Received: 6 April 2016 | Revised: 13 March 2017 | Accepted: 14 March 2017
DOI: 10.1002/stvr.1634

WILEY

RESEARCH ARTICLE

An automated framework to support testing for process-level race conditions

Tingting Yu[1] | **Witty Srisa-an**[2] | **Gregg Rothermel**[2]

[1]Department of Computer Science, University of Kentucky, Lexington, KY, USA
[2]Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, NE, USA

Correspondence
Tingting Yu, Department of Computer Science, University of Kentucky, Lexington, KY, USA.
Email: tyu@cs.uky.edu

Funding information
National Science Foundation, Grant/Award Number: CCF-1464032; Air Force Office of Scientific Research, Grant/Award Number: FA9550-10-1-0406; Army Research Office, Grant/Award Number: W911NF-13-1-0154; DARPA, Grant/Award Number: FA8750-14-2-0053; Maryland Procurement Office, Grant/Award Number: H98230-14-C-0140

Summary

Race conditions are difficult to detect because they usually occur only under specific execution interleavings. Numerous program analysis and testing techniques have been proposed to detect race conditions between threads on single applications. However, most of these techniques neglect races that occur at the process level due to complex system event interactions. This article presents a framework, SIMEXPLORER, that allows engineers to effectively test for process-level race conditions. SIMEXPLORER first uses dynamic analysis techniques to observe system execution, identify program locations of interest, and report faults related to oracles. Next, it uses virtualization to achieve the fine-grained controllability needed to exercise event interleavings that are likely to expose races. We evaluated the effectiveness of SIMEXPLORER on 24 real-world applications containing both known and unknown process-level race conditions. Our results show that SIMEXPLORER is effective at detecting these race conditions, while incurring an overhead that is acceptable given its effectiveness improvements.

KEYWORDS
kernel, processes, race conditions, software testing, virtual platforms

1 | INTRODUCTION

Modern software systems are prone to various classes of concurrency faults due to their reliance on concurrent programming and various uses of interrupts, system calls, and polling to fully use processing power and sense environmental factors. These characteristics render these systems highly complex and can result in various concurrency faults in deployment environments that can be difficult to identify, isolate, and correct. Race conditions are one such class of faults. Race conditions are difficult to detect primarily because they are sensitive to execution interleavings. Unless a specific interleaving is exercised that can cause races to occur and cause their effects to be visible, they will go undetected and remain in the system.

In practice, testing is the most commonly used approach for detecting race conditions in modern software systems. Thus, over the past decade, much research has been conducted on techniques that facilitate this (eg [1–5]). Most existing techniques, however, focus on multithreaded applications within single processes and have not been adapted to handle *process-level race conditions* such as races caused by improper shared resource accesses among multiple processes, software signals, and hardware interrupts.

Laadan et al. [6] examined 150 bugs in several Linux distributions, which they located by applying a search query of "race" to each distribution's website. Of these 150 bugs, they found that 109 (73%) resulted in what we are here calling "process-level races." While the authors note that their results likely under-represented the races present in multithreaded programs, their data do support the claim that these races are numerous. Furthermore, compared to thread-level races that typically corrupt volatile application memory, process-level races are arguably more dangerous because they often corrupt persistent and system resources that in ways that can lead to system-wide consequences, including security vulnerabilities and complete operational failures.

A recent mishap at NASDAQ during Facebook's initial public offering illustrates the effects that process-level races can have [7]. In spite of "unprecedented" testing efforts used on the auction system used to conduct the offering, races occurred between the auction process and the calculation process. These races corrupted data, causing the system to go into an infinite loop; this in turn produced a backlog of unfulfilled orders that resulted in over $40 million dollars in damages.

As a further complication, process-level races often involve shared resources accessed through system calls. Thus, testing for process-level races requires the effects of read/write and synchronization operations involving system calls to be accounted for. However,

 Copyright © 2017 John Wiley & Sons, Ltd.

system calls operating on shared resources are typically treated as black boxes by engineers who use them to develop applications. Furthermore, as Laadan et al. [6] report, most race detection algorithms are designed to work only with memory read/write and well-defined synchronization operations at the thread level.

Many state-of-the-art systems for detecting thread-level races [1,3,4,8–10] can dynamically *observe* events of interest and then analyze information collected from those observations to determine whether races occur. Despite the ability of dynamic approaches to achieve greater precision than static analysis approaches (eg [11]) in identifying races, they are imprecise and can result in both false positives and false negatives. For example, race detection based on a happens-before algorithm [1,6] can identify only the sources of races that appear during an execution; other sources that exist in the application but are not exercised cannot be identified (incompleteness). On the other hand, such approaches may also fail to recognize user-defined synchronization primitives and report the existence of races that are not actual races (false positives).*

Many techniques have been proposed to improve dynamic analysis techniques by providing both *observability* (the ability to observe events) and *controllability* (the ability to permute execution interleavings during test runs) [12–15]. These techniques attempt to abstract away scheduling nondeterminism in concurrent programs to achieve greater execution control. For example, active testing [13] has been used to first identify potential races and then to control the underlying scheduler by inserting delays at context switch points. These techniques have been shown to be effective at detecting concurrency faults with low rates of (or no) false positives.

While 1 way to test for process-level races could be to attempt to adapt one of the foregoing thread-level testing techniques, these techniques cannot be directly adapted to provide the same degrees of observability and controllability at the system level that they achieve at the thread level. Where *observability* is concerned, existing techniques tend to focus only on application execution and do not monitor events occurring in lower-level software components such as device drivers and Operating System (OS) modules. As such, they can be ineffective for revealing subtle faults that can appear in hardware, device drivers, and kernels. In addition, these approaches can obscure lower-level information because they rely on instrumentation that can perturb lower-level system states (eg, cache, bus, and register usage).

Where *controllability* is concerned, sources of unpredictability in program execution due to scheduling and interrupt events must be controllable during testing. As an example, to reveal an interrupt-related fault, engineers must be able to force interrupts to occur at a particular location. Unfortunately, existing techniques that rely on binary/bytecode instrumentation to insert yield points at the application level (eg, CHESS [12]) have rarely been adapted to detect concurrency faults that occur due to shared hardware resources or across different applications in modern software systems. To be effective, process-level race detection techniques must be able to control the kernel's scheduler. Furthermore, these techniques must have access to various system states to ensure process and event schedulability. Achieving even partial system control often requires custom kernel modifications, limiting the practicality of the techniques. For example, RACEPRO can be used only in a custom Linux kernel that has been modified to provide event recording and replay capabilities [6]. It also cannot precisely control process scheduling, making it ineffective as a platform for testing for various types of process-level races. (We provide a detailed analysis of RACEPRO's limitations in Section 2.3).

In this article, we present a testing framework for modern software systems, SIMEXPLORER, that allows engineers to effectively test for process-level races by providing deterministic execution in multiprogramming systems.† Our framework uses the Simics Virtualization Platform [17] to observe system execution, to deterministically control occurrences of events, and to allow testing to target specific types of races. The framework allows engineers to focus only on applications under test without worrying about other events or applications in the system. Side effects from execution of applications not under test can be easily determined by querying current system states (eg, using the Simics Process Tracker or examining stack frames). By using Simics, our framework also operates at the binary level so it is capable of detecting races between applications written in different languages. It is also generalizable, as it can be used on any system that can run on Simics. We instantiate the SIMEXPLORER framework to test for races at different system levels, including the following: (1) improper shared resource access among multiple processes; (2) interactions between applications and hardware interrupts; and (3) incorrect arrivals of software signals.

To evaluate our SIMEXPLORER framework, we conducted an empirical study using 24 Unix programs that contain actual process-level races. Our goal was to answer 2 research questions: (1) How effective is SIMEXPLORER at detecting process-level races, and (2) how effective is SIMEXPLORER at detecting observable faults related to those races. The results of our study show that our approach is effective at revealing races, and far more effective than commonly used stress testing approaches at revealing faults related to those races. Further, while our approach takes longer than stress testing to execute test runs due to virtualization overhead, it can reveal more races while using far fewer test runs.

The remainder of this article is structured as follows. Section 2 further motivates the work by providing several examples of real-world process-level races. Section 3 presents our SIMEXPLORER framework and the algorithms that it relies on. Section 4 provides details on our implementation of the framework. Section 5 describes our empirical study design and presents its results. Section 6 provides additional discussion of the results and their implications. Section 7 reviews related work, and Section 8 concludes.

2 | MOTIVATION

The problem we consider in this work relates to race conditions, which generally involve 1 or more processes accessing a shared resource (such a file or variable), where this multiple access has not been properly controlled [18]. We say that a *process-level race condition* occurs

*We say that a potential racing pair reported by a race detector is a false positive if the order of the 2 events in the pair cannot actually be switched in any other executions.

†This article is a revised and expanded version of a previously published conference paper [16]; the differences between that paper and this article are described in detail in Section 7.1.

when (1) two system events (eg, processes, software signals, and hardware interrupts) access a shared resource, and (2) they could access the shared resource in an order different from the intended order. This definition of a race condition is broader than the standard definition of a data race (ie, a data race occurs when 2 threads access the same shared memory location at the same time). Thus, as we define it, a race condition is a necessary but not sufficient condition for a typical data race.

2.1 | Real-world examples of system-level races

We provide 7 real-world examples to illustrate how process-level races can occur.

Corrupted file. Figure 1 provides an instance of a process-level race modeled after a pattern of commonly occurring races in Linux systems [19–21]. This race happens between a parent and its child process. The race occurs on a file, causing the recorded information to be incorrect. The parent process *P*1 creates a file "file1" and writes to it (lines 1-3). Next, *P*1 spawns a child process *P*2 using `fork`. While *P*2 is sleeping (line 6), *P*1 renames "file1" to "file2" and waits for *P*2 to exit (lines 16-17). After *P*2 wakes up, it writes a string into "file2" (lines 7-8). Next it checks for the existence of "file1" and writes a message into "file2" (lines 9-10). The expected output is a "file2" file containing the string "hello@kitty". However, races can occur if *P*1 is suspended while *P*2 is sleeping. In such a scenario, *P*2 creates a new file "file2", and writes "kitty" into it (lines 7-8). Next, *P*1 overwrites the contents of "file2" by renaming "file1" to "file2" (line 16). As a result, the output information in "file2" is just "hello".

Files not deleted. Figure 2 provides an example of a race between a regular process and a software signal handler. This race occurs in a program, CSPLIT, considered in our empirical study. Under normal execution, the process calls `create_output_file` to create a set of files (line 3) and increments the counter (line 5). If a signal occurs, the process iterates over the files that have been created (lines 8-9) and deletes them (line 10). However, if a signal occurs just before the counter is incremented (line 4), the files will not be deleted by the handler.

Blocked processes. Figure 3 provides an example of a deadlock due to a race condition in Apache 2.0.49 [22] that can be detected by our technique. It involves the CGI process and the HTTPD process that handles the CGI request. Apache redirects CGI's stderr and stdout outputs into 2 pipes. Under normal execution, the HTTPD process reads data after the CGI process writes stderr and stdout into the 2 pipes. However, a deadlock occurs when line 2 happens before line 8. The HTTPD process is blocked, waiting for the CGI process to write data to pipe2, and the CGI process is blocked, waiting for the HTTPD process to read data from pipe1.

```
1  fd1 = fopen("file1", "w");
2  fwrite("hello", 1, 256, fd1);
3  fclose(fd1);
4  childpid = fork();
5  if(childpid == 0) { /*child process*/
6        sleep(1);
7        fd2 = fopen("file2", "a+");
8        fwrite("@", 1, 256, fd2);
9        if(stat("file1") != 0){
10           fwrite("Kitty\n",1,256,fd2);
11       }
12       fclose(fd2);
13       exit(0);
14 }
15 else { /*parent process*/
16       rename("file1", "file2");
17       wait(&status);
18       exit(0);
19 }
```

FIGURE 1 Two processes race on a file

Permission denied. Figure 4 provides an example of a race condition in an Android application [23]. The `init` process watches for a new device (line 1) and sets the permission on it (line 3). At the same time, if the server process tries to open this device, this results in an error because the server does not yet have access permission for this device.

Nonsynchronized data transmission. Figure 5 provides an example of a process-level race in a UART device driver that runs in kernel mode [24]. This race occurs on multicore systems and can be detected by our approach. When the UART port is buggy (the interrupt does not work initially), the driver calls a backup timer periodically. The backup timer on core 1 first disables interrupts by setting the interrupt enable register (IER) (line 1) and later checks the IER to see if it has been enabled. If it has, the timer uses an interrupt handler to transmit data (line 4); otherwise, it uses polling to slowly transmit the data (line 6). If *start_tx* is called on core 2, interrupts are enabled (line 9) and races can occur on the IER (line 1 races with line 9), causing both the backup timer and transmission routines to simultaneously transmit data through the same serial port.

Incorrect interrupt enabling. Figure 6 provides an example of a race condition that occurs in a UART driver program from the early releases of the 2.6 Linux kernel. This fault remained in the source code for 3 years before being fixed. Under normal operating conditions, the transmission is enabled (`serial8250_start_tx`), and the interrupt handler `serial8250_interrupt` is responsible for transmitting data. When the UART port is buggy (the interrupt does not work initially) (line 4), workaround code based on polling instead of interrupts is used (lines 17-20). Unfortunately, the enabled interrupts (lines 7-8) are not disabled in the workaround code region (lines 17-20), so by the time that code is executed, both the interrupt handler (line 25, invoked by line 10) and the workaround code (line 19) could be transmitting or receiving data through the same serial port at the same time by calling the routine `transmit_chars`, and simultaneously read/write the shared variable `xmit→tail` (line 29).

Misuse of longjmp. Figure 7 provides an example of a race stemming from the use of `longjmp` between a regular process and a software signal handler. Invoking `longjmp` from a signal handler causes the code after the `setjmp` to be called under the same conditions as the signal handler itself. The intent is to execute code in the loop in the `main` routine and log some data (lines 9-13). Upon receiving a SIGINT, the program transfers out of the loop, logs the error, and terminates. However, if a signal occurs just before the second `if` statement in `log_message` (right before line 32), this results in `longjmp` transferring control back to `main`, where `log_message` at line 16 is called. As such, the first `if` statement (line 26) would not be executed this time (because buf is not set to NULL as a result of the interrupt), and the program would write to the invalid memory location referenced by `buf0`.

Corrupted memory. Figure 8 provides an example of a concurrency fault that occurs between signal handlers (CWE-831). This can be best seen as an example of a problem that can be exploited by an attacker,

```
1  int files_created = 0; /*global variable*/
2  create_output_file() {
3        output_stream = fopen(output_filename, "w");
4        /*signaled*/
5        files_created++;
6  }
7  signal() {
8        for (i = 0; i < files_created; i++) {
9              name = make_filename(i);
10             if (unlink(name))
11                   error(0, errno, "%s", name);
12       }
13 }
```

FIGURE 2 A process-level race due to a software signal

```
HTTPD{                                  CGI{
1. ...                                  6. write(pipe1,stderr);
2. read(pipe2, stdout);                 7. ...
3. ...                                  8. write(pipe2,stdout);
4. read(pipe1, stderr);                 9. ...
5. }                                    10. }
```

FIGURE 3 A deadlock in the Apache Web server

```
init{                                   server{
1. open(dev);                           5. ...
2. ...                                  6. open(dev);
3. set_permission(dev);                 7. ...
4. }                                    8. }
```

FIGURE 4 A process-level race in an Android application

```
backup_timer{                           start_tx{
1.write_reg(IER);/*disable interrupts*/ 8. ...
2. ...                                  9.write_reg(IER);/*enable interrupts*/
3.if(isEnabled(IER)==TRUE);             10./*transmit by ISR*/
4./*transmit by ISR*/                   11. ...
5.else                                  12. }
6./*transmit by polling*/
7.}
```

FIGURE 5 A process-level race in a UART device driver. IER, interrupt enable register

although it might also occur due simply to sequences of events occurring on their own in the system. The handler function uses global state (logMessage), and it can be called by both the SIGHUP and SIGTERM signals. The program begins its "normal" functionality, which is simplified as `sleep()`, but could be any functionality that consumes some time. An attacker sends SIGHUP, which invokes the handler. This in turns calls `syslog`; `syslog()` calls `malloc()`, which is nonreentrant, and `malloc()` begins to modify metadata to manage the heap. Next, the attacker sends SIGTERM, which interrupts the SIGHUP handler, but `syslog`'s malloc call is still executing and has not finished modifying its metadata. The SIGTERM handler is invoked, which records the log message using `syslog()`, then frees the `logMessage` variable. At this point, the state of the heap is uncertain because `malloc` is still modifying the metadata for the heap; the metadata might be in an inconsistent state. The SIGTERM-handler call to free() is assuming that the metadata is inconsistent, possibly causing it to write data to the wrong location while managing the heap. The result is memory corruption, which could lead to a crash or even code execution, depending on the circumstances under which the code is running.

