Automated Classification and Prioritization of Data Races

Baris Kasikci
DSLAB, I&C, EPFL

Abstract—Multithreaded software is full of data races. State of the art data race detectors tend to report a large number of races that are mostly harmless. Therefore, it is difficult to choose to fix races that actually matter. Moreover, one cannot eliminate all data races, as this would hurt system performance. Developers are faced with the need to prioritize such race reports in order to address them in decreasing order of impact.

In this writeup, we first review a bug classification and prioritization system and then explore two papers that attempt to classify data races according to their impact.

Inspired from these papers, we propose a novel approach to data race classification. Our preliminary results, obtained with our prototype, PORTEND, are promising. PORTEND manages to accurately classify a few dozen races without human intervention, in reasonably short time.

Index Terms—software testing, data races, classification, triage, prioritization, thesis proposal, candidacy exam write-up, EDIC, EPFL

I. INTRODUCTION

A data race occurs when two threads can access a shared memory location concurrently and at least one of these accesses is a write. This typically happens because ordering of those concurrent accesses is not enforced by synchronization operations like barriers. As software becomes increasingly parallel, we expect the number of data races it contains to increase.

Data races can be harmless or harmful, depending on how they affect program correctness. Harmless data races may be present in programs for various reasons such as performance or mere convenience for the developer. Multiple, unsynchronized updates of a progress bar, or a race that does not ever affect the program output are examples of harmless data races. Harmful data races typically violate a program’s correct operation specification by causing the system to hang, crash, etc. Harmful races are some of the worst bugs because they caused people to lose their lives [1] and companies to be subject to massive economic loses [2].

It has been shown that 69% to 90% of data races reported by static or dynamic race detectors are harmless, and therefore they do not need to be fixed [3], [4], [5], [6]. Compounding the issue of harmless races, both static and dynamic race detectors report false positives, that is reports that do not represent actual races. In the case of static race detectors, false positives arise because of races that cannot occur in practice, but are flagged due to the lack of runtime information. Dynamic race detectors do not have false positives if they can monitor all synchronization operations in an application. However, with the exception of very few detectors [7], [8], most dynamic race detectors cannot track user-defined synchronizations, and therefore, also report false positives.

Another problem with data races is they are typically difficult to reproduce and analyze. Practitioners from industry report that diagnosing a data race bug can take days to weeks [9]. A recent study that we have conducted showed that if a state of the art race detector like Google ThreadSanitizer [10] is used while loading a single web page in Firefox, around 1000 data races are detected.

Given the critical losses caused by harmful data races, it is not possible to ignore them. However, considering all the races reported by a race detector as having the same priority (whether static or dynamic) is impractical for the aforementioned reasons. All these results lead us to the conclusion that the software community can benefit from a data race prioritization approach.

Classification and prioritization of bug reports has been treated before in the context of general bugs [11] and to a limited extent for data races [5], [12]. The system that gathers data from the largest number of computers is Windows Error Reporting (WER) that we discuss in section §II-A [11]. Then we present a technique to observe the two possible
interleavings that can be taken to resolve a race (that is make one of the threads the winner of the race) and examine the effect of this resolution on program state in section §II-B [5]. Finally we explore a technique that shows how random fuzzing of thread schedules can be used to uncover harmful races in section §II-C [12].

Previous race classification approaches either try to find a definitely harmful race [12] (one that crashes or hangs the system, etc.) or try to categorize races in “potentially harmful” or “potentially harmless” classes [5]. Our experiments showed that it is possible to have a finer grained classification than what was proposed by previous work with a different approach (§III-B).

In this writeup, we suggest a new dynamic data race detector that is capable of prioritizing detected races with higher accuracy than previous work, according to our preliminary experiments. Our prototype PORTEND, manages to classify several dozen races from real programs with full accuracy, according to a novel four-category classification scheme.

