RRF: A Race Reproduction Framework for Use in Debugging Process-Level Races

Supat Rattanasuksun*, Tingting Yu†, Witawas Srisa-an*, Gregg Rothermel*
*Department of Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588, USA
{supat, witty, grother}@cse.unl.edu
†Department of Computer Science, University of Kentucky, Lexington, KY 40506, USA
tyu@cs.uky.edu

***Abstract*—Process-level races are endemic in modern systems. These races are difficult to debug because they are sensitive to execution events such as interrupts and scheduling. Unless a process interleaving that can result in the race can be found, it cannot be reproduced and cannot be corrected. In practice, however, the number of interleavings that can occur among processes in practice is large, and the patterns of interleavings can be complex. Thus, approaches for reproducing process-level races to date are often ineffective. In this paper, we present RRF, a race reproduction framework that can help software engineers reproduce reported process-level races, enabling them to potentially debug these races. RRF performs a hybrid analysis by leveraging existing static program analysis tools, dynamic kernel event reporting tools, and yield points to provide the observability and controllability needed to reproduce races. We conducted an empirical study to evaluate RRF; our results show that RRF can be effective for reproducing races.**

I. Introduction

Concurrency faults are difficult to detect and debug because they are sensitive to execution interleavings. *Process-level races*, where multiple processes access operating system resources without proper synchronization, are particularly challenging to handle. Process-level races can occur across different applications written in different languages and accessing diverse resources. A race detector designed for a specific language or operating at the thread level may not be able to detect and reproduce such races. In addition, race detection and reproduction require observability of shared memory accesses and synchronization operations. Process-level races, however, can occur in kernel services (e.g., system call routines and device drivers) that are not observable by software engineers. Finally, process-level races can occur at times well past system deployment due to changes in system configurations. Because the scenarios in which process-level races occur are complex, these races often elude in-house testing efforts and remain undiscovered until after system deployment. In fact, up to 73% of all races reported after deployment in various Linux distributions are process-level races [14].

When process-level races are discovered they are typically reported to engineers. Examples of bug reports describing races can be found in various bug repositories including those for Red Hat, Ubuntu and Debian. In most cases, bug reports include only general descriptions of symptoms; in some cases, bug reports are more detailed and provide inputs that cause the races. Engineers use this information to identify potential sources of races (e.g., file systems, network stacks, or device drivers) and attempt to reproduce them as part of the debugging process. Unfortunately, reported information alone is often not sufficient to reproduce races. We have encountered cases in which 50,000 execution instances cannot reproduce a reported race due to the small execution interleaving windows that must occur in order for the races to manifest themselves.

In practice, engineers often use `sleep` or `sched_yield` calls to pause the execution of pairs of processes [19], [28], to increase the chances of enforcing execution windows that lead to races. Inserting yield points at the source code level, however, is an error-prone approach when dealing with process-level races. Shared resource usage can be implicit; that is, an application may make a request to use a shared resource indirectly by invoking a system call that invokes another system call to access the racy resource. Engineers may miss the opportunity to observe the behavior of an implicit system call if they fail to insert a yield point prior to the event that can result in that call. In another class of approaches known as *record and replay approaches* [14], engineers may fail to observe problems due to replay divergence – a mismatch between the actual actions recorded in a trace and the modified actions intended to enforce execution windows. Replay divergence occurs when engineers directly modify a trace of a program's execution and cannot replay it after the modifications [14].

Process-level races can also involve more than two pairs of shared resource accesses. In such cases, to localize races based on a reported symptom, engineers need to identify locations that share the same resource and then observe the behavior of *each* pair-wise access to the shared resource. (A pair-wise access is the minimum requirement for exposing a race [17].) If engineers fail to exercise any pair-wise accesses, they may be unable to reproduce not only the reported race, but also to identify other locations that can possibly cause additional races on the same shared resource.

It follows that engineers need a debugging framework that allows them to observe the *system-level events* such as system calls and signals that occur during program execution, without encountering replay divergence, so that they can uncover both explicit and implicit system calls that might be relevant to the race they are trying to reproduce. They also need an approach for automatically mapping system-level events to invocation points in source code and inserting yield points

prior to and after these invocation points. They then need to be able to observe the runtime behaviors of *all* pair-wise accesses of shared resources, and validate system behaviors against specific correctness properties that can determine whether a race occurs and whether it is the reported race.

In this paper we present RRF, a race reproduction framework that helps software engineers automatically reproduce reported races and verify whether these races have been reproduced through execution. RRF performs a hybrid analysis by leveraging existing static program analysis tools and kernel event reporting tools including DTrace [7], iNotify [16], and Scribe [13]. These tools provide the observability needed to identify explicit and implicit accesses to shared resources and automatically map them to application events. To exercise a specific interleaving, RRF automatically injects yield points at these application events and executes the racing applications to observe pair-wise accesses to a potentially racy resource. RRF also provides a repository in which engineers can store specified correctness properties that can be used to validate whether a pair of resource access operations experience races and locate where they occur in the programs.

We conducted an empirical study to evaluate RRF. We located seven systems containing real-world races that had been reported in Linux bug repositories. We then applied RRF to reproduce the races. We studied whether our framework can reproduce the reported symptoms better than a baseline approach involving stress testing, and found that our approach was both more efficient and more effective.

II. Background

A. Motivation

In this work, we define a *process-level race* as a race that occurs when (1) two processes access a shared resource, and (2) they could have accessed the shared resource in an order different from the original order. This definition of a race is broader than the standard definition [1], [28]. Our definition also includes order violations, in which the desired order of shared resource accesses is reversed and the accesses may or may not be protected by a common lock [17], [21]. An order violation is a necessary but not sufficient condition for a race.

We provide examples to illustrate how process-level races occur and why they are difficult to debug. The first issue that makes debugging process-level races difficult is the lack of execution observability. Because system-level events including system calls and signals are typically treated as black boxes by engineers, the sequences of events that occur inside system-level events are not seen. For example, in the Linux kernel there was a change in the implementation of the MV command. Previously, to accomplish its task, MV used a combination of UNLINK and RENAME calls. The new implementation, however, just uses RENAME. As such, a previously working shell script that attempts to replace a file using the sequence of commands RM, LN, and MV no longer works correctly. In this example, engineers must be able to observe events occurring inside the target of a system call. Existing instrumentation tools that work at the statement level miss this race because they cannot detect implicit system calls.

The second issue that renders debugging difficult is the lack of execution controllability. Typically, shared resources can potentially be accessed by multiple locations in a pair of racy programs. For example, there is a scenario in which two instances of BASH shells can corrupt the history file. In the BASH code, there are at least eight locations that access the history file. These locations can *potentially* cause races when two instances of BASH are running simultaneously and they must be exercised in a pair-wise fashion to reproduce the race. We executed these two instances of BASH over 50,000 times, and were not able to reproduce the reported race. Clearly, an approach is needed that provides the controllability necessary to exercise interleavings to reproduce such races, and this approach must be capable of addressing situations in which multiple locations are involved in causing races.

