Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2020-IJ-002

2020-IJ-002

Automatic Detection, Validation, and Repair of Race Conditions in Interrupt-Driven Embedded Software

Yu Wang, Fengjuan Gao, Linzhang Wang, Tingting Yu, Jianhua Zhao, Xuandong Li

Technical Report 2020

Most of the papers available from this document appear in print, and the corresponding copyright is held by the publisher. While the papers can be used for personal use, redistribution or reprinting for commercial purposes is prohibited.

Automatic Detection, Validation, and Repair of Race Conditions in Interrupt-Driven Embedded Software

Yu Wang, Fengjuan Gao, Linzhang Wang, Tingting Yu, *Member, IEEE*, Jianhua Zhao, and Xuandong Li

Abstract—Interrupt-driven programs are widely deployed in safety-critical embedded systems to perform hardware and resource dependent data operation tasks. The frequent use of interrupts in these systems can cause race conditions to occur due to interactions between application tasks and interrupt handlers (or two interrupt handlers). Numerous program analysis and testing techniques have been proposed to detect races in multithreaded programs. Little work, however, has addressed race condition problems related to hardware interrupts. In this paper, we present SDRacer, an automated framework that can detect, validate and repair race conditions in interrupt-driven embedded software. It uses a combination of static analysis and symbolic execution to generate input data for exercising the potential races. It then employs virtual platforms to dynamically validate these races by forcing the interrupts to occur at the potential racing points. Finally, it provides repair candidates to eliminate the detected races. We evaluate SDRacer on nine real-world embedded programs written in C language. The results show that SDRacer can precisely detect and successfully fix race conditions.

Index Terms—Embedded software, interrupts, race condition, software testing, repair suggestion

For Research Only

1 Introduction

Modern embedded systems are highly concurrent, memory, and sensor intensive, and run in resource constrained environments. They are often programmed using interrupts to provide concurrency and allow communication with peripheral devices. Typically, a peripheral device initiates a 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 frequent use of interrupts can cause concurrency faults such as data races to occur due to interactions between application tasks and ISRs. Such faults are often difficult to detect, isolate, and correct because they are sensitive to execution interleavings.

As an example, occurrences of race conditions between interrupt handlers and applications have been reported in a previous release of uCLinux [33], a Linux OS designed for real-time embedded systems. In this particular case, the serial communication line can be shared by an application through a device driver and an interrupt handler. In common instances, the execution of both the driver and the handler would be correct. However, in an exceptional operating scenario, the driver would execute a rarely executed path. If an interrupt occurs at that particular time, simultaneous transmissions of data is possible (Section 2 provides further details).

Many techniques and algorithms have been proposed to address concurrency faults, such as race conditions. These include static analysis [23], [36], [52], [77], [81], dynamic monitoring [10], [20], [37], [50], schedule exploration [11], [17], [51], [68], [69], [74], test generation [54], [58] and concurrency fault repair [34], [38], [43], [44], [73]. These techniques, however, focus on thread-level races. Applying these directly to interrupt-driven software is not straightforward. First, interrupt-driven programs employ a different concurrency model. The implicit dependencies between asynchronous concurrency events and their priorities complicate the happens-before relations that are used for detecting races. Second, controlling interrupts requires fine-grained execution control; that is, it must be possible to control execution at the machine code level and not at the program statement level, which is the granularity at which many existing techniques operate. Third, occurrences of interrupts are highly dependent on hardware states; that is, interrupts can occur only when hardware components are in certain states. Existing techniques are often not cognizant of hardware states. Fourth, repairing race conditions in interrupt-driven embedded systems usually requires disabling and enabling interrupt sources in hardware; this is different from repairing thread-level concurrency faults.

There are several techniques for testing embedded systems with a particular focus on interrupt-level concurrency faults [26], [42], [63]. For example, Higashi *et al.* [26] improve random testing via a mechanism that causes interrupts to occur at all instruction points to detect interrupt related data

- *Yu Wang, Fengjuan Gao, Linzhang Wang, Jianhua Zhao, and Xuandong Li are with the State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing 210093, China. E-mail: {yuwang_cs, fjgao}@smail.nju.edu.cn, {lzwang, zhaojh, lxd}@nju.edu.cn.*
- *Tingting Yu is with the University of Kentucky, Lexington, KY 40506 USA. E-mail: tyu@cs.uky.edu.*

Manuscript received 16 June 2019; revised 27 Mar. 2020; accepted 17 Apr. 2020. Date of publication 20 Apr. 2020; date of current version 10 Jan. 2022. (Corresponding author: Linzhang Wang.) Recommended for acceptance by P. Eugster. Digital Object Identifier no. 10.1109/TSE.2020.2989171

0098-5589 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

races. However, these techniques rely on existing test inputs and could miss races that could otherwise be detected by other inputs. In addition, these techniques do not account for the implicit dependencies among tasks and interrupts due to priorities. Furthermore, none of the existing techniques can automatically repair race conditions.

This paper presents SDRacer (**s**tatic and **d**ynamic *race* detection), an automated tool that combines static analysis, symbolic execution, and dynamic simulation to detect, validate and repair race conditions in interrupt-driven embedded systems. SDRacer first employs static analysis to identify code locations for potential races. SDRacer then uses symbolic execution to generate input data and interrupt interleavings for exercising the potential racing points; a subset of false positives can be eliminated at this step. To further validate race warnings, SDRacer leverages the virtual platform's abilities to interrupt execution without affecting the states of the virtualized system and to manipulate memory and buses directly to force interrupts to occur. Finally, SDRacer provides repair suggestions for each validated race and developers decide which repair strategy is more suitable for the race.

To evaluate the effectiveness and efficiency of SDRacer, we apply the approach to nine embedded system benchmarks with race conditions. Our results show that SDRacer precisely detected 190 race conditions and successfully repaired them without causing deadlocks or excessive performance degradation. Furthermore, the time taken by SDRacer to detect, validate and repair races is typically a few minutes, indicating that it is efficient enough for practical use.

In summary, this paper contributes the following:

- An automated framework that can detect, validate and repair race conditions for interrupt-driven embedded software systems.
- A practical tool for directly handling the C code of interrupt-driven embedded software.
- Empirical evidence that the approach can effectively and efficiently detect and repair race conditions in real-world interrupt-driven embedded systems.

The rest of this paper is organized as follows. In the next section we present a motivating example and background. We then describe SDRacer in Section 3. Our empirical study follows in Sections 4 and 5, followed by discussion in Section 6. We present related work in Section 7, and end with conclusions in Section 8.

2 Motivation and Background

In this section we provide background and use an example to illustrate the challenges in addressing race conditions in interrupt-driven embedded software.

2.1 Interrupt-Driven Embedded Systems

In embedded systems, an interrupt alerts the processor to a high-priority condition requiring the interruption of the current code the processor is executing. The processor responds by suspending its current activities, saving its state, and executing a function called an interrupt handler (or an interrupt service routine, ISR) to deal with the event. This interruption is temporary, and, after the interrupt handler finishes, the processor resumes normal activities.

We denote an interrupt-driven program by P = `Task` $\|$ `ISR`, where `Task` is the main program that consists of one or more tasks (or threads) and `ISR` = $ISR_1 \| ISR_2 \| \ldots \| ISR_N$ indicates interrupt service routines. The subscripts of ISRs indicate interrupt numbers, with larger numbers denoting lower priorities. Typically, P receives two types of *incoming data*: command inputs as entered by users and sensor inputs such as data received through specific devices (e.g., a UART port). An *interrupt schedule* specifies a sequence of interrupts occurring at specified program locations. In this work, we do not consider reentrant interrupts (interrupts that can preempt themselves); these are uncommon and used only in special situations [63].

2.2 Race Conditions in Interrupt-Driven Programs

A race condition occurs if two events access a shared resource for which the order of accesses is nondeterministic, i.e., the accesses may happen in either order or simultaneously [53], [76]. It broadly refers to data races (if accessing the shared resource simultaneous) and order violations (if accessing the resources in either order). Specifically, in our context, a race condition is reported when two conditions are met: 1) the execution of a task or an interrupt handler T is preempted by another interrupt handler H after a shared memory access m, and 2) H manipulates the content of m. More formally

$$\begin{aligned} e_i = MEM(m_i, a_i, T_i, p_i, s_i) \wedge e_j = MEM(m_j, a_j, T_j, p_j, s_j) \\ \wedge m_i = m_j \wedge (a_j = WRITE \vee a_i = WRITE) \\ \wedge s_i = s_j.enabled \wedge p_j > p_i. \end{aligned}$$

MEM$(m_i, a_i, T_i, p_i, s_i)$ denotes a task or an ISR T_i with priority p_i performs an access $a \in$ {WRITE, READ} to memory location m_i while in an hardware state s_i. The above condition states that two events e_i and e_j are in race condition if they access the same memory location and at least one access is a write. Here, e_i is from a task or an ISR and e_j is from a different ISR, the interrupt of H_j is enabled when e_i happens, and the priority p_j is greater than p_i.

We consider the definition as a variant of order violations. Data races [59] are not applicable between a task and an ISR or between ISRs, because a memory cannot be simultaneously accessed by the tasks or the ISRs. That said, a memory is always accessed by a task (or a low-priority ISR) and then preempted by an ISR. Interrupts have an asymmetric preemption relation with the processor's non-interrupt context: interrupts can preempt non-interrupt activity (i.e., tasks) but the reverse is not true [63]. Atomicity violations [45] are not applicable because they require three shared variable accesses. Traditional order violations [45] are also not applicable since there is no enforced execution order. However, we regard it as a variant of order violations because interrupts have an asymmetric preemption relation with the processor's non-interrupt context.

2.3 A Motivating Example

In prior releases of uCLinux version 2.4, there is a particular race condition that occurs between the UART driver program uart start and the UART ISR serial8250 interrupt [33]. We provide the code snippets (slightly modified for ease of

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

```
1  int transmit(struct uart_port *port){
2    ...
3    if (iir & UART_IIR_NO_INT) {
4      if (!(port->bugs & UART_BUG_TXEN)) {
5        port->bugs |= UART_BUG_TXEN;
6        ...
7      }
8    }
9    serial_out(port, UART_IER, flags); /*disable irq2*/
10
11   ...
12   if (port->bugs & UART_BUG_TXEN) { /*workaround*/
13     ...
14     p = xmit->tail + 1;
15     serial_outp(port, UART_TX, p.x_char); /*incorrect
           output*/
16   }
17 }
18
19 static irqreturn_t irq1_handler(...){
20
21   if (thr == 0x1101) {
22     xmit->tail = a + 1;
23   }
24   b = xmit->tail;
25   ...
26 }
27
28 static irqreturn_t irq2_handler(...){
29
30   if (thr != 0x1101) {
31     xmit->tail = c + 1;
32     ...
33   }
34 }
```

Fig. 1. Race condition in a UART device driver.

presentation) that illustrate the error in Fig. 1. The variables marked with bold indicate shared resources accessed by both tasks and ISRs.

Under normal operating conditions, the interrupt service routines (ISRs) are always responsible for transmitting data. There are two ISRs: `irq1_handler` has a higher priority than `irq2_handler`. However, several sources have shown that problems such as races with other processors on the system or intermittent port problems can cause the response from the ISRs to get lost or cause a failure to correctly install the ISRs, respectively. When that happens, the port is registered as "buggy" (line 5) and workaround code based on polling instead of using interrupts is used (line 12-16). Unfortunately, the enabled `irq1_handler` is not disabled in the workaround code region so by the time the workaround code is executed, it is possible that `irq1_handler` preempts and modifies the shared variable `xmit->tail` (line 14); this causes the serial port to receive the wrong data (line 15).

