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

Yu Wang
Linzhang Wang
State Key Laboratory of Novel
Software Technology
Nanjing University
Nanjing 210023
yuwang@seg.nju.edu.cn
lzwang@nju.edu.cn

Tingting Yu
Department of Computer Science
University of Kentucky
Lexington, KY 40506
tyu@cs.uky.edu

Jianhua Zhao
Xuandong LI
State Key Laboratory of Novel
Software Technology
Nanjing University
Nanjing 210023
zhaojh@nju.edu.cn
lxd@nju.edu.cn

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. 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 and validate 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. We evaluate SDRacer on nine real-world embedded programs written in C language. The results show that SDRacer can precisely detect race conditions.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Embedded Software, Interrupts, Race Condition, Software Testing

ACM Reference format:
Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong LI. 2017. Automatic Detection and Validation of Race Conditions in Interrupt-Driven Embedded Software. In *Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10-14, 2017 (ISSTA'17),* 12 pages.
https://doi.org/10.1145/3092703.3092724

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 [29], 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 data races. These include static analysis [20, 30, 40, 60, 63], dynamic monitoring [9, 17, 31, 38], schedule exploration [10, 14, 39, 53, 54, 58], and test generation [41, 45]. 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.

There are several techniques for testing embedded systems with a particular focus on interrupt-level concurrency faults [22, 34, 48]. For example, Higashi et al. [22] improve random testing via a mechanism that causes interrupts to occur at all instruction points to detect interrupt related data races. However, these techniques rely

ISSTA'17, July 10-14, 2017, Santa Barbara, CA, USA

ACM ISBN 978-1-4503-5076-1/17/07…$15.00
https://doi.org/10.1145/3092703.3092724

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.

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 and validate 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. Finally, 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.

To evaluate the effectiveness and efficiency of SDRacer, we apply the approach to nine embedded system benchmarks with previously unknown race conditions. Our results show that SDRacer precisely detected 190 race conditions. Furthermore, the time taken by SDRacer to detect and validate races is typically a few minutes, indicating that it is efficient enough for practical use.

In summary, this paper contributes the following:

- A fully automated framework that can detect and validate 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 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 – 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` = $ISR_1 || ISR_2 || \dots || ISR_N$ 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 [48].

2.2 Race Conditions in Interrupt-driven Programs

A race condition occurs 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,

$$e_i = MEM(\alpha_i, m_i, a_i, T_i, p_i, s_i) \wedge e_j = MEM(\alpha_j, m_j, a_j, T_j, p_j, s_j)$$
$$\wedge m_i = m_j \wedge (a_j = WRITE \vee a_i = WRITE) \wedge s_j = s_j.enabled \wedge p_j > p_i$$

MEM(α, 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 ∈ {WRITE, READ} to memory location m 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.

A race condition is broadly referred to data races, atomicity violations, and order violations. In this work, we consider order violations. Data races 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 [48].

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 [29]. We provide the code snippets (slightly modified for ease of presentation) that illustrate the error in Figure 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

```
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
35  }
```

Figure 1: Race condition in a UART device driver

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 read 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.

Finally, 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.

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 [49]. As such, thread-level race detection techniques [9, 17, 31, 38, 41, 45] 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 vs. ISR and ISR vs. 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.

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 [54, 64] 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 [66]. 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 Figure 2. The rectangular boxes contain the major components. SDRacer

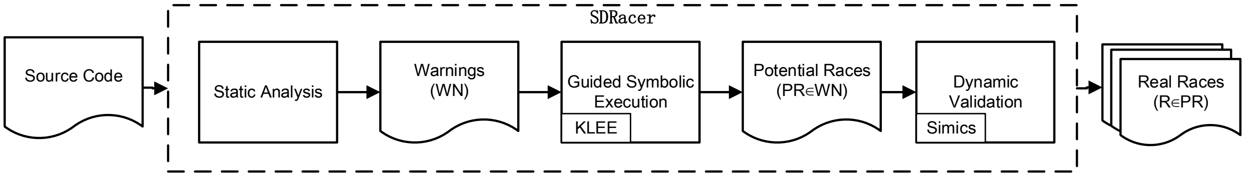

Figure 2: Overview of SDRacer framework.