B. Achieving Observability and Controllability

RRF is motivated by RACEPRO—a process-level race detection technique—together with the idea of active testing. We briefly introduce these two techniques below.

RACEPRO is an approach for reproducing races by using in-vivo execution [14]. RACEPRO uses a tool, SCRIBE, to record system-level event traces, and thus provide observability. These event traces can be used to model system calls and identify system-level shared resources (SRs). Specifically, RACEPRO tracks system calls that operate on shared resources (e.g., `open`, `read`, `write`, `lstat`, `clone`, `wait`, `execve`, and `exit`); these involve *reads* and *writes* on SRs. For example, the `lstat` system call on file f reads the metadata of f, and the `write` system call on f writes to both the data and metadata of f.

RACEPRO next analyzes systems for execution branches that can potentially lead to races, and reorders system-level events in these branches to expand the possible interleavings beyond those in the recorded trace. These reordered traces are then run on an in-vivo execution engine, thus achieving controllability. This approach has benefits: traces can be executed at near zero overhead, and access to source code is not required. The approach suffers, however, from two problems. First, reordering system calls can lead to replay divergence, causing the engine to produce an incorrect report or fail to execute. Second, the in-vivo execution engine is complex and as of this writing, is no longer being maintained. As such, it is difficult to transfer the approach to real-world usage (we discuss this further in Section V-B).

Another approach for inducing races to occur is *active testing*. In this approach, two steps are commonly performed. First, static or dynamic analysis is used to identify code locations at which races can occur. Second, functions such as `yield` are used to suspend threads around code locations that may cause races. The approach attempts to expose races by pausing one thread's execution and then checking to see whether a race actually occurs. *CalFuzzer* [27] implements

active testing for Java to detect races occurring in shared memory, and *UPC-Thrille* [20] implements it for Unified Parallel C. Active testing is slower than RACEPRO due to its need to pause threads, but it does not suffer from replay divergence as it does not explicitly reorder events. SIMRACER [35] is another approach based on active testing; it exercises interleavings to detect races by using a virtual platform to pause and resume binary execution. One difference between these testing approaches and RRF is that RRF is not a race *detection* system; rather, it is a race *reproduction* system designed to help engineers reproduce (and ultimately debug) reported races. As such, RRF need not consider all possible locations at which races can occur; instead, it focuses only on accesses to shared resources that have been reported to to race. This allows RRF to filter out unimportant shared resources. Testing approaches, on the other hand, consider all potential sources in order to expose as many races as possible.

III. INTRODUCING RRF

Figure 1 provides an overview of RRF. The input to RRF includes an application configuration, two or more programs undergoing debugging (PuDs), test inputs, bug reports, and failed output. When program inputs are provided (which is typically the case), RRF can use these to exercise the program. If inputs are not provided, test case generation techniques can be used to generate inputs. The effectiveness of RRF does depend, however, on the quality of the test inputs used.

In Component A in Figure 1, inputs are used to configure and run each PuD individually. As a `race-reproduction` (not race detection) framework, RRF needs bug reports that describe the fault that must be reproduced. In some cases, bug descriptions clearly specify the resources that need to be observed; in others, engineer expertise is needed to narrow down resources to be observed The foregoing is true, however, of `all` race reproduction approaches, so we are requiring nothing beyond those.

The output after each execution of the PuD is a log of system-level events invoked during the execution. Each event log is analyzed concurrently with the bug report to identify resources that can potentially race and system-level events that access those shared resources. The analysis result is then used to cull out events that cannot access the suspected resources. The output of this phase is a log of events of each application that may be relevant to the reported races. In Component B, the outputs of the prior phase are mapped to the original source code. The result of this task is a list of potential interleaving event pairs between the two applications that can race. In Component C, yield points are injected into the source code for each of the potential interleaving event pairs to amplify the chance of exposing races. To reproduce races, Component D compiles and executes the two modified PuDs (*mPuDs* – the original PuDs with `sleep` calls injected). As part of execution, a system-level event log is generated, containing interaction information about the PuDs. Component E uses interaction information to determine whether a race has occurred by validating the output for correctness and analyzing

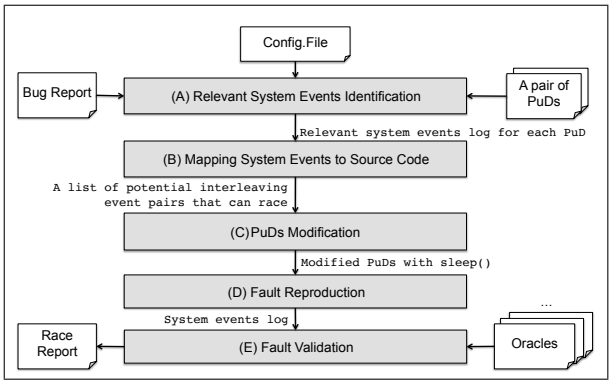


Fig. 1. Workflow in RRF

the event logs against internal property-based oracles (e.g., race detectors). A race report that includes information on whether races occur and the locations of event pairs that are involved in them is then generated.

We classify RRF as mostly automatic. An engineer must specify resources that should be observed (by analyzing information in bug reports). This is also true of manual approaches for reproducing process-level races. Unlike manual approaches, however, RRF then automatically performs analysis to identify code sections that can access the resources and injects yield points to amplify the likelihood of reproducing those races. RRF is also systematic, extensible, and general. Developers can perform instrumentation either at the source code level or by using other techniques or tools (e.g., PIN [10]) to perform instrumentation at the binary code level.

Since RRF injects `sleep` calls at the pair-wise level for each execution, a difference between the normal execution and the instrumented execution is likely to be caused by the faulty interleaving induced by the pair of events. This can help developers pinpoint the racing pair leading to the failure.

As noted in Section II, RRF combines part of RACEPRO with active testing. Specifically, we leverage SCRIBE, a system-level event monitoring and logging tool, to provide observability. SCRIBE has been used to detect process races in deployed systems by logging system-level events in Linux; these can be processed off-line to detect races and other types of concurrency faults [13]. Performance evaluations of SCRIBE in server and desktop environments showed that its average overheads are 2.5% and 15%, respectively [12]. We use SCRIBE to detect events due to its support for Linux and its low runtime overhead. If our framework needs to be deployed in other operating systems, however, other tools could be used (e.g, DTrace would function for Solaris).