PORTEND uses a combination of multi-schedule and multi-path analysis (§III-B) in order to explore the effects that various schedules and inputs may have on the outcome of a data race. PORTEND allows developers to reason about effects of a data race: Can they lead to hangs or crashes? Can they alter the system state? Can their effects be visible in program output, or are they harmless?

In our approach to data race classification, we were inspired by WER for the general principle of bug prioritization, and by RACEFUZZER [12] and Record/Replay Analyzer [5] regarding the application of prioritization to data races. In the rest of this writeup, we first review the state of the art in bug prioritization (§II-A) and race classification (§II-B, §II-C). Then, we explore the limitations of these approaches (§III-A) and propose our own solutions (§III-B). We present preliminary evaluation results of our approach (§III-C), suggest future work directions (§III-D), and conclude (§III-E).

II. SURVEY OF THE SELECTED PAPERS

This section summarizes three papers that illustrate the importance of classifying and prioritizing bugs with a particular emphasis on data races. The summary of these papers follow the same structure: First, the paper is briefly summarized with an emphasis on its main contributions. Then, the system’s design and implementation are presented. Finally, experimental evaluation of the system is reviewed. We defer the limitations of the papers until section §III-A, to discuss them collectively and put them in perspective.

A. Debugging in the (Very) Large: Ten Years of Implementation and Experience

Summary

Windows Error Reporting (WER) is a distributed system aimed at automatically processing a large number of error reports [11]. The errors are gathered from a base of a billion machines (2009). WER is capable of both automatically generating and automatically classifying error reports. The primary goal of WER is to prioritize the error reports in order for developers to fix bugs that affect the most users. WER is adopted by all Microsoft product teams and has helped developers fix thousands of bugs (e.g., 5000 bugs in Windows Vista). Finally, WER, thanks to its scale, has been able to identify Heisenbugs [13], which only occur under rare circumstances.

WER Design and Implementation

Before delving into WER’s design and implementation, we define error and bug in the context of WER. An error is an event in the code whose outcome is different from what the developer expected. A bug is the root cause that can result in one or more errors.

WER has a client-server architecture with client nodes situated on every machine running any Microsoft software, and central Microsoft servers that gather error reports from those clients to process them.

WER design and implementation rely on six fundamental strategies: error report bucketing, progressive data collection, automation, preserving privacy, guidance to solutions, and generalization to the entire software products of Microsoft. We now briefly explain each strategy.

Error report bucketing is one of the most important components of WER and consists of error labeling and classifying. Labeling is performed at the client side with all the available evidence in an effort to assign the error reports from a single bug in a single bucket. Classification is performed on the server side with more error reports available to achieve the one to one mapping between error reports and buckets.

Orthogonal to labeling and classifying, expanding and condensing heuristics are used to further increase the accuracy of bug classification. These heuristics are derived empirically with the goal of having one bug per bucket and making sure that error reports from the same bug don’t end up being classified in different buckets.

WER collects information from client machines in a progressive manner in order to scale to a large number of machines. Most error reports contain simple bug labels. WER only asks clients for further data in case the available information is insufficient to fix the bug. This is done through instructing the related WER client to gather more data to be sent when the same error is reproduced subsequently.

WER aims for automation at both ends of the system, namely the user and the developer ends. The only step requiring authorization from the user is when the error report needs to be submitted. WER system allows one time “opt-in” and “opt-out” features to automate this process as well. On the developer site, only error reports that fall into unresolved bug buckets are processed.

Microsoft applies company wide data access restriction to WER data in order to preserve user privacy. Moreover, all serial number and hardware configuration information is zeroed prior to being sent to WER servers. Finally the entire WER system is based on informed user consent with all the default settings configured to not participate to the WER system.

WER service maintains a link to the URL of a solution to
a known bug. This solution can be an update or a site with a relevant patch that fixes the bug. Solutions such as massive system updates are observed to be generally applicable to more than a single bug.

WER is used by all product teams within Microsoft, and it is provided as a service that third party software companies can access. In this way, both the teams within Microsoft and third party vendors benefit from WER.