2.2 | Comparing system-level and thread-level race detection techniques

As noted in Section 1, thread-level race detection techniques [1,3–5,8–10,12–15,25–27] cannot be directly adopted to address process-level race conditions. There are several reasons for this.

First, existing thread-level techniques assume the existence of specific, well-defined read and write operations to shared memory (eg, load and store) and thread synchronization primitives (eg, lock/unlock

```
1  static int serial8250_startup(...){
2     up = (struct uart_8250_port *)port;
3     ...
4     if (!(up->bugs & UART_BUG_TXEN)) {
5           up->bugs |= UART_BUG_TXEN;
6     }
7     ...
8     /* Finally, enable interrupts. */
9     up->ier = UART_IER_RLSI | UART_IER_RDI;
10    serial_outp(up, UART_IER, up->ier);
11    ...
12 }
13
14 static void serial8250_start_tx(...){
15    serial_out(up, UART_IER, up->ier);
16    ...
17    if (up->bugs & UART_BUG_TXEN) {
18          ...
19          transmit_chars(up);
20    }
21 }
22
23 static irqreturn_t serial8250_interrupt(...){
24    ...
25    transmit_chars(up);
26    ...
27 }
28
29 static void transmit_chars(...){
30    struct circ_buf *xmit = &up->port.info->xmit;
31    ...
32    xmit->tail = (xmit->tail + 1)
33                      & (UART_XMIT_SIZE - 1);
34    ...
35 }
```

FIGURE 6 Faulty code that causes race conditions in the Linux UART driver

```
1  int main(void) {
2     if (signal(SIGINT, handler) == SIG_ERR) {
3        /* Handle error */
4        ...
5     }
6
7 /* info1 and info2 are set by user input here */
8     if (setjmp(env) == 0) {
9        while (1) {
10       ...
11       log_message(info1, info2);
12       ...
13      }
14 }
15    else {
16       log_message(info1, info2);
17     }
18  }
19
20 void handler(int signum) {
21    longjmp(env, 1);
22 }
23
24 void log_message(char *info1, char *info2) {
25    ...
26    if (buf == NULL) {
27       buf = buf0;
28       bufsize = sizeof(buf0);
29    }
30
31    /* write a message into buf */
32    if (buf == buf0) {
33       buf = NULL;
34    }
35 }
```

FIGURE 7 A fault from inside a signal handler

```
1  void handler (int sigNum) {
2     syslog(LOG_NOTICE, "%s\n", logMessage);
3     free(logMessage);
4     sleep(10);
5     exit(0);
6  }
7
8  int main (int argc, char* argv[]) {
9     logMessage = strdup(argv[1]);
10    /* Register signal handlers. */
11    signal(SIGHUP, handler);
12    signal(SIGTERM, handler);
13    sleep(10);
14 }
```

FIGURE 8 Signal handler function associated with multiple signals

and wait/notify). As such, these techniques have been designed to observe specific operations that access and protect shared memory. On the classes of systems we are targeting, however, the read and write effects on shared resources, as well as the process synchronization primitives, are very different from those used in multithreaded programs. For instance, in Figure 1, the `stat` system call has a write effect on the shared resource "file1". Furthermore, some system calls also have both read and write effects. For example, when `execve` is called to execute a binary executable, in addition to reading the inode of the binary, it also conceptually writes to the kernel resource `/proc`. Thread-level techniques that monitor load and store operations have not been modeled to identify more complex read-write operations. Also, `wait-exit` system calls synchronize access events between parent and child processes. As such, existing techniques that observe thread-level synchronization primitives cannot be applied. In general, all read/write operations and synchronization primitives must be remodeled to detect race conditions at the process level.

As a second bar to adaptation, existing thread-level race detection techniques cannot handle system-wide shared resources that are beyond shared memory accesses within single applications. For example, files and kernel objects (eg, `/proc`) cannot be monitored by these techniques.

As a third bar to adaptation, thread-level techniques that rely on binary/bytecode instrumentation to control memory access ordering between threads cannot be used to control the execution order of system calls across different processes. In contrast, process-level race detection techniques must be able to control the kernel's scheduler.

Generalizing from the foregoing, existing thread-level race detection techniques cannot handle process-level race conditions. Therefore, new techniques are needed.

2.3 | Limitations of RACEPRO: a state-of-the-art approach

RACEPRO [6] is a state-of-the-art race detection technique that can detect certain types of races between application processes. However, there are several limitations to RACEPRO that can affect its effectiveness. First, to detect races, RACEPRO leverages the vector-clocks algorithm that has been widely used in dynamic techniques for thread-level race detection [1,4,28]. As such, RACEPRO can detect a race only if it occurs during a test execution, and thus it may miss other potential races; this is one of the fundamental limitations of the vector-clock algorithm.

Second, to handle false alarms generated by imprecise race detectors, replay techniques have been used to identify false positives [6,29,30]. ThreadSanitizer [29], for example, inserts sleeps after racing instructions; a real thread-level race is reported if the 2 sleeps happen simultaneously. RACEPRO [6], on the other hand, replays the original execution as well as its variations (reordering system calls) to validate whether a detected race is harmful by observing the output. However, some races that are classified as benign through this method may not always be benign; that is, different inputs may be able to show that such races are harmful. As such, RACEPRO can erroneously classify a race as benign if there are no detectable failures in the output.

Third, RACEPRO cannot detect races that occur due to asynchronized hardware accesses by monitoring hardware states, or races that occur specifically on multicore systems.

Fourth, RACEPRO is implemented as Linux kernel components with some non-trivial kernel modifications. As such, it works only for Linux-based applications. The requirement for kernel modifications creates severe challenges for using RACEPRO in any evaluation of real-world problems, because kernel expertise and a significant amount of implementation effort are needed.

Finally, the biggest drawback of RACEPRO is replay divergence failure (ie, a mismatch between replayed system calls and actual system calls). Replay divergence failure can occur when the actual system states do not allow the system to replay or switch system calls.

2.4 | An evaluation of replay divergence in RACEPRO

Given the foregoing drawbacks associated with RACEPRO, along with the near impossibility of actually applying its existing implementation to the real-world race conditions we encountered, we do not include RACEPRO as a baseline approach in our empirical evaluation. We did, however, evaluate the effects of replay divergence on RACEPRO's

operations. To do this, we downloaded and installed the latest version of RACEPRO released in August 2011 [31].

We next sought programs on which we could execute RACEPRO. We located programs that had been previously reported to have process level races in several bug repositories (eg, GNU, bugzilla, and debian). On the basis of bug descriptions, we identified applications written in C or shell scripting languages for which existing descriptions contained reports of concurrency faults that RACEPRO should be able to detect. These apps included *updatedb*, *mv*, *rm*, *locate*, *pxz*, *csplit*, and *bash* version 3.0.

To attempt to execute these programs on RACEPRO, we first created an application to modify these programs so that they create processes in certain orders using specific syntaxes supported by RACEPRO. However, the problems outlined in the prior section prevented us from applying the program to the vast majority of the subjects we considered. Ultimately, we were able to execute RACEPRO on 2 pairs of real-world programs, in which 2 instances of pairs of programs; one in which 2 *bash* shells are opened and concurrently execute, and one involving *rm* and *symlink* commands.

We next used RACEPRO to attempt to identify races in these programs; we performed this process on an Apple MacBook Pro laptop running OS-X version Yosemite (10.10) with Intel 2.4 GHz Core I5 Processor and 16 GB of memory. We also used Virtual Box to host Linux Kernel 2.6.35 as the guest operating system that support RACEPRO.

The process proceeded as follows. First, we used RACEPRO to identify all potential races; these should include locations that can cause a race we are trying to replicate. Second, we let RACEPRO perform its replay step to cause races to occur. At the end of this step, there are 3 potential outcomes. First, the replay finishes, but races are not detected. Second, the replay finishes, and races are detected. Third, replay divergences occur.

Our results show that replay divergences occurred prolifically. While RACEPRO was able to successfully identify locations in which races can potentially occur, it was not able to reproduce any of these races. For the first pair of programs, *bash* vs *bash*, RACEPRO made 18 attempts to detect races. It was able to complete 10 attempts without replay divergences, but it also did not find any races. Six attempts resulted in replay divergences. Two attempts resulted in deadlocks. For the second pair of the programs, *rm* vs *symlink*, RACEPRO made 27 attempts to detect races. All attempts resulted in replay divergences.

On the basis of these results, we chose not to use RACEPRO as a baseline system due to its failure to run most of the real-world applications. For the limited ones that can run, it also experienced frequent occurrences of replay divergences.

3 | APPROACH

SIMEXPLORER, which is implemented on Simics, is designed to support testing for process-level race conditions. We consider 4 classes of system components that instantiate events that can be involved in such races, as follows:

- Regular user processes (created through `fork` and `execv`). A user-level program can invoke multiple user processes by calling the `fork` and `execv` functions. The scheduling of processes is controlled by the kernel process scheduler.
- Software signals and handlers (software interrupts used for event notifications). A software signal in Linux is a software interrupt delivered to notify a user process about a particular event. When a software signal is sent to a process, a signal handler may be entered. The servicing of the signal may or may not immediately depend on the system state and the current disposition of the signal.
- Hardware interrupts and handlers. A hardware interrupt initiates the device communication by issuing an interrupt that is then serviced by an *interrupt service routine* (ISR), which is a procedure that is invoked when a particular type of interrupt is issued. The servicing of the interrupt may or may not immediately depend on the hardware state.
- Kernel threads that run in kernel mode. A kernel thread is similar to a user process except that it operates in the kernel space and has access to kernel functions and data structures.

SIMEXPLORER operates in 2 phases. Phase 1 analyzes the system under test for resources that can potentially lead to race conditions under normal execution. This phase takes existing inputs, following a random schedule to produce an event trace that is used in the second phase to test for faults. Phase 2 (the primary contribution of this work) permutes event interleavings to force faults to occur at potential faulty points obtained in phase 1. SIMEXPLORER outputs both potential faults and observable results.

SIMEXPLORER has 2 major components: an *Execution Observer* and an *Execution Controller* (Figure 9). The Observer monitors the system under test in phase 1 and generates runtime information that is used to analyze the system for potential sources of races. The Observer also monitors runtime events to direct the scheduling effort in phase 2. The Controller specifies events (eg, process scheduling, signal handler invocations, and ISR invocations) to be invoked at particular points in execution. This is critical to supporting testing because it allows nondeterministic events to occur predictably (the implementation of the Observer and Controller is described in Section 4). The output of phase 2 is a list of real races. SIMEXPLORER also checks the program's output against a test oracle to determine whether the race condition fault is revealed.

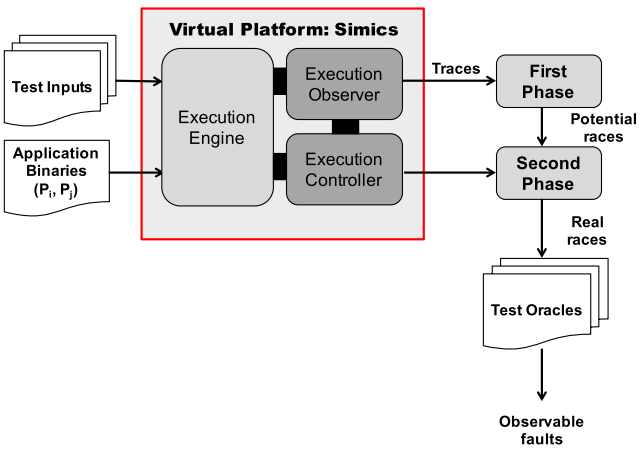


FIGURE 9 Overview of SimExplorer

3.1 | Recording events

As when detecting thread-level races, the Observer component of SIMEXPLORER records 2 types of events—shared resource accesses and synchronization events.

Shared resource accesses. The Observer records operations that access the following shared resources (*SR*s):

- shared memory,
- file descriptors (file handles of opened files),
- file inodes (metadata about files, such as time and access permission),
- file data (contents of a file),
- process data (process file table, data in */proc*, eg, PID), and
- hardware components (eg, registers).

Many of these *SR*s are also used by Laadan et al. [6] to perform process-level race detection; however, we also consider hardware components that are accessible by applications and device drivers. This enables our approach to detect more types of race conditions, such as races between device drivers and interrupt handlers. *SR*s need not be limited to the aforementioned resources; however, in this work, we consider only those resources that are likely to produce harmful races.[‡] For example, we do not track accesses to standard I/O, because races related to these are likely to be benign (when running multiple events, outputs are expected to be nondeterministic). In addition, we consider only *SR*s that are directly accessible by application developers. For example, we do not track kernel process data. However, SIMEXPLORER provides a user interface that lets developers specify other *SR*s.

After identifying *SR*s, the Observer tracks operations that access them in the system under test. We denote these operations as `OP`s. The system stores recorded information in a log file for future use. When engineers wish to identify potential sources of race conditions involving a pair of events, they can provide the traces of the events as inputs to our phase 1 shared resource analyzer.

We categorize `OP`s as *reads* or *writes*. For `OP`s related to memory accesses, SIMEXPLORER identifies *reads* and *writes* using the observability support implemented in our prior work [16,33]. For `OP`s related to hardware accesses, SIMEXPLORER uses the features provided by Simics to directly identify *reads* and *writes*. For `OP`s related to system calls on shared objects, SIMEXPLORER models their effects on *reads* and *writes*. SIMEXPLORER tracks 23 common system calls (eg, `open`, `read`, `write`, `lstat`, `clone`, `wait`, `execve`, and `exit`); these involve *reads* and *writes* on the defined *SR*s. 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*.

The `clone` system call creates a new process inode under the */proc* directory (write). The `wait` system call changes the state of the process id of its child process and removes the inode of its child process inode under the */proc* directory (write). The `exit` call changes the state of the pid of the current process. A system call passed with different options may have different effects on an object. For example, the `open` system call on file *f* reads the data for *f* with the O_RDONLY option, writes to the inode of *f*'s parent directory with O_CREATE, and writes to the data for *f* with the O_TRUNC option. SIMEXPLORER provides a user interface that lets developers define other system calls of interest.

Synchronization primitives. Synchronization primitives are used to control event interactions and avoid race conditions. For process-level races, the Observer monitors synchronization primitives related to processes, signals, and interrupts. In the case of regular processes, the Observer records `fork`, `wait`, `exit`, and `pipe` system calls. These system calls are used to synchronize shared resource accesses between processes (details are provided in Section 3.2).