Like active testing, we use process yielding to provide the controllability needed to reproduce process-level races. We inject `sleep` calls at specific code locations in the two programs to enforce a desired interleaving. RRF is systematic in the approaches it uses to identify relevant system-level events, map those events to application events, generate execution interleavings that should be explored, and validate whether a reported race has been reproduced. RRF is largely automated so it can be more precise than manual approaches for gener-

```
[39] ioctl(0, 5401, 0xbfe59d68) = 0
[39]     data: output, ptr = 0xbfe59d68, size = 36, 00000500 00000005 000000bf 00008a3b ...
[39] fstatat64(-100, 0x8059b1c, 0x8059abc, 256) = 0
[39]     data: input string, ptr = 0x8059b1c, size = 23, "/usr/local/var/locatedb"
[39]     resource lock, type = inode, access = read, id = 1, serial = 0, desc = none
[39]     resource lock, type = inode, access = read, id = 29, serial = 0, desc = none
[39]     resource lock, type = inode, access = read, id = 44, serial = 0, desc = none
[39]     resource lock, type = inode, access = read, id = 261, serial = 5, desc = none
[39]     resource lock, type = inode, access = read, id = 446, serial = 0, desc = none
[39]     data: non-det output, ptr = 0x8059abc, size = 96, 00000803 00000000 00000000 ...
[39] unlinkat(-100, 0x80588c0, 0) = 0
[39]     data: input string, ptr = 0x80588c0, size = 23, "/usr/local/var/locatedb"
[39]     resource lock, type = inode, access = read, id = 1, serial = 0, desc = none
[39]     resource lock, type = inode, access = read, id = 29, serial = 0, desc = none
[39]     resource lock, type = inode, access = read, id = 44, serial = 0, desc = none
[39]     resource lock, type = inode, access = write, id = 261, serial = 9, desc = none
[39] llseek(0, 0, 0, 0xbfe59d10, 1) = -29 ESPIPE (Illegal seek)
[39]     resource lock, type = files_struct, access = read, id = 441, serial = 14, desc = none
[39]     resource lock, type = file, access = write, id = 262, serial = 1, desc = /dev/pts/0
[39] close(0) = 0
[39]     resource lock, type = files_struct, access = write, id = 441, serial = 15, desc = none
```

Fig. 2. Sample SCRIBE output

ating interleavings. It is also extensible so that engineers can specify new oracles for use in detecting races, and identify interacting events that cause those races. Next, we describe the components of RRF.

A. Identifying Relevant System Events

For RRF to work, Component A requires inputs, including a bug report, a pair of PuDs, and a configuration file. First, engineers must analyze the bug report to ascertain information relevant to revealing shared resources on which races can possibly occur. Some bug reports provide comprehensive information (e.g., inputs or execution scripts, operating environments, other running applications, and behavioral descriptions of detected races [2], [3], [4], [15], [25]); in these cases engineers can use the information to generate specific traces of system-level events (e.g., use the provided execution scripts). Some reports, however, are less comprehensive (e.g., providing no inputs and providing only vague behavioral descriptions of the detected races [8], [9]); in these cases engineers need to perform additional tasks including additional test input generation and shared resource analysis to increase the chance of exercising code sections that can potentially cause the reported races. These activities are common to other approaches for reproducing races, so by requiring them ourselves we are requiring nothing extra. At a minimum, we expect information regarding the incorrect symptoms experienced by users of the system (e.g., file corruption, missing signals, and socket not found) to be provided as part of the bug report.

To illustrate the process for identifying resources that can potentially race we provide an example. Suppose that process UPDATEDB can race with process LOCATE, but the bug report for this race describes only what can happen without providing execution scenarios in which the race occurs. Based on the reported symptom, which is a corrupted database file located at */usr/local/var/locatedb*, an engineer could individually run the two PuDs using an existing test suite while turning on an event reporting tool such as SCRIBE. At the end of each run, a report of a list of system-level events is generated. For example, Figure 2 provides an example of SCRIBE output, that shows system-level events initiated by UPDATEDB. The associated command snippet is shown in Figure 3.

```
141 # To avoid breaking locate while this script is running, put the
142 # results in a temp file, then rename it atomically.
143 if test -s $LOCATE_DB.n; then
144   rm -f $LOCATE_DB
145   mv $LOCATE_DB.n $LOCATE_DB
146   chmod 644 $LOCATE_DB
147 else
148   echo "updatedb: new database would be empty" >&2
149   rm -f $LOCATE_DB.n
150 fi
```

Fig. 3. Mapping a block of system calls to a line of source code in UPDATEDB

In most cases, traces generated by tools like SCRIBE contain extraneous information that includes other system-level events not relevant to a reported symptom. To cull out extraneous information, we rely on the analysis of the bug report. In our example, the bug report indicates that process-level races occur on a particular database file; thus, only events that can manipulate that file are considered relevant. In the case in which a symbolic link is also made to the database file, SCRIBE also reports the actual file name as well as the symbolic name so we can still accurately remove extraneous information. In our example, the highlighted system calls in Figure 2 (`fstatat64` and `unlinkat`) are relevant to the reported race because they access the database file; remaining system calls in the figure are not relevant.

Even when an execution script is provided as part of a bug report, the foregoing process should be applied to find possible faulty locations that may exist in the path where the reported race exists. This is because RRF can determine whether there are other code locations that can produce damaging races instead of focusing only on the reported interleaving. As shown in Section V, we have found examples in which a reported race can have multiple sources within two programs. If engineers focus on a single source, they may leave other sources that result in similar races untouched.

B. Mapping System-level Events to Source Code

Component A of RRF produces a list of relevant system-level events for each application. These events must be mapped to the source code locations in the application that invoke their corresponding system calls. Component B constructs this mapping by combining static analysis and system-level

event modeling (discussed in Section II). (Note that we intend RRF to be used by engineers who are debugging applications developed by their own organizations, so it is reasonable to assume the availability of source code.)

RRF uses static analysis to identify source code locations involving application events (e.g., using `rm` to remove the database file) and then translates each event into a sequence of system calls based on the previously constructed models. The result is a mapping of each block of system level events (e.g., system calls to write to a file and manipulate the corresponding inode structure) to the corresponding event in the source code of the PuD. Figure 3 illustrates how the highlighted block of system calls in Figure 2 is mapped to line 144 of UPDATEDB. The output of this component is a list of locations where a race may be initiated.

C. PuD Modification

Given a set of locations in two PuDs that can potentially race, interleavings must be enforced at these locations. This is done by using `sleep` calls to pause the execution of one process at a particular shared resource access point and then pause the other process at an execution point that accesses the same resource. Currently, we inject `sleep` calls into source code, but binary instrumentation could also be used. To identify pause points, Component C performs pairwise analysis of relevant application events between the two processes. While a more optimized pairwise algorithm can be applied in the case of large numbers of events [4], [8], [9], [15], [25], by culling out extraneous events the number of system calls utilized can be rendered small enough to employ all combinations.

