JBench: A Dataset of Data Races for Concurrency Testing

Jian Gao[1], Xin Yang[1], Yu Jiang[1]✉, Han Liu[1], Weiliang Ying[2], Xian Zhang[2]

School of Software, Tsinghua University, Key Laboratory of Information System Security, Ministry of Education, China[1]

Huawei Technologies Co., Ltd., China[2]

ABSTRACT

Race detection is increasingly popular, both in the academic research and in industrial practice. However, there is no specialized and comprehensive dataset of the data race, making it difficult to achieve the purpose of effectively evaluating race detectors or developing efficient race detection algorithms.

In this paper, we presented JBench, a dataset with a total number of 985 data races from real-world applications and academic artifacts. We pointed out the locations of data races, provided source code, provided running commands and standardized storage structure. We also analyzed all the data races and classified them from four aspects: variable type, code structure, method span and cause. Furthermore, we discussed usages of the dataset in two scenarios: optimize race detection techniques and extract concurrency patterns.

KEYWORDS

data race, classification, dataset, concurrency testing

ACM Reference Format:
Jian Gao[1], Xin Yang[1], Yu Jiang[1]✉, Han Liu[1], Weiliang Ying[2], Xian Zhang[2] School of Software, Tsinghua University, Key Laboratory of Information System Security, Ministry of Education, China[1] Huawei Technologies Co., Ltd., China[2] . 2018. JBench: A Dataset of Data Races for Concurrency Testing. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196398.3196451

1 INTRODUCTION

With the prevalence of concurrent programs, data races are also introduced due to their non-deterministic thread scheduling, which often causes inconsistent data access resulting in enormous losses [3]. As a result, researchers pour a lot of energy into proposing effective techniques to detect data race. For example, happen-before based race detectors [6] and lockset based race detectors [15] are popular dynamic race detection techniques. Predictive trace analysis techniques [8, 14] exploit a SMT solver to predict data race.

However, there is no typically accepted benchmark to conduct the evaluation task among these detectors. For example, Lin *et al.* proposed JaConTeBe [13] containing 47 Java concurrency bugs taken from real-world code, but only 19 of them are related to race. Furthermore, each race bug corresponds to a test case written by themselves, and built-in test cases in open-source programs are not used. Java Parallel Grande benchmark [16] is originally designed for high performance computing, several programs of which are used to detect data race, such as *SOR*, *MolDyn*, *RayTracer*, etc. DataRaceBench [12] presents a benchmark suite for systematically evaluating data race detection tools on OpenMP programs, containing some microbenchmarks with and without data races. [2] conducts an empirical study on five Java race detectors with each evaluation program containing around 20-700 lines of code. But none of the benchmarks mentioned above are tailor-made for Java data race.

Containing only several race programs are insufficient to evaluate race detectors. To make matters worse, no one explains how many data races are present in each evaluation program. In addition, the choice of test cases is biased during the evaluation process, which makes evaluation results are not persuasive. As a result, maintaining a dataset for data races is meaningful for future research and evaluation of race detectors.

Towards tackling these problems, we present JBench, a dataset of data race that can be used to compare different race detection tools. We explore 179 evaluation programs from academic artifacts and real-world applications, filter out 48 programs with the code size ranging from 100 LOC-300,000 LOC, and record a total of 985 data races. We point out the locations of data races, provide source code and provide running commands of evaluation programs. Meanwhile, we also analyze and categorize each data race from four aspects: variable type, code structure, method span and cause.

JBench is publicly available on GitHub[1]. We encourage researchers to use and extend JBench to evaluate different data race detectors for its convenience and comprehensiveness.

2 DATASET PREPARATION

We first investigated the existing academic Java concurrency benchmark suites in the manner that was used in [13]. We collected a total of 168 academic concurrent programs. Furthermore, we also selected popular and representative programs in each domain. For example, *Tomcat* is the first choice for the web application server domain. We collected 11 such real-world applications. Finally, we exploited three steps to screen out suitable evaluation programs and used three race detectors to detect data races. As a result, 42 academic programs and 6 real-world applications were left. And there were 204 and 781 data races in them respectively.

2.1 Source of Data

Evaluation programs for race detectors are usually from the following two categories: academic artifacts and real-world applications. At this stage, we collected 179 evaluation programs.

