VDTest: An Automated Framework to Support Testing for Virtual Devices

Tingting Yu
Dept. of Comp. Sci.
University of Kentucky
Lexington, KY, 40506, USA
tyu@cs.uky.edu

Xiao Qu
ABB Corporate Research
Raleigh, NC, 27606, USA
xiao.qu@us.abb.com

Myra B. Cohen
Dept. of Comp. Sci. & Eng.
Univ. of Nebraska - Lincoln
Lincoln, NE, 68588, USA
myra@cse.unl.edu

ABSTRACT

The use of virtual devices in place of physical hardware is increasing in activities such as design, testing and debugging. Yet virtual devices are simply software applications, and like all software they are prone to faults. A full system simulator (FSS), is a class of virtual machine that includes a large set of virtual devices – enough to run the full target software stack. Defects in an FSS virtual device may have cascading effects as the incorrect behavior can be propagated forward to many different platforms as well as to guest programs. In this work we present VDTEST, a novel framework for testing virtual devices within an FSS. VDTEST begins by generating a test specification obtained through static analysis. It then employs a two-phase testing approach to test virtual components both individually and in combination. It leverages a differential oracle strategy, taking advantage of the existence of a physical or golden device to eliminate the need for manually generating test oracles. In an empirical study using both open source and commercial FSSs, we found 64 faults, 83% more than random testing.

CCS Concepts

•Software and its engineering → Software defect analysis; •Computer systems organization → *Embedded software*;

Keywords

Testing, Virtual Devices, Device Drivers, Test Oracles

1. INTRODUCTION

A full-system simulator (FSS), is a virtual machine (VM) (or software implementation) of the complete environment that executes target software on a physical machine [5, 14, 36]. Unlike process VMs or hypervisor-based systems (such as VMware) that rely on the host architecture to run the target machines [9], an FSS encompasses a variety of virtual devices that can simulate not only the processor cores and

memory, but also the entire hardware platform including the network, buses, and peripheral devices (e.g., keyboards, USBs, video adaptors). Usually, different devices can be combined to provide a large number of unique platforms.

FSSs are becoming widely used for many purposes in embedded and mobile domains where hardware is diverse, and new platforms are being released at a rapid pace. FSSs are used for tasks including system design, development, testing, debugging and security analysis. This relieves the engineer from having to own many different physical devices, and allows them to adapt quickly during hardware and software evolution. Developers can also interact with the FSS to implement customized tools for their target device, such as test case generators, or they can use additional FSS features such as profiling and provisioning.

Developing a virtual device can be a challenging task. The official documentation of hardware devices often contain inaccuracies and ambiguities [29], and thus the corresponding software implementation is unlikely to be fault-free. Yet defects in an FSS can have cascading effects. One of the earlier versions (v4.4) of the Android emulator did not rotate screens, preventing developers from testing or debugging any application that rotated [21]. Several studies have shown that software faults in virtual devices can cause security vulnerabilities [2–4]. For example, a critical vulnerability (called Venom) exists in the virtual floppy disk controller (FDC) code in the Quick Emulator (QEMU) FSS.¹ The fault stems from the FIFO buffer that the virtual FDC simulates to store commands from the CPU. The FIFO fails to reset the index allowing writes by the FDC to overflow. This security fault can propagate to the programs operating on the host platform [3].

Despite the existence of many automated software testing techniques, applying these directly to virtual devices is not straightforward. First, execution environments require special test drivers. Second, failures are often triggered by interactions between components. Third, failures often do not lead to crashes, making oracles difficult to obtain. Fourth, output states can be masked by input and output data points that are shared.

To overcome these challenges we propose an automated framework, VDTEST, that allows engineers to effectively test virtual devices within an FSS. First, VDTEST provides a test template generator that extracts basic device properties with little manual effort and uses these to generate test cases. Second, it utilizes a two phase testing approach. The first phase tests the behavior of individual components (e.g.,

¹QEMU has both a hypervisor and FSS version.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE '16, May 14–22, 2016, Austin, TX, USA

© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...\$15.00

DOI: <http://dx.doi.org/10.1145/2884781.2884866>

registers and data buffers), and the second phase integrates components, testing for interactions among them. Third, to address the oracle problem, VDTTEST employs a physical device or a gold-standard virtual device, (also called the *oracle device*), which leverages differential testing [29, 30]. Finally, to avoid masking effects [13], VDTTEST manipulates only necessary input parameters in a test case, leaving in-ert parameters unchanged. It uses a feedback-driven testing process to identify invalid/ineffective parameters that do not impact the hardware state (e.g., read-only registers), and then enforces constraints to omit these parameters in further iterations.