To perform pair-wise injection, the list of system calls S_i from process P_i and the list of system calls S_j from process P_j are used as inputs. System calls in S_i and S_j are paired if they access the same shared resource. σ stores a list of such pairs (σ_i, σ_j), which is initialized to empty. Since the order of the two system calls in a pair matters, we choose to include (σ_i, σ_j) and (σ_j, σ_i) in σ. The results of this process are instrumented PuDs with `sleep` calls (mPuDs).

Because our implementation uses `sleep` calls, our framework needs to use an appropriate sleep time for each injected `sleep` call. To select an initial sleep time, we consider the amount of time necessary for the critical operation to be completed. We then refine this initial time through trial runs to determine an optimal sleep time for each case. In the cases we studied, we found that one second worked well in all cases. RRF also needs to ensure that we can execute pairs of processes in specific orders. For example, we may want to ensure that a suspected system call in one process is executed before another suspected system call in another process. In this scenario, we set the sleep time of the second process to be longer than that of the first.

RRF also injects `sleep` calls before and after each suspected system call to maximize the likelihood of exposing races. In this scenario, we would explore the execution interleaving that configures P_i to execute before P_j and then explore the interleaving that configures P_j to execute before P_i. As such, two `sleep` calls (with one and two seconds of sleep time, respectively) would be injected immediately before a system call and another pair of `sleep` calls would be injected immediately after the system call. Conditional compilation is used to specify the desired execution interleaving (e.g., P_j executes before P_i).

Algorithm 1: Sleep Injection

Input : a list of suspected SysCalls from P_i (S_i),
a list of suspected SysCalls from P_j (S_j),
source code of P_i (SRC_i),
source code of P_j (SRC_j),
two targeted system calls from the Pairwise Algorithm (σ),
sleep time (ST)
Output: a set of mPuDs MP

1 $MP \leftarrow \emptyset$
2 **foreach** (σ_i, σ_j) $\in \sigma$ **do**
/* places to inject sleep before sys. call */
3 $L_{i_b} = \emptyset$
4 $L_{j_b} = \emptyset$
/* places to inject sleep after sys. call. */
5 $L_{i_a} = \emptyset$
6 $L_{j_a} = \emptyset$
7 **if** ($\sigma_i \in S_i$) **then**
8 L_{i_b} = sleep(ST) + σ_i
9 L_{j_b} = sleep($2 \times ST$) + σ_i
10 L_{i_a} = sleep(ST) + σ_i
11 L_{j_a} = σ_i + sleep($2 \times ST$)
12 **else**
13 L_{i_b} = sleep($2 \times ST$) + σ_j
14 L_{j_b} = sleep(ST) +σ_j
15 L_{i_a} = σ_j + sleep($2 \times ST$)
16 L_{j_a} = sleep(ST) +σ_j
17 MP ←GenModPuD(S_i, S_j,SRC_i, SRC_j,L_{i_b}, L_{j_b}, L_{i_a}, L_{j_a});
/* one sleep is injected before a system call;
the other after the system call */
18 **return** MP

Algorithm 1 presents our `sleep` call injection approach. For each pair of system calls σ (Line 2), the algorithm creates four mPuD MPs (Line 17). It first determines which process to execute first, by comparing the specified system call (σ_i) to system calls in P_i (S_i). If found, P_j needs to execute first and thus uses ST as the sleep time (Line 8) and P_j uses $2 \times ST$ as its sleep time (Line 13). If σ_i is not found in S_i, it is in P_j; therefore, P_j needs to execute before P_i (Lines 9 and 14). This information is used to inject `sleep` calls into the source files of P_i and P_j, which are SRC_i and SRC_j, respectively. Lines 10-11 and 15-16 specify locations at which to inject the `sleep` calls *after* σ_i and σ_j. The algorithm returns the mPuDs with injected `sleep` calls. If there are multiple sources of potential races, the same mPuDs are used and conditional compilation flags are used to enable a particular `sleep` call. Figure 4 illustrates a modified UPDATEDB.

D. Fault Reproduction

Next, the mPuDs must be executed to attempt to reproduce the reported race. To do this, RRF compiles each mPuD and runs them in pair-wise fashions as specified in σ. SCRIBE records relevant OS activities that can be used by oracles (described next) to detect whether reported races occur and determine the causes of these races. It is possible that even

```
if test -s $LOCATE_DB.n; then

#ifdef L10
  sleep 1
#endif
.
.
#ifdef L27
  sleep 1
#endif

  rm -f $LOCATE_DB

#ifdef L110
  sleep 2
#endif
.
.
```

Fig. 4. A modified UPDATEDB source file with injected `sleep` calls

with amplification by `sleep` calls, our approach can suffer from execution non-determinism because a system call is a large block of execution steps and thus, execution interleavings between two system calls have variability. To address this we ran trials using different numbers of runs and found that we could reliably reproduce races within 10 runs.

E. Race Validation

An important component of our framework is an *oracle* that can be used to observe whether a reported race occurs in each run. Because races are dynamic in nature, we employ both internal oracles [36] and output based oracles to help detect whether a race occurs and isolate the locations of its causes. The information generated by our oracles is then used to further classify each detected race into three possible types: *targeted*, *incidental*, and *new*. If a run is intended to exercise a particular race (based on where we inject `sleep` calls into the two PuDs), and that same race is detected, it is classified as targeted. If a race is detected but it is not due to our `sleep` calls, it is either an incidental race or a new race. An incidental race is a detected race that we did not intend to exercise in a given run but would attempt to exercise later through a different set of `sleep` calls; however, it naturally occurs as part of the run. A new race is a race that is not related to the possible locations, determined by our analysis, at which a reported race can occur.

We designed our internal oracles to look for property violations such as TOCTOU (time of check to time of use) violations, atomicity violations, and races. Our oracles are stored in an expandable repository so that engineers can extend our framework to detect types of faults beyond those covered in this work. At the end of the RRF process, a report is generated to specify whether races occur, classify their types, and provide their locations in the two PuDs. This information can help engineers localize faults.

In terms of execution complexity, there are three factors that can determine the number of runs that should be used to try to reproduce races. The first factor is the number of system calls in the two applications that must be executed in pairwise fashion (S_i and S_j). In our example, LOCATE and UPDATEDB have three relevant system calls each. As such, the number of pairs (interleavings) is 3×3. The second factor is the number of patterns that we want to explore for each interleaving. Currently, we have four patterns: P_i before P_j before the system call, P_j before P_i before the system call, P_i before the system call and P_j after the system call, and P_j before the system call and P_i after the system call. As such, we need to multiply each interleaving by four. The third factor is the number of runs that we need to overcome non-determinism (r). As such, the number of runs we need to reproduce a reported race between two applications is $sizeof(S_i) \times sizeof(S_j) \times 4r$.