The first challenge is that embedded systems use special operations to control interrupts, some of which may not even be recognized by existing static and dynamic analysis techniques. For example, `serial_out` disables `irq2_handler` by directly flagging an interrupt bit at the hardware level using the variable `flags` (line 9). Failing to identify such operations would report false positives. For example, conservative analysis techniques would falsely report that there is a race condition between line 14 and line 31 on the variable `xmit->tail` even if the `irq2_handler` is disabled in the task. Therefore, hardware states and operations must be known when testing for race conditions in interrupt-driven embedded systems.

Second, task and interrupt priorities affect the order relations between concurrency events. For example, the content of `xmit->tail` at line 24 cannot be modified by the write of `xmit->tail` at line 31 due to the reason that the `irq1_handler` has a higher priority than the `irq2_handler`. Therefore, existing techniques that neglect the effect of priorities would lead to false positives.

Third, exposing this race condition requires specific input data from the hardware. For example, only when the IIR register is cleared (i.e., `iir & UART_IIR_NO_INT` is true) and the port is set to "buggy" will the true branch (line 4) be taken in the `transmit` function. Existing techniques on testing interrupt-driven programs that rely on existing inputs are inadequate. While automated test case generation techniques, such as symbolic execution can be leveraged, adapting them to interrupt-driven software is not straightforward. For example, IIR is a read-only register and thus cannot be directly manipulated; the value of IIR is controlled by the interrupt enable register (IER). Therefore, hardware properties must be considered when generating input data.

Fourth, races detected by static analysis or symbolic execution without considering the change of program control flow due to interrupts may report false positives. In this example, static analysis would report there exists a race on `xmit->tail` between line 22 and line 31. But the two lines can never get executed in the same run. Therefore, controllability is needed to validate whether a race detected by these techniques are real or not. While several existing approaches have tried to abstract away scheduling non-determinism in concurrent programs to achieve greater execution control (e.g., [69]). These approaches often control thread scheduling, but cannot force hardware interrupts to occur at arbitrary point.

Finally, repairing interrupt-related race conditions requires enforcing synchronization operations that are specific to interrupts. In this example, the interrupt disable operation associated with `irq1_handler` should be inserted before line 14. However, finding the right place to insert interrupt operations is challenging because the correctness of program semantics must be guaranteed. In addition, the length of a critical section must be considered because a long critical section may lead to timing or performance violations. Existing techniques on repairing thread-level concurrency faults cannot be directly adapted to address this problem.

2.4 Leveraging Virtual Platforms in Testing

Virtual platforms such as Simics provide observability and fine-grained controllability features sufficient to allow test engineers to detect faults that occur across the boundary between software and hardware. SDRacer takes advantage of many features readily available in many virtual platforms to tackle the challenges of testing for race conditions in interrupt-driven embedded software. Particularly, we can achieve the level of observability and controllability needed to test such systems by utilizing the virtual platform's abilities to interrupt execution without affecting the states of the virtualized system, to monitor function calls, variable values and system states, and to manipulate memory and buses directly to force events such as interrupts and traps. As such, SDRacer is able to stop execution at a point of interest and force a traditionally non-deterministic event to occur. Our system then monitors the effects of the event on the system and determines whether there are any anomalies.

2.5 Comparing to Thread-Level Race Detection Techniques

Although interrupts are superficially similar to threads (e.g., nondeterministic execution), the two abstractions have subtle semantic differences [64]. As such, thread-level race detection

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

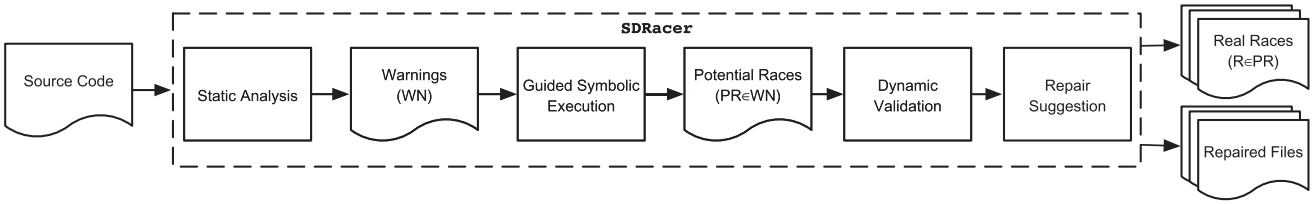

Fig. 2. Overview of SDRacer framework.

techniques [10], [20], [37], [50], [54], [58] cannot be adapted to address interrupt-level race conditions.

First, threads can be suspended by the operating system (OS) and thus the insertion of delays (e.g., sleep or yield instructions) can be used to control the execution of threads. The status of each thread is also visible at the application level. However, interrupts cannot block—they run to completion unless preempted by other higher-priority interrupts. The inability to block makes it impossible to use advanced OS services for controlling the occurrences of interrupts in race detection. In addition, the internal states of interrupts are invisible to tasks and other interrupt handlers because of the non-blocking characteristics. As such, it is impossible to use code instrumentation for checking the status of interrupts.

Second, threads typically employ symmetrical preemption relations—they can preempt each other. In contrast, tasks and interrupt handlers (i.e., task versus ISR and ISR versus ISR) have asymmetrical preemption relations. Specifically, interrupts cannot be preempted by normal program routines; instead, they can be preempted only by other interrupts with higher priority, and this can occur only when the current interrupt handler is set to be preemptible. The asymmetric relationship between interrupt handlers and tasks invalidates the happens-before relations served as the standard test for detecting thread-level races. Because their computed happens-before relation will not be precise if an interrupt is triggered during their computation, but thread-level methods do not handle interrupts.

Third, the concurrency control mechanisms employed by interrupts are different. A thread synchronization operation uses blocking to prevent a thread from passing a given program point until the synchronization resource becomes available. However, concurrency control in interrupts involves disabling an interrupt from executing in the first place. This is done by either disabling all interrupts or disabling specific interrupts that may interfere with another interrupt or task. As such, thread-level techniques that rely on binary/bytecode instrumentation [69], [82] to control memory access ordering between threads cannot be used to control the occurrences of hardware interrupts. In contrast, interrupt-level race detection techniques must be able to control hardware states (e.g., registers) to invoke interrupts at specific execution points [84]. In addition, occurrences of interrupts are highly dependent on hardware states; that is, interrupts can occur only when hardware components are in certain states. Existing thread-level race detection techniques are not cognizant of hardware states.

3 SDRacer Approach

We introduce SDRacer whose architecture is shown in Fig. 2. The rectangular boxes contain the major components. SDRacer first employs lightweight static analysis (SA) to identify potential sources of race conditions. The output of this step is a list of static race warnings, $\{< e_i = (T_i, L_i, A_i), e_j = (T_j, L_j, A_j) >\}$. However, the event pair $< e_i, e_j >$ is unordered and thus SDRacer attempts to force e_j to occur after e_i in the following components to validate the race order. Here, T is a task or an ISR, L is the code location, and A is the access type. In the example of Fig. 1, the output of this step is: WN_1 = <(`transmit`, 14, R), (`irq1_handler`, 22, W)>, WN_2 = <`transmit`, 14, R), (`irq2_handler`, 31, W)>, WN_3 = <`irq2_handler`, 31, W), (`irq1_handler`, 22, R)>, and WN_4 = <`irq2_handler`, 31, W), (`irq1_handler`, 24, R)>.

Next, SDRacer invokes symbolic execution to generate input data that can reach the code locations of the static race warnings. In Fig. 1, the input data t_1 = {`IIR` = 0 x 0111, `THR` = 0 x 0111, `port->bugs = 0`} is generated to exercise WN_1, and t_2 = {`IIR` = 0 x 0111, `THR` = 0x0110, `port->bugs = 0`} is generated to exercise WN_2 and WN_4. This step can also eliminate infeasible racing pairs. For example, WN_3 cannot be covered due to the conflict path conditions between `irq1_handler` and `irq2_handler`. Therefore, WN_3 is a false positive. The output of symbolic execution is a list of potential races PR and their corresponding input data.

Then, SDRacer utilizes the virtual platforms to exercise the inputs on the potential races generated from the symbolic execution and force the interrupts to occur at the potential racing points. The output of this step is a set of real races. In the example of Fig. 2, WN_1 and WN_4 are real races because we can force the `irq1_handler` to occur right after line 14 and the `irq2_handler` to occur right after 24. Therefore, WN_1 and WN_4 are real races, whereas WN_2 is a false positive; `irq2_handler` cannot be issued after line 14 because its interrupt line is disabled.

Last, the race repair component repairs races by enforcing the interrupt disable and enable operations, adding locks or extending critical sections. It is worth noting that disabling interrupt will not lead the lost of interrupt because the interrupt is queued and will be saved to execute later [18]. In order to fix races, SDRacer first determines where to enforce these operations and then generates patches by static code transformation. In Fig. 1, to repair WN_1, an interrupt disable operation `irq_disable(1)` is added right before line 14 to disable `irq1_handler`, and an interrupt enable operation `irq_enable(1)` is added right after line 14. Likewise, to repair WN_4, `irq_disable(1)` is added before line 31, and `irq_enable(1)` is added right after line 31.

3.1 Static Analysis

In the static analysis phase, SDRacer first identifies shared resources and interrupt enable and disable operations. It then analyzes a list of potential racing pairs, i.e., static race

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

warnings. The racing pairs are used for guiding symbolic execution and dynamic race validation.

Algorithm 1. Shared Resources Identification

Input: Task set $Tasks$, ISR set $ISRs$
Output: shared resources set (SRS)

```
1: /* S denotes all tasks and ISRs. aliasSet is an alias set. */
2: S = Tasks ∪ ISRs, aliasSet = ∅
3: /* Whole-program alias analysis */
4: for T ∈ S do
5:   aliasSet = aliasSet∪ andersenPointsToAnalysis(T)
6: end for
7: /* Construct inter-procedure aliases in aliasSet. */
8: aliasSet = aliasSet∪ connectAliases(S, aliasSet)
9: for each ISR ∈ ISRs do
10:   /* r denotes a variable in ISR */
11:   for each r ∈ ISR do
12:     /* S_global denotes all global variables, parm(ISR) denotes
        parameters of ISR */
13:     if r ∉ S_global and ∀v ∈ parm(ISR), (v, r) ∉ aliasSet then
14:       continue
15:     end if
16:     if ∃T ∈ (S − {ISR}), r ∈ T or alias ∈ T where
        (r, alias) ∈ aliasSet, then
17:       SRS = SRS ∪ r
18:       for each (r, alias) ∈ aliasSet do
19:         SRS = SRS ∪ alias
20:       end for
21:     end if
22:   end for
23: end for
```

3.1.1 Identifying Shared Resources

Race conditions are generally caused by inappropriate synchronized access to shared resources. So precisely detecting shared resources is key to race detection. In addition to shared memory that is considered by thread-level race detection techniques, SDRacer also accounts for hardware components that are accessible by applications and device drivers, including device ports and registers.

SDRacer automatically decomposes tasks based on the specific patterns of device drivers. For example, the first parameter of $kthread\_create$ refers to the function name of a task. Another type of task is the function callback, which is triggered by a specific device operation (e.g., device read). Then, we use Algorithm 1 to identify all possible shared resources. Specifically, we use Andersen's pointer analysis [7] to identify resources accessed by at least: 1) two ISRs, or 2) one task and one ISR. For each detected shared resource, we also add all its aliases to the shared resource set SRS. We first perform intra-procedure alias analysis for each task and ISR (at lines 3 to 6). Then, in order to connect alias relation in tasks/ISRs, we assume interrupt request operations (e.g., $request\_irq$) and task registration operations (e.g., $kthread\_create$) is called right after its registration. For example, $threadfn(threadArg)$; and $interruptFN(dev)$ are called right after $kthread\_create(threadFn, threadArg, \ldots)$; and $request\_irq(irqNum, interruptFn, flag, \ldots)$; separately. Based on the assumption, we perform inter-procedure alias analysis (at line 8). In this step, we analyze the program from top to bottom according to its call graph. For each function, when a variable is used as an argument of a callee, we record its aliases in the caller, aliases of the parameter in the callee, and their relation. Therefore, a caller's alias relation will be passed to its callees iteratively, and thus the assumption will not miss possible aliases. Then, we identify shared resources based on the whole program alias relation. For each ISR, we only analyze a variable (i.e., r) in the ISR if it is a global variable (including hardware components like device ports) or it may be an alias of the ISR's parameter (at lines 12 to 15). Finally, for each remaining variable, we add it and its aliases to SRS if 1), it is also used in other tasks or ISRs (i.e., $r \in T$ denotes r is a global variable), or 2) at least one of its aliases may be used in other tasks or ISRs (i.e., $alias \in T$ where $(r, alias) \in aliasSet$).