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 $< e_i, e_j >$ does not imply that the two events are truly ordered. In the dynamic validation phase, SDRace attempts to force e_j to occur after e_i. Here, T is a task or an ISR, L is the code location, and A is the access type. In the example of Figure 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 Figure 1, the input data t_1 = {`IIR = 0x0111`, `THR = 0x0111`, `port->bugs = 0`} is generated to exercise WN_1, and t_2 = {`IIR = 0x0111`, `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.

Last, 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 Figure 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.

3.1 Static Analysis

In the static analysis phase, SDRacer first identifies shared resources and interrupt enable and disable operations. It then performs context-sensitive analysis to compute a list of potential racing pairs, i.e., static race warnings. The racing pairs are used for guiding test input generation and dynamic race validation.

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.

We use the Thread Safety Analysis tool [5] from the LLVM Clang static analyzer [1] to identify resources accessed by at least: 1) two ISRs, and 2) one task and one ISR. 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). Each detected shared resource SV is denoted by a 6-tuple: $RSL(SV)$ = {$< T, L, V, AV, R, A >$}, where T denotes the name of the task or ISR in which SV is accessed, L denotes the code location of the access, V denotes the name of the SV, AL 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 Figure 1, the RSL 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 Figure 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 (`IER`s). Since these operations are often not recognized by static analysis, SDRacer conservatively assumes they are equivalent to interrupt enabling (e.g., `enable_irq_all()`); false positives can be eliminated in the dynamic validation phase. In Figure 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 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 Figure 1, the $ITRL$ is: <`transmit`, 9, *all*, *enable*>, where *all* denotes all interrupts are enabled.

Algorithm 1 Static race detection

```
Input: IICFGs of $P$
Output: potential racing pairs ($PR$)
for each $< G_i, G_j >$ in $IICFGs$ do
    for each $sv_i \in G_i$ do
        for each $sv_j \in G_j$ do
            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
                $PR = PR \cup (sv_i, sv_j)$
            end if
        end for
    end for
end for
```

3.1.3 Identifying Static Race Warnings. In this step, we identify shared resource 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 (IICFG) for the task and each of the ISR that contains that contains at least one shared resource. IICFG 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.

Algorithm 1 describes the computation of potential racing pairs based on the IICFGs of the program. SDRacer traverses each IICFG by a depth-first search to examine the interrupt status (i.e., enable or disable) of every instruction. 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 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.

For each shared resource sv_i at the location L of an IICFG G_i, if there exists the same shared resource sv_j in an IICFG 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 Figure 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.

Note that our lightweight static analysis does not consider loops or context-sensitivity, which may lead to inaccuracies. For example, ignoring loops may cause false negatives because a new racing pair may be discovered in subsequent iterations. However, such cases were not found in the experiment. The context-insensitive analysis may lead to false positives because it does not distinguish between different calling contexts of a function. On the other hand, precise static analysis is more expensive [62]. 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 [19, 37], 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 [11] 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 [65]. 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 automatically identify these input points based on the specific patterns of device drivers – this is a per-system manual process.

In the example of Figure 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 a inter-context control flow graph (ICCFG) by connecting the inter-procedural control flow graphs (ICFGs) of the tasks and ISRs. For each 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 of the ICFG in which e_j exists. In the example of Figure 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 *stateset* denote the set of program states that could reach e. *stateset* can be analyzed based on the backward reachability analysis of IICFG. At each step of the symbolic execution procedure, we select a promising state $s_i \in stateset$, which is likely to reach e. Internally, SDRacer estimates the distance between each program state s_i and e before selecting the next state. The distance is defined as the number of instructions to be executed from s_i to e and is computed by statically traversing the ICCFG. 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 [21] and coverage-guided symbolic execution [11, 33]), 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. 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 that we statically analyze the source code of the program to prune away paths that do not lead to the shared resources – they correspond to the irrelevant 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 avoid generating a large number of invalid inputs.

In the example of Figure 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 weather races reported in the second 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 first 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.