IV. EMPIRICAL STUDY

RRF provides an automated process for reproducing specific classes of process-level races. Ultimately we are concerned with whether engineers who use RRF can localize and correct process-level races more quickly than with current approaches. Studies of humans, however, are expensive, and before embarking on these it makes sense to study whether the approach can indeed reproduce races reliably, and in a reasonable amount of time. We therefore designed an empirical study considering the following research questions:

RQ1: Given process-level races that fall within the type of races RRF targets, how effective is RRF at reproducing these?

RQ2: Given process-level races that fall within the type of races RRF targets, how costly is it to run RRF on these?

A. Objects of Analysis

To answer our research questions we needed to locate instances of systems in which process-level races had been reported, that are of the classes of races RRF targets.

To do this we turned to several bug report repositories, including those for Red Hat, Ubuntu, Debian, and GNU, and searched them using the keywords “race”, “concurrency”, “file”, “signal”, and “socket”. We considered only bug reports about code written in C or shell scripting language. We then narrowed our focus further to bug reports that involve file or socket resources that are shared among multiple processes. We did this because, while RRF has been designed to detect a broad class of concurrency faults, in practice it can detect only the types of races for which we have internal oracles. For each case considered, we analyzed the bug reports to determine what pairs of programs were involved in the races. Finally, we replicated the environments needed to potentially reproduce the races, creating shell scripts and oracle processes.

The pairs of programs we selected are shown in Table I. In the table, the character in parentheses after the application name indicates whether the application is a C program (C) or a shell script (S). We also report other basic characteristics including lines of code (LoC) and bug repository information. Each pair of applications has particular behaviors when they interact with each other, as we describe next.

A pair of UPDATEDB processes from version 4.1.20 [8] can race with each other while performing database updates. This classic race occurs when one of the two UPDATEDB processes, without proper synchronization of file accesses, naively modifies the database while the other is using it.

TABLE I
BASIC CHARACTERISTICS OF OBJECT PROGRAMS

Program Pair				
Program 1	**LoC**	**Program 2**	**LoC**	**Type of Races**
UPDATEDB (S)	14,405	UPDATEDB (S)	14,405	Improper Synchronization
MV (C)	47,023	RM (C)	45,850	Unsynchronized File Accessing
LOCATE (C)	14,643	UPDATEDB (S)	14,405	Improper Synchronization
PXZ (C)	407	STEALER (S)	6	Bypass a restriction or similar
CSPLIT (C)	29,905	MSIGSHOOTER (C)	46	Signal Handler Race Condition
BASH (S)	144,536	BASH (S)	144,536	Improper File Accessing Mangement
RM(FTS) (C)	69,524	SYMLINK (S)	3	TOCTTOU and Improper Link Resolution Before File Access

MV and RM are applications from version 6.9 of the GNU Core Utilities [3]. MV is a basic Unix command used to move files from a source location to a destination location. RM is a basic Unix command that deletes files. Typically, to perform atomic file replacement using a shell script, RM, LN, and MV are used in sequence. However, an implementation change in MV in coreutils-6.9-16.fc8 creates a race in a commonly used atomic rename script. When a race occurs, the link previously created is removed before the file is moved, causing the source file for the MV operation to be missing.

LOCATE [15] is an application from version 4.1.20 of the GNU Find Utilities. This GNU package provides a modular directory and file search system. LOCATE is used to locate all types of files in the system based on a database created by UPDATEDB (also from version 4.1.20 of the GNU Find Utilities). A bug report indicates that these two can race, causing the database file to be prematurely deleted by UPDATEDB.

PXZ or Parallel XZ [25] is a C utility that uses the LZMA compression algorithm to compress files using multiple processing cores [24]. The reported race is that an intermediate file created as part of the compression process can be seen by another process after creation but just prior to applying the correct permission setting. To reproduce this race we created STEALER, a customized shell script that exploits the vulnerability (which involves a race between PXZ and STEALER) by attempting to steal the information PXZ is working on.

CSPLIT [2] is an application from version 5.2.0 of the GNU Core Utilities, and MSIGSHOOTER is a customized application that instigates an Interrupt Signal (`SIGINT`) to terminate a CSPLIT process. A bug report indicates that these can race, leaving temporary files undeleted after CSPLIT ends. MSIGSHOOTER can instigate multiple interrupt signals, but in this case only two instances of `SIGINT` are needed.

BASH (Bourne Again Shell version 3.0) [9] is an sh-compatible shell for Unix operating systems. Like other shells, BASH works as a command manager. It receives command lines from a user through standard input devices such as the keyboard or mouse, sends these commands to the operating system, and interacts with the user by displaying results on an output device such as the monitor. By default, BASH can save a history file as a log for the system or the developer to use. The reported race in this program occurs when two instances of BASH interact, causing the history file to be corrupted.

RM (or remove [4]) is a command from GNU Coreutils 8.4, designed to remove files and directories recursively. Using the option `-r` or `-R`, the RM command can hierarchically delete a file from a given location in the root directory. The race that can occur with this program is classified as a TOCTTOU race. When the race occurs, the directory traversing status will be `NOFOLLOW`, meaning that there are no symbolic links or `symlinks` underneath the root directory and the deletion can proceed. This race allows attackers to create a SYMLINK, causing the deletion of the `symlink` target. We created a shell script, SYMLINK, to create a symbolic link inside an existing directory and act like an attacker in this scenario.

B. Variables and Measures

Independent Variable. We wish to assess the effectiveness and cost of RRF. To assess effectiveness (RQ1), we wished to compare RRF to a baseline technique, and an initial candidate for this was RACEPRO. However, we discovered that there are several limitations of RACEPRO that prevent it from running on four of our seven object program pairs; this rendered it unsuitable for use as a baseline technique. We provide further discussion of the limitations we encountered, along with the data we *were* able to obtain, in Section V-B.

As an alternative baseline technique for use in this study, we compare RRF to a *stress-testing* approach typical of those commonly used in practice to detect races. The stress-testing approach (STA) that we utilize involves randomly launching the pairs of programs that are known to race a large number of times. We do this *with* the RRF observation tool turned on in order to gather execution information that would be needed by engineers to begin to understand the causes of the race, if it occurs. We also use the same oracles in both cases.

To provide a fair comparison of the approaches, for each object program, we first executed RRF on it to completion, measuring the amount of time required. We then executed the STA approach for exactly that amount of time. This lets us assess the relative effectiveness of the approaches when they are given the same amounts of time.

Dependent Variables. To assess effectiveness (RQ1), we measure the number of test runs needed by RRF and STA to detect each *targeted* process-level race, in the cases in which the race was revealed. To assess the cost of RRF (RQ2), we measure the wall clock time required for a complete run of the approach. In this case, there is no baseline approach to compare RRF against.