Each detected shared resource access SRA is denoted by a 6-tuple: $SRA = \{< T, L, V, AV, R, A >\}$, where T denotes the name of the task or ISR in which the shared resource (SR) is accessed, L denotes the code location of the access, V denotes the name of the SR, AV denotes whether the name V is an alias ($false$) or a real name ($true$) (real name is the declared name), R means the real name of this resource, and A denotes the access type—read (denoted by R) or write (denoted by W).

In the example of Fig. 1, all SRA for the `xmit` → `tail` is: $<$ `transmit`, 14, `xmit` → `tail`, true, `xmit` → `tail`, R $>$, $<$ `transmit`, 22, `xmit` → `tail`, true, `xmit` → `tail`, W $>$, $<$ `irq_handler`, 24, `xmit` → `tail`, true, `xmit` → `tail`, R $>$, and $<$ `transmit`, 31, `xmit` → `tail`, true, `xmit` → `tail`, W $>$.

3.1.2 Identifying Interrupt Operations

To track interrupt status (i.e., disabled or enabled) of a shared resource, SDRacer identifies interrupt-related synchronization operations, which typically involve interrupt disable and enable operations. In many embedded systems, coding interrupt operations can be rather flexible. An interrupt operation can be done by directly manipulating hardware bits (e.g., line 9 of Fig. 1). In addition, these operations vary across different architectures and OS kernels.

SDRacer considers both explicit and implicit interrupt operations. For the explicit operations, SDRacer considers standard Linux interrupt APIs, including `disable_irq_all()`, `disable_irq(int irq)`, `disable_irq_nosync(int irq)` and `enable_irq(int irq)`, where the `irq` parameter indicates the interrupt vector number (i.e., the unique ID of an interrupt). For the implicit operations, SDRacer tracks operations that manipulate interrupt-related hardware components, such as the interrupt enable registers (`IERs`). Since the effect of these operations are often not recognized by static analysis, SDRacer conservatively assumes they are equivalent to interrupt enabling (e.g., `enable_irq_all()`) to avoid false negatives. The consequence is false positives, which can be validated by the dynamic validation. In Fig. 1, the hardware write operation at line 9 is considered to be an interrupt enable operation.

To handle interrupts in different kernels or architectures. SDRacer provides a configuration file that allows developers to specify the names of interrupt APIs. The output of this step is a 4-tuple list: ITRL = $\{< M, L, I, T >\}$, where M denotes the function name, L denotes the code location

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

where the interrupt operation is called, I denotes the interrupt vector number and T denotes the type of interrupt operation (i.e., enable or disable). In the example of Fig. 1, the $ITRL$ is: $<$`transmit`, 9, all, $enable>$, where all denotes all interrupts are enabled.

3.1.3 Identifying Static Race Warnings

In this step, we identify shared resource access pairs that may race with each other from all identified shared resources. These pairs are used as targets for guiding symbolic execution to generate test input data.

To statically identify potential racing pairs, we first build a reduced inter-procedural control flow graph (RICFG) for the task and each of the ISR that contains at least one shared resource. RICFG prunes branches that do not contain shared resources in the original inter-procedural control flow graph (ICFG) in order to reduce the cost of analysis. Additionally, we use a bit vector *INTB* to record the interrupt status. For example, *INTB* = $< 1, 0, 0 >$ indicates that the first interrupt is disabled and the second and the third interrupts are enabled. *INTB* is updated when an interrupt disable/enable instruction is visited. Note that when visiting an instruction inside the ISR, the bit associated with the ISR is always set to 1 because an ISR is non-reentrant. Additionally, it is possible that an interrupt disabled by a task or an ISR is re-enabled immediately by another interrupt. In order to avoid false negatives, for each interrupt disable instruction, we check all enabled interrupts right after this operation. If there exists another interrupt that re-enables the disabled interrupt, we ignore the interrupt disable operation and its corresponding interrupt enable operation. By this conservative analysis method, we can avoid disabling interrupts that can be enabled by other ISRs.

Algorithm 2. Static Race Detection

Input: RICFGs of P
Output: potential racing pairs (PR)
1: **for** each $< G_i, G_j >$ in $RICFGs$ **do**
2: **for** each $sv_i \in G_i$ **do**
3: **for** each $sv_j \in G_j$ **do**
4: **if** $sv_i.V == sv_j.V$ and ($sv_i.A == W$ or $sv_j.A == W$) and $G_i.pri < G_j.pri$ and INTB.`get`(sv_i).`contains`(G_j) **then**
5: $PR = PR \cup (sv_i, sv_j)$
6: **end if**
7: **end for**
8: **end for**
9: **end for**

Algorithm 2 describes the computation of potential racing pairs based on the RICFGs of the program. SDRacer traverses each RICFG by a depth-first search to examine the interrupt status (i.e., enable or disable) of every instruction. For each shared resource sv_i at the location L of a RICFG G_i, if there exists the same shared resource sv_j in a RICFG G_j, at least one shared resource is a write, the priority of G_j is higher than that of G_i, and the interrupt for G_j is enabled at L, the pair (sv_i, sv_j) forms a potential race condition. For example, in Fig. 1, the bit vector at line 14 is $< 0, 0 >$, indicating that both irq1 and irq2 are enabled. Also, both irq1 and irq2 have higher priorities than `transmit`. The bit vector at line 13 is $< 1, 0 >$, because `irq1_handler` is non-reentrant. Therefore, WN_1, WN_2, WN_3, and WN_4 are reported as static race warnings.

About loops, our lightweight static analysis does not fully analyze loops because we unroll loops twice (i.e., transform a loop into two if statements) to balance accuracy and efficiency. Moreover, the analysis is context-insensitive, which may lead to false positives because it does not distinguish between different calling contexts of a function call. On the other hand, precise static analysis is more expensive [80]. As future work, we will evaluate cost-effectiveness by adopting precise static analysis techniques.

3.2 Guided Symbolic Execution

We propose a new symbolic execution procedure to generate input data for exercising static race warnings reported in static analysis and eliminating a portion of false races. Unlike traditional guided symbolic execution [22], [49], symbolic execution on interrupt-driven programs needs to consider the asymmetrical preemption relations among tasks and ISRs. The symbolic execution of SDRacer consists of two steps: 1) identifying entry points that take symbolic inputs; 2) generate inputs that exercise racing pairs reported by static analysis. Internally, we leverage the KLEE symbolic virtual machine [12] to implement the goal-directed exploration of the program to traverse the program locations involving potential races.

3.2.1 Identifying Input Points

Execution paths in embedded systems usually depend on various entry points that accept inputs from external components, such as registers and data buffers [83]. One challenge for our approach involves dealing with multiple input points in order to achieve high coverage of the targets. SDRacer considers two kinds of input points: 1) hardware-related memories (e.g., registers, DMA), and 2) global data structures used to pass across components (e.g., buffers for network packages, global kernel variables that are accessible by other modules). SDRacer can identify these input points based on the specific patterns of device drivers—this is a per-system manual process.

In the example of Fig. 1, The input points include the UART registers and the UART port. Specifically, the values in the registers IIR (line 3) and THR (line 21 and line 30) determine the data and control flow of the program execution. As such, we make these register variables symbolic. We also make the data fields of the UART port symbolic (e.g., `port->bug` at line 4) because they accept inputs from users and external components.

3.2.2 Guided Symbolic Execution.

For each static race warning $WN = < e_i, e_j >$, SDRacer calls the guided symbolic execution to generate a test input to exercise the WN or report that the WN is a false positive. Since each call to the symbolic execution targets a pair of events in two different tasks or ISRs, we build an inter-context control flow graph (ICCFG) by connecting the inter-procedural control flow graphs (ICFGs) of the tasks and ISRs. For an instruction that is equal to the first racing event e_i in a WN, we add an edge that connects e_i to the entry function

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

of the ICFG in which e_j exists. In the example of Fig. 1, to generate inputs for WN_1 = <(`transmit`, 14, R), (`irq1_handler`, 22, W)>, the entry of `irq1_handler` is connected to the instruction right after the `xmit->tail` read access.

SDRacer guides the symbolic execution toward the two ordered events of each WN by exploring the ICCFG. Let $e \in WN$ denote the current event to be explored, and s denotes the current program state that symbolic execution is exploring. A program state [13] is a representation of symbolic process. For each branch and loop, the program state will be cloned to explore different paths. Based on s, we refer to $\mathcal{S}_s$ as the set of next program states that symbolic execution could explore and reach e. Given e and s, $\mathcal{S}_s$ can be analyzed by the backward reachability analysis of ICCFG. At each step of the symbolic execution procedure, we select a promising state $s_i \in \mathcal{S}_s$, which is likely to reach e. Internally, SDRacer estimates the distance between each program state s_i and e before selecting the next state. Note that s_i is a next program state cloned based on branches or loops. Therefore, each s_i contains the next instruction to be explored. We define the distance from s_i to e to be the minimum number of instructions from the next instruction in s_i to the instruction in e according to the ICCFG. Among all program states in $\mathcal{S}_s$, we select s_i to be the next program state that symbolic execution should explore if its next instruction leads to the shortest path to the target instruction. If multiple states have the same distance to e, SDRacer randomly selects one. In this sense, the search strategy of SDRacer differs from prior symbolic execution techniques such as state prioritization (e.g., assertion-guided symbolic execution [25] and coverage-guided symbolic execution [12], [41], [46]), because they do not target the exploration of potential racing points.

If no state in $stateset$ can reach e, we check if e is in a loop. If e is in a loop, we increase the number of loop iterations by a fixed number of times given a timeout threshold and try again. This will increase our chance of reaching the goal. The iteration number is increased until reaching the loop bound L_{max} (L_{max} = 1000 in our experiments).

Otherwise, we backtrack and search for another path to the current event. If backtracking is repeated many times, eventually, it may move back to the first event, indicating that the current racing pair cannot be exercised. In such case, we move to the next racing pair. When our symbolic execution reaches the first event (i.e., e_i), the next program state to explore is the entry instruction in e_j. We continue exploring program states until we reach the second event. After reaching the second event (i.e., e_j), we traverse the current program path to compute the path condition (PC). Then, we compute the data input by solving the path condition using an SMT solver.

The main problem in guided symbolic execution is to make the procedure practical efficient by exploring the more "interesting" program paths. Toward this end, we propose several optimization techniques. Recall the way of constructing RICFG that we statically analyze the source code of the program to prune away paths that do not reach the shared resources—they are irrelevant to the potential races. We also skip computationally expensive constraint solver calls unless the program path traverses some unexplored potential races. In addition to these optimizations, we prioritize the path exploration based on the number of potential races contained in each path to increase the likelihood of reaching all static races sooner. Furthermore, we leverage concrete inputs (randomly generated) to help solving complex path constraints.

In the example of Fig. 1, the symbolic execution successfully generates input data for exercising WN_1 and WN_2, and WN_4. For WN_3, the symbolic execution explores the two events at line 31 and line 21 in the ICCFG that connects `irq1_handler` and `irq2_handler`. The path constraint $thr == 0x1101 \wedge thr \neq 0x1101$ is unsolvable, so WN_3 is a false positive.