Algorithm 2 Algorithm SDRacer: 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($P$, $t_\sigma$)
 4:     end if
 5:     if $e_i$ in $H$ then
 6:         $E$ = Execute($P$, $e_i.H$, $t_\sigma$)
 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
```

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 2). Given a potential racing pair $\sigma = (e_i, e_j)$, 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 which e_j exists to occur immediately after e_i (line 9). If a race occurs, it is added to *RaceSet* (line 15).

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 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_j 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 Figure 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

Algorithm 3 Algorithm to determine whether it is possible to issue an interrupt: *ISR_enabled(int p)*

```
Input: P
Output: enabled /*p is the pin number for a certain interrupt*/
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
```

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 Figure 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 3 is the routine in the Controller used to determine whether it is possible to issue an interrupt.

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 3). 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 Implementation

The static analysis component of SDRacer was implemented using the Clang Tool 3.4 [6]. Our alias analysis leveraged the algorithm in [57] to handle the alias of shared resources. Our guided symbolic execution was implemented based on KLEE 1.2 [2] with STP solver [4] and KLEE-uClibc [3]. 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 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.

4 EMPIRICAL STUDY

To evaluate SDRacer we consider two research questions:

RQ1: How effective is SDRacer at detecting interrupt-level race conditions across the three phases?

RQ2: How efficiency is SDRacer at detecting 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.

Table 1: 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	37	4	62
module3	99	2	8	1	40

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.

We also selected two driver programs from LDD [15]: 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 1 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 [59], 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 (3.10GHz) 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. 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.

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 [43, 56]. 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 [18]. 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 two variables of interest where effectiveness is concerned. Other metrics such as the cost of manual analysis could be valuable.

5 RESULTS AND ANALYSIS

Table 2 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 2 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 races detected in the three industrial programs to developers and the races were confirmed. We also reported the races detected in the four device driver programs and are waiting for the confirmation.

As the results show, the symbolic execution reduced the number of false positives contained in the sets of static race warnings by 55.6% overall, with reductions ranging from 27% to 100% across all 11 subjects. The dynamic validation reduced the number of races reported by symbolic execution by 59.1% overall, with reductions ranging from 0% to 100%. The manual examination revealed that among all races 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 races were found on this program 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 due to its 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 2 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% of total testing time overall. The time spent on symbolic execution was 235 seconds in arithmetic mean, accounting for 88.1% of total testing time. The remaining (31 seconds) time was spent on dynamic validation, which accounted for 11.8% 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 2 lists the average platform overhead associated

Table 2: Experimental Results

Programs	Race Detected				Execution Time (second)			Simulation Overhead	Controlled Interrupts 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

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.

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%. Our static analysis is able to handle nested interrupts with different priorities, as opposed to deal with race conditions only between tasks and ISRs [13]. SDRacer's symbolic execution reduced the false positive rate to 49.8%. The VM-based dynamic validation eliminated all false positives. Meanwhile, SDRacer detected all races with an average testing time of 4.5 minutes on each program.

If these results generalize to other real objects, then *if engineers wish to target 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. As discussed in Section 7, there have been techniques for detecting concurrency faults that occur due to interactions between application and interrupt handlers [23, 34, 48, 66]. 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 2 reports the numbers of races detected when using only the dynamic validation component. As the data shows, in total, it detected only 137 races – 28.2% 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.

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.

7 RELATED WORK

There has been a great deal of work on analyzing, detecting, and testing for thread-level data races [7, 12, 16, 25, 35, 36, 42, 44, 46, 52, 54, 67]. However, as discussed in Section 2.5, existing techniques

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

on testing for thread-level concurrency faults have rarely been adapted to work in scenarios in which concurrency faults occur due to asynchronous interrupts.