In the case of signal handlers, the Observer tracks system calls involving signal registration, signal enabling, and signal disabling (ie, `sigaction`, `signal`, and `sigprocmask`). The Observer also tracks system calls and their parameters that can affect status of signals, including `sigfillset`, `sigaddset` and `sigemptyset`. A specific signal can be blocked by first adding it to the signal set through calling `sigaddset`. Then the signal set is passed to the `sigprocmask` with `SIG_BLOCK` option.

For each signal handler, the Observer maintains an attribute that records both the signal number and entry address of a signal handler, denoted by `<num, entry>`. This is because 1 signal may register 1 or more signal handlers during execution; the entry address can be used to distinguish handlers of the same signal. The observer also maintains a status list for each signal handler to record the instruction ranges in which the signal is enabled. This information guides phase 2 to find feasible locations to invoke event handlers. Specifically, each element in the list is denoted by `<start, end>`, where `start` is the instruction point where the signal is enabled and `end` is the instruction point where the signal is disabled.

In the case of hardware interrupts, the Observer monitors interrupt bits to track interrupt disabling and enabling operations. These bit-level operations are then mapped into the instruction-level statement, because phase 2 happens at the instruction level. Similar to signal handling, the Observer records attributes of ISRs, denoted by a 2-tuple `<num, entry>`.

3.2 | Phase 1: identify potential fault sources

The phase 1 algorithm computes a set of potential sources of faults, *PRaceSet*, between processes P_i and P_j, between processes and event handlers P_i and S_j, and between event handlers S_i and S_j. Two events have the *potential to race* if P_i and P_j (S_j) access the same *SR*, at least one of the accesses is a write, and 1 access can *potentially* happen both before and after the other. We next introduce notation used to describe our algorithms.

- L_i and L_j denote the log files of P_i and P_j (S_j) obtained from a normal execution, respectively.
- $\texttt{OP}(SR_n)$ denotes an *SR* access by P_n, and includes both the entry and exit of an *SR* access operation.
- $\sigma = (\texttt{OP}(SR_i), \texttt{OP}(SR_j))$ denotes a pair of access operations involving the same *SR* in P_i and P_j (S_j), and σ_i and σ_j denote the 2 individual accesses within the pair.
- $\texttt{ENT}(\texttt{OP}(SR_n))$ denotes the entry of an *SR* access in process P_n. If $\texttt{OP}(SR_n)$ is a system call, $\texttt{ENT}(\texttt{OP}(SR_n))$ is the instruction

[‡] A harmful data race can lead to observable faults detected by test oracles, while a harmless race does not affect program correctness [32].

that invokes that system call. If $\texttt{OP}(SR_n)$ is a memory or hardware access, $\texttt{ENT}(\texttt{OP}(SR_n))$ is the instruction that executes the load/store instruction.

- $\texttt{EXT}(\texttt{OP}(SR_n))$ denotes the exit of an *SR* access in process P_n. If $\texttt{OP}(SR_n)$ is a system call, $\texttt{EXT}(\texttt{OP}(SR_n))$ is the return instruction of the system call. If $\texttt{OP}(SR_n)$ is a memory or hardware access, $\texttt{EXT}(\texttt{OP}(SR_n))$ is the instruction immediately after $\texttt{ENT}(\texttt{OP}(SR_n))$.
- $\texttt{State}(P_n)$ returns the current state of process P_n.
- $\texttt{Execute}(P_n)$ returns the state after executing the current instruction of process P_n.
- $\texttt{Output}(P_i, P_j)$ returns the outputs of P_i and P_j.

Given an event e_i and an event e_j, we use $\prec$ to denote the *happens-before* relationship for e_i and e_j; this is a relationship that can be satisfied by the following rules:

- if e_i and e_j are from the same process, and e_i happens before e_j, then $e_i \prec e_j$;
- if e_i and e_j are from different processes, signal handlers, or interrupt handlers, and there is a dependency such that e_j would not happen unless e_i happens, then $e_i \prec e_j$;
- if e_i is from a process and e_j is from a signal handler *S*, and $e_i \prec e_j$, then $e_i \prec S_{all-the-events}$;
- if e_i is from a process and e_j is from a signal handler *S*, and $e_j \prec e_i$, then $S_{all-the-events} \prec e_i$;
- if e_i is from a process and e_j is from an interrupt handler *I*, and $e_i \prec e_j$, then $e_i \prec I_{all-the-events}$;
- if e_i is from a process and e_j is from an interrupt handler *I*, and $e_j \prec e_i$, then $I_{all-the-events} \prec e_i$; and
- $\prec$ is transitively closed.

Note that the foregoing rules are defined in relation to the list of shared resources, previously mentioned in Section 3.1, that our system focuses on. However, the approach can be extended to cover other types of resources that include semaphores and locks as well as other types of faults including improper order of file opening and accessing, nested locking relations, and atomicity violations including read/write in consistencies. However, there have been several prior efforts that focus on detecting races that occur due to accesses to such shared resources and detecting these fault types (eg [34–38]). Thus, we choose to focus here on other resources that can suffer from races due to system call, signals, and interrupts.

On the second of the foregoing rules, we infer dependencies from the semantics of system calls. These semantics are captured through the Linux system call specification [39]. For example, if Process P_i `forks` a child process P_j, all events in P_i prior to calling `fork` happen before all events in P_j. The following 2 semantics are also identified as *happens-before* system calls: (1) a `wait` occurs in the parent process, and an `exit` occurs in the child process; (2) a `pipe` in 1 process blocks the data in another process until the data is available. For the third and fourth of the foregoing rules, dependencies are inferred from the properties of signal handlers. Because a signal handler can never be preempted by a regular process, all events are synchronized in the signal handler.

On the third and fourth rules, we infer dependencies between processes and signal handlers from system states. If a signal associated with a signal handler *S* is disabled from the entry point of process *P* (not registered), all events in *S* should happen after the signal is enabled in the process. If a signal is disabled at a nonentry point of *P* and not enabled until the exit point of *P*, all events prior to the disabled point in *P* must happen before all events in *S*.

The fifth and sixth rules also apply to the case of hardware interrupts. If the interrupt bit associated with an interrupt handler I_i is disabled from the entry point of a process, all events in I_i should happen after the interrupt is enabled in the process. If an interrupt is disabled at a nonentry point of *P* and not enabled until the exit point of *P*, all events prior to the disabled point in *P* must happen before all events in *S*.

The basic notion behind rules 3-6 is that a signal hander or an interrupt handler cannot run in parallel with the process it interrupts.

Figure 10 provides the algorithm for phase 1. The algorithm takes traces L_i and L_j as inputs, and outputs a set of potential races, *PRaceSet*. For each shared resource sr_i being accessed in L_i, the algorithm iterates

Algorithm SimExplorer: Phase1

```
1: Inputs: Li, Lj
2: Outputs: PRaceSet
3: begin
4: PRaceSet = φ
5: for each SRi ∈ Li
6:     for each SRj ∈ Lj
7:         if SRi equals to SRj
8:             if ¬(OP(SRi) ≺ OP(SRj)) ∧¬(OP(SRj) ≺ OP(SRi)) ∧
               ¬(OP(SRi) is read ∧ OP(SRj) is read)
9:                 if OP(SRi) comes before OP(SRj)
10:                    σ = (OP(SRi), OP(SRj))
11:                else
12:                    σ = (OP(SRj), OP(SRi))
13:                endif
14:                PRaceSet = PRaceSet ∪ σ
15:            endif
16:        endif
17:    endfor
18: endfor
19: end
```

FIGURE 10 Algorithm used in phase 1

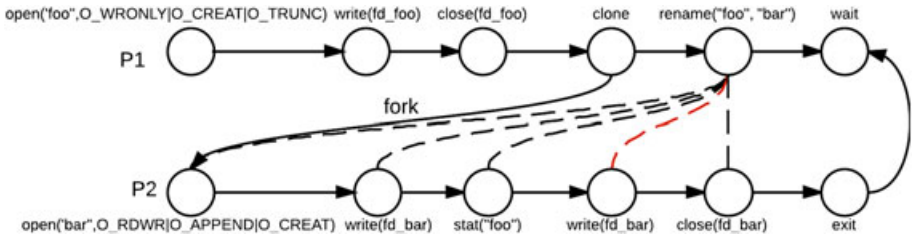


FIGURE 11 Happens-before example (solid arrows denote happens-before relationships, dashes denote potential races)

over L_j to find the same shared resource being accessed (lines 5-7). If the 2 events do not have a *happens-before* relationship and at least 1 access is a write, the pair of events is marked as a potential race and added to *PRaceSet* (line 14).

To simplify the phase 2 algorithm, event pairs are recorded based on execution order. For example, an event pair recorded as (σ_1, σ_2) indicates that σ_1 occurs before σ_2 in the original execution trace, and the phase 2 algorithm will need to reverse the 2 events to make σ_2 happen before σ_1. Note that only distinct event pairs are identified in terms of instruction locations; we ignore redundant pairs that occur due to recursive calls or loops. As an example, we refer to the program in Figure 1, which has the following execution trace:

*P*1:`open`(*"foo"*)→ *P*1: `write`(*fd_foo*)→ *P*1:`close`(*fd_foo*)→ *P*1:`clone`()
→ *P*1:`rename` (*"foo"*,*"bar"*)→ *P*1:`wait`()→ *P*2:`open`(*"bar"*)→ *P*2:`fstat`(*fd_bar*)
→ *P*2:`write`(*fd_bar*)→ *P*2:`stat`(*"foo"*)→ *P*2:`write`(*fd_bar*)→ *P*2:`close`(*fd_bar*)→ *P*2:`exit`()

Figure 11 shows the *happens-before* relationships and the potential-race relationships between operations in this trace. Solid arrows indicate *happens-before* relationships between pairs of events. For example, all events in *P*2 happen after events in *P*1 prior to the `fork` call. The return from `wait` in *P*1 happens after the `exit` of *P*2 because *P*1 has to wait for *P*2 to finish. The system call `rename` in *P*1 modifies the inodes of "foo" and "bar", thus it races with the system calls accessing the inodes in *P*2. Events connected by dashes are potential race pairs.

This phase 1 algorithm may generate false positives for 2 reasons. First, the algorithm does not identify all synchronization operations. In Figure 1, the statement in line 10 cannot be executed unless the statement in line 16 is executed. Thus, the potential race pairs (*P*1:`rename` (*"foo"*, *"bar"*), *P*2:`write`(*fd_bar*)) (the second `write`(*fd_bar*) in *P*2) and (*P*1:`rename` (*"foo"*, *"bar"*), *P*2:`close`(*fd_bar*)) in Figure 1 can never result in real races.

In addition, the phase 1 algorithm (as well as other existing dynamic techniques) fails to consider hardware states. As embedded software and device drivers are highly dependent on hardware, it is difficult to predict feasible interleavings. For example, interrupts usually rely on several operations among different hardware bits. Even if a potential race can be predicted from 1 execution, this race may not happen since the hardware bits may be changed when the predicted interleaving is exercised. We describe how our proposed system deals with false positives as part of phase 2 in the next section.

3.3 | Phase 2: test for real races

The main objective of the phase 2 algorithm is to automatically control process scheduling based on each pair in *PRaceSet*. Since software signals and hardware interrupts use different synchronization mechanisms than regular events, we also develop an algorithm to automatically control asynchronous events. We present these 2 algorithms as *Phase2-RegPro* (Figure 12) and *Phase2-Asnyc* (Figure 13).

3.3.1 | Processes

We first describe the *Phase2-RegPro* algorithm. Let $\sigma = (\sigma_i, \sigma_j)$ be a pair of instructions that have been identified as potential race sources. In this case, σ_i happens before σ_j when P_i and P_j are originally executed. The algorithm takes σ, P_i and P_j, and test oracle *O* as inputs; it outputs real races and observable faults. The goal of the algorithm is to switch the order of σ_i and σ_j to let σ_i happen after σ_j. Specifically, whenever a process is about to execute an instruction in σ, it is suspended (lines 7-9). After both events have been suspended, P_j is allowed to resume (lines 10-12) until it reaches the exit of its *SR* access (lines 13-14).

Next, P_i is allowed to resume (line 15) until it reaches the exit instruction of its *SR* access (line 17). At this point, we have switched the execution order of σ_i and σ_j. A real race is added into *RaceSet* and the events continue executing normally (lines 18-19). If 1 process terminates, the algorithm checks the state of the other and allows it to resume if it is still suspended (lines 21-23); in this test execution, the potential race is not a real race because the terminated process does not reach the *SR* after the other process reaches the *SR* and is suspended. When both events terminate, the algorithm checks their outputs against a test oracle to determine whether a fault has been identified (lines 24-26).

To see how this algorithm detects process-level races, consider the example shown in Figure 1. Suppose we select the potential race pair σ = (*P*1:`rename`(*"foo"*, *"bar"*), *P*2:`write`(*fd_bar*)) (the first `write`(*fd_bar*) in *P*2) identified in phase 1. The system call `rename` comes before the `write` system call in the original execution trace; thus the goal of phase 2 is to let the `rename` happen after the `write`. First, the algorithm suspends the execution of *P*1 and *P*2 when they are about to execute the instructions in the race pair. Next, *P*2 is allowed to resume until the `write` returns. *P*2 is then suspended again, and *P*1 is allowed to resume until it returns from `rename`. At this point, we conclude that σ is a real race. In fact, if we let *P*1 and *P*2 continue, we will discover that this race is harmful.

By default, SIMEXPLORER switches the order of operations to allow one to happen immediately after the other. However, in a pair (σ_i, σ_j),

Algorithm SimExplorer: Phase2-RegPro

```
1: Inputs: PRaceSet, P_i, P_j, O
2: Outputs: RaceSet, Faults
3: begin
4: RaceSet = φ
5: for each σ = (σ_i, σ_j) ∈ PRaceSet
6:       execute P_i and P_j
7:       if Execute(P_n) ∈ σ and n ∈ (i, j)
8:           suspend P_n
9:       endif
10:      if State(P_i) is suspended and State(P_j) is suspended
11:          continue P_j
12:      endif
13:      if Execute(P_j) == EXT(σ_j)
14:          suspend P_j
15:          continue P_i
16:      endif
17:      if Execute(P_i) == EXT(σ_i)
18:          RaceSet = RaceSet ∪ σ /*race occurs*/
19:          continue P_j
20:      endif
21:      if P_m terminates ∧ State(P_n) is suspended ∧ m ∈ (i, j) ∧ n ∈ (i, j) ∧ m ≠ n
22:          continue P_n
23:      endif
24:      if Output(P_i, P_j) ≠ O
25:          print "Error: fault found"
26:     endif
27: endfor
28: end
```

FIGURE 12 Phase 2 algorithm to test for races between processes

Algorithm SimExplorer: Phase2-Async

```
1: Inputs: PRaceSet, P, S
2: Outputs: RaceSet, Faults
3: begin
4: RaceSet = φ
5: for each σ = (σ_i, σ_j) in PRaceSet
6:       execute P
7:       if (Execute(P) == EXT(σ_j) ∧ σ_i ∈ S) ∨ (Execute(P) == ENT(σ_i) ∧ σ_j ∈ S)
8:           if State(S) is not masked
9:               raise signal S
10:          else
11:              find another possible location
12:          endif
13:          if S accesses SR
14:              RaceSet = RaceSet ∪ σ /*race occurs*/
15:          endif
16:      endif
17:      if Output(P, S) ≠ O
18:          print "Error: fault found"
19:      endif
20: endfor
21: end
```

FIGURE 13 Phase 2 algorithm to test for races between a process and a signal

it may not always be possible to let σ_i happen immediately after σ_j due to changes in system states (eg, disabled process scheduling and page fault). In such cases, SIMEXPLORER continues the execution of P_j (line 14) until it is possible to schedule P_i and execute σ_i (line 17). If the entry instruction of the operation in another potential race pair is reached and it is still not possible to execute P_i, then (σ_i, σ_j) is not considered to be a race.