For each static warning, there are three types of output generated by the symbolic execution. The first type of output is a potential race together with its input data, which means that this race is possible to be exercised at runtime. The second type of output is an unreachable message (unsolvable path constraints), which indicates that the static warning is a false positive. The third type of output is a message related to timeout or crash. The reason could be the execution time-out, the limitation of constraint solver or the unknown external functions. In the next phase of dynamic validation, we validate whether races reported in the first and third types are real races or not.

3.3 Dynamic Validation of Race Conditions

We propose a hardware-aware dynamic analysis method to validate the remaining race conditions from the symbolic execution. In this phase, SDRacer simulates virtual environment for interrupt-driven programs, which provides an execution observer and an execution controller. First, it employs an execution observer to monitor shared resource accesses and interrupt operations, and then uses an execution controller to force each race condition to occur.

3.3.1 Executing Observer

The Observer records operations that access shared memory and hardware components. The observer also monitors interrupt bits (IER and IIR registers) to track interrupt disabling and enabling operations. These bit-level operations are then mapped into the instruction-level statement, because the control of interrupts happens at the instruction level.

For each shared resource access, SDRacer can retrieve the current interrupt status of all IRQ lines to check whether it is possible to force a specific interrupt to occur.

3.3.2 Execution Controller

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.

We now describe the algorithm of execution controller (Algorithm 3). Given a potential racing pair σ = (e_i, e_j), The goal of this algorithm is to force an ISR that contains e_j to occur right after the access to e_i. The algorithm first executes the program under test P (line 6). If the the first shared resource access e_i occurs in a task, the algorithm executes the input data (generated from the symbolic execution) on P (line 3). If e_i occurs in an ISR, it executes P together with the interrupt issued at the arbitrary location of P (line 6). If the execution covers e_i, the algorithm forces the interrupt in

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

which e_j exists to occur immediately after e_i (line 9). If a race occurs, it is added to *RaceSet* (line 15).

Algorithm 3. Execution Controller

Input: *PRaceSet*, P, S
Output: *RaceSet*
1: **for** each $\sigma = (e_i, e_j) \in PRaceSet$ **do**
2: **if** e_i in T **then**
3: E = `Execute(`P`,` t_σ`)`
4: **end if**
5: **if** e_i in H **then**
6: E = `Execute(`P`,` $e_i.H$`,` t_σ`)`
7: **end if**
8: **if** E covers e_i **then**
9: **if** `ISR_enabled(`e_j`.H)` is true **then**
10: raise interrupt $e_j.H$
11: **else**
12: find another possible location
13: **end if**
14: **if** $e_j.H$ accesses e_j **then**
15: $RaceSet = RaceSet \cup \sigma$ /*race occurs*/
16: **end if**
17: **if** `Output(`P`,` S`)` $\neq O$ **then**
18: print "Error: fault found"
19: **end if**
20: **end if**
21: **end for**

Note that our algorithm can also force the interrupt to trigger immediately before e_i. In fact, the effect of triggering an interrupt immediately after the first event covers that of triggering an interrupt before the first event because a failure is usually caused by reading the incorrect value modified by the interrupt handler. It is not critical to choose either case because if we can trigger the second event right after the first event, then we can also trigger the second event right before the first event.

Because it may not be possible to raise an interrupt immediately (e.g., if the interrupt is currently disabled), the algorithm checks the current state of the interrupt associated with e_j (line 9) before raising an interrupt. The algorithm also checks outputs on termination of the events (lines 17-18) to determine whether a fault has been identified. If the interrupt (S) cannot be raised *immediately after* the shared resource access e_i in P (lines 9-10), the algorithm postpones $e_i.H$ (the ISR in which e_j exists) until it can feasibly be raised, or until the entry instruction of the operation in another potential race pair is reached.

To illustrate the algorithm's operation, using Fig. 1 as an example. Considering WN_1, given the input t_1, the `transmit` covers the read of `xmit`→ `tail` at line 14. Thus, the algorithm forces `irq1_handler` to be raised right after the read of `xmit`→ `tail` at line 14, In this scenario, `xmit`→ `tail` is modified by the `irq1_handler`, causing `transmit` to read the wrong value. As a result, WN_1 is real and harmful.

It is not always realistic to invoke an interrupt whenever we want. For example, the interrupt enables register and possibly other control registers have to be set to enable interrupts. In the example of Fig. 1, before invoking an interrupt, the interrupt enable register IER of the UART must be set while the interrupt identification register IIR must be cleared. Interrupts can be temporarily disabled even if they are enabled. Algorithm 4 is the routine in the Controller used to determine whether it is possible to issue an interrupt.

Algorithm 4. Algorithm to Determine Whether it is Possible to Issue an Interrupt: *ISR_enabled(int p)*

Input: P, p /* p is the pin number for a certain interrupt */
Output: enabled
1: **if** $eflags[9]$!= 0 and `ioapic.redirection[`p`]` == 0 and `ioapic.pin_raised[`p`]` == LOW **then**
2: **return**true
3: **end if**
4: **return** false

There are two 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 1 in Algorithm 4). 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 (APIC) as its interrupt controller. As such, our system checks whether the bit controlling the UART device is masked or not.

3.4 Race Conditions Repair Suggestions

Once a race condition is validated, the next step is to repair it. Before designing repair suggestions for race condition, we conduct an empirical study about the real-world repair methods for data race and race condition.

3.4.1 Learning From Practice

The repair of race condition requires strong domain knowledge and technical background. At present, it relies mainly on the manual repair by developers, which is time-consuming and laborious, and is also prone to errors. In order to explore and summarize the successful repair patterns that can be reused or can be automated, we conduct a lightweight empirical study to analyze how developers repair the race condition. Then we summarize the repair insight and patterns, in order to guide the developers for their future race repair as a reference.

To survey the repair strategies, we choose two data sources, the industry collaborators for embedding programs and the Linux kernel community. The industry collaborators provide their commonly used repair strategies in interrupt-driven programs. About collecting data in the Linux kernel, we inspect 532 bugs collected by Shi *et al.* [71] from 2011 to 2015 and validate 387 of them. Among the 387 races, 59 of them are related to interrupts since a part of races in the kernel are task-level races. The remaining bugs are not race conditions or too complex to identify their repair strategies. The reason for choosing Linux kernel as the subject of the empirical study is that it is the closest project that consists of many interrupt-driven programs (e.g., drivers).

Table 1 shows the result of race condition repair strategies. For each repair strategy, we describe how the method achieves eliminating races (column 2), along with examples (column 3). The fourth column indicates whether this type

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

TABLE 1
Summary of Repair Strategies From the Industry and Linux Kernel

Repair strategy	Description	Example	From ind.	From Linux		Application condition
				Task	Int	
Change operation orders (COO)	Change the order of operations so that the racing operations happen in separate timing $e_i \wedge e_j = false$	Move codes to a position where interrupts are finished	✓	88 26.8%	17 28.8%	The separate timing is available
Add additional checks (AAC)	Add additional checks to check program states to avoid race $e_i \wedge e_j = false$	if (!dev_initialized()) wait_until_init();	×	85 25.9%	5 8.5%	There is an available and race-free program state
Add locks (AL)	Add additional locks and unlocks $e_i \wedge e_j = false$	spin_lock/ spin_unlock	✓	81 24.7%	0 0%	It will not introduce deadlocks
Interrupt disable and enable (IDE)	Disable and enable interrupts $disable \wedge e_j = false$	disable_irq/ enable_irq	✓	0 0%	26 44.1%	It will not introduce deadlocks
Add atomic instructions (AAI)	Add atomic instructions $e_i \wedge e_j = false$	atomic_set	×	19 5.8%	4 6.8%	Its corresponding atomic instruction is available
Synchronization (Sync)	Synchronization $e_i \wedge e_j = false$	Read-copy update, memory barrier	✓	23 7.0%	0 0%	It should be used judiciously to avoid impeding performance
Remove race codes (RRC)	Remove race codes $remove\ e_i\ or\ e_j$	Remove unnecessary but buggy codes	×	12 3.7%	2 3.4%	The racy code is no longer needed
Extend critical sections (ECS)	Extend critical sections $e_i \wedge e_j = false$	Move spin_unlock after the racing code.	×	10 3.0%	4 6.8%	It will not introduce deadlocks
Minimize the use of shared resources (MinUse)	Minimize the use of shared resources $m_i \neq m_j$	Use bit operation instead of value assignment	×	3 0.9%	0 0%	Some SV accesses are redundant
Add try-again marks (ATM)	Retry interrupted tasks $\exists e_i \wedge e_j = false$	T(){if(flag==0)...} ISR() {flag=1;...}	✓	2 0.6%	0 0%	Performance insensitive tasks or interrupt handlers
Restrict users (ResUser)	Restrict users by documents or user manual $e_i \wedge e_j = false$	Forbid user sending requests right after starting a device.	✓	0 0%	0 0%	A general method
Change priorities of tasks or interrupts (ChgPrio)	Change priorities of tasks or interrupts $p_j > p_i = false$	Reverse priorities of two interrupts	✓	0 0%	0 0%	It will not lead to other races
Ad hoc repairs (Others)	Mostly $e_i \wedge e_j = false$	NA	×	5 1.5%	1 1.7%	NA

of repair strategy is used by our industry collaborators. The fifth and sixth columns are collected from the Linux kernel, we provide the number of occurrence for each repair method and the corresponding percentage among task-level races (i.e., *Task*) and interrupt-level races (i.e., *Int*). The last column discusses the conditions of applying repair suggestions. *NA* in the last row denotes that the two fields are not applicable.

3.4.2 Repair Suggestions

As shown in Table 1, in order to repair the race condition, developers can add code snippets, remove code snippets, and modify code snippets at specific locations to eliminate the root causes of the race condition.

In real-world repair, the race condition repair requires the location of shared resources, the location of their read and write operations, the interleaving of interrupts and tasks, which are very complicated. Moreover, in some embedded systems, the race condition repair method may consider the state of devices, in order to give a specific and ad hoc fix.

As far as we know, it is impossible to fully automate the repair process without programmers' participation [39]. Therefore, developers can choose repair strategies according to Table 1 and repair the program manually by themselves or automatically by SDRacer.

Based on Table 1, we can see that only a few of them are general purpose repair strategies, including `AL`, `AAI`, `IDE`, `Sync` and `AMB`. Others depend on semantics and pattern of the race. Among the top 5 repair strategies, `AL` and `IDE` are applicable to automatic repair. In comparison, `COO` and `AAC` depend on the semantics of the program and thus are more difficult to be done fully automatically. `AAI` only provides a limited set of operations, and often these operations are not enough to synthesize more complicated operations efficiently. It is worth noting that among all interrupt-level races, 44.1 percent of them are fixed by `IDE`. Among the remaining strategies, `Sync` and `AMB` are relatively uncommon and others depend on the pattern of races. Therefore, we focus on `AL` and `IDE` considering difficulty, practicality and universality. We leave other repair strategies as future work.

Interrupt Disable and Enable Strategy (`IDE`). This strategy automatically enforces interrupt disable and enable operations (e.g., `disable_irq(int irq)` and `enable_irq (int irq)`) on tasks or ISRs to avoid triggering the interrupt that can result in races. Note that we only disable and enable the interrupt line leads to races. Moreover, without loss of generality, we assume the interrupt disable operation waits for any pending IRQ handlers for this interrupt to complete before returning [4].

The main challenge of applying disabling/enabling interrupts strategy is that improper use of interrupt disable operation may lead to deadlock. For example, when a task disables an interrupt while holding a mutual exclusion shared resource (e.g., spin_lock [3]) the interrupt handler also needs to hold. Then the interrupt has to wait for the resource, but it leads to deadlock since the task that holds the resource also has to wait for the interrupt to return.