There are several techniques for testing embedded systems with a particular focus on interrupt-level concurrency faults [23, 34, 48, 66]. For example, Regehr et al. [48] 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 [66] 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 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.

There has been some work on using static analysis to verify the correctness of interrupt-driven programs [13, 28, 32, 49]. For example, Regehr et al. [49] 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 [49], SDRacer has two advantages. First, proof of the correctness of code transformation is often non-trivial; [49] 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 directly 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 [49] on seven Tiny OS applications does not evaluate the precision of their technique.

Jonathan et al. [32] 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 [61] 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.

There has been some research on testing for concurrency faults in event-driven programs, such as mobile applications [8, 26, 27, 35] and web applications [24, 47]. 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.

There has been some research on combining static analysis and symbolic execution to test and verify concurrent programs [19, 21, 50, 51, 55]. For example, Samak et. al. [51] 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. [21] 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.

8 CONCLUSION AND FURTHER WORK

This paper presents SDRacer, an automated tool to detect, validate 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. Finally, SDRacer leverages the ability of virtual platforms and employs a dynamic simulation approach to validate the remaining potential 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. 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

The paper was partially supported by the National Key Research and Development Plan (No.2016YFB1000802), the National Natural Science Foundation of China (No.61632015, 61472179, 61561146394, 61572249), and United States NSF grant CCF-1464032.

REFERENCES

[1] Clang Static Analyzer. https://clang-analyzer.llvm.org, 2016.

[2] KLEE LLVM Execution Engine. https://klee.github.io/, 2016.

[3] KLEE-uClibc. https://github.com/klee/klee-uclibc, 2016.

[4] STP constraint solver. http://stp.github.io/, 2016.

[5] Thread safety analysis, 2016. http://clang.llvm.org/docs/ThreadSafetyAnalysis.html.

[6] Using Clang Tools - LLVM. http://clang.llvm.org/docs/ClangTools.html, 2016.

[7] D. Aspinall and J. Ševčík. Formalising javaâĂŹs data race free guarantee. In *Theorem Proving in Higher Order Logics*, pages 22–37. Springer, 2007.

[8] P. Bielik, V. Raychev, and M. Vechev. Scalable race detection for android applications. In *ACM SIGPLAN Notices*, volume 50, pages 332–348. ACM, 2015.

[9] M. D. Bond, K. E. Coons, and K. S. McKinley. PACER: proportional detection of data races. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 255–268, 2010.

[10] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010.

[11] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementations (OSDI)*, pages 209–224, 2008.

[12] D. Callahan, K. Kennedy, and J. Subhlok. Analysis of event synchronization in a parallel programming tool. In *ACM SIGPLAN Notices*, volume 25, pages 21–30, 1990.

[13] R. Chen, X. Guo, Y. Duan, B. Gu, and M. Yang. Static data race detection for interrupt-driven embedded software. In *Secure Software Integration & Reliability Improvement Companion (SSIRI-C), 2011 5th International Conference on*, pages 47–52, 2011.

[14] K. E. Coons, S. Burckhardt, and M. Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 15–24, 2010.

[15] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux device drivers.* " O'Reilly Media, Inc.", 2005.

[16] E. Duesterwald and M. L. Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *symposium on Testing, analysis, and verification*, pages 36–48, 1991.

[17] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm. Ifrit: Interference-free regions for dynamic data-race detection. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 467–484, 2012.

[18] J. Engblom. Systematically exposing os kernel races - an interview with ben blum, 2012. http://blogs.windriver.com/tools/2012/09/systematically-exposing-os-kernel-races-an-interview-with-ben-blum.html.

[19] A. Farzan, A. Holzer, N. Razavi, and H. Veith. Con2colic testing. In *ACM SIGSOFT International Symposium on Foundations of software engineering (FSE)*, pages 37–47, 2013.

[20] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 338–349, 2003.

[21] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta. Assertion guided symbolic execution of multithreaded programs. In *ACM SIGSOFT International Symposium on Foundations of software engineering (FSE)*, pages 854–865, 2015.

[22] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue. An effective method to control interrupt handler for data race detection. In *Workshop on Automation of Software Test*, pages 79–86, 2010.

[23] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue. An effective method to control interrupt handler for data race detection. In *Workshop on Automation of Software Test (AST)*, pages 79–86, 2010.

[24] S. Hong, Y. Park, and M. Kim. Detecting concurrency errors in client-side java script web applications. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 61–70, 2014.

[25] S. Hong, M. Staats, J. Ahn, M. Kim, and G. Rothermel. Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation. *Journal of Software Testing, Verification and Reliability*, 25(4), 2015.

[26] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, 2014.