3.3.2 | Signal handlers

We now describe Algorithm *Phase2-Async*. The goal of this algorithm is to manipulate the order of potentially racing accesses to *SR* between a user process *P* and the signal handler *S*. According to the third and fourth rules described in the *happens-before* relationship, the goal of this algorithm can be restated as that of swapping the order of an *SR* access in *P* and the arrival of *S*. The algorithm first executes process *P* (line 6). If the signal occurs before *SR* is accessed by *P* in the original execution trace, the algorithm forces the signal to occur immediately after *SR* is accessed by *P* (first condition in line 7). If the signal occurs after *SR* is accessed by *P* in the original execution trace, the algorithm forces the signal to occur just before *SR* is accessed by *P* (second condition in line 7). If a race occurs, it is added to *RaceSet* (line 14).

Because it may not be possible to raise a signal immediately (eg, if the signal is currently masked), the algorithm checks the current state of *S* (line 8) before raising a signal. Similar to *Phase2-RegPro*, the algorithm also checks outputs on termination of the events (lines 17-19) to determine whether a fault has been identified.

If the signal handler (*S*) cannot be raised *immediately after* the *SR* access in *P* (lines 10-12), the algorithm postpones *S* until it can feasibly be raised, or until the entry instruction of the operation in another potential race pair is reached. In the case in which the signal handler cannot be raised *immediately before* the *SR* access in *P*, the algorithm finds another possible location in *P* at which to raise *S*, before the *SR*

access and after the entry instruction of the operation for another potential race pair.

To illustrate the algorithm's operation, using Figure 2 as an example, the variable *files_created* is a global variable and its memory read and write accesses are denoted by `load` and `store`. The execution trace is as follows:

P:`open`(*file*1) → *P*:`load`(*files_created*) →

P:`store`(*files_created*) → *P*:`open`(*file*2) →

P:`load`(*files_created*) → *P*:`store`(*files_created*) →

S:`load`(*files_created*) → *S*:`unlink`(*file*1) →

S:`load`(*files_created*) → *S*:`unlink`(*file*2).

From this trace, there are 2 potential race pairs, σ_1=(*P*:`open`(*file*1), *S*:`unlink`(*file*1)) and σ_2=(*P*:`store`(*files_created*), *S*:`load`(*files_created*)). Although (*P*:`open`(*file*2), *S*:`unlink`(*file*2)) can potentially race, the elements of the pair are redundant because they are part of loops. Considering σ_2, *S* happens after *P*:`store`(*files_created*) occurs in the original execution; thus, the algorithm forces *S* to be raised before *P*:`store`(*files_created*) in phase 2. In this scenario, *files_created* is incorrectly read by *S*. As such, the race is real and harmful.

A signal handler cannot run in parallel with the process that it interrupts. However, it is possible for a signal handler to race with another process running concurrently with the handler. In this scenario, we can apply the *Phase2-RegPro* algorithm to detect races between the handler and the concurrently running process. Using Figure 14 as an example, the program has 1 process (`main`) and 2 signal handlers (`sh1` and `sh2`) with different signal types (`SIGINT` and `SIGALARM`). Variables SV1, SV2, and SV3 are global variables. The execution trace is as follows:

P:`sigaction`(*SIGINT*, *sh*1) → *P*:`write`(*SV*1) →

P:`sigaction`(*SIGALARM*, *sh*2) → *sh*1:`read`(*SV*1) →

*sh*1:`write`(*SV*3) → *P*:`read`(*SV*2) →

P:`write`(*SV*3) → *P*:`sigemptyset`(*nmask*) →

P:`sigaddset`(&*nmask*, *SIGINT*) → *sh*2:`write`(*SV*1) →

*sh*2:`write`(*SV*2) → *P*:`read`(*SV*3)

```
1  main{
2      sigaction (SIGINT, sh1)
3      write SV1
4      ...
5      sigaction (SIGALARM, sh2)
6      ...
7      write SV2
8      ...
9      sigemptyset(nmask);
10     sigaddset(&nmask, SIGINT);
11     ...
12     read SV3
13     exit
14 }
15
16 sh1{
17    read SV1
18    ...
19    write SV3
20 }
21
22 sh2{
23    write SV1
24    ...
25    write SV2
26 }
```

FIGURE 14 Nested Software Signals

From this trace, there are 3 potential race pairs: σ_1=(*P*:`write`(*SV*1), *sh*1:`read`(*SV*1)), σ_2=(*P*:`read`(*SV*2), *sh*2:`write`(*SV*2)) and σ_3=(*sh*1:`read`(*SV*1), *sh*2:`write`(*SV*1)). While pairs (*P*:`write`(*SV*1), *sh*2:`write`(*SV*1)) and (*P*:`write`(*SV*3), *sh*1:`read`(*SV*3)) modify the same shared resource, they are synchronized by the happens-before. Considering σ_3, *sh*1:`read`(*SV*1) happens before *sh*2:`write`(*SV*1) in the original execution; the algorithm forces *sh*2 to be raised before *sh*1:`read`(*SV*1) in Phase 2.

3.3.3 | Interrupt handlers

The *Phase2-Async* algorithm can be adapted to handle interrupts related to races. In fact, controlling interrupts is easier than controlling processes and software signals. This is because Simics allows us to issue an interrupt on a specific IRQ line from the simulator itself. As such, when the Observer reaches an *SV*, an interrupt is invoked at a feasible location after the access to this *SV*.

Similar to the case of signal handlers, it is not always realistic to invoke an interrupt whenever we want. For example, the IER and possibly other control registers have to be set to enable interrupts. In the example of Figure 6, before invoking an interrupt, the IER of the UART must be set while the interrupt identification register IIR must be cleared. Even if interrupts are enabled, they can be temporarily disabled. Figure 15 is the routine in the Controller used to determine whether it is possible to issue an interrupt.

There are 2 general steps that our system takes prior to invoking a *controlled interrupt*. First, the controller module checks the status of the local and global interrupt bits to see if interrupts are enabled. In an X86 architecture, the global interrupt bit is the ninth bit of the eflags register (line 46 in Figure 15). When this bit is set to 1 the global interrupt is disabled, otherwise it is enabled. For local interrupts, Simics uses the Advanced Programmable Interrupt Controller as its interrupt controller. As such, our system checks whether the bit controlling the UART device is masked or not. Our system also checks whether a *self-generated* interrupt is about to be issued by examining the current pin status. If this is true, the *controlled* interrupt will not be invoked.

Where nested ISRs are concerned, as opposed to nested signals, we can apply the *Phase2-Async* algorithm to detect races between interrupt ISRs. This is because interrupts are typically assigned priorities. The higher priority ISR cannot be preempted by a lower priority ISR. As such, the Observer applies the third and fourth happens-before rules to detect races between ISRs.

3.4 | Algorithm scalability

Where scalability is concerned, a typical active testing approach such as RACEFUZZER [13] verifies every potential race reported in the first phase for a given input regardless of whether this race has already been verified by previous inputs. This can be expensive when given large test suites, particularly when each test case reports redundant potential races in the first phase. To alleviate this problem, we adjusted SIMEXPLORER such that once a potential racing pair is confirmed to be a real race, it is not verified again under another input. Specifically, for each test input, SIMEXPLORER reports potential races. If there exist any potential races that have not been confirmed to be real races,

```
procedure ISR_enabled(int p)
/*p is the pin number for a certain interrupt*/
45: begin
46:     if eflags[9] != 0 and ioapic.redirection[p] == 0
and ioapic.pin_raised[p] == LOW:
47:         return true
48:     else
49:         return false
50:     endif
51: end
```

FIGURE 15 Algorithm to determine whether it is possible to issue an interrupt

SIMEXPLORER verifies each of them, under the same test input, before proceeding to the next test input. As such, the actual number of test runs for an entire test suite T is

$$TR = |T| + \sum_{i=1}^{|PRaceSet|} NT_i$$

Here, $|T|$ is the total number of original tests, $|PRaceSet|$ is the number of potential races, and NT_i is the number of tests required to verify the ith pair in *PRaceSet*. Thus, if the number of potential races is fixed, the total number of test runs is linear in the number of test cases.

3.5 | Discussion and limitations

Several additional aspects of our approach and the foregoing algorithms bear further discussion. First, SIMEXPLORER tests only 1 potential race pair in each test run. This is done to avoid *fault masking* effects, which may occur in cases where the presence of a single conflicting pair is "masked" by the presence of the second. Thus, in a given run, our system needs to first determine whether a pair has already been tested; if it has, the system does not monitor events related to that pair in that run.

Second, in phase 2, SIMEXPLORER may encounter a potential race pair more than once during a test run (eg, due to a loop). Once a pair is identified, there is no need to keep recording it after each encounter; thus, our algorithm records only the first instance. This reduces the amount of redundant information in the race report.

Third, SIMEXPLORER reports both harmful and benign races. There are 2 steps that can be followed to distinguish these. First, races that cause failures that are observable through test oracles are clearly harmful. Second, although SIMEXPLORER is fully automated, additional manual effort can be used to determine whether the program may fail under other test inputs or interleavings.

Following these 2 steps, we manually classify races reported in the phase 2 as harmful or benign. We will study the automatic identification of benign races in future work by leveraging a recently proposed symbolic execution technique [32].

Fourth, SIMEXPLORER can also be used to *replay* concurrent executions of multiple threads or events. This is achieved by using the Controller to stop the execution of threads or events at the same execution points as in prior runs. Note that in Linux, every thread is a lightweight process with its own Process Identification Number (PID), so SIMEXPLORER can track it just like a process.

Fifth, SIMEXPLORER can cause events to deadlock in ways in which they should not. Deadlock occurs when 1 process is waiting for an event in another process that has been suspended by SIMEXPLORER. For example, process A calls a `read` in a loop to read the data written by process B. Process B calls a `write` to write the data. The loop exits until all data gets read. However, when process B is suspended before calling `write`, process A will hang and keep checking the availability of the data. We observed such a deadlock in 4 of the object programs we considered in our empirical study. To address this problem, SIMEXPLORER periodically checks the states of the events and allows the process that is being waited for to resume.

Sixth, the second phase of SIMEXPLORER verifies all events leading up to the 2 events in the potential race pair. It decomposes a complex system into a collection of controllable events. This allows us to focus only on the 2 events under test; events other than these run as normal. SIMEXPLORER monitors and controls the 2 events without interfering with others to guarantee that the system can execute. While this approach is sound and cost-effective, it can be incomplete. For example, replaying a racing pair may require the control of more than system-level events, because some other sources of nondeterminism within a process could also affect the execution behaviour of the process. On the other hand, complete replay techniques such as PENELOPE [40] may incur much higher overhead due to context switches. While we did not encounter any instances of incompleteness related to this sort of effect in our empirical study, in our future work, we will further evaluate the cost-effectiveness trade-offs of different approaches by studying complete replay techniques.

Seventh, by viewing every shared resource as a potential race source, SIMEXPLORER can produce false positives in phase 1. We use the happens-before relationship to reduce these false positives. In some cases, false positives can be reduced in phase 1 by strictly modeling communication primitives as happens-before relationships. However, when synchronization occurs at the process level, there can be ambiguities at the implementation level because the implementations may be unknown (many system calls are blackbox), and their synchronization effects cannot easily be modeled. Instead, SimExplorer takes an approach that does not involve preconceptions about implementation details.

Finally, SIMEXPLORER focuses on race conditions related to order violations between 2 access events. Thus, it can miss races involving more than 2 events, such as atomicity violations. As part of our future work, we can take advantage of existing concurrency fault detection algorithms (eg, for atomicity detection [27,41,42] and atomic-set serializability detection [10,43]) to handle a broader class of process-level race conditions.

4 | IMPLEMENTATION

As stated in Section 1, SIMEXPLORER is implemented on SIMICS. The *Observer* and *Controller* components of SIMEXPLORER use the virtual platform's abilities to interrupt execution without affecting the states of the virtualized system, to monitor events such as memory accesses, hardware accesses, system calls, and system states, and to manipulate memory and buses to directly invoke events such as interrupts and traps (eg, setting registers as described in Section 3.3.3). As such, SIMEXPLORER is able to stop program execution at a particular point of interest and force process scheduling or a signal to occur. SIMICS provides APIs that can be accessed via Python scripts; thus, SIMEXPLORER algorithms are programmed as Simics tools and written in Python.

Observer. We configured the *Observer* component so that it sets memory breakpoints on shared resource operations. The observer invokes callback functions when events of interest occur (eg, interrupts, memory read/write operations). As stated in Section 3, SIMEXPLORER monitors only operations that are initiated by a process under test (*PuT*). Therefore, we need to distinguish events invoked by the *PuT* from other events. To do this, we keep a calling stack for the *PuT* at runtime. When a function in the *PuT* is invoked, its frame pointer is added to the call stack. When a shared resource is accessed, we compare the current frame pointer with the frame pointer on top of the call stack. This mechanism allows us to ignore shared resources that might be accessed by kernel threads or user-level processes that are not under test.

Controller. One challenge in the implementation of the *Controller* component is enabling it to consider the state of the entire system. Currently, existing techniques for detecting thread-level data races perform dynamic analysis to permute thread interleavings [13,44]. One common approach is to insert yield instructions or sleeping points at the locations to pause execution. Because these techniques focus on races among threads within a process, they can abstract away execution of threads belonging to other processes and assume that such execution has no effect on the application under test. To reveal data races, these approaches execute threads under test to observe whether there are races between these threads and the suspended thread.

Techniques for revealing process-level races, in contrast, must consider execution of threads across processes and changes to the systems due to their execution. As such, the approaches must work despite the existence of unpredictable kernel scheduling. An approach introduced in [6] does not modify kernel scheduling but instead creates an event recorder to generate runtime traces that can be analyzed for races. In doing so, the approach does not control execution orders and can detect only races that are exercised by scheduled interleavings, and therefore, the approach is not suitable for testing. Once an execution trace is generated, the approach is capable of reordering system call events as a way to determine of a detected race is harmful.

Our approach exploits the ability to observe and suspend execution inside a virtual platform to achieve the desired controllability. We implemented the *Controller* component by installing a dummy device and its driver program. A dummy device is a virtual device that can be integrated with the Virtual Machine (VM). A VM consists of a variety of virtual devices that can simulate not only the processor cores and memory but also the entire hardware platform including the network, buses, and peripheral devices (eg, keyboards, USBs, and video adaptors). A dummy device does not simulate a real device but has the basic elements of a regular hardware device, including registers and interrupts. A user can write a device driver to control the dummy device.

When an event such as a shared resource access occurs, the *Controller* is notified by the *Observer*. Next, the *Controller* writes to the programming interrupt controller to raise an interrupt for the dummy device, and the interrupt handler in the device driver is immediately executed. The interrupt handler checks the state of the target process using the *task_struct* data structure and determines whether to suspend or resume a process or send out a software signal. Our mechanism can precisely force events to occur at particular binary instructions because hardware interrupts always have the highest priority to be executed. In addition, our approach is scalable to any operating system that can run on Simics.

Since SIMICS supports multicore systems, SIMEXPLORER is capable of detecting races due to concurrent execution on multiple cores. As an example, a device driver that is invoked simultaneously on 2 different cores can race if it manipulates the same hardware resource. To detect races in such systems, we use a dummy device for each core. During testing, we use the Linux *sched_setaffinity* command to assign kernel threads to 2 cores. Instead of suspending the kernel thread when necessary, SIMEXPLORER makes the thread spin on its core; this is because interrupt handlers cannot call sleep functions.