Evaluation
The evaluation results of WER represent data collected from over a billion computers. We first look at WER’s effectiveness in finding bugs, then we give results showing how well WER scales. Finally, we consider effectiveness of bucketing heuristics.

WER is intended to be used in conjunction with other bug detection methods used in Microsoft. Although the number of bugs discovered by WER is about 20 times less compared to those found by static analysis and model checking, the bugs found by WER are those that did not show up in such prior analysis. As a consequence of using WER, developers of Windows Vista fixed 5,000 bugs that were not detected during prior testing.

WER bug reports show that a small number of bugs are responsible for a large number of error reports. In particular 50% percent of all error reports belong to 20 buckets (which account for around 0.2% of the total number of buckets). The bugs whose error reports are reported by WER, rather than by human operators, are found to be up to 5.1 times more likely to be fixed.

WER proved to be helpful in fixing Heisenbugs, bugs that only occur under rare circumstances. These bugs are mostly related to concurrency errors and require special conditions to hold before manifesting themselves. For example, WER was able to identify a bug in MSN that was encountered more than twice by less than 0.18% of its users over a month.

From 1999 until 2009, WER collected several billions of error reports. Over this period, the number of bug buckets grew, whereas the number of errors reported per user decreased consistently. This suggests that bucketing heuristics were effective and that overall, the WER system helped Windows software achieve a higher quality.

The WER system relies on heuristics to achieve classification and prioritization, and therefore, is not always accurate. In particular, condensing heuristics, which should direct error reports from the same bug to a single bucket failed up to a maximum of 37%. Expanding heuristics that should direct error reports from different bugs to different buckets proved to me more effective with at most 4% inaccuracy.

B. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis

Summary
This paper introduces the state of the art in data race classification, namely record/replay analysis. Record/Replay analysis (RRA) is based on the insight that for a race to be classified as harmless, the order in which the two racing threads make their accesses to a shared variable should not alter the system state. The primary goal of RRA is to classify races into two categories, namely that of “potentially harmful” and “potentially harmless”. In order to achieve this, a multi-threaded program’s execution is recorded and then replayed offline while detecting races in this offline execution. The replacer is capable of exercising the two possible interleaveings that a race can have, and it compares program states after each interleaving to classify races. The races are classified as “potentially harmful” if these states differ and “potentially harmless” otherwise.

RRA Design and Implementation
RRA uses iDNA record/replay infrastructure to deterministically record and replay multithreaded program executions [14]. iDNA records initial system state (register values and the program counter) when the program starts executing. iDNA records all the load instructions during recording to eliminate all sources of non-determinism. iDNA optimizes to not log the load instructions unless the value in a particular memory location remains unchanged.

iDNA records special log items called sequencers every time a thread performs a synchronization operation or a system call. A global clock is incremented at each such operation to maintain a total order among sequencers. Two consecutive sequencers in a single thread are called a sequencing region.

iDNA replays perform offline replay of the recorded log, one sequencer region at a time and uses a happens before based race detection algorithm to detect races. A happens before based algorithm checks whether accesses (one of which must be a write) to a shared memory location are ordered by a synchronization operation. If there is an ordering imposed by synchronization operations, then there are no races during that execution.

RRA detects races if sequencing regions that overlap in time (i.e., according to the total order) contain accesses to a shared variable of which at least one is a write. RRA replays every race execution by enforcing the two possible orderings that a race can exhibit. This is achieved by using a “virtual processor” that is added to the iDNA infrastructure, which orchestrates the thread schedule in order to enforce the two possible orderings.

The fundamental insight behind RRA’s race classification is a classification scheme based on the impact of the race on program state. According to RRA, program state is defined as the contents of the registers and memory locations after a race is resolved in one of the two possible ways. RRA will compare the memory state after the two possible executions the race can have and will classify the race as “potentially harmful” if the states differ and as “potentially harmless” if the states are the same.