C. Study Methodology

To automate RRF we minimized the involvement of the user by encoding all information necessary for a run into a configuration file. This information includes the project name, shared resource name, first process number, second process

TABLE II
EMPIRICAL RESULTS

PuD1 (A)	PuD2 (B)	Total Time (Minutes) (C)	RRF			STA		
			Total Runs (D)	Races (E)	% (F)	Total Runs (G)	Races (H)	% (I)
UPDATEDB	UPDATEDB	18	360	110	30.55	938	68	7.24
MV	RM	40	640	60	9.37	4595	0	0
LOCATE	UPDATEDB	13	360	20	5.55	1286	0	0
PXZ	STEALER	20	320	20	6.25	613	0	0
CSPLIT	MSIGSHOOTER	11	160	60	37.50	2502	0	0
BASH	BASH	613	7,840	1308	16.68	56,570	0	0
RM(FTS)	SYMLINK	11	240	7	2.91	2717	0	0

number, location of the source code for the first and second processes, and log files. When RRF begins, the full list of system calls is filtered to exclude system calls from processes not being targeted. Then, a second filter returns just the system calls that use the shared resources. After RRF cleans up the log, it generates mPuDs by pairing up the system calls. By calculating and injecting `sleep` calls into the source code (before or after) that maps to the system calls, each mPuD can represent a set of interleaving patterns. RRF then executes the pair of mPuDs, and passes the output of each through the validation component to determine whether the race has occurred. We repeat the foregoing process ten times for each case, because even when using forced interleavings, non-determinism can still occur.

D. Threats to Validity

The primary threat to external validity in this study involves the object programs utilized. We have studied just seven pairs of racing programs. However, the programs are real and the races are real and non-trivial to reproduce. A second threat involves the representativeness of our oracles. These threats can be addressed only through further studies.

The primary threat to internal validity involves potential errors in the implementations of RRF and the infrastructure used to run RRF and STA. To limit these we extensively validated all of our tool components and scripts.

The primary threat to construct validity relates to the fact that we study effectiveness and cost measures relative to applications of RRF, but do not yet assess whether the approach helps engineers localize and correct process-level races more quickly than current approaches. A second threat involves the dynamic and non-deterministic nature of races. When we inject `sleep` calls into the two PuDs and run the programs, the goal is to exercise a race that can occur around those two `sleep` calls. However, other races can also naturally occur during that run, so the data we use to answer RQ1 must distinguish between races we intend to exercise (*targeted races*) and races that unexpectedly occur outside of the injected `sleep` calls (*incidental races*). We address this threat by analyzing the reported results to classify every detected race into the three types previously introduced in Section III: targeted, incidental, and new. This classification allows us to measure the effectiveness of RRF more precisely.

E. Results

Table II provides data for both of our research questions. Column C indicates the total execution time required by RRF in minutes. Columns D and G show the total numbers of runs for RRF and STA, respectively. Columns E and H show the total numbers of runs in which races were detected for RRF and STA, respectively. Columns F and I show the percentages of runs in which races were detected relative to runs in which they were not, for RRF and STA, respectively.

We turn first to RQ1, considering the comparative effectiveness of RRF and STA. Recall that the two techniques were executed for the same amounts of time (the amount of time required by RRF). In that amount of time, STA exercised between 1.9 (PXZ-STEALER) and 15 times (CSPLIT-MSIGSHOOTER) more runs that RRF. (Because we utilized monitoring on both approaches, the differences in numbers of runs are due to the fact that RRF injects `sleep` calls, whereas STA does not.) Nevertheless, these additional runs did not allow STA to outperform RRF. In fact, STA was unable to reproduce reported races at all on six of the seven application pairs; RRF, in contrast, was able to reproduce the races on all seven. On one pair (UPDATEDB-UPDATEDB), STA was able to reproduce races, but the numbers and percentages of runs in which it did so were smaller than those in which RRF succeeded.

Turning to RQ2, in six of seven cases, RRF required 40 minutes or less to execute. In one case (BASH-BASH), however, it required 613 minutes (over 10 hours). The execution time of RRF is dictated primarily by the number of executions it needs to perform, and this in turn is dictated by the number of pair-wise interleavings that must be targeted. Table III (rightmost column) shows the numbers of interleavings required, and indicates the much larger number needed in the case of BASH-BASH. Still, 40 minutes of machine time is arguably affordable, given the difficulty of revealing races and the fact that STA could not do so in this case.

V. DISCUSSION

As just noted, RRF was able to reproduce reported races in all seven pairs of our object programs. In the first two pairs of programs, for which both RRF and STA both reproduced reported races, RRF reproduced them more times using fewer runs. Larger numbers of race-revealing interleavings may help engineers localize the causes of races more efficiently. On the other programs, RRF reproduced races that STA, run an equivalent amount of time, could not. Overall, these results suggest that *RRF is more effective than stress testing for reproducing process-level races.*

TABLE III
OPTIMIZATION OF NUMBERS OF SYSTEM CALLS

PuD1 (A)	PuD2 (B)	System Calls Using Share Resources			Pair-wise Inter leaving (F)
		Before Opt. (C)	After Opt. PuD1 (D)	After Opt. PuD2 (E)	
UPDATEDB	UPDATEDB	5,232	3	3	36
MV	RM	391	8	2	64
LOCATE	UPDATEDB	2,818	3	3	36
PXZ	STEALER	4,975	5	1	32
CSPLIT	MSIGSHOOTER	462	4	1	16
BASH	BASH	8,518	6	8	784
RM(FTS)	SYMLINK	1,343	3	1	24

TABLE IV
INTERLEAVINGS THAT CAN CAUSE REPORTED RACES

PuD1	PuD2	# of Potential Sources of Races	# of Sources That Failed
UPDATEDB	UPDATEDB	9	9
MV	RM	16	5
LOCATE	UPDATEDB	9	2
PXZ	STEALER	8	7
CSPLIT	MSIGSHOOTER	4	2
BASH	BASH	196	79
RM(FTS)	SYMLINK	6	1

It is also worth noting that RRF possesses one characteristic that STA does not: it provides additional information to test engineers that STA cannot, in the form of execution window amplification. Typically, it is challenging to reproduce races that occur only in very small execution windows. This is why an enormous number of runs can be needed to expose an instance of an elusive race. By providing a systematic way to amplify execution windows, RRF should have a better chance of reproducing such elusive races than STA.

A. Additional Analysis

Each PuD that we studied invokes a large number of system-level events during its execution. Many of these events, however, are not relevant to reported races. RRF culls out these extra events, and to see to what extent it succeeds, we gathered data on the numbers of system calls using shared resources before and after this culling. Table III presents the results, and shows that culling reduces the number of system calls to be considered by several orders of magnitude (Columns D and E). This also reduces the number of interleavings that must be exercised as shown in Column F.