5 | EMPIRICAL STUDY

To evaluate SIMEXPLORER we consider 3 research questions:

RQ1: How effective is SIMEXPLORER at detecting system-level races across the phases of its operation?
RQ2: How effective is SIMEXPLORER at detecting observable faults related to those races?
RQ3: How efficient is SIMEXPLORER?

RQ1 allows us to evaluate the effectiveness of our phase 2 algorithm at reducing false positives, RQ2 lets us consider the ultimate effectiveness of our approach, and RQ3 provides an assessment of the overhead associated with the approach.

5.1 | Objects of analysis

To obtain objects of analysis, we first searched several bug repositories (eg, GNU, bugzilla, and debian) using the key words "concurrency", "race", "deadlock", "atomicity", "signal", and "interrupt". By examining bug descriptions (not code), we identified applications written in C or

TABLE 1 Object program characteristics

	Program		NLOC		Faults	Tests
	Basic	Paired	Basic	Paired		
Process level	BASH1	BASH1	39102	39102	1	30
	BASH2	BASH2	40944	40944	1	30
	TCSH	TCSH	47167	47167	1	5
	STRACE	MYPROG	25192	491	1	49*
	UPDATEDB	UPDATEDB	1924	1924	1	25*
	UPDATEDB	LOCATE	1924	669	1	48*
	FIND	RACEPROG1	4004	491	1	25*
	PYTHON	PYTHON	681	681	1	12*
	UART	UART	1944	1944	1	16
	MV	RACEPROG	346	491	1	45
	PS	GREP	4695	4695	1	57
	LN	RACEPROG2	410	491	1	28
	MKDIR1	RACEPROG3	143	491	1	23
	MKDIR2	RACEPROG4	146	491	1	23
	MKNOD	RACEPROG5	184	491	1	16*
	MKFIFO	RACEPROG6	112	491	1	17*
Signal level	BASH2	SIG (5)	40944	493	1	30
	BASH3	SIG (5)	48263	553	2	30
	STRACE	SIG (10)	25192	1814	1	49
	CSPLIT	SIG (4)	1032	73	3	30*
	TELNETD	SIG (4)	10323	118	1	59*
	SIG31	SIG (2)	35	3	1	3*
	EG	SIG (3)	38	8	2	7*
	NANO	SIG (7)	14331	629	1	30*
	ZSH	SIG (5)	96190	843	2	32*
	ED	SIG (3)	2384	157	2	45*
Interrupt level	UART	INT	3341	132	2	12*
	UART1	INT	3339	132	1	12*
	UART2	INT	3339	132	1	12*
	UART3	INT	3339	132	1	12*
	UART4	INT	3339	132	1	12*
	UART5	INT	3339	132	1	12*
	UART (smp)	INT	3341	132	1	14*
	UART	TIMER	3341	102	1	14*

C++ for which existing descriptions contained reports of concurrency faults related to resource dependencies among multiple processes and between processes and software signal handlers. We randomly selected 40 of these applications. We next eliminated from consideration those applications that could not compile in our environment; this process left us with 18 applications.[§]

We next identified 3 applications with signal handlers studied by Tchamgoue et al. [45] (NANO, ZSH, and ED). We further examined the CWE[¶] and securecoding[‖] websites and located 4 examples with vulnerabilities related to signal handlers.[**]

Three out of 4 examples were too small (fewer than 20 lines of non-commented code), so we made a new program EG, that integrates all functionality in the 3 examples (eg, they use different types of signal handlers). The other program is SIG31.

To study race conditions related to hardware interrupts, we applied SIMEXPLORER to the UART device driver on 2 preemptive kernel versions of Fedora Core 2.6.15 and Ubuntu 12.04. The driver includes 2 files, serial_core.c and 8250.c. The main application transmits character strings to and receives character strings from the console via the UART port. Note that in this study, we apply our testing process only to the UART driver. However, the same process is also applicable to other types of device drivers.

Our object programs are listed in column 1 of Table 1 (some are listed multiple times, for reasons explained below). We categorize the pairs of programs into process level, signal level, and interrupt level, as these are the types of races that occur among the pairs. The 3 categories are separated in the table by double horizontal lines. (Some programs are

[§]The objects of analysis and other artifacts used in this study can be obtained by contacting the first author.
[¶]https://cwe.mitre.org
[‖]http://www.cert.org/secure-coding/
[**]cwe-364,sourcecoding1,sourcecoding2

placed in multiple categories because they do apply to both.) Columns 3 and 4 in the table list the numbers of lines of noncomment code in the programs.

The top section of the table lists process-level pairs. BASH and TCSH are command interpreters that execute commands read from standard input or from a file. We used BASH versions 3.0, 3.2, and 4.2 and TCSH version 6.17. STRACE is a debugging tool that intercepts and records the system calls made by a process and the signals received by a process; we used version 4.5.18. UPDATEDB, LOCATE, and FIND are Linux utilities: UPDATEDB creates or updates a database used by LOCATE to find files, FIND is used to locate files in the file system; we used versions 4.1.2 of each of these. UART is a UART device driver from the Linux kernel; we used version 2.6.31. PYTHON1 is the subprocess module of Python version 2.4 that allows applications to spawn new processes. All other programs are Linux core utilities, including MV (version 6.1.9), PS (version 6.19), GREP (version 6.1.9), LN (version 5.97), MKDIR (version 5.97), MKDIR (version 5.2), MKNOD (version 5.2), and MKFIFO (version 5.2).

The middle section of the table lists the signal-level programs. The BASH2 program and STRACE also apply to this category. CSPLIT is used to split a file into 2 or more smaller files determined by context lines; we used version 5.2. SIG31 and EG are vulnerable programs described above. TELNET is an protoal program; we used version 0.17. NANO is a text editor for Unix-like computing systems; we used version 2.0. ZSH is an interactive command line interpreter; we used version 4.2. ED is a GNU line-oriented text editor; we used version 0.6.

The bottom section of the table lists multiple versions of the interrupt-level program, UART, described above. Note that the last 2 pairs are special cases. Since having only 1 interrupt handler prevents us from studying nested interrupts, we need to simulate an environment with additional interrupt handlers. In the case of (UART (SMP), INT), we enable SMP to increase the chance of letting the UART ISR preempt itself, since ISR normally does not race with itself on the same CPU. This allows us to study re-entrant interrupt handlers [46]. In the case of (UART, TIMER), we asked an unbiased person (a person who was not familiar with our technique and not involved in this research) to write a dummy timer device and its associated interrupt handler (ie, TIMER). This pair involves a premature read from the UART buffer [47].

To address our research questions, we also needed to identify specific faults in the object programs that could result in system-level races. By examining descriptions in the bug repositories for each object program, we located 36 naturally occuring faults related to the race types.

Since Simics has only 1 device that allows us to study interrupt handlers, the numbers of naturally occurring faults found in the UART program was low, rendering comparisons of the fault detection effectiveness of the approaches less meaningful (beyond the 2 faults created to correspond to special cases, as discussed above, we found only 2 naturally occuring faults). To further investigate fault detection effectiveness, we created additional seeded race conditions for UART. We injected 12 potential races into 8250.c by making syntactic changes to the code. For race conditions, we removed statements corresponding to critical section protection (e.g., `spin_lock`, `spin_lock_irq`). Of the 12 potential faults thus created, further examination revealed that 7 of the potential race condition faults could not possibly be triggered on the system on its given hardware platform, so we removed those. This left us with 5 additional potentially revealable race condition faults.

Column 5 of Table 1 lists the total numbers of faults that we ultimately studied, for each of the program pairs.

To enable system-level racing pairs to occur, the basic programs shown in column 1 of Table 1 must each be paired with at least 1 other process, signal handler, or interrupt handler. To select second programs to pair with the basic programs, we again consulted the bug repositories for the basic programs, and where possible, selected as paired programs those programs indicated as having led to the faults previously identified when run concurrently with the basic programs.

In several cases, bug reports did not specify problematic paired programs. However, exposing races requires 2 programs, so in these cases, we needed to create a second program. To achieve this, we asked an unbiased person who had no knowledge of our purposes to create a program, RACEPROG, that can be paired with multiple basic programs. This program accepts shared resources as inputs and uses various operations to manipulate the shared resources accessed by the basic programs. For example, on 1 test for MV, "`mv file1 file2`", that operates on 2 shared resources `file1` and `file2`, RACEPROG uses system calls (eg, `open, stat, access`) to access `file1` or `file2` or both.

The paired programs and signal handlers we used are listed in column 2 of Table 1, and the numbers of lines of code contained in these programs are reported in Column 4. In 3 cases (on programs BASH2, STRACE, and UPDATEDB) 2 different paired programs or signal handlers involved in races were identified, yielding 2 pairs of programs that could be studied—this accounts for the repetitions in column 1. The number in parentheses following each signal handler indicate the number of signal types registered in the program.

To address our research questions, we also required test suites. For basic programs that had been released with test suites, we selected, from those, test cases that could potentially exercise shared resources (eg, test cases of operations on BASH history files). Seven of our object programs, however, were not equipped with test suites. For these programs, we created black-box test suites. Engineers often use such test suites designed based on system parameters and knowledge of functionality [48]. We followed this approach, using the category-partition method [49], which uses a test specification language to encode choices of parameters and environmental conditions that affect system operations and combine them into test inputs. We asked an unbiased person who had no knowledge of our purpose to employ this approach.

Next, for each of the basic programs, we augmented test cases for the system based on a code-coverage-based test adequacy criterion. We first statically identified *SR*s in both basic and paired programs. We then generated a set of test cases that cover the feasible *SR*s (*SR*s for which there exists a possible execution of the program that executes them). This is similar to the *thread-pair-interleavings* (TPAIR[††]) criterion proposed by Lu et al. [50]. Column 6 of Table 1 lists the numbers of test cases ultimately used for each object program pair; numbers marked with a "⋆" indicate instances in which test cases were augmented.

Testing also requires test oracles. For programs released with existing test suites and with built-in oracles provided, we used those.

[††] An interleaving space gets complete coverage under the TPAIR criterion if all feasible interleavings of all shared memory accesses from any pair of threads are covered.

Otherwise, we checked program outputs, including messages printed on the console and files generated and written to by the programs. The oracles were obtained by running the 2 processes under test sequentially.

5.2 | Variables and measures

Independent variable. RQ1 assesses factors related to the effectiveness of SIMEXPLORER's phases, and SIMEXPLORER'S efficiency, and does not involve independent variables.

When considering RQ2 and RQ3, our independent variable is the testing technique used. One technique that we use, of course, is SIMEXPLORER, but we also wished to compare it with appropriate baseline techniques. Although techniques for thread-level concurrency fault detection exist, as Section 2.2 explains, they cannot be used to detect system-level concurrency faults and thus are not appropriate as baseline techniques. RACEPRO [6] can detect system-level races, and we considered using it; however, RACEPRO has not been updated for several years, and we were unable to use it on our object programs. More important, however, RACEPRO has several severe limitations (discussed in Section 2.3) that would render even a robust implementation notably less effective than SIMEXPLORER.

Another alternative for use as baseline techniques involves stress testing techniques. For the purpose of our evaluation, these techniques have several advantages. First, stress testing is a practical approach that currently represents the state-of-the-practice in industrial settings [51–53]. Second, the bug reports for several of our object programs describe stress testing as the approach used to reproduce those bugs. Third, stress testing has been well studied as a baseline approach for detecting thread-level concurrency faults [12,54], but it has not been studied with respect to system-level concurrency faults, rendering such a study potentially interesting. Finally, stress testing approaches, unlike RACEPRO, are directly applicable to multiple kernel systems without the need for kernel modifications, and they are capable of detecting races at the hardware level.

While there have been several systematic stress testing techniques proposed for use on multithreaded programs [27,44,55], they require the ability to *observe* and *control* event accesses. However, as discussed in Section 2.2, thread-level techniques that require observability and controllability cannot be directly adapted to the context of testing for system-level races. Therefore, we chose 2 randomized stress testing techniques as baselines for comparison with SIMEXPLORER. In the case of program pairs that do not involve signal handlers or interrupt handlers, the stress testing technique that we used runs a target program pair multiple times and invokes 20 processes (the largest number obtained from bug repositories) for each of the basic and paired programs. In the case of program pairs that do involve signal handlers or interrupt handlers, the stress testing technique that we used runs the basic program multiple times and randomly raises N software signals or hardware interrupts at arbitrary locations in the program during each test run. The number of events N to be issued is generated in terms of the specified event density. Our previous work [56] also showed that appropriately adjusting interrupt density can enhance testing effectiveness. Based on our empirical observations for the object programs (Section 4), SIMEXPLORER chooses the interrupt density to be 60%; ie, the system spends 60% of its instruction cycles executing interrupt handlers. Specifically,

$$N = \frac{T \cdot D}{\sum_{i=1}^{n} T_{H_i}},$$

where T is the program execution time, D is the event density, n the number of event events, and T_{H_i} is the execution time for the event handler H_i.

To determine signal types[‡‡] to be issued, the *Observer* module identified all signal types exercised by the test cases for each object program (eg, SIGINT and SIGCHILD). As in SIMEXPLORER, a signal was raised only when it was determined to be possible to do so by examining system states; otherwise, the signal was postponed to another arbitrary location in the program. In the case of program pairs that do involve interrupt handlers, N is the number of interrupts installed in the basic program.

For each stress testing technique, we used 3 different numbers of runs (levels). At the first level ($S1$), we used the same number of test runs as the number of test runs required by SIMEXPLORER; this lets us examine the relative effectiveness of the 2 approaches on equivalent numbers of test runs. At the second level ($S2$), we used the same amount of testing time (wall clock time) required by SIMEXPLORER; this lets us examine the relative effectiveness of the approaches when each is given the same amount of time. At the third level ($S3$), we used the number of tests run at the second level multiplied by 100; this lets us examine how the effectiveness of SIMEXPLORER compares to that of a more robust stress testing process. In all cases, tests are randomly selected from the sets of tests used by SIMEXPLORER.

To implement the stress testing techniques, we wrote a script that performed the techniques at the 3 levels of effort described in Section 5.2. Columns 5, 6, and 7 list the number of test runs at the first, second, and third levels of stress testing, respectively.

Dependent variables. We consider several dependent variables. Our first dependent variable measures technique *effectiveness* in terms of the number of potential races detected. This variable applies only to SIMEXPLORER, not the stress testing approach, but it lets us evaluate the effectiveness of our phase 2 algorithm at reducing false alarms (RQ1).

Our second dependent variable focuses on technique *effectiveness* measured in terms of the *ability of techniques to detect faults* (RQ2). For both SIMEXPLORER and the stress testing techniques that we consider, we measure the number of observable faults reported by test oracles. These can include both the known faults discovered through inspection of bug repositories and newly discovered faults.

To assess the *efficiency* of SIMEXPLORER (RQ3), we rely on 4 dependent variables, each of which measures 1 facet of efficiency, as follows:

- Our third dependent variable measures the number of test runs needed in phases 1 and 2 of SIMEXPLORER. This number is rel-

[‡‡] Linux has 31 types of signals.

evant only for assessing SIMEXPLORER because our experiment setup (as a matter of proper control) selects test run numbers for stress testing in a manner that attempts to match those used by SIMEXPLORER.