Academic artifacts CalFuzzer [10] is an active testing framework for concurrent programs. Partial programs of the IBM benchmark suite [4] and Java Parallel Grande benchmark suite [16] are in

MSR '18, May 28–29, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196451

[1]https://github.com/buptsseGJ/ComRaDe/tree/master/benchmark-suite

it. RV-Predict [8] is a specialized detection tool for data race. There are 12 programs in its research tool. VeriFIT [1] contains 16 test cases for concurrency testing. But only 4 programs are explicitly marked as data race according to its repository. JaConTeBe [13] contains 47 real-world Java concurrency bugs from 8 open source projects. There are 19 race bugs in it.

Real-world applications They contain all of the built-in test cases, unlike academic artifacts that contain only one test case each program. We have chosen open-source, widely used programs that are popular in the paper. Each of them are well-known in the corresponding domain. *Tomcat* is popular for the web application server. *Log4j* is a reliable, fast and flexible logging framework written in Java. We also consider well-known applications such as *ZooKeeper*, *FtpServer*, etc. as candidates.

2.2 Processing of Data

Not all 179 collected programs can be included in the dataset due to the presence of repetitive or race-free programs. We used three steps to collect data races. First, we ran collected programs to filter out those which cannot work correctly (for example, throwing an exception terminates execution). Second, we removed repetitive programs and kept only one left. Finally, we used three state-of-the-art race detectors (CalFuzzer [10], RV-Predict [8], DATE) to detect data races and deleted race-free programs. The number of final evaluation programs decreases from 179 to 48 through the above three steps.

Our goal is not only to collect evaluation programs, but also to arrange and classify data races. It will help to conveniently compare the effectiveness of different detection tools and analyze in depth the characteristics of data races.

At this stage, we solved three main challenges: (1) **Completeness of Data Races.** In order to contain as many data races as possible, we used the three race detectors mentioned above to perform five runs of race detection for each program. Then we checked and identified them manually. (2) **Complexity of Compiling Process.** *Tomcat* used *ant* script to finish compiling task. We reorganized the dependency files for the JaConTeBe programs so that our unified file organization structure can be used. (3) **Diversity of Detection Process.** Different race detectors have different running modes on different test cases. For CalFuzzer, we manually wrote the test script for each program to be detected. For RV-Predict, we used Java agent mode to finish detection process on real-world applications. Although there are diverse running modes, it does not affect our dataset, because we simply gather all the real data races.

Table 1 shows detailed statistics and distributions of programs and data races. Since real-world programs such as *Tomcat*, *DBCP*, *FtpServer*, *Log4j*, *Guava* and *ZooKeeper* contain more data races, we split them for demonstration. The total amount of code for evaluation programs exceeds 1,800,000LOC.

Table 1: The number of programs and races for each source.

Category	Source	#Size(LOC)	#Programs	#Data race
Academic artifacts	CalFuzzer	11,433	25	62
	RV-Predict	1,522	8	51
	VeriFIT	2,552	2	66
	JaConTeBe	893,756	7	25
Real-world applications	tomcat-8.0.26	289,169	1	286
	zookeeper-3.5.2	80,455	1	192
	log4j-2.8.1	233,902	1	155
	guava-21.0	251,051	1	32
	ftpserver-1.1.0	20,760	1	70
	dbcp2-2.1.1	19,880	1	46
#Total		1,804,480	48	985

3 DATASET DESCRIPTION

3.1 Data Race Storage

We follow the file structure shown in Figure 1 to construct JBench. It contains forty-eight independent programs and one XML file. Each program is standardized to contain 6 files or directories: *Bin, src, jar, lib, bug* and *text specification*. The first three directories store bytecode, source code or jar file of the evaluation program. The *lib* directory stores dependency files that need be provided during the compiling or running process. The *test specification* file describes how the program is executed, which contains input parameters or compiling and running commands. The *bug* directory has a *race report* file, which contains information about the number of races and race locations. We adopt the format of RV-Predict to record race information because of its intuitiveness.

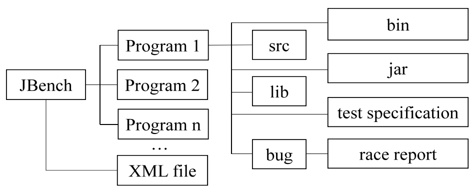

Figure 1: Organization structure of JBench.