RRA records the two executions in which the racing accesses are resolved in the two different ways. Whenever RRA detects a difference in the memory and register contents, it provides the two recordings to the developer so that he/she can replay these and reason better about the consequences of the races.
Evaluation

In order to evaluate the effectiveness of RRA, 18 different execution scenarios from Windows Vista and Internet Explorer were examined. In total, RRA classified 68 unique data races in these executions and the results were cross-checked by performing a manual analysis of each data race.

RRA classified 36 out of 68 races into the “potentially harmful category” out of which only 9 turned out to be really harmful after these races were discussed with the developers. There are two main reasons for misclassification of harmless races: 1) Replay failures and 2) Intentional races.

Replay failures occur due to limitations of the record replay infrastructure. iDNA infrastructure will abandon replay if the original address in the log is not encountered or if the replay diverges from the recording because of a slight change in control flow. In such cases, RRA preemptively classifies races as “potentially harmful”. In particular, 6 out of the 36 “potentially harmful” races were wrongly classified as such merely due to replay failures.

Intentional races are present in programs mostly because of performance reasons. For example, a statistics counter or a progress bar update by multiple threads may not need to be synchronized simply because having slight inconsistencies on their values does not affect the system in a critical way. In particular, 23 out of 29 races were wrongly classified as “potentially harmful” because RRA cannot reason about developers’ intentions.

Further study of the races that were wrongly classified revealed several patterns of races that are intentionally present in the tested programs. The first type of benign race, is user defined synchronization. User defined synchronization — which is also referred to as ad-hoc synchronization — can be implemented in various ways that an automated recording system cannot be aware of. Therefore race detectors typically report races for accesses to shared variables that are in fact protected by user defined synchronizations. The second common type of benign races happens when two threads write the same value to a shared variable. The third type of frequent race is when two threads write different values to a shared variable but the final outcome of the program does not change. The fourth type of benign race is the double checked locking paradigm where a variable is read without a lock before it is written to, and a lock is used only if the read value satisfies a certain criteria. The last type of benign race is detected by race detectors when threads modify disjoint bits of a bitfield.

RRA classified the remaining 32 races as “potentially benign” and achieved full accuracy in classification as these races were manually classified to be actually benign. This high accuracy of RRA in finding benign races can be attributed to its conservative criterion of race classification based on program state comparison.

The performance overhead of RRA is on the average 6x when it performs recording and 10x when it replays a recorded trace compared to native execution. The overhead of running the happens before relationship based race detection on the recorded trace is 45x and the overhead of performing the analysis on the two possible ways a race can be resolved is 280x.

C. Race directed Random Testing of Concurrent Programs

Summary

This paper introduces RACEFUZZER, a system that uses random thread schedule fuzzing in order to uncover harmful races. RACEFUZZER uses information from an existing test case of a multithreaded program where a race has been detected. RACEFUZZER requires knowledge of the racing accesses. Then it can orchestrate the execution to resolve races in a random fashion and continue executing the programs after these random resolutions. RACEFUZZER checks to see if the program will crash or deadlock after this random resolution. RACEFUZZER can replay executions without recording any information by simply using the same random seed, and therefore it incurs low replay overhead.

RACEFUZZER Design and Implementation

Race directed random testing combines race detection with a random scheduler in order to discover a real race with high probability and then examines to see whether that race could cause a program failure such as a crash or a deadlock. The stages that RACEFUZZER goes through are 1) race detection, 2) random fuzzing of the thread schedule until the race is actually detected, 3) random resolution of a race and 4) further program exploration. We now describe each stage in more detail.

RACEFUZZER uses a hybrid race detection method in order to discover races. Hybrid race detection is essentially the same as the happens before relationship based race detection described in section II-B with the additional check to see whether the accesses from the two threads were done holding the same lock. If a happens before relationship is not found or there is no common lock when accessing the shared memory, a race is flagged. Hybrid race detection is less accurate than pure happens before relationship based race detection as not holding a common lock when accessing a shared memory does not necessarily mean that accesses are racing (e.g., they may be required to execute in a strict order using barriers).