[27] Y. Hu, I. Neamtiu, and A. Alavi. Automatically verifying and reproducing event-based races in android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 377–388, 2016.

[28] W. Huo, H. Yu, X. Feng, and Z. Zhang. Static race detection of interrupt-driven programs. *Journal of Computer Research and Development*, 12:016, 2011.

[29] I. Jackson. IRQ handling race and spurious IIR read in 8250.c. Web page. https://lkml.org/lkml/2009/3/12/379.

[30] S. Joshi, S. K. Lahiri, and A. Lal. Underspecified harnesses and interleaved bugs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 19–30, 2012.

[31] V. Kahlon, N. Sinha, E. Kruus, and Y. Zhang. Static data race detection for concurrent programs with asynchronous calls. In *ACM SIGSOFT International Symposium on Foundations of software engineering (FSE)*, pages 13–22, 2009.

[32] J. Kotker, D. Sadigh, and S. A. Seshia. Timing analysis of interrupt-driven programs under context bounds. In *Formal Method in Computer-Aided Design (FMCAD)*, pages 81–90, 2011.

[33] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 193–204, 2012.

[34] Z. Lai, S.-C. Cheung, and W. K. Chan. Inter-context control-flow and data-flow test adequacy criteria for nesc applications. In *ACM SIGSOFT International Symposium on Foundations of software engineering (FSE)*, pages 94–104, 2008.

[35] P. Maiya, A. Kanade, and R. Majumdar. Race detection for android applications. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[36] J. Manson, W. Pugh, and S. V. Adve. *The Java memory model*, volume 40. ACM, 2005.

[37] P. D. Marinescu and C. Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *International Conference on Software Engineering (ICSE)*, pages 716–726, 2012.

[38] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace: effective sampling for lightweight data-race detection. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 134–143, 2009.

[39] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *USENIX Symposium on Operating Systems Design and Implementations (OSDI)*, pages 267–280, 2008.

[40] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering (ICSE)*, pages 386–396, 2009.

[41] 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 *International Conference on Software Engineering (ICSE)*, pages 727–737, 2012.

[42] R. O'callahan and J.-D. Choi. Hybrid dynamic data race detection. *ACM SIGPLAN Notices*, 38(10):167–178, 2003.

[43] L. Osterman. Larry Gets Taken to Task on Concurrency, 2005. https://blogs.msdn.microsoft.com/larryosterman/2005/02/11/larry-gets-taken-to-task-on-concurrency/.

[44] E. Pozniansky and A. Schuster. *Efficient on-the-fly data race detection in multithreaded C++ programs*, volume 38. ACM, 2003.

[45] M. Pradel and T. R. Gross. Fully automatic and precise detection of thread safety violations. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 521–530, 2012.

[46] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Scalable and precise dynamic datarace detection for structured parallelism. In *Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[47] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *ACM SIGPLAN Notices*, volume 48, pages 151–166. ACM, 2013.

[48] J. Regehr. Random testing of interrupt-driven software. In *ACM international conference on Embedded software (EMSOFT)*, pages 290–298, 2005.

[49] J. Regehr and N. Cooprider. Interrupt verification via thread verification. *Electronic Notes in Theoretical Computer Science*, 174(9):139–150, 2007.

[50] M. Samak, M. K. Ramanathan, and S. Jagannathan. Synthesizing racy tests. In *ACM SIGPLAN Notices*, volume 50, pages 175–185. ACM, 2015.

[51] M. Samak, O. Tripp, and M. K. Ramanathan. Directed synthesis of failing concurrent executions. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 430–446. ACM, 2016.

[52] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[53] K. Sen. Effective random testing of concurrent programs. In *International Conference on Automated Software Engineering*, pages 323–332, 2007.

[54] K. Sen. Race directed random testing of concurrent programs. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 11–21, 2008.

[55] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *International Conference on Computer Aided Verification*, pages 419–423, 2006.

[56] SSE Instructions: Which CPUs Can Do Atomic 16B Memory Operations?, 2014. http://stackoverflow.com/questions/7646018/sse-instructions-which-cpus-can-do-atomic-16b-memory-operations.

[57] B. Steensgaard. Points-to analysis in almost linear time. In *ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 32–41, 1996.

[58] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Software. Eng.*, 10(2):203–232, 2003.

[59] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: The goblint approach. In *IEEE/ACM International Conference on Automated Software Engineering*, pages 391–402, 2016.

[60] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI)*, pages 115–128, 2003.

[61] 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 *Asia-Pacific Symposium on Internetware*, pages 199–202, 2015.

[62] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[63] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for java libraries. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 602–629, 2005.

[64] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a coverage-driven testing tool for multithreaded programs. In *International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 2012.

[65] T. Yu, X. Qu, and M. B. Cohen. Vdtest: an automated framework to support testing for virtual devices. In *International Conference on Software Engineering (ICSE)*, pages 583–594, 2016.

[66] T. Yu, W. Srisa-an, and G. Rothermel. Simtester: a controllable and observable testing framework for embedded systems. In *ACM SIGPLAN Notices*, volume 47, pages 51–62, 2012.

[67] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 221–234, 2005.