Given a racing pair ({$< e_i$ = (T_i, L_i, A_i), e_j= (T_j, L_j, A_j) $>$}.), Equation (1) is a sufficient condition to insert an

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

```
/*          Race program 1          */
void task1(){            void ISR1(){
  spin_lock(&lock1);
  ...                      //race code.
  spin_unlock(&lock1);     ...

  /* race code. */       }
  ...

}
```

```
/*          Repaired by AL          */
void task1(){            void ISR1(){
  spin_lock(&lock1);       spin_lock(&lock2);
  ...                      //race code.
  spin_unlock(&lock1);     ...
  spin_lock(&lock2);       spin_unlock(&lock2);
  /* race code. */       }
  ...
  spin_unlock(&lock2);
}
```

Fig. 3. An example of adding locks.

```
/*          Race program 2          */
void task2(){            void ISR2(){
  spin_lock(&lock1);       spin_lock(&lock2);
  ...                      ...
  spin_unlock(&lock1);     spin_unlock(&lock2);
  /* race code. */         //race code.
  ...                      ...
  spin_lock(&lock2);     }
  ...
  spin_unlock(&lock2);
}
```

```
/*          Repaired by ECS          */
void task2(){            void ISR2(){
  spin_lock(&lock1);       spin_lock(&lock2);
  ...                      ...
  spin_unlock(&lock1);     //race code.
  spin_lock(&lock2);       ...
  /* race code. */         spin_unlock(&lock2);
  ...                    }
  ...
  spin_unlock(&lock2);
}
```

Fig. 4. An example of extending critical sections.

interrupt disable operation right before instruction I_d while avoiding this kind of deadlock. Because there is no mutual exclusion shared resource between I_d and T_j. I_d is an instruction in T_i, indicating the location of inserting an interrupt disable operation. $\mathcal{I}_{T_j}$ is a set of all instructions in T_j. $hold(i)$ denotes all possible mutual exclusion shared resources (e.g., spin_lock) that instruction i may hold and not released. $hold(i)$ is calculated based on the alias analysis in Algorithm 1. First, we analyze all mutual exclusion shared resources that instruction i holds, and add it to $hold(i)$. For example, a spin_lock is in $hold(i)$ if and only if it is acquired before i and released after i according to RICFG. Second, for each resource in $hold(i)$, we analyze its aliases based on the alias analysis result and add them to $hold(i)$

$$hold(I_d) \cap \left(\bigcup_{k \in \mathcal{I}_{T_j}} hold(k) \right) = \emptyset. \tag{1}$$

Then, let $Preds(i)$ be all predecessors of the instruction i (inclusive) in RICFG and let I_i be the instruction at L_i. Based on the equation, we identify I_d if 1) $I_d \in Preds(I_i)$, 2) I_d meets Equation (1), and 3) $\forall I_j \in Preds(I_i), dis(I_d, I_i) \leq dis(I_j, I_i)$. $dis(i, j)$ is the minimum number of instructions from instruction i to j in RICFG.

After locating I_d, to identify the instruction of inserting interrupt enable operation (denoted as I_e), we analyze T_i to find an instruction I_e *s.t.* 1) $\mathcal{S}_d = \text{post-dom}(I_d), \mathcal{S}_e = \text{dom}(I_e)$ in RICFG, 2) $I_d \in \mathcal{S}_e$, $I_e \in \mathcal{S}_d$, and 3) $\forall I_k \in \mathcal{S}_d, dis(I_e, I_d) \leq dis(I_k, I_d)$. In other words, I_e post-dominates I_d, I_d dominates I_e and I_e is the closest location to I_d. If we cannot find I_e or I_d, then we report this race pair is unable to repair by this strategy. Finally, we insert an interrupt disable instruction right before I_d and an interrupt enable instruction right after I_e.

In the example of Fig. 1, to repair the race WN_1, SDRacer first inserts an `disable_irq(1)` right before line 14. Then the `enable_irq(1)` is inserted right after line 14.

Add Locks Strategy (`AL`) and Extend Critical Sections (`ECS`). The core idea of this repair strategy is to automatically create or extend a critical section that protects shared variables. The principle is that keeping the order of existing locks will not introduce deadlock problem [14].

First, we define the lock order. If a task or an ISR first acquires a lock l_i and then acquires another lock l_j, there is a lock order from lock l_i to lock l_j, denoted by (l_i, l_j). There is a special case that, if a task or an ISR acquires two locks l_m and l_n at the same time (e.g., aliases), then the two locks have the same lock order. For example, if l_m and l_n are acquired at the same time after acquiring l_j, then the lock order is $(l_i, l_j, l_m/l_n)$. The lock order is computed by traversing the IRCFG of a program, recording each lock acquire operation. Given a racing pair (e_i, e_j), assume the order of acquiring locks for e_i is $LS_i = (l_{i1}, \ldots, l_{ik})$. Similarly, the relation for e_j is $LS_j = (l_{j1}, \ldots, l_{jm})$. Rule 2 is a sufficient condition to identify deadlocks because it avoids any inconsistent lock order between LS_i and LS_j. $l_i \rightarrow l_j$ denotes that lock l_i is acquired before holding l_j. Unlike (l_i, l_j), $l_i \rightarrow l_j$ indicates that there may be one or more locks between l_i and l_j

$$\begin{aligned} &\forall l_i, l_j \in LS_i \text{ and } l_i \rightarrow l_j, \text{if } l_i \in LS_j \text{ and } l_j \in LS_j, \\ &\text{then } l_i \rightarrow l_j \text{ also holds in } LS_j. \end{aligned} \tag{2}$$

Based on the rule, we propose two ways of repair, 1) adding new locks and 2) extending the lock scope. We first try to repair a race by adding a new lock. If the lock order of the program violates Rule 2, we will try to extend the lock scope instead of adding new locks. Otherwise, we insert a lock operation right before L_i and L_j, and also insert an unlock operation right after L_i and L_j. Then we validate the lock orders of the repaired program according to Rule 2. If the new lock meets the rule, we try to fix other races. `task1` and `ISR1` in Fig. 3 show a race repaired by adding a new lock. If Rule 2 is violated, we try to extend the lock scope. In order to meet Rule 2, we first locate the closest locks after L_i and L_j, which are denoted as l_{ii} and l_{jj}. Then, we extend one of the locks in $LS_{cand} = (l_{i1}, \ldots, l_{ii}) \cap (l_{j1}, \ldots, l_{jj})$, such that e_i and e_j can be protected by the same lock and meet Rule 2. If LS_{cand} is empty, we report this race pair is unable to repair by ECS. `task2` and `ISR2` in Fig. 4 show a race repaired by extending a critical section.

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

3.4.3 Race Repair Process

To repair a validated race, we propose a human-computer corporation based repair process. The overall procedure is shown in Algorithm 5. First, it analyzes the race report to locate its source lines (at line 1). After that, for each validated race warning, the algorithm chooses feasible repair strategies based on Table 1 and conducts the repair (at lines 3 to 6). Previous works claim that the majority of the programs patched by generate-and-validate patch generation systems do not produce acceptable outputs [60]. Therefore, we propose an additional subcomponent, which validates repaired programs, in order to avoid such situations. In this subcomponent, the algorithm validates the fixed program, either by the validation component from SDRacer or manual code review from developers (at lines 7 to 9). Then, the algorithm tries to extend critical sections for unfixed races (at line 8). Finally, it reduces repair operations by merging critical sections (at line 10). Next, we discuss how to validate repaired programs and reduce repair operations.

Algorithm 5. Race Repair and Validation

Input: P, $WNSet$
Output: P'
1: locateLine($WNSet$)
2: $P' = P$ /* Backup P */
3: **for** each $WN \in WNSet$ **do**
4: chooseRepairStrategies($WNSet$)
5: P' = repairProg(P', WN)
6: **end for**
7: **while** validation(P') is not passed **do**
8: extendCriticalSection(P')
9: **end while**
10: mergeFix(P')

Repaired Program Validation. To validate repaired programs, we reuse the first three components (i.e., static analysis, guided symbolic execution and dynamic validation) of SDRacer. Each generated patch is validated by SDRacer to check whether the bug has been fixed. Specifically, we check whether the dynamic validation component still reports the race.

Note that patches generated by SDRacer are still required to be validated. The reason for validation is that the position of inserting locks (or interrupt controlling operations) are lines nearest the racing points. Although it can repair the races defined in Section 2.2, it is not enough to repair atomicity violations [45], which are the combination of two or more races of the same shared resource. For example, there are two races and one atomicity violation in the code `Task(){write(sv);read(sv);}` `ISR(){write(sv);}`. Adding locks to create separate critical sections can fix the races but cannot fix the atomicity violation. Therefore, after repairing, we generate a patched program and validate it by the dynamic validation phase. If we can still detect races or failures, we will gradually enlarge the corresponding critical sections according to the repair algorithms until no bugs and no failures. In this way, extending critical sections could fix the atomicity violation in the example. The repaired program is `Task(){lock(l);write(sv);read(sv);unlock(l);}` `ISR(){lock(l);write(sv);unlock(l);}`.

Reduce Repair Operations. After repairing all races, we analyze critical sections created by `IDE`, `AL` or `ECS`. Then, we merge two critical sections if one of the following conditions are met: 1) the two critical sections have overlap. 2) there is no instruction between two critical sections. We only merge critical sections with the same IRQ if they are created by `IDE`. When merging critical sections created by `AL` or `ECS`, we replace two locks with the same lock. Merging critical sections reduces the number of repair operations by removing redundant operations.

3.5 Implementation

The static analysis component of SDRacer was implemented using the Clang Tool 3.4 [6]. Our alias analysis leveraged the algorithm in [7] to handle the alias of shared resources. Our guided symbolic execution was implemented based on KLEE 1.2 [1] with STP solver [5] and KLEE-uClibc [2]. Since most kernel functions are not supported by KLEE and KLEE-uClibc, we have extended KLEE-uClibc to support kernel functions such as *request_irq()*. In order to guide the symbolic execution toward specific targets (i.e., potential racing points), we modified KLEE to only gather constraints related to the paths that are generated by static analysis. We used Simics virtual platforms with a simulated X86 CPU to implement the dynamic validation phase. Simics provides APIs that can be accessed via Python scripts to monitor concurrency events and to manipulate memory and buses directly to force interrupts to occur. Finally, our automatic race repair component was implemented in Python to repair races from the dynamic validation phase.

4 Empirical Study

To evaluate SDRacer we consider three research questions:

RQ1. How effective is SDRacer at detecting interrupt-level race conditions?
RQ2. How efficient is SDRacer at detecting interrupt-level race conditions?
RQ3. How effective and efficient is SDRacer at repairing interrupt-level race conditions?

RQ1 allows us to evaluate the effectiveness of our approach in terms of the number of races detected at different phases, and their abilities to reduce false positives. RQ2 lets us consider the efficiency of our approach in terms of analysis/testing time and platform overhead. RQ3 examines whether our repair strategy is effective and efficient.

4.1 Objects of Analysis

As objects of analysis, we chose both open source projects and industrial products. First, we selected 118 device driver programs that can be compiled into LLVM bitcode from four versions of Linux Kernel. We next eliminated from consideration those drivers that could not execute in Simics environment; this process left us with four drivers: keyboard, mpu401_uart, i2c-pca-isa, and mv643_eth. The 114 drivers were not executable because their corresponding device models were not available in Simics—they need to be provided by developers. As part of the future work, we will develop new device models for Simics in order to study more device driver programs.

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

TABLE 2
Objects of Analysis

Program name	*LOC*	*#INT*	*# Func*	*#SR*	*#BB*
keyboard_ driver	84	1	4	5	45
mpu401_ uart	630	1	16	2	316
i2c-pca-isa	225	1	11	9	111
mv643xx _eth.c	3256	1	29	7	1076
short	704	5	18	20	315
shortprint	531	1	11	22	266
short (EI)	707	5	18	20	317
shortprint (EI)	530	1	11	22	266
module1	168	1	3	1	55
module2	154	2	7	4	62
module3	99	2	8	1	40

We also selected two driver programs from LDD [18]: short and shortprint. To create more subjects, we manually seeded a concurrency fault to each of the two LDD programs. Specifically, we injected a shared variable increment operation and a decrement operation in their interrupt handlers. The fault injection did not change the semantics of the original programs but induced new races to these programs. The two programs are denoted as short (EI) and shortprint (EI).