Recall that RRF calculates the pair-wise interleavings that can access shared resources in each application. For example, CSPLIT has four possible locations and MSIGSHOOTER has one. As such, in this case RRF produces four pairs of interleavings. We can also view each pair of interleavings as a *potential source of races*. The third column in Table IV reports the numbers of potential sources of races observed in each pair of programs, and the fourth column reports the number of these potential sources that produced the reported race. On all seven of our program pairs, multiple potential sources of races were identified. On all program pairs except (RM(FTS)-SYMLINK), more than one of these potential sources did in fact produce the reported race. In practice, if the engineer debugging the applications does not monitor all sources, it is possible that a patch may not repair all defective sources. By using RRF, developers can locate additional causes of races.

TABLE V
ADDITIONAL FAULT TYPES FOUND

PuD1	PuD2	Additional Fault Types Found	Number of Faults Found
UPDATEDB	UPDATEDB	Only one UPDATEDB can really update.	204
LOCATE	UPDATEDB	UPDATEDB delete the database file while LOCATE is using that database.	120

We also applied different oracles to assess whether other types of concurrency faults can occur on the same shared resource in an application. To do this, we chose oracles that operate on similar resources (e.g., files, sockets) and applied those oracles to validate the event traces. Table V reports our findings. In addition to the reported races, two pairs of programs also contained other types of faults involving the shared resources that we monitor. First, UPDATEDB-UPDATEDB can suffer from a second type of race that leads to a nullified update; we found 204 additional instances of this type of fault. Second, we also found 120 instances of premature file removal due to races in LOCATE-UPDATEDB. Validating traces using additional oracles incurred only modest overhead. For example, using one additional oracle to validate LOCATE and UPDATEDB, cost only 18 more seconds per run.

The quality of a bug report can affect the effectiveness of RRF. If a report is incomplete, RRF must conservatively identify more relevant events; this can result in increased debugging time. Vaguely described symptoms can also lead to imprecise fault localization by requiring RRF to consider additional events and resources. Because RRF is dynamic, the quality of test suites can also affect its ability to localize races; inadequate test suites can lead to fewer events that can be identified as relevant.

B. RRF versus RacePro

According to Laaden et al. [13], RACEPRO is capable of detecting the types of races that occur in our object programs, as well as additional race types. To do this, first, RACEPRO identifies all potential locations that may lead to a race that needs to be reproduced. Second, it uses replay to cause races to occur. Based on this information, we also attempted to compare RRF to RACEPRO, but we were able to run it on only three of seven pairs of programs.

One limitation of RACEPRO involves its ability to perform resource accessing and anchoring of `sleep` calls needed to exercise races. This ultimately results in excessive replay divergence. We also discovered several faults in RACEPRO that have not been addressed, perhaps because RACEPRO has not been actively maintained since 2012. These faults prevented several of our programs from running [31], [32]. Even for pairs of programs that we were able to run, we needed to carefully rewrite them first. For example, we needed to change the programs so that they created processes in certain orders using specific syntaxes.

For the three object program pairs that we were able to run RACEPRO on (PXZ-STEALER, BASH-BASH, and RM-SYMLINK), RACEPRO was able to identify some locations that

TABLE VI
NUMBER OF TARGETED RACES AND INCIDENTAL RACES

PuD1	PuD2	Type of Races	
		Targeted	Incidental
UPDATEDB	UPDATEDB	85	25
MV	RM	40	0
LOCATE	UPDATEDB	20	0
PXZ	STEALER	20	0
CSPLIT	MSIGSHOOTER	60	0
BASH	BASH	562	746
RM(FTS)	SYMLINK	7	0

can potentially cause the desired races (in BASH-BASH and RM-SYMLINK). However RACEPRO was not able to reproduce any of these races. We found that 33 of 45 replay attempts resulted in divergences. There were 10 attempts that did not result in divergences but they also failed to reproduce the races. There were two attempts that also resulted in deadlocks by the replay mechanism. For PXZ-STEALER, we further investigated why RACEPRO could not detect any potential races. First, RACEPRO could only intermittently run the program pair. In the instances in which they ran, RACEPRO did not detect any violations. We found that in these scenarios, atomicity violations occurred where STEALER was able to copy a compressed file to another location. This is due to the small window in which a file permission change occurs, allowing STEALER, which was executed under a different user process, to access the file. While RACEPRO is reportedly able to detect atomicity violations [14], it did not detect this one.

C. Incidental Races

In some programs, races are particularly difficult to reproduce and in these cases, amplification is necessary to exercise them. In other cases, races occur naturally even without amplification, and such other races, in addition to the ones we intend to exercise, can also occur in a program execution. We report the results of our investigation of incidental races during our experiments in Table VI.

As the table shows, two of our pairs of programs exhibit race behavior more frequently than others. In these cases, the races we observed included both targeted and incidental races and additional analysis of the data was required to distinguish them. All of the cases show, however, that RRF was effective for reproducing targeted races.

D. Detecting More Races

We also discovered that in some cases, the repair specified in a bug report [15] does not completely fix races. In our application pair (LOCATE, UPDATEDB), the repair described in the report fixes the UPDATEDB script but races can still occur in LOCATE. Using RRF we were still able to reproduce races after the suggested repair has been applied.

VI. RELATED WORK

There has been a great deal of research on debugging concurrent programs at the thread level [5], [6], [11], [26], [37]. These techniques, however, focus on multi-threaded applications within single processes, and have rarely been adapted to deal with concurrency faults occurring across processes.

There are several dynamic techniques that replay or reproduce concurrency faults by permuting thread interleavings. We have already described active testing techniques including CALFUZZER [27] and UPC-THRILLE [20]. CHESS [18] captures information on nondeterminism during program execution and allows the scheduler to later reproduce a faulty execution by replaying nondeterministic choices. These techniques, however, focus on thread-level concurrency while ignoring concurrency faults at the process level.

There has been much research on fault localization for concurrent programs [21], [22], [30], [33], [38]. For example, Park et al. [21] monitor memory-access patterns among threads and detect data-access patterns associated with a program's pass/fail results. They then use statistical debugging to report data access patterns with suspiciousness scores. Wang et al. [34] identify of shared memory access pairs that behave distinctively in failed and successful runs, and pinpoint root causes using different test procedures. Again, these techniques focus on thread-level concurrency faults. In addition, they are applied only during pre-deployment.

There have been a few approaches created for detecting races at the process level [14], [23], [29], [35]. We have already discussed RACEPRO [14]. In addition, detecting TOCTOU races [29] has been a topic in the security community. TOCTOU races typically occur when the permission check and use of a file in one process is not atomic, allowing a malicious process to slip in and attack the file. Our prior work on SIMRACER leverages active testing [28] to reduce false positives during dynamic race detection. However, these techniques all focus on testing and detection and do not consider the debugging of process-level concurrency faults.

