catch (InterruptedException ix) {}
Racer.java:42      : }
Racer.java:35      : int c = 420 / racer.d;                // (4)
----- transition #5 thread: 0
gov.nasa.jpf.vm.choice.ThreadChoiceFromSet {id:"SHARED_OBJECT", 1/2,isCascaded:false}
Racer.java:35      : int c = 420 / racer.d;                // (4)

===== results
error #1: gov.nasa.jpf.listener.PreciseRaceDetector "race for field Racer@15f.d  main at Racer.main(Ra..."

===== statistics
elapsed time:      00:00:01
states:            new=9,visited=1,backtracked=4,end=2
search:            maxDepth=6,constraints=0
choice generators: thread=8 (signal=0,lock=1,sharedRef=2,threadApi=3,reschedule=2), data=0
heap:              new=366,released=33,maxLive=361,gcCycles=7
instructions:      3435
max memory:        30MB
loaded code:       classes=63,methods=1482

===== search finished: 5/9/19 2:37 PM

```

References

1. https://en.wikipedia.org/wiki/Race_condition#Tools
2. <https://runtimeverification.com/predict/2.1.2/docs/runningexamples/>
3. <https://runtimeverification.com/blog/category/predict/>
4. <https://web.stanford.edu/class/archive/cs/cs107/cs107.1194/resources/valgrind.html>
5. <https://code.google.com/archive/p/data-race-test/wikis/ThreadSanitizer.wiki>
6. <https://github.com/stephenfreund/RoadRunner/blob/master/INSTALL.txt>
7. https://lp.skyboxsecurity.com/rs/440-MPQ-510/images/Skybox_Report_Vulnerability_and_Threat_Trends_2019.pdf
8. <https://github.com/javapathfinder/jpf-core/wiki/Writing-JPF-tests>
9. <https://runtimeverification.com/predict/>
10. <https://www.sciencedirect.com/topics/computer-science/avoid-data-race>
11. <https://github.com/stephenfreund/RoadRunner>