Meanwhile, a manually constructed XML file is used to record race locations, race features, and running time without detection process, as shown in Figure 2. The XML file, as a comparison baseline, can be input into an evaluation platform to automatically generate result report for race detection tools. The outermost layer is the *reports* element representing the whole dataset. It contains 48 *report* elements meaning a single evaluation program with *name* and *totalTime* arributes indicating its name identifier and pure running time (under a certain machine configuration). One *report* consists of data races marked as *race* element. The meanings of elements in *race* is literal. Lines 8-12 describe the race features introduced in Section 3.2.

```
1   <reports>
2     <report name="testRace1" totalTime="0.074">
3       <race>
4         <line1>41</line1> <line2>47</line2>
5         <variable>x</variable>
6         <packageClass>TestRace1</packageClass>
7         <detail><![CDATA[Call stack causing data race.]]></detail>
8         <variableType>PT</variableType>
9         <codeStructure>bs-bs</codeStructure>
10        <methodSpan>SS</methodSpan>
11        <sensitiveBranch>no</sensitiveBranch>
12        <cause>NS</cause>
13        <race>
14    <report>
15  <reports>
```

Figure 2: The XML structure used by dataset.

3.2 Data Race Classification

We not only collected 985 data races, but also analyzed and classified them from four aspects: *variable type*, *code structure*, *method span* and *cause*. These classification results were also recorded in the XML file mentioned above. The analysis results allow researchers to refine their detection techniques with different patterns and perform empirical studies.

3.2.1 Variable Type. The performance of race detectors on variable types directly affects the likelihood of their industrial practice. In terms of variable type, we referred to the Java data type, combined with data race, then divided the variable type into four categories: primitive types, reference types, collection types and mixed usage. We checked all 985 data races and categorized them manually. Table 2 shows in detail the number of data races for each category. Column 1 refers to the abbreviation of a category. Columns 2-3 list our categories and their basic compositions. Column 4 shows the number of static declarations of variables in each category. Column 5 records the total number of data races in each category.

Table 2: The number of data races for each type.

Category	Type	Components	#Number	
			Static	Total
PT	primitive types	byte, short, int, long, float, double, boolean, char	78	375
RT	reference types	class/interface object	23	374
		array	3	132
CT	collection types	primitive collection	0	9
		object collection	2	63
MU	mixed fix	field of object	10	31
		field of array object	1	1
		#Total	117	985

3.2.2 Code Structure. The detection capabilities of these data race detection tools may be affected by the program structure because the specific statements that cause data race may be not actually executed. It is reported that 50% (17 out of 34) of the schedule-sensitive branches found in the experiments resulted from concurrency bugs [9]. For these purposes, it is important to identify the code structures in which the data race occurs.

Taking into account the control flow structure of Java program language, the following code structures are related to data race: *basic statement*, *branch condition*, *statement in branch*, *loop condition* and *statement in loop*. *Branch condition* and *loop condition* are statements that control scheduling. Typically, the conditions of the if statement and switch statement belong to *branch condition*. *Statement in branch* refer to the statements that will be executed in the branch after the branch decision. *Statement in loop* are similar to *statement in branch*. We call the statement that does not satisfy the above situation as the *basic statement*. The data race involves a pair of access to the shared variable. So, any pair of data race is one of the fifteen combinations of any two kinds among the five code structures described above. For nested structures, we record the nearest code structure that contains the data race, because the code structure closer to the data race is more likely to determine whether it can be detected by the race detection tool.

The results of code structures of all 985 data races are shown in Table 3. Column 1 refers to the abbreviations of categories. Column 2 lists code structures of data races. Column 3 shows the number of data races. Column 4 shows the number of potential sensitive branch in each category.

3.2.3 Method Span. Another concern about the data race is that the two operations are intra-method or inter-method. This result may provide guidance for testing or detecting concurrent programs. We divide the race locations into three categories: the same method of the same class, different methods of the same class, different methods of different classes. From the perspective of source code, each class file may include anonymous classes, besides the class which has the same name as the file name. But we only consider whether the data races are in the same class file. Statistical results are shown in Table 4. Column 1 refers to the abbreviations of categories. Column 2 lists possible locations of data races. Column 3 shows the number of data races in each category.

Table 3: The number of code structures of data races. (The item marked with '–' means code structure may not have schedule-sensitive branch.)