The other three subjects are real embedded software from China Academy of Space Technology. Module1 is an UART device driver. Module2 is a driver for the lower computer. Module3 is used to control the power of engine. Table 2 lists all eleven programs, the number of lines of non-comment code they contain, the number of interrupts (with different priorities), the number of functions, the number of shared resources, and the number of basic blocks. The number of basic blocks indicates the complexity of symbolic execution. The size of the benchmarks is consistent with a prior study of concurrency bugs in device driver programs [75], which ranges from less than a hundred line of code to thousands of lines of code.

All our experiments were performed on a PC with 4-core Intel Core CPU i5-2400 (3.10 GHz) and 8GB RAM on Ubuntu Linux 12.04. For the simulation, the Host OS was Ubuntu 12.04 and the guest OS was 10.04. Simulation was based on real-time mode and conducted without VMP (In order to run Intel Architecture (IA) targets quickly on IA-based hosts.). The timeout for symbolic execution was set to 10 minutes.

4.2 Dependent Variables

We consider several measures (i.e., dependent variables) to answer our research questions. Our first dependent variable measures technique *effectiveness* in terms of the *number of races detected*. We measure the number of races detected in each of the three phases. Similar to data races in multi-thread programs, the race condition we detect also has benign races and harmful races. We also inspected all of the reported real races (from the dynamic validation phase) that did not result in detectable failures to determine whether they were harmful or benign.

To assess the *efficiency* of techniques we rely on four dependent variables, each of which measures one facet of efficiency. The first dependent variable measures the analysis and testing time required by SDRacer across the three phases. 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.

Our second variable regarding efficiency measures the extra *platform overhead* associated with SDRacer. 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 SDRacer because it is independent of techniques for locating harmful races.

To measure the effectiveness of repair, we patch the program as suggested by SDRacer and run its test cases (generated from symbolic execution). We consider a repair is valid if it does not fail any test case in the dynamic validation. To measure the efficiency of repair, we compare the program execution time without the patch to the execution time with the patched applied.

4.3 Threats to Validity

The primary threat to external validity for this study involves the representativeness of our programs and faults. Other programs may exhibit different behaviors and cost-benefit tradeoffs, as may other forms of test suites. However, the programs we investigate are widely used and the races we consider are real (except the seeded races on the two LDD 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 (e.g., Simics and KLEE) to implement the various modules in our approach. As an additional threat to internal validity, race manifestation can be influenced by the underlying hardware [56], [72]. 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 [21]. 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 previously documented races existing 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 races detected are just one variable of interest where effectiveness is concerned. Other metrics such as the cost of manual analysis could be valuable. Furthermore, the performance also depends on the experiment setup, such as the time of unrolling loops, symbolic execution timeout, maximum repair attempts, etc.

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

TABLE 3
Experimental Results

Programs	Race Detected				Execution Time (second)			Sim Overhead	Dyna Only
	Static Analysis	Symbolic Execution	Dynamic Validation	Manual Checking	Static Analysis	Symbolic Execution	Dynamic Validation		
keyboard_driver	4	4	4	4	0.073	1.03	1.65	892x	4
mpu401_ uart	146	129	47	47	0.088	1251.83	75.2	245.3x	12
i2c-pca-isa	4	4	1	1	0.078	1.00	42.1	530.1x	1
mv643xx _eth	16	14	10	10	0.183	1207.97	102.2	64.4x	2
short	127	35	18	18	0.109	41.53	26.8	297x	14
shortprint	4	2	0	0	0.088	1.25	21.61	445.6x	0
short (EI)	149	41	24	24	0.106	48.28	24.13	285.6x	18
shortprint (EI)	14	8	6	6	0.091	4.44	49.3	425.8x	6
module1	4	4	4	4	0.076	0.91	1.54	669.2x	4
module2	93	65	64	64	0.075	21.48	1.25	590.1x	64
module3	15	15	12	12	0.073	3.39	1.06	426x	12

For Research Only

5 Results and Analysis

Table 3 reports the effectiveness and efficiency results observed in our study; we use this table to address our research questions.

5.1 RQ1: Effectiveness of SDRacer

Columns 2-5 in Table 3 show the number of races reported by static analysis, the number of races remained after symbolic execution, the number of real races reported by the dynamic validation across all 11 subjects, and the number of true races validated manually by us. We reported the detected races in the 9 subjects to developers and 4 of them were confirmed (i.e., keyboard_driver, module1, module2 and module3). Others are waiting for the confirmation. Note that we do not report bugs in the 2 subjects with injected races because they are introduced to evaluate the effectiveness of SDRacer.

As the results show, the symbolic execution reduced the number of false positives contained in the sets of static race warnings by 40.3 percent overall, with reductions ranging from 0 to 96.6 percent across all 11 subjects. The dynamic validation reduced the number of races reported by symbolic execution by 36.7 percent overall, with reductions ranging from 0 to 100 percent. The manual examination revealed that among all real races (i.e., exclude inject bugs) reported by the dynamic validation, all races are real and harmful. In total, SDRacer detected 190 races. Only on `shortprint` did SDRacer not detect any races. No false negatives were found on all programs by the manual inspection.

On two out the 11 subjects, symbolic execution reported equal number races to the dynamic validation (`keyboard_driver` and `module1`). In other words, symbolic execution did not report false positives on the two programs. On the other nine programs, symbolic execution did report false positives. By further examining the programs, we found two reasons that led to the false positives. The first reason is due to the unknown access type (read and write) in external functions. For example, on `mpu401_uart`, the ISR calls an external library (`snd_mpu401_input_avail`) taking an SV as the argument. The symbolic execution treats this access as a write since static analysis incorrectly identifies it as a write. The second reason is due to the conflict path constraints between the main task and ISRs, which resulted in time-out. In this case, the race reported by the static analysis is directly sent to the dynamic validation phase. The third reason is that it is incapable of recognizing the implicit interrupt operations; This case happened to the program `short`.

5.2 RQ2: Efficiency of SDRacer

Columns 6-8 in Table 3 report the analysis time of static analysis, symbolic execution, and dynamic validation. On two programs (`mpu401_uart` and `mv643xx_eth`), the symbolic execution reached the time limit (i.e., 10 minutes) on the two static warnings of each program due to the unsolvable path constraints. Therefore, their times of symbolic execution were much higher than the other programs. Overall, the total testing time spent by SDRacer ranged from 2 seconds to 23 minutes across all 11 subjects. Specifically, the time for static analysis never exceeded 0.2 second, which accounted for less than 0.01 percent of total testing time overall. The time spent on symbolic execution was 235 seconds in arithmetic mean, accounting for 88.1 percent of total testing time. The remaining (31 seconds) time was spent on dynamic validation, which accounted for 11.8 percent of total testing time. The time for symbolic execution and dynamic validation varied with the number of detected static warnings.

SDRacer incurred platform overhead due to the use of VMs. Column 9 of Table 3 lists the average platform overhead associated with SDRacer across all test runs. As the table shows, the average platform overhead ranged from 64x to 669x. As we can see from the result, the less complex a subject is, the more platform overhead it incurred. This is because our execution observer was implemented using the callback functions provided by the Simics VM; it took time for the MV to trigger callback functions. However, considering the benefits of virtual platforms and the difficulty of detecting interrupt-level race conditions, such overhead is trivial.

5.3 RQ3: Effectiveness and Efficiency of Automated Repair

We patched the program as suggested by SDRacer and run its test inputs along with the controlled interrupts. We consider a repair is valid if it does not fail any test case (i.e., no races are reported by SDRacer). Our results indicate that the repairs on all eleven programs are valid and did not incur new races or deadlock. To measure the repair overhead, we randomly choose three inputs from guided symbolic

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

TABLE 4
Repair Results

Programs	#New operations		Repair Overhead	
	IDE	AL/ECS	IDE	AL/ECS
keyboard_driver	2	4	0x	0x
mpu401_uart	28	46	0.04x	0.01x
i2c-pca-isa	2	4	0.75x	0.01x
mv643xx_eth	20	26	0.01x	0x
short	14	28	0.03x	0.01x
shortprint	NA	NA	NA	NA
short (EI)	16	30	0.03x	0.01x
shortprint (EI)	6	10	0.05x	0.01x
module1	2	4	0.8x	0.01x
module2	12	18	0.07x	0.01x
module3	4	10	0.08x	0.01x

execution (including interrupt trigger time) and average the performance overhead.

Columns 2-3 of Table 4 show the number of new operations inserted to fix races. The repair for *shortprint* is not applicable since there is no report. The number of new operations is not very high even for programs with many races. The main reason is that we merge critical sections to reduce redundant interrupt or lock operations. For example, there are two races in `Task(){read(x);read(y);}`, `ISR(){write(x);write(y);}`, which can be fixed by `Task(){lock(m);read(x);read(y);unlock(m);}`, `ISR(){lock(m);write(x);write(y);unlock(m);}`. Note that there are only 4 new operations instead of 8 operations. Columns 4-5 of Table 4 reports the overhead of the repair. On nine out of 11 programs, the overhead was less than 0.09. These overheads are in the same order of magnitude as that of other thread-level concurrency fault repair techniques [34], [44]. We consider such small overheads to be negligible. On the other hand, the overheads on the `module1` and the `i2c-pca-isa` are much higher. The reason is that after enforcing the interrupt disable and enable operations, it changed the control flow of the main tasks since it was preempted by ISR, causing the main tasks to execute longer.

Note that different interrupt request lines induced different overhead. In the experiment, we found that the higher priority interrupt tends to induce higher overhead when disabling it. We conjecture that disabling higher priority interrupts are likely to prevent lower priority ISRs from being executed and thus cause interrupt latencies.

Finally, we report fixed versions of `module1`, `module2` and `module3` to the developer and all of them are confirmed.

6 Discussion

In this section, we first summarize our experimental results and then explore additional observations and limitations relevant to our study.

6.1 Summary of Results

SDRacer's static analysis component can detect potential race conditions with a false positive rate 72.0 percent. Our static analysis is able to handle nested interrupts with different priorities, as opposed to deal with race conditions only between tasks and ISRs [16]. SDRacer's symbolic execution reduced the false positive rate to 49.8 percent. The VM-based dynamic validation eliminated all false positives. Meanwhile, we manually validated the results from SDRacer and found that they are all true races. The average testing time is 4.5 minutes on each program. SDRacer's automated race repair component effectively repaired all detected race conditions while inducing little overhead. There are only two exceptions, which are i2c-pca-isa and module1. The reason is that the functions contain race pair is too simple to ignore the overhead of introducing additional interrupt controlling operations. Therefore, adding locks is a better choice but adding interrupt controlling operations is simpler.

If these results generalize to other real objects, then *if engineers wish to target and repair race detection in interrupt-driven embedded system, SDRacer is a cost-effective technique to utilize.* In the case of non-existing VMs, developers can still use static analysis and symbolic execution to detect races.

6.2 Further Discussion

*Influence of Test Input Generation.*There have been techniques for detecting concurrency faults that occur due to interactions between application and interrupt handlers [27], [42], [63], [84]. However, these techniques neither handle nested interrupts nor considers priority constraints among tasks and ISRs. Also, they do not have the static analysis and symbolic execution components, which could miss races that can only be revealed by certain inputs. In addition, these techniques are not applicable in the case of non-existing VMs or runtime environment. To further investigate whether the use of static analysis and symbolic execution can improve the race detection effectiveness, we disabled the two components and did see missing races. Columns 10 in Table 3 reports the numbers of races detected when using only the dynamic validation component. When only the dynamic validation stage is used, we regard it as a dynamic testing tool since it can detect races. We regarded the test subjects as a black box and manually fed inputs. As the data shows, in total, it detected only 137 races—28.2 percent less effective than SDRacer.