RACEFUZZER then starts executing the multithreaded program by randomly selecting a thread from the set of enabled threads and executing its next instruction unless that instruction belongs to the set of potentially racing instructions discovered in the previous stage. In such an event, the thread that is about to make the allegedly racing access is blocked and other threads are repeatedly selected to run. The goal is to try and bring the second racing access temporally next to the first one. If that can be done, it means that the race detected in the previous stage was indeed a real race.

The resolution of the race is done by selecting one of the racing accesses to be performed first. This selection is also randomly performed. The main insight of this random selection is as follows: Races generally get resolved in one particular way, however they tend to cause problems once they are resolved in the less frequent way. The rationale behind this insight is that if the common resolution case of the races were to cause a problem, that would have been detected in early stages of program development and testing. By resolving
the race in a random fashion, both resolution scenarios are effectively equally likely and therefore it should be possible to see the effects of resolving the race differently.

RACEFUZZER continues executing the program after random resolution of the race to explore the effects of the race on the program. Unlike RRA, which stops exploring the effects of data races immediately after the race is reproduced, RACEFUZZER continues execution to obtain more information about the consequences of the race.

As an implementation detail, it should be mentioned that RACEFUZZER performs thread switches not on every statement but rather on every synchronization operation. This is based on the previous observation that concurrency errors can be reproduced if thread schedule manipulation at the synchronization operation granularity is performed [15].

The scheduling method adapted by RACEFUZZER is capable of detecting data race bugs that are not likely to occur during normal execution. This situation can be described by a code snippet as follows:

```java
thread1{ thread2{
1. lock(L);
2. f1();
3. f2();
4. f3();
5. unlock(L);
6. if(x == 0)
7. ERROR;
}
```

The condition to hit the error in this program can only be satisfied if statement 6 is reached before statement 8 is executed given x has an initial value of 0. However, in a regular execution, this is not very likely to happen as thread1 can be preempted in favor of thread2 all the way from statement 1 until statement 5 which would mean the bug would not surface. However, the race on shared variable x will be detected by RACEFUZZER’s race detector and since the thread schedule is controlled to bring the racing accesses temporally next to each other and then resolve the race, the bug will be detected with 50% probability.

**Evaluation**

RACEFUZZER is evaluated on a set of 14 Java programs and it managed to find both previously known and unknown data race bugs. These bugs manifest themselves in the form of unhandled exceptions. The experiments were run 100 times to determine bug finding effectiveness of RACEFUZZER as well as to assess its performance.

RACEFUZZER’s runtime overhead is 1.1x to 3x greater than native execution for I/O intensive benchmarks. On the other hand, for CPU intensive computing applications, overhead is more than three orders of magnitude. However, RACEFUZZER is intended to be used as a testing and debugging tool and, therefore, high runtime overheads are tolerable to a certain extent.

The probability of experiencing concurrency bugs with RACEFUZZER’s scheduler is higher compared to merely using Java Virtual Machine’s (JVM) own scheduler. In particular, with the default scheduler of JVM, only 3 out of 30 unhandled exceptions in all the benchmark programs were observed.

The probability of hitting a data race bug with RACEFUZZER’s scheduler was higher than for all but one of the tested programs. Moreover, for 11 out of 13 programs had an average probability of hitting a bug higher than 85%.

**III. Research Proposal**

**A. Limitations of the State of The Art**

WER system’s impact within Microsoft and its success on prioritizing bugs inspired us to attack a similar problem for data races, which are difficult to reproduce in practice. WER leverages its scale in order to uncover obscure Heisenbugs. Nevertheless, this approach is not systematic, and data race bugs like the one we presented in section §II-C may still not be discovered. Moreover, WER system does not provide a way to easily reproduce such concurrency bugs but it rather presents the developers, information related to the final state of the program.