To evaluate VDTTEST we conducted an empirical study using 11 pairs of devices, obtained from one physical machine, two open-source FSSs and one commercial FSS. The results show that VDTTEST is effective at revealing faults; 82.3% more faults were detected than random testing. When compared with a traditional combinatorial interaction testing technique [11], it reveals 42% more faults given the same testing budget. The contributions of this work are:

- VDTTEST, a framework for testing virtual devices in an FSS; and
- An empirical study that shows VDTTEST can find new faults in existing FSS virtual devices, many of which have been confirmed by developers.

In the next section we present a motivating example and background. We then describe VDTTEST 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

An FSS sits on top of host operating system emulating all of the hardware for each supported device. It also can run guest operating systems and applications (both of which are normal versions of their respective programs). Figure 1 is a snippet of code extracted from a real virtual device, the PL031 timer device in the QEMU FSS. It contains a bug at line 19 [27]. The interrupt mask bit `s->im` is incorrectly set. The interrupt status register `s->is` should be set instead. The result is that interrupt alarms are not fired as expected.

The first challenge is that testing this code requires special test drivers – the execution environment must be properly modeled. For instance, I/O interface functions (i.e., entry functions) such as `pl031_write` on line 1 are invoked by the FSS to pass inputs to the virtual device. And the device transaction functions such as `pl031_set_alarm` (line 4) are invoked to perform I/O commands that may fire interrupts by calling `qemu_set_irq` (line 20).

Second, the function (line 16) containing the faulty statement can not be executed unless certain bits are set in both the data load register (RTC_LR) and interrupt mask (RTC_MR) register. As such, interactions between device components must be considered when testing a virtual device.

Third, faults often fail to propagate their effects to program outputs (e.g., crash). In such cases, output-based test oracles are inadequate. Thus, *internal oracles* are needed that allow engineers to inspect internal device states (e.g., register values) for correctness. Such states are referred to as *observable output points* [8]. In Figure 1 the results of RTC_LR and RTC_MR registers are observable output points.

```

1. static void pl031_write(...)
2. {
3.     case RTC_MR:
4.         pl031_set_alarm(s);
5.     case RTC_LR:
6.         ticks +=value - pl031_get_count(s); /*value is register content*/
7.         s->mr = value & 1 /*change RTC_MR*/
8.     ...
9. }

10. static void pl031_set_alarm(...)
11. {
12.     if (ticks == 0)
13.         pl031_interrupt(s);
14.     ...
15. }

16. static void pl031_interrupt(...)
17. {
18.     PL031State *s = (PL031State *)state;
19.     s->im = 1; // should be s->is = 1;
20.     qemu_set_irq(s->irq, s->is & s->im);
21. }

```

Figure 1: Example virtual device code

Finally, the observable output points are often the same as the input points, so they can be masked by an input value written to the same point. The input register components (RTC_LR and RTC_MR) are also used to observe device state. Setting the inputs of RTC_LR may affect the output of RTC_MR (line 7). If both registers are set with input values, the output effect of RTC_MR may be masked by its input.

These issues motivate our need for a special testing framework for virtual devices.

2.1 Virtual Devices

Figure 2 illustrates a typical I/O system with peripheral devices (the gray area reflects the devices and their I/O memory). A peripheral device is a device that is connected to a computer but is not part of the core computer architecture (e.g., CPU, motherboard and memory). It is controlled by reading and writing to its registers either within the memory address space (memory-mapped I/O) or the I/O address space (port-mapped I/O). For memory-mapped I/O, the device registers are mapped into the CPU’s address space. Thus, the device is accessed in the same way as a regular memory access. For port-mapped I/O, I/O device registers have a separate address space from the CPU address space. Thus, the device is often accessed through a special class of CPU instructions (e.g., `inb` and `outb` in X86).

Most devices have at least two types of registers. The first type is the *data register*, through which the input/output data is read from or written to the device. The second type is a *control register*, which selects and shows the mode of operations of the device. Certain bits in the registers are write-only or read-only. A device can also have data buffers that temporarily store the data from the CPU or its peripheral devices. Other devices interact with it, for example, the UART device use a FIFO buffer to transmit and receive data from the CPU.

In this work we are concerned with virtual peripheral devices, hence when we use the term *virtual device* we are referring to those which are peripheral.

2.2 Device Modeling

We represent a generic device D as a 5-tuple, $\langle base[word], mmap[bool], regs[R], bufs[B], deps[D] \rangle$, where the value within each bracket indicates the *type* of the property. For example, *base* is the base address of the device of the *word*

Figure 2: System with peripheral devices

type. The *mmap* is a *bool* type that describes if the device is memory-mapped (*true*) or port-mapped (*false*). The *regs*[*R*] describes the register set in the device, which includes varieties of registers. The type *R* (register) is modeled by $\langle \text{offset}, \text{size}, \text{value} \rangle$, describing the *offset* from the base address, the *size* of the register (e.g., 16-bit), and the content *value* in the register, where $0 \leq \text{value} < 2^{\text{size}}$. The *bufs*[*B*] describes the set of buffers in the device. This property is optional as not all devices have buffers. The type *B* (buffer) is modeled as $\langle \text{addr}, \text{value} \rangle$ where *addr* is the memory address of the buffer and *value* is the content in the buffer. The registers and buffers are device components that compose the basic structure of the device under test (DUT). The last element *deps*[*D*] represents other hardware devices whose states can be affected by this device (i.e., dependent devices DD). For example, an interrupt controller can change the state when a DUT sets its interrupt bit.