VII. CONCLUSION

Debugging process-level races is challenging because the sources of errors can occur in system-level events, and reproducing them requires favorable process scheduling. To empower engineers with the tools needed to effectively debug process-level races, we have introduced RRF, a framework that provides the observability and controllability necessary to reproduce races. Our evaluation shows that RRF is effective at reproducing challenging races. Given seven previously reported, real-world races, RRF was able to reproduce all of them while stress testing was able to reproduce only two. RRF also reveals other sources within a program that can produce reported races, allowing engineers to formulate repairs that address all sources. The framework also allows engineers to explore whether other types of concurrency faults can occur on these sources at modest cost.

ACKNOWLEDGMENT

This material is based on research sponsored by DARPA and Maryland Procurement Office under agreement numbers FA8750-14-2-0053 and H98230-14-C-0140, respectively. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies or the U.S. Government.

REFERENCES

[1] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional Detection of Data Races. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 255–268, 2010.

[2] Bug-Coreutils. Bug-Coreutils: Signal-Handling Cleanup for csplit, install, ls, nohup, sort, tee. https://lists.gnu.org/archive/html/bug-coreutils/2004-04/msg00126.html. Accessed: 2015-10-20.

[3] Red Hat Bugzilla. Bug 438076 - mv Creates Race Condition by Calling unlink() Before rename(). https://bugzilla.redhat.com/show_bug.cgi?id=438076. Accessed: 2015-10-20.

[4] Red Hat Bugzilla. CVE-2015-1865 Coreutils: "Time of Check to Time of Use" Race Condition fts.c. https://bugzilla.redhat.com/show_bug.cgi?id=1211300. Accessed: 2015-10-20.

[5] Jong-Deok Choi and Andreas Zeller. Isolating Failure-inducing Thread Schedules. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 210–220, 2002.

[6] Holger Cleve and Andreas Zeller. Locating Causes of Program Failures. In *Proceedings of the International Conference on Software Engineering*, pages 342–351, 2005.

[7] Greg Cooper. DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X, and Free BSD by Brendan Gregg and Jim Mauro. *Software Engineering Notes*, 37(1):34–34, 2012.

[8] Debian Bug Report Findutils. Debian Bug Report Logs - 67782: find has Race Condition Which Sometimes Causes Spurious Error Messages. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=67782. Accessed: 2015-10-20.

[9] Ubuntu in Launchpad. Bug 10809 Race Condition when Writing 'bash' History. https://bugs.launchpad.net/ubuntu/+source/bash/+bug/10809. Accessed: 2015-10-20.

[10] Naftaly S. (Intel). Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool. Accessed: 2016-08-15.

[11] Nicholas Jalbert and Koushik Sen. A Trace Simplification Technique for Effective Debugging of Concurrent Programs. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 57–66, 2010.

[12] Oren Laadan, Chia-Che Tsai, Nicolas Viennot, Chris Blinn, Peter Senyao Du, Junfeng Yang, and Jason Nieh. Finding Concurrency Errors in Sequential Code: OS-level, In-vivo Model Checking of Process Races. In *Proceedings of the Conference on Hot Topics in Operating Systems*, pages 20–30, 2011.

[13] Oren Laadan, Nicolas Viennot, and Jason Nieh. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, pages 155–166, 2010.

[14] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive Detection of Process Races in Deployed Systems. In *Proceedings of the Symposium on Operating Systems Principles*, pages 353–367, 2011.

[15] Debian Bug Report Log. 461585 locate: Strange Interpretation of Atomic. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=461585. Accessed: 2015-10-20.

[16] Robert Love. Kernel Korner: Intro to Inotify. *Linux J.*, 2005(139):8–, November 2005.

[17] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[18] Madanlal Musuvathi and Shaz Qadeer. Iterative Context Bounding for Systematic Testing of Multithreaded Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 446–455, 2007.

[19] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 267–280, 2008.

[20] Chang-Seo Park, Koushik Sen, Paul Hargrove, and Costin Iancu. Efficient Data Race Detection for Distributed Memory Parallel Programs. In *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 51:1–51:12, 2011.

[21] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the International Conference on Software Engineering*, pages 245–254, 2010.

[22] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. A Unified Approach for Localizing Non-deadlock Concurrency Bugs. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 51–60, 2012.

[23] Mathias Payer and Thomas R. Gross. Protecting Applications Against TOCTTOU Races by User-space Caching of File Metadata. In *Proceedings of the Conference on Virtual Execution Environments*, pages 215–226, 2012.

[24] PXZ. PXZ: Parallel LZMA Compressor Using Liblzma. https://jnovy.fedorapeople.org/pxz/. Accessed: 2015-10-20.

[25] Dabian Bug Report PXZ. 775306 pxz: CVE-2015-1200: Race Condition in Setting Permissions. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=775306. Accessed: 2015-10-20.

[26] Jeremias Rossler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. Reconstructing Core Dumps. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 114–123, 2013.

[27] Koushik Sen. Effective Random Testing of Concurrent Programs. In *Proceedings of the International Conference on Automated Software Engineering*, pages 323–332, 2007.

[28] Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 11–21, 2008.

[29] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably Solving File TOCTTOU Races with Hardness Amplification. In *Proceedings of the Conference on File Storage Technologies*, pages 189–206, 2008.

[30] Rachel Tzoref, Shmuel Ur, and Elad Yom-Tov. Instrumenting Where It Hurts: An Automatic Concurrent Debugging Technique. In *Proceedings of the International Conference on Software Engineering*, pages 27–38, 2007.

[31] Columbia University. linux-2.6-racepro: Recording Stuck on File Operations. https://github.com/columbia/linux-2.6-racepro/issues/3. Accessed: 2016-01-07.

[32] Columbia University. racepro: signal.handled not in the Graph. https://github.com/columbia/racepro/issues/6. Accessed: 2016-01-07.

[33] Wenwen Wang, Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Xipeng Shen, Xiang Yuan, Jianjun Li, Xiaobing Feng, and Yong Guan. Localization of Concurrency Bugs Using Shared Memory Access Pairs. In *Proceedings of the International Conference on Automated Software Engineering*, pages 611–622, 2014.

[34] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. DrDebug: Deterministic Replay Based Cyclic Debugging with Dynamic Slicing. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 98:98–98:108, 2014.

[35] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. SimRacer: An Automated Framework to Support Testing for Process-level Races. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 167–177, 2013.

[36] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. An Empirical Comparison of the Fault-Detection Capabilities of Internal Oracles. In *Proceedings of the Symposium on Software Reliability Engineering*, pages 11–20, November 2013.

[37] Cristian Zamfir and George Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the European Conference on Computer Systems*, pages 321–334, 2010.

[38] Wei Zhang, Chong Sun, Junghee Lim, Shan Lu, and Thomas Reps. ConMem: Detecting Crash-Triggering Concurrency Bugs Through an Effect-Oriented Approach. *ACM Transactions on Software Engineering and Methodology*, 22(2):10:1–10:33, 2013.