RRA performs record/replay analysis and examines the effects a race can have on system memory and register state by considering the two possible ways in which the race can be resolved. Our experimental evaluation (§III-C) revealed that program state comparison leads to inaccurate classification mainly because contents of memory tend to differ for simple reasons such as dynamic memory allocations. Therefore, we concluded that a higher level of abstraction than program states is more applicable for classifying races (§III-B) such as comparing program outputs. Moreover, RRA stops exploring the two possible race interleavings immediately after the race occurs. We found that proceeding like RACEFUZZER, and executing the program till it terminates is more effective in gaining more insight into the consequences of data races.

RACEFUZZER allows developers and testers to reason about consequences of data races in more informative ways than RRA, as it can actually pinpoint the actual failures. RACEFUZZER is helpful in identifying harmful races but it does not attempt to classify harmless races and does not give guarantees that some races are not going to cause any problems. Moreover, the approach taken is random in nature and therefore does not provide a systematic method for classification.

None of the previous approaches consider how different program inputs may affect the outcome of races. The consequences of resolving the race in the two possible ways may differ (and we have seen that it indeed differs (§III-C)) depending on the inputs with which the program is ran. We call this multi-path analysis and together with RACEFUZZER’s approach, (that we term multi-schedule analysis) we have found the classification to be more accurate.

**B. A New Approach to Race Classification**

Our approach introduces a novel four category classification for races which is finer grained than what was proposed by previous work.
According to this classification, “spec violated” corresponds to races for which, at least one of the racing accesses leads to the violation of program specification like causing hangs or crashes. The second category of races is “output differs” for which one of the orderings of racing accesses leads the program to generate different outputs and hence makes the outputs schedule and/or input dependent. If the outputs of the program do not differ for various executions (with different thread schedule and inputs) of the program for both orderings of the racing accesses, we classify the race as “k-bounded harmless”. The notion of “k-bounded harmless” races will be discussed in more detail shortly. If it is impossible to enforce the racing accesses in the two possible orderings we classify those races as “single ordering”. This suggests that race detection had false positives because it did not manage to spot a user defined synchronization.

Our prototype PORTEND for which we implemented our approach, is a dynamic race detector that is based on the happens before relationship. It can run developer’s test suites or any other execution of the program to detect races while recording a thread schedule trace. PORTEND possesses a record/replay infrastructure that can control the scheduling of threads and it uses KLEE symbolic execution engine [16] to enumerate program paths and gather symbolic constraints that propagate through the program’s execution.

In the first step of the analysis (shown in Fig 1a), PORTEND replays the recorded schedule until the race under investigation is reproduced. Then it explores the two possible executions by resolving the race once in the original way it was recorded (primary) and once in which the alternate access ordering of racing statements is enforced (alternate). This corresponds to “single-pre/single-post” analysis and is mainly performed by PORTEND to determine whether the alternate schedule is even possible or there is a user defined synchronization that makes such an access sequence impossible. If it is indeed not possible to enforce the alternate schedule, then the race is classified as “single ordering”.

PORTEND does not immediately classify races if there is a difference between post-race program states of the primary and alternate as RRA does. Instead it continues executing the program to observe the consequences. If the program crashes or hangs, we classify the race as “spec violated”. If on the other hand we observe a difference in program outputs, we classify the race as “output differs”.

However, even if the primary and alternate executions behave identically during “single-pre/single-post” analysis, it does not mean that the race is harmless. Therefore PORTEND explores multiple paths in the pre-race execution as well as in the post-race execution leveraging symbolic execution to find those multiple paths that take the program through the race.

Finally, we combine this multi-path analysis with multi-schedule analysis, because the same path through a program may generate different outputs depending on how its execution segments are interleaved. To enable such combined analysis, we augmented the KLEE symbolic execution engine [16] to support threads and allowed PORTEND to control their schedule.

Of course, exploring all possible paths and schedules that experience the race is impractical, because their number can grow exponentially with the number of threads, branches, and preemption points in the program. Instead, we provide developers a mechanism to control the number $k$ of path/schedule alternatives explored during analysis, in essence allowing them to control the “volume” of paths and schedules in the shaded areas of Fig. 1. If PORTEND classifies a race as “k-bounded harmless”, then a higher value of $k$ offers higher confidence that the race is harmless for all executions (i.e., including the unexplored ones), but it entails longer analysis time.