Atomicity Violations. SDRacer considers one type of definition of race conditions—order violations. In practice, testers can adopt different definitions because there is not a single general definition for the class of race conditions that occur between an *ISR* and a task/an *ISR*. SDRacer may miss faults due to atomicity violations. For example, if a read-write shared variable pair in the main program is supposed to be atomic, the *ISR* can read this shared variable before it is updated in the main program. Since SDRacer does not capture the read-read access pattern, this fault may be missed.

Inline Functions. In the dynamic validation phase, we use memory breakpoints to detect when concurrency events are executed. However, some simple functions are optimized as inline functions by compilers. In this case, breakpoints for these functions cannot be triggered. To handle this case, we need to disable optimization for these functions.

Dynamic Priority Assignment. Many false positives in the static analysis phase are caused by nested interrupts, because SDRacer does not recognize priorities that are dynamic assigned. These false positives can result in more validation time in symbolic execution and dynamic simulation. As part of future work, we will consider operations involving dynamic priority adjustment.

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

Complex Repair Strategies. As we can see from the survey of race repairs, some repair strategies (e.g., changing operation orders, adding additional checks) depend on semantics of a program, which is difficult to propose an accurate repair method to support these strategies. However, among the top 4 repair strategies, the two semantics-based repair strategies (i.e., COO and AAC) strategies take 50.4 percent among all repair methods, which means that both of them are important and popular repair methods.

Scalability to the Entire System. In our study, the analysis involves a test program, the interrupt handler that interacts with the device driver, and the device driver code. The key point here is that the tester focuses on a specific component[1] and how it interacts with the rest of the components. If the focus changes to a different component, the same analysis can be applied to test the new component. As such, the proposed approach is more suitable for component testing instead of testing the entire system at once.

Multi-Core Systems. Our experiment focuses on single-core systems. When it comes to multi-core systems, our method can detect races happening across cores because the way of detecting shared resources in the dynamic validation phase is to monitor the memory address of shared resources. As for repairing races in multi-core systems, disabling interrupts is not a choice because enabling/disabling interrupts is on a per-core basis [24]. Therefore, it is impractical to globally disable interrupts on all cores. However, adding locks or extending critical sections can fix this kind of race.

7 Related Work

There has been a great deal of work on analyzing, detecting, and testing for thread-level data races [8], [15], [19], [29], [47], [48], [55], [57], [61], [67], [69], [85]. However, as discussed in Section 2.5, existing techniques on testing for thread-level concurrency faults have rarely been adapted to work in scenarios in which concurrency faults occur due to asynchronous interrupts.

Dynamic Testing in Interrupt-Driven Programs. There are several techniques for testing embedded systems with a particular focus on interrupt-level concurrency faults [27], [42], [63], [84]. For example, Regehr *et al.* [63] use random testing to test Tiny OS applications. They propose a technique called restricted interrupt discipline (RID) to improve naive random testing (i.e., firing interrupts at random times) by eliminating aberrant interrupts. However, this technique is not cognizant of hardware states and may lead to erroneous interrupts. SimTester [84] leverages VM to address this problem by firing interrupts conditionally instead of randomly. Their evaluation shows that conditionally fired interrupts increase the chances of reducing cost. However, all the foregoing techniques do not consider interrupt-specific event constraints (e.g., priorities) and may lead to imprecise results. In addition, they are incapable of automatically generating test inputs or repairing race conditions. In contrast, our approach can cover all feasible shared variables in the application instead of using arbitrary inputs; this can help the program execute code regions that are more race-prone.

1. A component is a device driver program. The list of components can be identified by popular Linux commands such as "modprobe".

Static Analysis in Interrupt-Driven Programs. There has been some work on using static analysis to verify the correctness of interrupt-driven programs [16], [32], [40], [64]. For example, Regehr *et al.* [64] propose a method to statically verify interrupt-driven programs. Their work first outlines the significant ways in which interrupts are different from threads from the point of view of verifying the absence of race conditions. It then develops a source-to-source transformation method to transform an interrupt-driven program into a semantically equivalent thread-based program so that a thread-level static race detection tool can be used to find race conditions, which is the main benefit of their approach. Comparing to [64], SDRacer has two advantages. First, proof of the correctness of code transformation is often non-trivial; [64] does not provide proofs showing the transformation is correct or scalable. In contrast, SDRacer is transparent and does not require any source code transformation or instrumentation and can be applied to the original source code. Second, SDRacer uses dynamic analysis to validate warnings reported by static race detectors. Our evaluation showed that SDRacer can eliminate a large portion of false positives produced by static analysis, whereas Regehr's work [64] on seven Tiny OS applications does not evaluate the precision of their technique.

Jonathan *et al.* [40] first statically translate interrupt-driven programs into sequential programs by bounding the number of interrupts, and then use testing to measure execution time. While static analysis is powerful, it can report false positives due to imprecise local information and infeasible paths. In addition, as embedded systems are highly dependent on hardware, it is difficult for static analysis to annotate all operations on manipulated hardware bits; moreover, hardware events such as interrupts usually rely on several operations among different hardware bits. SDRacer leverages the advantages of static analysis to guide precise race detection. Techniques combined with static and dynamic method [78] could also detect and verify races. However, due to the lack of test case generation method, Manually efforts are required to inspect codes and generate test cases to reach race points.

Dynamic Testing in Event Driven Programs. There has been some research on testing for concurrency faults in event-driven programs, such as mobile applications [9], [30], [31], [47] and web applications [28], [62]. Although the event execution models of event-driven and interrupt-driven have similarities, they are different in several ways. First, unlike event-driven programs that maintain an event queue as first-in, first-out (FIFO) basis, interrupt handlers are often assigned to different priorities and can be preempted. Second, interrupts and their priorities can be created and changed dynamically and such dynamic behaviors can only be observed at the hardware level. Third, the events in event-driven programs are employed at a higher-level (e.g., code), whereas hardware interrupts happen at a lower-level (e.g., CPU); interrupts can occur only when hardware components are in certain states. The unique characteristics of interrupts render inapplicable the existing race detection techniques for event-driven programs.

Hybrid Techniques. There has been some research on combining static analysis and symbolic execution to test and verify concurrent programs [22], [25], [65], [66], [70], [79]. For

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

example, Samak *et al.* [66] combine static and dynamic analysis to synthesize concurrent executions to expose concurrency bugs. Their approach first employs static analysis to identify the intermediate goals towards failing an assertion and then uses symbolic constraints extracted from the execution trace to generate new executions that satisfy pending goals. Guo *et al.* [25] use static analysis to identify program paths that do not lead to any failure and prune them away during symbolic execution. However, these techniques focus on multi-threaded programs while ignoring concurrency faults that occur at the interrupt level. As discussed in Section 2.5, interrupts are different from threads in many ways. On the other hand, we can guide SDRacer to systematically explore interrupt interleavings or to target failing assertions.

Automatic Race Repair Techniques. There have been several techniques on automatically repairing concurrency faults in multi-threaded applications [34], [38], [43], [44], [73]. For example, AFix [34] can fix single-variable atomicity violations by first detecting atomicity violations, and then construct patches by adding locks.

CFix [35] is designed to have more general repair capabilities than AFix. CFix addresses the problem of fixing concurrency bugs by adding locks or condition variables to synchronize program actions. HFix [43] is proposed to address the problem that fixing concurrency bugs by adding lock operations and condition variables increase the fix complexity in some cases. HFix produces simpler fixes by reusing the code that is already present in the program rather than generating new code. ARC [38] employs a genetic algorithm without crossover (GAC) to evolve variants of an incorrect concurrent Java program into a variant that fixes all known bugs. However, none of existing techniques have been proposed to repair concurrency faults in interrupt-driven embedded programs. Surendran *et al.* [73] present a technique that focuses on repairing data races in structured parallel programs. This technique identifies where to insert additional synchronization statements that can prevent the discovered data races.

Interrupt-driven programs require dedicated repair strategies (e.g., control interrupts), in order to achieve better performance. Moreover, instead of only applying repair strategies for race instructions, SDRacer also validates fixed programs to ensure that races are correctly addressed. Although some techniques (e.g., AFix [34]) also verify their patches, the effectiveness depends on the fault detection tools while the detection and validation components in SDRacer further improve the effectiveness.

8 Conclusion and Further Work

This paper presents SDRacer, an automated tool to detect, validate and repair race conditions in interrupt-driven embedded software. SDRacer first employs static analysis to compute static race warnings. It then uses a guided symbolic execution to generate test inputs for exercising these warnings and eliminating a portion of false races. Then, SDRacer leverages the ability of virtual platforms and employs a dynamic simulation approach to validate the remaining potential races. Finally, SDRacer automatically repairs the detected races. We have evaluated SDRacer on nice real-world embedded programs and showed that it precisely and efficiently detected both known and unknown races. Therefore, it is a useful addition to the developers' toolbox for testing for race conditions in interrupt-driven programs. It also successfully repaired the detected races with little performance overhead. In the future, we will further improve the accuracy of static analysis. We also intend to extend our approach to handle other types of concurrency faults.

Acknowledgments

This work was supported by the National Key R&D Program of China (No. 2017YFA0700604), and the National Natural Science Foundation of China (No.61632015).

References

[1] KLEE LLVM Execution Engine, 2020. [Online]. Available: https://klee.github.io/
[2] KLEE-uClibc, 2020. [Online]. Available: https://github.com/klee/klee-uclibc
[3] Lesson 1: Spin locks, 2020. [Online]. Available: https://www.kernel.org/doc/Documentation/locking/spinlocks.txt
[4] Linux generic IRQ handling — The Linux Kernel documentation, 2020. [Online]. Available: https://www.kernel.org/doc/html/v4.18/core-api/genericirq.html#c.disable_irq
[5] STP constraint solver, 2020. [Online]. Available: http://stp.github.io/
[6] Using Clang Tools - LLVM, 2020. [Online]. Available: http://clang.llvm.org/docs/ClangTools.html
[7] L. O. Andersen, "Program analysis and specialization for the C programming language," PhD thesis, Dept. Comput. Sci., Univ. Cophenhagen, Copenhagen, Denmark, 1994.
[8] D. Aspinall and J. Ševčík, "Formalising Java's data race free guarantee," in *Proc. Int. Conf. Theorem Proving Higher Order Logics*, 2007, pp. 22–37.
[9] P. Bielik, V. Raychev, and M. Vechev, "Scalable race detection for android applications," *ACM SIGPLAN Notices*, vol. 50, pp. 332–348, 2015.
[10] M. D. Bond, K. E. Coons, and K. S. McKinley, "PACER: Proportional detection of data races," in *Proc. ACM SIGPLAN Symp. Program. Lang. Des. Implementation*, 2010, pp. 255–268.
[11] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte, "A randomized scheduler with probabilistic guarantees of finding bugs," in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2010, pp. 167–178.
[12] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. USENIX Symp. Operating Syst. Des. Implementations*, 2008, pp. 209–224.
[13] C. Cadar *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.
[14] Y. Cai, L. Cao, and J. Zhao, "Adaptively generating high quality fixes for atomicity violations," in *Proc. 11th Joint Meet. Found. Softw. Eng.*, 2017, pp. 303–314.
[15] D. Callahan, K. Kennedy, and J. Subhlok, "Analysis of event synchronization in a parallel programming tool," *ACM SIGPLAN Notices*, vol. 25, pp. 21–30, 1990.
[16] R. Chen, X. Guo, Y. Duan, B. Gu, and M. Yang, "Static data race detection for interrupt-driven embedded software," in *Proc. 5th Int. Conf. Secure Softw. Integr. Rel. Improvement Companion*, 2011, pp. 47–52.
[17] K. E. Coons, S. Burckhardt, and M. Musuvathi, "GAMBIT: Effective unit testing for concurrency libraries," in *Proc. 15th ACM SIGPLAN Annu. Symp. Princ. Practice Parallel Program.*, 2010, pp. 15–24.
[18] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2005.
[19] E. Duesterwald and M. L. Soffa, "Concurrency analysis in the presence of procedures using a data-flow framework," in *Proc. Symp. Testing Anal. Verification*, 1991, pp. 36–48.