- Our fourth dependent variable is the testing time required by SIMEXPLORER. Although measuring time is undesirable in cases in which there are nondeterministic shared resource accesses among processes, this is not a problem in our case because we use a VM that behaves in a deterministic manner. Here too, however, the number is relevant only for assessing SIMEXPLORER, for reasons similar to those relevant to our first variable.
- Our fifth dependent variable considers the *runtime overhead* of SIMEXPLORER across test runs relative to stress testing; this overhead occurs because stress testing does not use observability or controllability. We calculate runtime overhead by dividing the average runtime per test run required for SIMEXPLORER by that required for stress testing.
- Our sixth dependent variable measures the extra *platform overhead* associated with SIMEXPLORER. This is important because using virtual platforms such as Simics for testing can increase costs, since virtualization times can be longer than execution times on real systems. We calculate platform overhead by dividing the average runtime per test run on Simics by the runtime per test on the real machine. Note that judging whether races are harmful is not taken into account when computing the overhead of SIMEXPLORER because it is independent of techniques for locating harmful races.

5.3 | Study operation

We conducted our study by configuring Simics to virtualize a single core X86 system running Linux 2.6.15 and 3.5.0 kernels. As noted in Section 3, when concurrency faults occur on single core systems, they can also occur in multicore systems, so results obtained on a single core system would be obtainable on a multicore system. To confirm that SIMEXPLORER can function in multicore environments, we did gather data on the UART object on a multicore system simulated by Simics.

To implement the testing process of SIMEXPLORER, as noted above, we ran both basic and paired programs simultaneously with each test case, and for pairs involving event handlers raised events at arbitrary locations in the basic program. As noted in Section 5.2, the number of events is determined by the event density. We collected the runtime trace for each test case and applied the phase 1 algorithm. We next applied the phase 2 algorithm to check for real races based on the potential races identified in the first phase. Finally, we inspected all of the reported real races that did not result in detectable failures to determine whether they were harmful or benign.

5.4 | Threats to validity

The primary threat to external validity for this study involves the representativeness of our programs and test suites. Other programs may exhibit different behaviours and cost-benefit tradeoffs, as may other forms of test suites. However, the programs we investigate are widely used, and the faults we consider are real. Furthermore, the test cases we use either are those provided with the programs or are created using commonly used black-box testing (test specification language) and white-box testing (code coverage) processes, rendering them representative of test cases that could be used in practice by engineers to test these programs.

The primary threat to internal validity for this study is possible faults in the implementation of our approach and in the tools that we use to perform evaluation. We controlled for this threat by extensively testing our tools and verifying their results against smaller programs for which we can manually determine the correct results. We also chose to use popular and established tools (eg, Simics) to implement the various modules in our approach.

Fault detection effectiveness can also depend on the test oracles used. We report failures using output-based oracles. Using internal oracles may increase fault detection effectiveness [57,58], but at the cost of manual effort, testing time, and false positives. Finally, it is possible that faults revealed by our test oracles are due to effects other than the propagation of races. We controlled for this threat by comparing error results to those described in the bug reports; we report only faults related to races.

As an additional threat to internal validity, race manifestation can be influenced by the underlying hardware [59,60]. For example, microprocessors that provide virtualization support may be able to prevent certain races from occurring due to fewer system calls. Our work uses SIMICS, a full platform simulator to provide us with the necessary controllability and observability to cause races. Simics has been widely used to expose difficult-to-reproduce faults including races [61]. The version of SIMICS that we used does not simulate the later Intel processors with hardware virtualization support—a feature that can affect our ability to produce races. Nonetheless, our system was able to detect and reproduce previously documented races exising in our experimental subjects. Therefore, the execution patterns seen using SIMICS should be comparable to those that would be observed in the real systems.

Where construct validity is concerned, numbers of faults and races detected are just 2 variables of interest where effectiveness is concerned. Other metrics such as the cost of trace analysis could be valuable. Regarding efficiency, other variables such as the number of event permutations may also be a variable of interest in some systematic schedule exploration techniques [44,62]; however, this does not provide much information in our case because we permuate 1 event pair each execution such that the permutation occurs only once.

5.5 | Results and analysis

RQ1: Race detection effectiveness of SIMEXPLORER Table 2, columns 3 to 5, report data gathered in our study relative to research question RQ1. The columns report, across the 3 classes of object programs, the number of potential process-level races detected by the phase 1 algorithm, the number of these identified as real races by the phase 2 algorithm, and the number of harmful races identified by further examination, respectively. The phase 2 algorithm reduced the number of false positives contained in the sets of potential races by

TABLE 2 RQ1 and RQ2: Effectiveness results

Program		SIMEXPLORER			Faults detected			
Basic	Paired	Ph.1	Ph.2	Hm.	SE	S1	S2	S3
BASH1	BASH1	13	8	2	1	1	1	1
BASH2	BASH2	9	4	0	0*	0	0	0
TCSH	TCSH	7	4	2	1	0	1	1
STRACE	RACEPROG	3	0	0	0*	0	0	0
UPDATEDB	UPDATEDB	17	3	3	1	0	0	1
UPDATEDB	LOCATE	12	4	1	1	0	0	0
FIND	RACEPROG	4	2	2	1	0	0	0
PYTHON	PYTHON	6	2	2	1	0	0	0
UART	UART	11	7	2	1	0	0	0
MV	RACEPROG	16	6	3	1	0	0	0
PS	GREP	19	13	3	1	0	0	0
LN 5.94	RACEPROG	14	4	2	1	0	0	0
MKDIR1	RACEPROG	10	5	2	1	0	0	0
MKDIR2	RACEPROG	10	5	2	1	0	0	0
MKNOD	RACEPROG	8	6	3	1	0	0	0
MKFIFO	RACEPROG	8	5	2	1	0	0	0
BASH2	SIG	6	3	2	**0***+	0	0	0
BASH3	SIG	5	4	3	1*	0	1	1
CSPLIT	SIG	9	3	3	2*	1	1	1
STRACE	SIG	5	1	1	1	0	0	0
TELNETD	SIG	6	2	2	1	0	0	0
SIG31	SIG	4	4	4	2	0	1	1
EG	SIG	6	6	6	2	1	1	1
NANO	SIG	12	5	2	1	0	0	0
ZSH	SIG	10	4	4	2	0	0	0
ED	SIG	11	6	2	**2**	0	0	0
UART	INT	4	2	2	2+	0	0	0
UART1	INT	4	2	2	1	0	0	0
UART2	INT	4	2	2	1	0	0	0
UART3	INT	4	2	2	1	0	0	0
UART4	INT	4	2	2	1	0	0	0
UART5	INT	4	2	2	1	0	0	0
UART (smp)	INT	4	2	2	1	0	0	0
UART	TIMER	4	2	2	1	0	0	0

52% overall, with reductions ranging from 0% to 100% across the program pairs for all 3 classes. For the first class of objects, the reduction ranged from 38.5% to 100%, with a total reduction of 53.8%. For the second class of objects involving signal handlers, the reduction ranged from 0% to 80%, with a total reduction of 48.6%. For the third class of objects involving interrupt handlers, the total reduction was 50% for all pairs of objects.

RQ2: Fault detection effectiveness of SIMEXPLORER. Table 2, columns 6 to 9, report data gathered in our study relative to research question RQ2. Column 6 reports the number of faults detected by SIMEXPLORER. (Note that the mapping between harmful races and faults can be many-to-one or one-to-one; this is because a single fault such as a missing lock pair may result in multiple races.) Across all program pairs, SIMEXPLORER detected 36 of the 41 previously known faults—a fault detection rate of 87.8%. The 5 cases in which known faults were not detected are indicated by a "*". SIMEXPLORER also detected 2 faults that had not been previously known; these faults were related to a signal handler in BASH2 and are indicated by a "+". Specifically, in the first class of objects, SIMEXPLORER detected 14 of the 16 known faults—a fault detection rate of 87.5%. In the second class of objects, SIMEXPLORER detected 14 of the 16 known faults—a fault detection rate of 87.5%. In the third class of objects, SIMEXPLORER detected all known faults—a fault detection rate of 100%.

There is 1 special case worth noting. One of the known faults missed by SIMEXPLORER, on BASH2 paired with a signal handler (indicated by bold font) failed to appear in the results reported by output-based test oracles. SIMEXPLORER did identify an actual race in this case, but the effects of the race did not propagate to output. A similar case happened on ED, where the real race did not propagate to output.

Columns 7 to 9 in Table 2 report the numbers of faults detected by the 3 levels of stress testing techniques. As the data shows, in total, stress testing detected only 6 of the 41 known faults—a fault detection rate of 14.6%. For program pairs (BASH1, BASH1), (CSPLIT, SIG), and (EG, SIG), all 3 stress testing levels detected the same known faults. For program pairs (BASH3, SIG), (TCSH, TCSH), and (SIG31, SIG), levels *S*2 and *S*3 detected the same known fault while level *S*1 detected none. For program pairs (UPDATEDB, UPDATEDB), only *S*3 detected the known fault. The data shows that given longer running times, stress testing may demonstrate more fault detection effectiveness than with shorter times; however, at all 3 levels, SIMEXPLORER performed as well as or better than stress testing.

RQ3: Efficiency of SIMEXPLORER. Table 3 provides data relevant to research question RQ3. Column 3 lists the number of test runs for SIMEXPLORER's 2 phases, ranging from 6 to 68. (The approach by which the number of test runs 3 determined has been described in Section 3.5.) The numbers in parentheses in this column indicate the numbers of test runs in phase 2, ranging from from 2 to 32. Overall, SIMEXPLORER requires only 40.3% additional test runs in the second phase, beyond those needed in the first phase.

Column 4 of Table 3 (TT-SE) reports the time required for all test runs reported in column 3, in seconds. Across all program pairs, the testing time ranges from 1.0 second to 669.6 seconds (11.2 min). The minimum testing time occurred in TELNETD, while the maximum time occurred on UPDATEDB, respectively. These testing times are very reasonable given the difficulty associated with detecting race conditions and imply that SIMEXPLORER is efficient with respect to testing time.

Columns 5 to 7 of Table 3 report the numbers of test runs used by stress testing. These numbers are tied to those observed for SIMEXPLORER as explained in Section 5.2, and by themselves indicate little, but are used in the discussion of dependent variable 5 that follows.

Column 8 of Table 3 indicates the average runtime overhead of SIMEXPLORER across all test runs. Overall, the average runtime overhead of SIMEXPLORER relative to stress testing ranged from 1.3 to 33.3, but in most cases (all but 6) was less than 6.2. We consider these overheads to be acceptable for in-house testing, which is the intended usage scenario of SIMEXPLORER.

Column 9 of Table 3 lists the average platform overhead associated with SIMEXPLORER across all test runs. As the table shows, the average platform overhead ranged from 12.4 to 35.8. Considering the benefits of virtual platforms and the difficulty of detecting race conditions, such overhead is trivial.

TABLE 3 RQ3: Efficiency data

Program		TR-SE	TT-SE	S1	S2	S3	OH_{sim}	OH_m
Basic	Paired							
BASH1	BASH1	42 (12)	87.3	42	170	17000	2.0	21.1
BASH2	BASH2	46 (16)	103.6	46	186	18600	1.8	21.0
TCSH	TCSH	37 (32)	28.7	9	37	3700	1.3	24.5
STRACE	RACEPROG	59 (10)	90.5	59	255	25500	2.8	30.2
UPDATEDB	UPDATEDB	35 (10)	570.4	35	144	14400	4.0	32.3
UPDATEDB	LOCATE	40 (8)	669.7	40	167	16700	4.2	33.1
FIND	RACEPROG	41 (16)	12.0	41	180	18000	1.5	19.5
PYTHON	PYTHON	41 (29)	10.1	41	180	18000	17.8	32.8
UART	UART	18 (2)	16.7	18	75	7500	4.5	15.6
MV	RACEPROG	52 (7)	159.3	52	290	29000	1.8	20.5
PS	GREP	64 (7)	58.4	64	365	36500	6.2	24.2
LN	RACEPROG	49 (21)	9.6	49	320	32000	33.3	19.2
MKDIR1	RACEPROG	45 (22)	7.8	45	195	19500	25.4	18.3
MKDIR2	RACEPROG	45 (22)	7.8	45	197	19700	22.3	17.4
MKNOD	RACEPROG	30 (14)	6.1	30	136	13600	22.3	17.4
MKFIFO	RACEPROG	22 (5)	5.2	22	24	2400	4.6	17.0
BASH2	SIG	38 (8)	97.3	38	184	18400	1.9	20.2
BASH3	SIG	36 (6)	77.6	36	127	12700	1.6	20.1
CSPLIT	SIG	51 (21)	10.4	36	25	2500	4.3	25.4
STRACE	SIG	68 (19)	90.2	59	253	25300	2.4	22.0
TELNETD	SIG	49 (10)	30.4	24	98	9800	3.2	28.1
SIG31	SIG	6 (3)	1.9	6	55	5500	2.9	12.4
EG	SIG	7 (2)	2.5	9	78	7800	3.1	13.2
NANO	SIG	49 (19)	14.4	49	41	4100	2.8	35.8
ZSH	SIG	41 (9)	16.8	41	42	4200	2.5	22.2
ED	SIG	54 (9)	19.6	54	48	4800	2.4	29.5
UART	INT	19 (7)	16.5	19	41	4100	2.5	17.2
UART1	INT	19 (7)	16.3	19	41	4100	2.5	17.0
UART2	INT	19 (7)	16.5	19	41	4100	2.5	17.0
UART3	INT	19 (7)	16.4	19	41	4100	2.5	17.1
UART4	INT	19 (7)	16.5	19	41	4100	2.5	17.0
UART5	INT	19 (7)	16.2	19	41	4100	2.5	17.0
UART (smp)	INT	25 (11)	18.3	25	41	4100	2.2	17.2
UART	TIMER	23 (9)	18.1	23	41	4100	2.2	17.0

6 | DISCUSSION

SIMEXPLORER was effective at detecting process-level race conditions. Meanwhile, SIMEXPLORER was efficient in terms of the number of test runs and testing time required. While SIMEXPLORER incurrs technique and platform overhead with respect to stress testing, additional runs of stress testing do not render testing more effective in most cases. As such, when we consider both fault detection rate and testing time, SIMEXPLORER is more cost-effective. In addition, the costs of testing can be amortized into the development workflow when using VMs.

We now explore additional observations relevant to our study.

6.1 | Faults detected

In the cases we studied, SIMEXPLORER detected many interesting faults; these are summarized in Tables 4 and 5. Table 4 describes races that happen between processes, and Table 5 describes races involving asynchronous system events. We describe both the consequences (ie, the program behaviour) and the root causes of the races. The second column of Table 5 describes the types of signals required to expose the faults.

In Table 4, three cases involve races that occur between 2 instances of 1 program (BASH1, BASH1), (TCSH, TCSH), (UPDATEDB, UPDATEDB), and (UART, UART). This implies that testing should be done by pairing instances stemming from the same program, if the program accesses system-level shared resources. On the (PYTHON, PYTHON) pair, the order of accessing shared resources between parent and child processes matters. This implies that programs that fork child processes should be tested for race conditions. On the (PS, GREP) pair, the race involves accessing kernel-level objects, which indicates that modeling system-level shared resources at lower levels is necessary. For the rest of the programs, races involve participation of another program. In practice, engineers can use SIMEXPLORER to first obtain shared

TABLE 4 Harmful Process-level races detected by SimExplorer

Program	Consequence	Description
BASH1, BASH1 TCSH, TCSH UPDATEDB, UPDATEDB	Corrupted history files [19,20]. Corrupted database files [20].	Two processes access the same resource at the same time.
UPDATEDB, LOCATE FIND, RACEPROG MV, RACEPROG LN, RACEPROG	Premature freeing of resources [21,63,64,65].	The RACEPROG process attempts to open the file in between the operations, causing the `open` to fail. (eg, the MV process calls `unlink` before `rename`, and they are not atomic [64]).
PS, GREP	When the command "`ps aux | grep string`" the command may (incorrectly) print the command "`grep string`" itself [66].	The `execve` of GREP and the `read` of PS both access the inode of `/proc/pid/cmd_line`, where `pid` corresponds to the GREP process. If the `read` happens after the `execve` is run, the GREP command itself will be printed.
PYTHON, PYTHON	When the 2 child processes are spawned, they do not exit correctly.	The check of child process return code (`exit`) and the wait for the child process (`waitpid`) are not atomic. As a result, the wait system call fails when the child process already returns and exits.
MKDIR1, RACEPROG MKDIR2, RACEPROG MKNOD, RACEPROG MKNOD, RACEPROG	`raceprog` slips in and changes the file permission, in between the file permission check and the use of the file in the basic program. [67,68].	TOCTTOU (time of check to time of use) races. A change is made by 1 process between the time at which the condition is checked and the time at which the results of that check are used by another process.
UART, UART	Data transmission is not synchronized.	IER is not disabled (See Figure 5).