Category	Code structure pair	#Number	#SSB
BS-BS	basic statement - basic statement	488	–
BC-BC	branch condition - branch condition	1	–
SB-SB	statement in branch - statement in branch	52	15
SL-SL	statement in loop - statement in loop	12	–
BS-BC	basic statement - branch condition	153	68
BS-LC	basic statement - loop condition	27	–
BS-SB	basic statement - statement in branch	125	23
BS-SL	basic statement - statement in loop	31	–
BC-SB	branch condition - satement in branch	58	39
BC-SL	branch condition - statement in loop	4	3
LC-SB	loop condition - statement in branch	14	–
LC-SL	loop condition - statement in loop	2	–
SB-SL	statement in branch - statement in loop	18	–
	#Total	985	148

Table 4: Statistical results of locations of data races.

Category	Race position	#Number
SS	the same method of the same class	203
DS	different methods of the same class	694
DD	different methods of different classes	88
	#Total	985

3.2.4 Cause of Data Race. The universally recognized definition of data race is that two operations of different threads that concurrently read and write or both write variable on a shared location without proper synchronization. In term of synchronization operations, we can divide them into three categories: no synchronization, partial synchronization and incorrect synchronization. The three categories are orthogonal. We classify them manually. In addition to these three categories, we also find out some common and typical usage habits causing data race.

Table 5 details the classification of all 985 data races. Column 1 lists the three categories. Column 2 counts the number of each cause. Column 3 refers to the abbreviations of categories. Columns 4-5 list the common usages which cause data race.

Table 5: Classification of all 985 data races in JBench.

Cause	Count	Category	Pattern	Count
No synchronization	820	N-Volatile	`volatile` declaration	10
		N-If	Race on branch condition	150
		N-Loop	Race on loop condition	28
		N-Accessor	Race on `getter`/`setter`	25
		N-Others	Other situations	607
Partial synchronization	142	P-If	Race on branch condition	46
		P-Loop	Race on loop condition	3
		P-Others	Other situations	93
Incorrect synchronization	23	I-SCB	Inconsistent `synchronized`	10
		I-MSS	`synchronized` method and block	5
		I-MSN	`static` and non-`static` `synchronized`	2
		I-Others	Other situations	6
#Total	985	–	–	985

4 DATASET USAGE

In this section, we discuss the usage scenarios of our dataset.

4.1 Optimize Detection Techniques

The dataset can be the basis for empirical studies on a variety of race detection techniques. Happen-before based [6], lockset based [15] and predictive trace analysis [8, 14] are typical techniques. But their effectiveness and efficiency are significantly different. Meanwhile, we can also observe the performance of race detectors on certain features since the XML file mentioned above records these features. Based on the evaluation results, researchers can combine these techniques to make up for certain shortcomings.

We have recently used the dataset to evaluate three race detectors. They are CalFuzzer [10], the commercial version of RV-Predict [8] and DATE. In one evaluation, the average false negative rate for these three tools was 58.62%, 43.19% and 55.40% on academic artifacts, respectively. Their average time overhead is 232.53 times, 248.31 times and 286.69 times, respectively. On the *sor* program, none of the three tools found any race. For RV-Predict, 10 of the 73 undetected data races are due to *incorrect synchronization*, which accounts for 18.52%, twice as much as other race detectors. We detected data races with RV-Predict and DATE on six open-source programs. There were 781 data races in these six programs, RV-Predict detected 220 of them, DATE detected 487 of them.

By summarizing race locations and observing race contexts, we may identify which class or method is more likely to have data race, which will reduce the number of instrumentation and speed up the detection process. Also, these information can be used to optimize and direct fuzzing techniques[11, 17].

4.2 Extract Concurrency Patterns

Some well-known patterns which have been proposed for concurrency bugs [5] are in our dataset, such as *double-checked locking*, *two-stage access bug pattern*, *wrong lock or no lock*, etc. By carefully analyzing the race characteristics (e.g. code structures, causes of data races) of the dataset, some meaningful patterns will be summarized sooner or later.

These patterns have two potential usage scenarios. One is that they can be used as anti-patterns to prevent developers from producing such race bugs. The other is that they can statically identify the possible race conditions in programs, similar to FindBugs [7]. The actual usage of bug patterns will ensure that the program contains fewer errors.

5 LIMITATIONS AND FUTURE WORK

The completeness of data races in a single program is a threat to our dataset. We ensure that academic artifacts include as many true data races as possible through manual inspection and tool detecting. But for real-world applications, we can only use different tools with multiple runs to detect data races inside due to large code size (for example, lines of code of *tomcat-8.0.26* is 289KLOC).