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

[20] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm, "IFRit: Interference-free regions for dynamic data-race detection," in *Proc. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2012, pp. 467–484.
[21] J. Engblom, "Systematically exposing OS kernel races - an interview with Ben Blum," 2012. [Online]. Available: http://blogs.windriver.com/tools/2012/09/systematically-exposing-os-kernel-races-an-interview-with-ben-blum.html
[22] A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Con2colic testing," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2013, pp. 37–47.
[23] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *Proc. ACM SIGPLAN Symp. Program. Lang. Des. Implementation*, 2003, pp. 338–349.
[24] J. Ganguly and M. D. Lemmon, "Theory of clock synchronization and mutual exclusion in networked control systems," Univ. Notre Dame, Notre Dame, IN, USA, Tech. Rep. ISIS-99–007, 1999.
[25] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta, "Assertion guided symbolic execution of multithreaded programs," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2015, pp. 854–865.
[26] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue, "An effective method to control interrupt handler for data race detection," in *Proc. Workshop Autom. Softw. Test*, 2010, pp. 79–86.
[27] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue, "An effective method to control interrupt handler for data race detection," in *Proc. Workshop Autom. Softw. Test*, 2010, pp. 79–86.
[28] S. Hong, Y. Park, and M. Kim, "Detecting concurrency errors in client-side java script web applications," in *Proc. IEEE 7th Int. Conf. Softw. Testing Verification Validation*, 2014, pp. 61–70.
[29] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel, "Are concurrency coverage metrics effective for testing: A comprehensive empirical investigation," *J. Softw. Testing Verification Rel.*, vol. 25, no. 4, pp. 334–370, 2015.
[30] C.-H. Hsiao *et al.*, "Race detection for event-driven mobile applications," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 326–336.
[31] Y. Hu, I. Neamtiu, and A. Alavi, "Automatically verifying and reproducing event-based races in android apps," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 377–388.
[32] W. Huo, H. Yu, X. Feng, and Z. Zhang, "Static race detection of interrupt-driven programs," *J. Comput. Res. Develop.*, vol. 12, 2011, Art. no. 016.
[33] I. Jackson, "IRQ handling race and spurious IIR read in 8250.c." 2020. [Online]. Available: https://lkml.org/lkml/2009/3/12/379
[34] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 389–400, 2011.
[35] G. Jin, W. Zhang, and D. Deng, "Automated concurrency-bug fixing," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, 2012, pp. 221–236.
[36] S. Joshi, S. K. Lahiri, and A. Lal, "Underspecified harnesses and interleaved bugs," in *Proc. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2012, pp. 19–30.
[37] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang, "Static data race detection for concurrent programs with asynchronous calls," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2009, pp. 13–22.
[38] D. Kelk, K. Jalbert, and J. S. Bradbury, "Automatically repairing concurrency bugs with ARC," in *Proc. Int. Conf. Multicore Softw. Eng. Perform. Tools*, 2013, pp. 73–84.
[39] S. Khoshnood, M. Kusano, and C. Wang, "ConcBugAssist: Constraint solving for diagnosis and repair of concurrency bugs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 165–176.
[40] J. Kotker, D. Sadigh, and S. A. Seshia, "Timing analysis of interrupt-driven programs under context bounds," in *Proc. Formal Method Comput.-Aided Des.*, 2011, pp. 81–90.
[41] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2012, pp. 193–204.
[42] Z. Lai, S.-C. Cheung, and W. K. Chan, "Inter-context control-flow and data-flow test adequacy criteria for NesC applications," in *Proc. ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 94–104.
[43] H. Liu, Y. Chen, and S. Lu, "Understanding and generating high quality patches for concurrency bugs," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 715–726.
[44] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 299–309.
[45] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *Proc. 13th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2008, pp. 329–339.
[46] K.-K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proc. Int. Static Anal. Symp.*, 2011, pp. 95–111.
[47] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for android applications," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 316–325.
[48] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," *ACM SIGPLAN Notices*, vol. 40, pp. 378–391, 2005.
[49] P. D. Marinescu and C. Cadar, "Make test-zesti: A symbolic execution solution for improving regression testing," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 716–726.
[50] D. Marino, M. Musuvathi, and S. Narayanasamy, "LiteRace: Effective sampling for lightweight data-race detection," in *Proc. ACM SIGPLAN Symp. Program. Lang. Des. Implementation*, 2009, pp. 134–143.
[51] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing Heisenbugs in concurrent programs," in *Proc. USENIX Symp. Operating Syst. Des. Implementations*, 2008, pp. 267–280.
[52] M. Naik, C.-S. Park, K. Sen, and D. Gay, "Effective static deadlock detection," in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 386–396.
[53] R. H. B. Netzer and B. P. Miller, "What are race conditions? Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992.
[54] A. Nistor, Q. Luo, M. Pradel, T. R. Gross, and D. Marinov, "Ballerina: Automatic generation and clustering of efficient random unit tests for multithreaded code," in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 727–737.
[55] R. O'callahan and J.-D. Choi, "Hybrid dynamic data race detection," *ACM SIGPLAN Notices*, vol. 38, no. 10, pp. 167–178, 2003.
[56] L. Osterman, "Larry gets taken to task on concurrency," 2005. [Online]. Available: https://blogs.msdn.microsoft.com/larryosterman/2005/02/11/larry-gets-taken-to-task-on-concurrency/
[57] E. Pozniansky and A. Schuster, "Efficient on-the-fly data race detection in multithreaded C++ programs," in *Proc. 9th ACM SIGPLAN Symp. Princ. Practice Parallel Program.*, 2003, pp. 179–190.
[58] M. Pradel and T. R. Gross, "Fully automatic and precise detection of thread safety violations," in *Proc. ACM SIGPLAN Symp. Program. Lang. Des. Implementation*, 2012, pp. 521–530.
[59] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Context-sensitive correlation analysis for race detection," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 320–331, 2006.
[60] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36.
[61] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav, "Scalable and precise dynamic datarace detection for structured parallelism," in *Proc. 33rd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2012, pp. 531–542.
[62] V. Raychev, M. Vechev, and M. Sridharan, "Effective race detection for event-driven programs," *ACM SIGPLAN Notices*, vol. 48, pp. 151–166, 2013.
[63] J. Regehr, "Random testing of interrupt-driven software," in *Proc. ACM Int. Conf. Embedded Softw.*, 2005, pp. 290–298.
[64] J. Regehr and N. Cooprider, "Interrupt verification via thread verification," *Electron. Notes Theor. Comput. Sci.*, vol. 174, no. 9, pp. 139–150, 2007.
[65] M. Samak, M. K. Ramanathan, and S. Jagannathan, "Synthesizing racy tests," *ACM SIGPLAN Notices*, vol. 50, pp. 175–185, 2015.
[66] M. Samak, O. Tripp, and M. K. Ramanathan, "Directed synthesis of failing concurrent executions," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2016, pp. 430–446.
[67] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.
[68] K. Sen, "Effective random testing of concurrent programs," in *Proc. Int. Conf. Automated Softw. Eng.*, 2007, pp. 323–332.
[69] K. Sen, "Race directed random testing of concurrent programs," in *Proc. ACM SIGPLAN Symp. Program. Lang. Des. Implementation*, 2008, pp. 11–21.
[70] K. Sen and G. Agha, "CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools," in *Proc. Int. Conf. Comput. Aided Verification*, 2006, pp. 419–423.

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.

[71] J. Shi, J. I. Weixing, Y. Wang, L. Huang, Y. Guo, and F. Shi, "Linux kernel data races in recent 5 years," *Chin. J. Electron.*, vol. 27, no. 3, pp. 556–560, 2018.
[72] SSE Instructions: Which CPUs can do atomic 16B memory operations?, 2014. [Online]. Available: http://stackoverflow.com/questions/7646018/sse-instructions-which-cpus-can-do-atomic-16b-memory-operations
[73] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar, "Test-driven repair of data races in structured parallel programs," *ACM SIGPLAN Notices*, vol. 49, pp. 15–25, 2014.
[74] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw.. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.
[75] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler, "Static race detection for device drivers: The Goblint approach," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 391–402.
[76] C. von Praun, *Race Detection Techniques*. Boston, MA, USA: Springer, 2011, pp. 1697–1706.
[77] C. von Praun and T. R. Gross, "Static conflict analysis for multi-threaded object-oriented programs," in *Proc. ACM SIGPLAN Symp. Program. Lang. Des. Implementation*, 2003, pp. 115–128.
[78] Y. Wang, J. Shi, L. Wang, J. Zhao, and X. Li, "Detecting data races in interrupt-driven programs based on static analysis and dynamic simulation," in *Proc. Asia-Pacific Symp. Internetware*, 2015, pp. 199–202.
[79] Y. Wang, L. Wang, T. Yu, J. Zhao, and X. Li, "Automatic detection and validation of race conditions in interrupt-driven embedded software," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 113–124.
[80] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2004, pp. 131–144.
[81] A. Williams, W. Thies, and M. D. Ernst, "Static deadlock detection for java libraries," in *Proc. Eur. Conf. Object-Oriented Program.*, 2005, pp. 602–629.
[82] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: A coverage-driven testing tool for multithreaded programs," in *Proc. Int. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2012, pp. 485–502.
[83] T. Yu, X. Qu, and M. B. Cohen, "VDTest: An automated framework to support testing for virtual devices," in *Proc. Int. Conf. Softw. Eng.*, 2016, pp. 583–594.
[84] T. Yu, W. Srisa-an, and G. Rothermel, "SimTester: A controllable and observable testing framework for embedded systems," *ACM SIGPLAN Notices*, vol. 47, pp. 51–62, 2012.
[85] Y. Yu, T. Rodeheffer, and W. Chen, "RaceTrack: Efficient detection of data race conditions via adaptive tracking," *ACM SIGOPS Operating Syst. Rev.*, vol. 39, pp. 221–234, 2005.

Yu Wang received the BS degree in computer science from the University of Electronic Science and Technology of China, Chengdu, China, in 2014. He is currently working toward the PhD degree at Nanjing University, Nanjing, China. His research is in software engineering, with focus on analyzing concurrent software defects.

Fengjuan Gao received the BS degree in computer science from the University of Electronic Science and Technology of China, Chengdu, China, in 2014. She is currently working toward the PhD degree at Nanjing University, Nanjing, China. Her research is in software engineering with focus on symbolic execution.

Linzhang Wang is a professor with the State Key Laboratory of Novel Software Technology, Nanjing University, China. His research interests include software engineering and software security.

Tingting Yu (Member, IEEE) received the BE degree in software engineering from Sichuan University, Chengdu, China, in 2008, and the MS and PhD degrees from the University of Nebraska-Lincoln, Lincoln, Nebraska, in 2014. She is an assistant professor of computer science with the University of Kentucky. Her research is in software engineering, with focus on developing methods and tools for improving reliability and security of complex software systems; testing for sequential and concurrent software; regression testing; and performance testing. She is a member of the IEEE Computer Society.

Jianhua Zhao received the BS, MS, and PhD degrees in computer science from Nanjing University, Nanjing, China, in 1993, 1996, and 1999, respectively. He is a professor with Nanjing University. His research interests include formal support for design and analysis of systems, software verification.

Xuandong Li received the BS, MS, and PhD degrees in computer science from Nanjing University, Nanjing, China, in 1985, 1991, and 1994, respectively. Since 1994, he has been with the Department of Computer Science and Technology, Nanjing University where he is currently a professor. His research interests include formal support for design and analysis of reactive, disturbed, real-time, and hybrid systems, software testing and verification, and model driven software development.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

For Research Only

Authorized licensed use limited to: Nanjing University. Downloaded on January 25,2022 at 05:14:35 UTC from IEEE Xplore. Restrictions apply.