When PORTEND determines that a race violates a specification, it provides the corresponding evidence: program inputs (including system call returns) and a thread schedule that reproduce the harmful consequences deterministically. This “evidence” can be replayed in a debugger, thus helping developers to debug the race, not just classifying it.

C. Evaluation

We applied PORTEND to 7 real applications: SQLite, a widely used embedded database engine which is considered highly reliable, with 99% test coverage [17]; Pbzip2, a parallel implementation of the widely used bzip2 file compressor [18]; Memcached, a distributed memory object cache system used by services such as Flickr, Twitter and Craigslist; Ctrace, a multithreaded debug library [19]; Bbuf, a shared buffer implementation with a configurable number of producers and consumers [20]; Fmm, an n-body simulator in the popular SPLASH2 benchmark suite [21]; and Ocean, a simulator of eddy currents in oceans, from SPLASH2.

PORTEND classifies with full accuracy all the 49 known data races in these programs, with no human intervention, in under 19 minutes total. It took one person-month to manually confirm that the races deemed harmless by PORTEND were indeed harmless—this is typical of how long it takes to classify races in the absence of an automated tool [9]. For the deemed-harmful races, we confirmed classification accuracy in minutes, by using the replayable debug information provided by PORTEND.

We also evaluate PORTEND on homegrown microbenchmarks that capture most classes of harmless races [10], [5]: “redundant writes” (RW), where racing threads write the same value to a shared variable, “disjoint bit manipulation” (DBM), where disjoint fields of a bit-field are modified by racing
threads, “all values valid” (AVV), where the racing threads write different but valid values, and “double checked locking” (DCL), a method used to reduce the locking overhead by first testing the locking criterion without actually locking. Table II describes all 11 experimental targets.

<table>
<thead>
<tr>
<th>System</th>
<th>Size (LOC)</th>
<th>Language</th>
<th># Forked Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite 3.3.0</td>
<td>113,326</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>ocean 2.0</td>
<td>11,603</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>ffmpeg 2.0</td>
<td>11,545</td>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>memcached 1.4.5</td>
<td>8,300</td>
<td>C</td>
<td>4</td>
</tr>
<tr>
<td>pbzip2 2.1.1</td>
<td>6,868</td>
<td>C++</td>
<td>4</td>
</tr>
<tr>
<td>ctrace 1.2</td>
<td>886</td>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>tsbuf 1.0</td>
<td>261</td>
<td>C</td>
<td>8</td>
</tr>
<tr>
<td>AVV</td>
<td>49</td>
<td>C++</td>
<td>3</td>
</tr>
<tr>
<td>DCL</td>
<td>45</td>
<td>C++</td>
<td>5</td>
</tr>
<tr>
<td>DBM</td>
<td>45</td>
<td>C++</td>
<td>3</td>
</tr>
<tr>
<td>RW</td>
<td>42</td>
<td>C++</td>
<td>3</td>
</tr>
</tbody>
</table>

**TABLE II**

Programs analyzed with Portend. Source lines of code are measured with the **loc** utility.

To evaluate Portend’s accuracy and precision, we had it classify all 49 races in our target applications and micro-benchmarks. Table I summarizes the results. The first two columns show the number of distinct races and the number of instances, i.e., the number of times those races manifested during race detection. The “spec violated” column includes all races for which the program either crashed or hanged. In the “k-bounded harmless” column we show for which races the post-race states differed vs. not. We used 5 for the value of k and found out that in practice it is a sufficient number for exploring the paths to reveal differences for the systems that we have tested. Finally, the “single ordering” column contains the largest number of races, thus confirming the observation that most data races are harmless [7], [8], [4].