Another threat is the correctness of data races. Concurrent programs do not have definite statement scheduling sequences as single-threaded programs do, so it is difficult to manually confirm real data races due to various factors. We need to be familiar with threading knowledge, understand the architecture of detected programs, analyze function call sequences, etc. As a result, complex concurrent accesses to variables may be incorrectly classified as data races.

Although we have considered well-known benchmark suites related to concurrency bugs, a large number of programs from repositories such as GitHub and SourceForge have not yet been contained. In the future, we will enrich the JBench and evaluate its representativeness in depth. Meanwhile, we will analyze the 985 data races in depth and select some representative ones (for example, the ones triggered at a low probability). Concentrating on data races that are difficult to detect will help to present more effective detection techniques and apply in industry.

6 CONCLUSION

In this paper, we present JBench, a dataset which contains 48 programs and 985 data races, which can be used to evaluate and optimize race detection techniques. We point out the locations of data races, provide source code, provide running commands and standardize storage structure. Meanwhile, we draft a taxonomy of data races from four aspects: variable type, code structure, method span and cause. We organize all information into an XML file. We have recently used this dataset to evaluate three race detectors. The average false negative rate was over 40%, and the average time overhead could reach about 200 times. We hope our dataset can help researchers to improve the capability of race detection tools due to its convenience and comprehensiveness.

ACKNOWLEDGMENTS

Thank Yilong Li for providing supports on the use of RV-Predict. This work is sponsored in part by NSFC (No. 61527812), NSTMP (No. 2016ZX01038101) and CPSF (No. 2017M620785).

REFERENCES

[1] [n. d.]. VeriFIT Repository of Test Cases for Concurrency Testing. http://www.fit.vutbr.cz/research/groups/verifit/benchmarks/. ([n. d.]). Accessed March, 2018.
[2] Jalal S Alowibdi and Leon Stenneth. 2013. An Empirical Study of Data Race Detector Tools. In *Chinese Control and Decision Conference*. 3951–3955.
[3] Jesdanun Anick. 2004. GE Energy acknowledges blackout bug. http://www.securityfocus.com/news/8032. (2004). Accessed June 4, 2017.
[4] Yaniv Eytani, Klaus Havelund, Scott D Stoller, and Shmuel Ur. 2007. Towards a framework and a benchmark for testing tools for multi-threaded programs. *Concurrency and Computation: Practice and Experience* 19, 3 (2007), 267–279.
[5] Eitan Farchi, Yarden Nir, and Shmuel Ur. 2003. Concurrent bug patterns and how to test them. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International.* IEEE, 7–pp.
[6] Cormac Flanagan and Stephen N Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *ACM Sigplan Notices*, Vol. 44. ACM, 121–133.
[7] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. *ACM Sigplan Notices* 39, 12 (2004), 92–106.
[8] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. *ACM SIGPLAN Notices* 49, 6 (2014), 337–348.
[9] Jeff Huang and Lawrence Rauchwerger. 2015. Finding schedule-sensitive branches. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ACM, 439–449.
[10] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. 2009. CalFuzzer: An extensible active testing framework for concurrent programs. In *International Conference on Computer Aided Verification.* Springer, 675–681.
[11] Jie Liang, Mangzhe Wang, Yuanliang Chen, Yu Jiang, and Renwei Zhang. 2017. Fuzz Testing in Practice: Obstacles and Solutions. (2017).
[12] Chunhua Liao, Pei Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *the International Conference for High Performance Computing, Networking, Storage and Analysis.* 1–14.
[13] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. Jacontebe: A benchmark suite of real-world java concurrency bugs (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.* IEEE, 178–189.
[14] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating data race witnesses by an SMT-based analysis. In *NASA Formal Methods Symposium.* Springer, 313–327.
[15] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 391–411.
[16] Lorna A Smith, J Mark Bull, and J Obdrizalek. 2001. A parallel java grande benchmark suite. In *Supercomputing, ACM/IEEE 2001 Conference.* IEEE, 6–6.
[17] Mangzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Han Liu, Xibin Zhao, and Jiaguang Sun. 2017. SAFL: Increasing and Accelerating Testing Coverage with Symbolic Execution and Guided Fuzzing. (2017).