Abbreviation: IER, interrupt enable register.

TABLE 5 Harmful asynchronous races detected by SimExplorer

Program	Event Type	Consequence	Description
STRACE, SIG	SIGINT	Error messages	A signal arrives before the statement `files_created` in the file deletion routine [69] is executed
CSPLIT, SIG	SIGINT	The target process is not cleanly detached(ie, it does not print correct messages).	SIGINT arrives to kill a process before an STRACE system call can complete a write to the */proc* directory.
BASH2, SIG	SIGINT	The history file is not saved when BASH terminates.	The signal handler reads the incorrect value of `history_lines_this_session` system call can complete a write to the */proc* directory.
TELNETD, SIG	SIGINT	Deadlock	The signal handler enters the `cleanup()` and grabs the lock while the lock file has already be obtained.
ZSH, SIG ED, SIG	SIGINT and SIGTERM	Error messages.	The signal is not disabled during the execution of the signal handler. The shared variable is modified by the reentrant or other signal handlers.
SIG31, SIG	SIGHUP and SIGTERM	Memory corruption.	SIGTERM is not disabled during the execution of the SIGHUP handler. The memory is prematurely released
EG, SIG	SIGINT	Memory corruption.	Calling `longjmp` inside a signal handler. Program control is transferred, causing invalid memory to be written.
UART, INT	INT	Deadlock [70].	The interrupt is not disabled. The spin lock in the `serial8250_handle_port` is preempted.
UART(SMP), INT	INT, INT	Error messages.	The interrupt handler is self-preempted. The shared variable in the ISR is modified.
UART(0,1,2,3,4,5), INT	INT	Error messages.	The interrupt is not disabled. The shared variable in the main program is modified by the interrupt handler.
UART, TIMER	INT, INT	Error messages.	Nested interrupt. The shared variable in the UART ISR is modified.

Abbreviation: ISR, interrupt service routine.

resource accesses for the program under test, and then write the paired program accessing these resources. It is also possible to automate this task, something we will consider in future work.

As Table 5 shows, in 7 cases races can be detected by invoking 1 event handler. The (UART(0,1,2,3,4,5), INT) pairs include both an original UART and a mutated UART. In the original UART,

SIMEXPLORER detected a unknown fault. Specifically, the shared variable in the `uart_write_room` function is modified by the interrupt handler. The other 4 cases involve more than 1 event handler. The implication of this is that event types should be considered when testing for asynchronous races.

6.2 | Benign races

When a race occurs but cannot cause any observable fault with any inputs, it is benign [71]. In this study, we found several benign races; these are described in Table 6.

On program pair (BASH1, BASH1), the 2 processes each write to `dev/null`, which is a device file that discards all data written to it but reports that the write operation succeeded. Such write operations do not disturb the system, and faults related to them will never occur. In fact, our technique can be tuned to exclude `dev/null` as an *SR* of interest.

In the case of BASH3, the race is benign because the modified variable is restored before being read by the handler. In the case of UART, since both kernel threads write the same value to the *SR*, even though races occur, the value read is always the same. This type of benign race can be identified by precise program analyses such as symbolic execution [72]. For example, if the latest write (ie, variable restoration) before the read *r* in the handler shares the same symbolic value as the write event *w* racing with *R* in the main program, the <r, w>pair is benign. However, precise program analysis can be expensive.

On program pair (NANO, SIG), 2 races were identified as benign. The first occurs when SIGWINCH modifies global variables COLs and LINES and is benign because the signal handler is expected to resize the window by writing the global variables. The second occurs when SIGINT modifies `pager_input_aborted`. This variable is a flag used to check whether the pager mode is "exit". Similarly, on program pair ED, SIG, the flag variable `sigint_pending` is used to check whether a SIGINT variable has been received. This type of benign race stems from non-standard synchronizations, which can be eliminated by user annotations. While techniques have been proposed to automatically identify such synchronizations, it is unclear whether these techniques can be applied in this context.

6.3 | Faults not detected

We also examined the faults that were not detected. On the (BASH2, BASH2) program pair, a race is exposed when the 2 BASH processes run under different sessions (a parent BASH and a child BASH) [73]. Initially, we did not consider this scenario in our study. However, once we created such a scenario to reproduce the fault, SIMEXPLORER*did detect this race*. The lesson to be learned here is that engineers may do well to realistically explore potential execution scenarios (eg, considered sessions and simultaneous logins) because doing so may uncover additional faults.

On the (BASH2, SIG) program pair, SIMEXPLORER detected a race that did not propagate to output. This occurred because BASH and the signal handler both wrote the same value to a shared memory location [74]. A different test may cause the signal handler to write a different value, potentially causing the program to fail. The lesson to be learned here is that our approach should consider redundant writes as a criterion to eliminate tests as part of the selection process in phase 2 (see Section 3.5).

On the (BASH3, SIG) program pair, the fault related to 1 race [75] was not detected because our tests did not exercise the shared variable that can cause the race to occur. Subsequently, we manually created a test to exercise this shared variable and SIMEXPLORER was then able to detect this race. The lesson to be learned here is that the effectiveness of our approach can depend on the initial test suite and the oracles used. Selecting a proper adequacy criterion (eg, data-flow) could increase the effectiveness of our approach.

On the (STRACE, RACEPROG) program pair, the race occurs only on an IA64 platform whereas our experiments were conducted on an X86 platform [76]. This confirms an argument made by Laadan et al. [6] that process-level races are sensitive to system configurations. The result also suggests that engineers may wish to test under different system configurations.

6.4 | Improving efficiency

To reduce costs, 1 important characteristic of our technique is that checking is performed before issuing a controlled event. Without checking, SIMEXPLORER would run the program until the end and rerun the program to find a different point at which to issue the event if the scheduling failed. When an *SR* is accessed in the program, the

TABLE 6 Benign races detected by SimExplorer

Program	Event Type	Description	Reason for being benign
BASH1, BASH1	-	When 2 BASH processes issued an `open("dev/null", O_TRUNC)` system call with the `O_TRUNC` option set (file length is truncated to 0), the 2 system calls race because they both write to the entire data range of the inode of `dev/null` without proper synchronization.	They are each immediately followed by a `close` system call.
UART, UART	-	Two threads write to the IIR register without proper synchronization.	Write the same value.
BASH3, SIG	SIGINT	The signal handler modifies a global variable of BASH.	The old value of this variable is restored before it is read again by the handler.
NANO, SIG ED, SIG	SIGWINCH SIGINT	The shared variable is modified by the signal handler.	The global variable is flag used to check whether a condition is met.

controllability module first checks to see whether it is possible to issue a signal or an interrupt, and if not, it proceeds to the next possible location. This approach can save test runs, but at the cost of checking. To quantify the tradeoffs involved, we measured the amount of time required to finish testing without the checking step enabled. Recall that Table 3 displays testing time with checking enabled. With checking disabled, on the other hand, the testing time did increase, ranging from 112.4% to 201.75%. Clearly, the checking approach saves time overall.

6.5 | Further discussion

System-specific properties. Our experiment with SIMEXPLORER was conducted on a system running on Linux. As such, the runtime properties including shared resources, system calls, and schedulers that we considered are Linux specific. Detecting different types of elusive faults does require engineers to look deeper into system-specific properties. In practice, however, companies tend to focus on specific systems, so they could implement our approach on those systems. Thus, the general idea of our approach should translate to other systems.

Supporting multiple languages. By operating at the binary level, our approach is not language dependent, even though our work thus far has focused on C/C++ code. In most modern languages including Java and JavaScript, dynamic compilation is often used to translate portable code (eg, Java bytecode) to binary prior to execution. Such binaries are fully observable within our system. Furthermore, most language virtual machines separate VM code and application executables by address space (eg, VM code resides in the text section while application code tends to reside in the code-cache, which is part of the heap); our approach would allow us to separately or collectively tackle concurrency issues that may exist in the VM, its libraries, and applications.

System evolution. Modern software systems evolve (eg, through software updates, availability of new configurations, and creation of new applications). SIMEXPLORER is designed with evolution in mind. When a new application is installed in a system, it is necessary to test the interactions between that application and other applications. One approach is to test the entire system again [51]. With SIMEXPLORER, developers can analyze the resource usage of this new application with previously tested applications. Only applications that share resources with this new application need to be tested again.

Environment dependencies. Given the same test input, it is possible that a potential race pair computed in phase 1 is not reachable in phase 2. Such a scenario can occur if the system environment or the status of a device changes (eg, in the case of a buggy device port), causing a program to take a path that does not access the targeted shared resource. We did not encounter any such cases in our study. This is because by using a virtual platform, we can maintain environment consistency across different runs.

Data injection.SIMEXPLORER injects data into device ports and forces an interrupt handler to execute 1 path. The data we inject is the same as the test input given to the program. For example, if an application sends the string "hello" to the UART console passed by UART *transmitter buffer*, a controllability module would inject "world" into the UART *transmitter buffer* to force an interrupt to occur after a certain access. It is also possible to have multiple paths by which shared variables can exist in interrupt handlers. Testers can extend our method by forcing interrupt handlers to execute different paths, which may increase the probability of revealing faults.

7 | RELATED WORK

There have been several approaches created for detecting races that result in various concurrency violations including time of check to time of use (TOCTOU) [77,78]. TOCTOU races typically happen when the permission check and use of a file in 1 process is not atomic so that a malicious process can slip in and attack the file. However, these techniques are nontesting-based approaches; the techniques depend on the current execution at deployment time. In addition, TOCTOU races are just 1 specific type of race while our technique deals with general races.

There has been a great deal of work on analyzing, detecting, and testing for thread-level data races [1–5,8,11,13,25,26]. We have discussed this in Section 2.2. However, as discussed in Section 2.2, existing techniques cannot be directly adapted to handle process-level races.

Nonetheless, the ideas behind existing thread-level approaches can be useful in our context if system call effects are modeled as read/write operations. For instance, we may leverage the approaches used in CHESS and CONTEST to ensure exhaustive interleavings of processes. Work by Maiya et al. [25] considers asynchronous event dispatch in addition to thread interleaving as a way to reduce false positives for race detection in Android applications. Similarly, SIMEXPLORER also considers event dispatch based on signals and system calls to distinguish real races from potential races.

There has been some work on testing for races caused by software signals and interrupts [33,79–81]. For example, Tahara et al. [80] issue signals when a PUT attempts to access a global variable, amplifying the chance of exposing faults. However, this technique does not attempt to reorder the global variable access and the signal handler. Thus, it is possible that the signal handler and the global variable will be synchronized, resulting in false positives. Again, these techniques can handle only signal races while SIMEXPLORER focuses on general races.

More recent approaches [82–84] have built on the 2-phase algorithm used by RACEFUZZER. Cai et al. [82] identify thread scheduling constraints to avoid checking for real deadlock in the confirmation phase; this can reduce the cost of testing. Eslamimehr et al. [84] use concolic testing to search for inputs and schedules to confirm potential races. Machado et al. [85] use symbolic execution to explore different scheduling possibilities to find a failing multithreaded schedule to aid debugging. Again, these approaches focus on thread-level concurrency faults. Nonetheless, it might be possible to improve SIMEXPLORER by leveraging the algorithms used in these approaches.

Dimitrov et al. [28] define a *commutative race* as a race between 2 operations that "do not commute and can happen in any order." They develop a commutativity race detector to detect such races in Java libraries. To help with detection, they specify each operation as in terms of a set of access points that it accesses. In SIMEXPLORER, races due to signals and system calls can be viewed primarily as a form of commutative race. The way we model system calls and signals also yield similar results to Dimitrov et al.'s access point representation.

7.1 | Our own prior work

As noted in Section 1, this article is a revised and expanded version of a previously published conference paper [16]. We now provide details on how this article differs from that, and what its additional contributions include.

Note, initially, that when the conference version of this work is placed into JSTVR format, it is 18 pages long, plus 1 page for references. In contrast, this article is 33 pages long, plus 2 pages for references. This is an 83.3% expansion, well over the 30% normally expected of revised and expanded conference papers.

Length, of course, is an imperfect measure of contribution, but in this case, the added length does reflect substantial new intellectual contributions, as follows.

First, we have improved our algorithms in 3 ways. (1) The algorithm presented in [16] did not handle hardware interrupts. This article introduces our new 2-phase algorithm and applies it to interrupt-related races. This extension is nontrivial because dealing with interrupts requires observing and controlling hardware states, which is technically different from handling process-level races. (2) Our new algorithm handles nested interrupts and nested software signal handlers; the prior ones did not. (3) Our new algorithm handles all types (31) of Linux signal handlers; the prior algorithms handled just a single type of signal handler. Our new algorithm, thus, represents a substantial improvement over the prior algorithms, vastly extending the power of our approach to detect significant classes of concurrency faults.

A second substantial enhancement to this article involves its empirical study. The study in [16] was concerned with only process-level races. This study is included in this article, but whereas the prior empirical study considered just 16 object programs, we now consider 24. Moreover, these new objects allow us to study the ability of our approach to detect concurrency faults of additional classes, as well as to empirically study our extended algorithms on a more substantial base of artifacts. Furthermore, locating and preparing object programs for studies involves substantial effort; these new objects are now available for others to use in experiments and thus represent a substantial additional contribution.

A third enhancement involves our inclusion of Section 4, which provides details related to the implementation of our approach; this will allow other researchers to replicate our approach and studies.

A fourth enhancement involves Section 6, which contains more than twice as much material as the corresponding section in the conference paper. The new material includes additional information on the races we were able to detect in the empirical study, on benign races, on technique efficiency, and on several additional aspects of the approach that were not discussed at all in the earlier paper.

Last, we have made a large number of changes related to the presentation of the work. This includes 3 more motivating examples in Section 2 that provide additional details on the types of races our technique can handle. It also includes, in the sections presenting our empirical work, more detailed descriptions of our objects of study and the types of data races our approach detected. Finally, we have modified the abstract and introduction of this article to reflect the added content and spell out the novelty of our approach, and we have improved our discussion of related work.

8 | CONCLUSION

We presented SIMEXPLORER, a framework for effectively testing for process-level races. We have conducted an empirical study applying SIMEXPLORER to 24 user-level programs and one device driver program. We have empirically compared SIMEXPLORER to traditional stress testing techniques, and our results suggest that SIMEXPLORER is more effective than these techniques at detecting faults that are caused by process-level races, with an overhead that is certainly acceptable given the improvement in effectiveness.

Some faults in our objects of study were not detected due to characteristics of the test cases and system environment used. In future work, we intend to investigate the possibility of automatically creating test inputs and setting up system environments that allow us to effectively expose system-level data races. As discussed earlier, many current approaches for dynamic race detection could be applied to process-level races once system call effects are modeled. We will leverage existing approaches (eg, CHESS) to ensure more exhaustive interleavings of processes. We will also leverage the algorithm in AVIO to detect atomicity violations involving multiple processes.