To determine accuracy, we manually classified every race and found that Portend had correctly classified all 49 races—this means it achieved full accuracy on our target races and applications. We found that all races classified “k-bounded harmless” by Portend are indeed harmless in an absolute sense. Finally, we found that all “single ordering” races involve ad-hoc synchronization.

Multi-path multi-schedule exploration proved to be crucial for Portend’s accuracy. In particular, for 5 out of 9 “output differs” races (3 in bbuf, 2 in ctrace) and for 1 “spec violated” race (in ctrace), single-path analysis revealed no difference in output; it was only multi-path multi-schedule exploration that revealed an output difference (all 6 races required multi-path analysis for classification, and 3 races required combined multi-path multi-schedule analysis). Without multi-path multi-schedule analysis, it would have been impossible to accurately classify those races by just using the available test cases and relying on any of the previous race classification schemes.

We also compared Portend’s accuracy to the accuracy of several other approaches: 1) RRA, which we presented earlier, 2) Helgrind” [8], and 3) Ad/Hoc Detector [7], which both specialize in detecting ad-hoc synchronization patterns. We also include comparison results to the ground truth (manual analysis). The results of this comparison can be seen in Fig 2. It can be seen that, Portend achieves full accuracy. Record/Replay-Analyzer does not tolerate replay failures and classifies races that exhibit a post-race state mismatch as harmful (shown in “spec violated”), causing 40 misclassifications. When comparing to Helgrind+ and Ad-Hoc-Detector, we are conservative and assume that these tools incur zero false positives when ad-hoc synchronization is present, even though it is unlikely, since both tools rely on heuristics. This notwithstanding, both tools weed out races due to ad-hoc synchronization, so they cannot properly classify the remaining races (21 out of 49). In contrast, Portend can classify a much wider range of races.

To evaluate precision, we ran 10 times the classification of all races. Portend consistently reported the same data set shown in Table I, which indicates that, for these races and applications, it achieves full precision.

**D. Future Research Directions**

There are various paths that we want to follow to improve Portend and also to attempt bug classification and prioritization in the context of other software bugs.

We would like to make Portend available through an IDE like Eclipse. The projected usage scenario for Portend is in the same vein with the testing as a software service idea [22]. Portend could work in the background whenever a developer is writing code and run developer tests in the background to see whether any potential race introduced by the developer should be fixed or not.

In order for Portend to reach its full potential, we intend to look at the possibility of integrating it with path synthesis capabilities of ESD [23]. ESD synthesizes a program execution from a bug report leveraging symbolic execution. Note that this does not necessarily mean that we need to have Portend system run together with ESD and this project is rather related to taking ESDs path synthesis capabilities to the next level. In doing so, Portend will have taken a step in becoming a data race classifier for both static and dynamic race detectors.

Once we can synthesize paths to bugs (not necessarily only data races) reported by static race detectors, we plan to extend Portend to be a “universal bug prioritization” system that can prioritize a large number of bugs that such detectors flag.
It is also possible to extend race classification into race avoidance. PORTEND would be valuable in such a case as avoiding all the races in an application will probably be very costly.

PORTEND can provide more information regarding the consequences of data races rather than mere output comparison. Specifically, if PORTEND will be able to give insight on the performance impacts of having the race versus not having it, this could be beneficial for the developers.

Finally, we intend to extend PORTEND to automatically suggest fixes for detected bugs and evaluate them to inform the testers and developers on the impacts of the proposed fixes. These impacts can range from performance penalties to causing previously unknown bugs to surface.

E. Conclusion

In this writeup, we first presented a recent technique related to bug prioritization and classification. Then we reviewed two papers that deal with classifying data races according to their severity and consequences.

We presented a technique for triaging data races based on their potential consequences through a multi-path and multi-schedule analysis. We perform triaging using a novel four-category data race classification scheme.

PORTEND, a prototype based on this technique, classified 49 different data races in 7 real-world applications and benchmarks with full accuracy and precision, without human effort. Moreover, we argued PORTEND can be used as a debugging aid to fix data races.

REFERENCES