ACKNOWLEDGEMENTS

This work has been supported in part by National Science FoundationCCF-1464032, the Air Force Office of Scientific Research through award FA9550-10-1-0406, the Army Research Office through award W911NF-13-1-0154, DARPA through award FA8750-14-2-0053, and Maryland Procurement Office under agreement numbers H98230-14-C-0140. We thank Supat Rattanasuksun for helping with empirical study.

REFERENCES

1. M. D. Bond, K. E. Coons, and K. S. McKinley, *PACER: Proportional detection of data races*, Proceedings of the Conference on Programming Languages, Design and Implementation, ACM, Toronto, 2010, pp. 255–268.
2. J. Erickson, M. Musuvathi, S. Burckhardt, et al., *Effective data-race detection for the kernel*, Proceedings of the Confernece on Operating System Design and Implementation, ACM, Vancouver, 2010, pp. 151–162.
3. R. Raman, J. Zhao, V. Sarkar, et al., *Scalable and precise dynamic datarace detection for structured parallelism*, Proceedings of the Conference on Programming Language Design and Implementation, ACM, Beijing, 2012, pp. 531–542.
4. T. Sheng, N. Vachharajani, S. Eranian, et al., *RACEZ: A lightweight and non-invasive race detection tool for production applications*, Proceedings of the International Conference on Software Engineering, ACM, Honolulu, 2011, pp. 401–410.
5. S. Hong, M. Staats, J. Ahn, et al., *Are concurrency coverage metrics effective for testing: A comprehensive empirical investigation*, J. Software Testing, Verification Reliab. **25** (2015), no. 4, 334–370.
6. O. Laadan, N. Viennot, C-C Tsai, et al., *Pervasive detection of process races in deployed systems*, Proceedings of the Symposium on Operating Systems Principles, ACM, Cascais, 2011, pp. 353–367.
7. *Facebook IPO Glitch*, 2012. http://www.cio.com.au/article/425234/nasdaq/_facebook/_glitch/_came/_from/_race/_conditions/.
8. S. Savage, M. Burrows, G. Nelson, et al., *Eraser: A dynamic data race detector for multithreaded programs*, ACM Trans. Comput. Syst. **15** (1997), no. 4, 391–411.

9. T. Yu, W. Srisa-an, and G. Rothermel, *Simrt: An automated framework to support regression testing for data races*, Proceedings of the International Conference on Software Engineering, ACM, Hyderabad, 2014, pp. 48–59.
10. J. Huang and C. Zhang, *Persuasive prediction of concurrency access anomalies*, Proceedings of the International Symposium on Software Testing and Analysis, ACM, Toronto, 2011, pp. 144–151.
11. D. Engler and K. Ashcraft, *RacerX: Effective, static detection of race conditions and deadlocks*, Proceedings of the Symposium on Operating Systems Principles, ACM, Bolton, 2003, pp. 237–252.
12. M. Musuvathi, S. Qadeer, T. Ball, et al., *Finding and reproducing heisenbugs in concurrent programs*, Proceedings of the Conference on Operating Systems Design and Implementation, ACM, San Diego, 2008, pp. 267–280.
13. K. Sen, *Race directed random testing of concurrent programs* , Proceedings of the Conference on Programming Language Design and Implementation, 2008.
14. J. Yu, S. Narayanasamy, C. Pereira, et al., *Maple: a coverage-driven testing tool for multithreaded programs*, Proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications, ACM, Tucson, 2012, pp. 485–502.
15. M. Eslamimehr and J. Palsberg, *Sherlock: Scalable deadlock detection for concurrent programs*, Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, Hong Kong, 2014, pp. 353–365.
16. T. Yu, W. Srisa-an, and G. Rothermel, *SimRacer: An automated framework to support testing for process-level races*, Proceedings of the International Symposium on Software Testing and Analysis, ACM, Lugano, 2013, pp. 167–177.
17. J. Engblom, D. Aarno, and B. Werner, *Full-System Simulation from Embedded to High-Performance Systems*, 2010.
18. D. A. Wheeler, *Secure programming for linux and unix howto*, 2003.
19. 2004. https://bugs.launchpad.net/debian/+source/bash/+bug/10809/comments/0.
20. 2011. http://www.mail-archive.com/debian-bugs-dist@lists.debian.org/msg918886.html.
21. 2000. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=67782.
22. *Apache Deadlock*, 2003. http://marc.info/?l=apache-httpd-bugs&m=105967988713871.
23. *Android Race*, 2010. https://android.googlesource.com/platform/frameworks-/base/+/e6b1bbd8acca3f6e174c24cf4eb23a66db2d08a2.
24. *UART Race*, 2011. http://lkml.org/lkml/2011/7/25/369.
25. P. Maiya, A. Kanade, and R. Majumdar, *Race detection for android applications*, Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Pittsburgh, 2014, pp. 316–325.
26. E. Farchi, Y. Nir, and S. Ur, *Concurrent bug patterns and how to test them*, Proceedings of the International Symposium on Parallel and Distributed Processing, 2003, pp. 22–26.
27. S. Park, S. Lu, and Y. Zhou, *Ctrigger: exposing atomicity violation bugs from their hiding places*, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, IEEE , Nice, 2009, pp. 25–36.
28. D. Dimitrov, V. Raychev, M. Vechev, et al., *Commutativity race detection*, Proceedings of the Conference on Programming Language Design and Implementation, ACM, Washington, DC, 2014, pp. 305–315.
29. K. Serebryany and T. Iskhodzhanov, *ThreadSanitizer: Data race detection in practice*, Proceedings of the Workshop on Binary Instrumentation of Applications, ACM, New York, 2009, pp. 62–71.
30. P. Thomson, A. F. Donaldson, and A. Betts, *Concurrency testing using schedule bounding: An empirical study*, Proceedings of the Symposium on Principles and Practices of Parallel Programming, ACM, Orlando, 2014, pp. 15–28.
31. C. University, *RacePro on GitHub*. https://github.com/columbia/racepro [Accessed: 2016-01-07].
32. B. Kasikci, C. Zamfir, and G. Candea, *Data races vs. data race bugs: Telling the difference with portend*, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, 2012, pp. 185–198.
33. T. Yu, W. Srisa-an, and G. Rothermel, *Simtester: a controllable and observable testing framework for embedded systems*, Proceedings of the International Conference on Virtual Execution Environments, ACM, London, 2012, pp. 51–62.
34. K. Serebryany, A. Potapenko, T. Iskhodzhanov, et al., *Dynamic race detection with llvm compiler*, Proceedings of the Second International Conference on Runtime Verification, RV'11, San Francisco, CA, 2012, pp. 110–114.
35. D. Marino, M. Musuvathi, and S. Narayanasamy, *Literace: Effective sampling for lightweight data-race detection*, Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, Dublin, Ireland, 2009, pp. 134–143.
36. M. Arnold, M. Vechev, and E. Yahav, *Qvm: An efficient runtime for detecting defects in deployed systems*, Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, Nashville, TN, USA, 2008, pp. 143–162.
37. S. V. Adve, M. D. Hill, B. P. Miller, et al., *Detecting data races on weak memory systems*, Proceedings of the 18th Annual International Symposium on Computer Architecture, ISCA '91, Toronto, Ontario, Canada, 1991, pp. 234–243.
38. J. Mellor-Crummey, *On-the-fly detection of data races for programs with nested fork-join parallelism*, Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, Albuquerque, New Mexico, USA, 1991, pp. 24–33.
39. J. He, *Linux system call quick reference*.
40. F. Sorrentino, A. Farzan, and P. Madhusudan, *Penelope: weaving threads to expose atomicity violations*, Proceedings of the International Symposium on Foundations of Software Engineering, ACM, Santa Fe, 2010, pp. 37–46.
41. C-S Park and K. Sen, *Randomized active atomicity violation detection in concurrent programs*, Proceedings of the International Symposium on Foundations of Software Engineering, ACM, Atlanta, 2008, pp. 135–145.
42. S. Lu, J. Tucek, F. Qin, et al., *Avio: detecting atomicity violations via access interleaving invariants*, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, San Jose, 2006, pp. 37–48.
43. C. Hammer, J. Dolby, M. Vaziri, et al., *Dynamic detection of atomic-set-serializability violations*, Proceedings of the international conference on software engineering, 2008, pp. 231–240.
44. M. Musuvathi and S. Qadeer, *Iterative context bounding for systematic testing of multithreaded programs*, Proceedings of the Conference on Programming Languages, Design and Implemenatation, ACM, San Diego, 2007, pp. 446–455.
45. S. Dorankula, GuyMartin Tchamgoue, and Y-K Jun, Efficient detection of first races in concurrent signal handlers, *Computer Applications for Graphics, Grid Computing, and Industrial Environments*, 2012.
46. *What's the race condition in these two interrupt service routines?*, 2012. http://stackoverflow.com/questions/13078394/whats-the-race-condition-in-these-two-interrupt-service-routines.
47. *Fix race between interrupt handler and tasklet*, 2013. https://ccsinfo.com/forum/viewtopic.php?p=170718.
48. A. Causevic, D. Sundmark, and S. Punnekkat, *An industrial survey on contemporary aspects of software testing*, Proceedings of the International Conference on Software Testing, Verification and Validation, IEEE, Paris, 2010, pp. 393–401.
49. Thomas Ostrand J. and Mark Balcer J., *The category-partition method for specifying and generating fuctional tests*, Commun. ACM **31** (1988), no. 6, 676–686.
50. S. Lu, W. Jiang, and Y. Zhou, *A study of interleaving coverage criteria*, Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, Dubrovnik, 2007, pp. 533–536.

51. *Test Strategies for Smartphones and Mobile Devices*, 2010. http://www.macadamian.com/images/uploads/whitepapers-/MobileTestStrategie/_Aug2010.pdf.
52. *Stress Testing*, 2006. http://msdn.microsoft.com/en-us/magazine/cc163613.aspx.
53. *Web Site Test Tools and Site Management Tools*, 2012. http://www.softwareqatest.com/qatweb1.html.
54. S. Park, S. Lu, and Y. Zhou, *CTrigger: Exposing atomicity violation bugs from their hiding places*, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, Washington, DC, 2009, pp. 25–36.
55. P. Fonseca, R. Rodrigues, and B. B. Brandenburg, Ski: exposing kernel concurrency bugs through systematic schedule exploration, *The USENIX Symposium on Operating Systems Design and Implementation*, ACM, Broomfield, 2014, pp. 415–431.
56. T. Yu, W. Srisa-an, and G. Rothermel, *Simlatte: A framework to support testing for worst-case interrupt latencies in embedded software*, Proceedings of the International Conference on Software Testing, Verification, and Validataion, IEEE, Cleveland, 2014, pp. 312–322.
57. M. Staats, M. W. Whalen, and M. P. E. Heimdahl, Programs, tests, and oracles: The foundations of testing revisited, *Int'l. Conf. Softw. Eng.*, ACM, Honolulu, 2011, pp. 391–400.
58. T. Yu, A. Sung, W. Srisa-an, et al., *Using property-based oracles when testing embedded system applications*, Proceedings of the International Conference on Software Testing, Verification and Validation, IEEE, Berlin, 2011, pp. 100–109.
59. L. Osterman, *Larry Gets Taken to Task on Concurrency*, 2005. https://blogs.msdn.microsoft.com/larryosterman/2005/02/11/larry-gets-taken-to-task-on-concurrency/.
60. *SSE Instructions: Which CPUs Can Do Atomic 16B Memory Operations?*, 2014. http://stackoverflow.com/questions/7646018/sse- instructions-which-cpus-can-do-atomic-16b-memory-operations.
61. J. Engblom, *Systematically exposing os kernel races an interview with ben blum*, 2012. http://blogs.windriver.com/tools/2012/09/systematically-exposing-os-kernel-races-an-interview-with-ben-blum.html.
62. S. Bindal, S. Bansal, and A. Lal, *Variable and thread bounding for systematic testing of multithreaded programs*, Proceedings of the 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, Lugano, Switzerland, 2013, pp. 145–155.
63. 2011. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=461585.
64. 2008. https://bugzilla.redhat.com/show/_bug.cgi?id=438076.
65. 2006. http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=357140.
66. 2001. https://bugzilla.redhat.com/show/_bug.cgi?id=54127.
67. 2005. http://www.securityfocus.com/archive/1/395489.
68. 2007. https://bugs.launchpad.net/ubuntu/+source/coreutils/5.97-5.6ubuntu1.
69. 2004. http://lists.gnu.org/archive/html/bug-coreutils/2004-04/msg00126.html.
70. K. Trap, *deadlock in 3c59x driver. Web page*. http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=4bf3631cdb012591667ab927fcd7719d92837833.
71. S. Narayanasamy, Z. Wang, J. Tigani, et al., *Automatically classifying benign and harmful data races using replay analysis*, Proceedings of the Conference on Programming Language Design and Implementation, ACM, San Diego, 2007, pp. 22–31.
72. C. Cadar, D. Dunbar, and D. Engler, *KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs*, Proceedings of the Conference on Operating Systems Design and Implementation, ACM, San Diego, 2008, pp. 209–224.
73. 2009. https://bugs.gentoo.org/show/_bug.cgi?id=231775.
74. 2008. http://ftp.gnu.org/gnu/bash/bash-3.2-patches/bash32-051.
75. 2011. http://ftp.gnu.org/gnu/bash/bash-4.2-patches/bash42-023.
76. 2010. https://rhn.redhat.com/errata/RHBA-2010-0174.html.
77. M. Payer and T. R. Gross, *Protecting applications against tocttou races by user-space caching of file metadata*, Proceedings of the International Conference on Virtual Execution Environments, ACM, London, 2012, pp. 215–226.
78. D. Tsafrir, T. Hertz, D. Wagner, et al., *Portably solving file TOCTTOU races with hardness amplification*, Proceedings of the USENIX Conference on File Storage Technologies, ACM, San Jose, 2008.
79. G. M. Tchamgoue, O-K Ha, K-H Kim, et al., *Lightweight labeling scheme for on-the-fly race detection of signal handlers*, Ubiquitous Computiong Multimedia Appl. (2011), 201–208.
80. T. Tahara, K. Gondow, and S. Ohsuga, *DRACULA: Detector of data races in signals handlers*, Proceedings of the Asia-Pacific Software Engineering Conference, 2008.
81. L. G. Murillo, R. L. Bucs, D. Hincapie, et al., *Swat: Assertion-based debugging of concurrency issues at system level*, Proceedings of the Design Automation Conference, IEEE, Chiba, 2015, pp. 600–605.
82. Y. Cai, S. Wu, and W. K. Chan, *Conlock: A constraint-based approach to dynamic checking on deadlocks in multithreaded programs*, Proceedings of the International Conference on Software Engineering, ACM, Hyderabad, 2014, pp. 491–502.
83. M. Samak and M. K. Ramanathan, *Trace driven dynamic deadlock detection and reproduction*, Proceedings of the Symposium on Principles and Practice of Parallel Programming, ACM, Orlando, 2014, pp. 29–42.
84. M. Eslamimehr and J. Palsberg, *Race directed scheduling of concurrent programs*, Proceedings of the Symposium on Principles and Practice of Parallel Programming, ACM, Orlando, 2014, pp. 301–314.
85. N. Machado, B. Lucia, and L. Rodrigues, *Concurrency debugging with differential schedule projections*, Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, ACM, Portland, OR, USA, 2015, pp. 586–595.

How to cite this article: Yu T, Srisa-an W, Rothermel G. An automated framework to support testing for process-level race conditions. *Softw Test Verif Reliab*. 2017;e1634. https://doi.org/10.1002/stvr.1634