We next model the state of a device, which will be later used in the VDTEST's algorithm. A state *S* of a device *D* is denoted as $S_D = (\text{regs}, \text{bufs})$, where $\text{regs} = \langle r_1, \dots, r_i, \dots \rangle$ and $\text{bufs} = \langle b_1, \dots, b_i, \dots \rangle$. r_i denotes the value contained in the i^{th} register, and b_i specifies the content in the i^{th} data buffer. There are two types of actions that can trigger state transitions — **read** and **write** commands issued by the CPU. Last, we model the behavior of a device *D* using a state transition system (S_D, δ_D) , where $\delta_D: S_D \rightarrow S'_D$ is the state-transition function which changes a device state $S_D = (\text{regs}, \text{bufs})$ into a new state $S'_D = (\text{regs}', \text{bufs}')$, driven by the **read** and/or **write** actions. In the case where a DUT has dependent devices (e.g., $\text{deps} \neq \text{null}$), the transition system is extended to $(S_{DUT} \cup S_{DD_1} \dots \cup S_{DD_i} \cup \dots, \delta_{DUT} \cup \delta_{DD_1} \dots \cup \delta_{DD_i} \cup \dots)$, where $DD_i \in \text{deps}$.

2.3 Test Case Modeling

The behavior of a device is often changed by flagging register bits (0 and 1) or manipulating buffer contents. Thus, a test case is defined as $tc = \{R_{0,0} \dots R_{i,j}, B_0 \dots B_k\}$, where $R_{i,j}$ indicates that the j^{th} bit in the i^{th} register is flagged, and B_k indicates contents in the k^{th} buffer are changed.

Test cases for testing hardware device can be classified as being *stateless* or *stateful*. A stateless test case does not depend on the previous test case; the device state depends only on new values written by the test case. On the other hand, a stateful test case is a sequence of ordered test cases (test sequence), where the j^{th} test case in a sequence depends on the state resulting from the execution of the i^{th} test case, where i happens before j . For example, a test case flagging an interrupt clear bit takes effect only when the interrupt

enable bit is set by its earlier test case. In this case, $TC = \{t_i, t_j\}$ forms a stateful test case. A stateless/stateful test case execution is guided by a state transition on the DUT. VDTEST models three state transitions:

1. *Read*(DUT): $S \rightarrow S'$. This describes the device state changed from *S* to *S'* after reading the registers of the DUT, where no test inputs are supplied. While it might seem counter-intuitive, a state change can occur by just reading from a register. Such a transition often applies to the registers with self-clearing bits. For example, the modem status register (MSR) in the UART device is reset each time when it is read after a write operation.
2. *Write*(DUT): $tc \Rightarrow S \rightarrow S'$. This describes the device state change from *S* to *S'* after flipping the bits specified in a test case (*tc*) in the DUT.
3. *WriteS*(DUT): $tc_1 \cup \dots \cup tc_l \Rightarrow S \rightarrow S'$. This describes the device state changed from *S* to *S'* after executing a stateful test case (test sequence) composed of $tc_1 \cup \dots \cup tc_l$ in the DUT, where *l* is the length of the test sequence.

2.4 Test Oracle Modeling

An expected device state can be used as a *test oracle*. The most accurate way to obtain this state, is to use a *hardware oracle* and to compare the output (i.e., state) of the DUT to the state of its corresponding physical device. In the case of non-existing physical devices, an alternative approach is to use a well-developed virtual device from another FSS to conduct differential testing [29, 30]. It is also possible that no readily available oracles exist, when a device is newly designed for example. VDTEST can still work in this context, presumably the device specifications can be converted into test oracles. In any of the above cases, the artifact (i.e., a physical device, a device model from other FSSs, or a device specification) that can produce an expected output state is considered a golden device, denoted by D_o in this paper.

3. VDTEST

We introduce VDTEST whose architecture is shown in Figure 3. The dark gray boxes contain the major components: a *static analyzer*, a *test case generator*, and an *output differ*. VDTEST first employs static analysis (SA) to generate test specifications. The static analyzer takes as input the device source code, a pre-defined test template skeleton that specifies device properties (e.g., register address offset), and a user provided annotation plugin. Next, VDTEST runs the *test generator* to translate the test specification into two *test drivers*, one for the oracle device (shown as part of the *Golden System*) and the other for the virtual device under test (shown as part of the *Target System*). The test generator uses combinatorial testing to create test cases.

After test cases are generated, VDTEST executes each test case both on the oracle device (i.e., D_o) and the DUT (i.e., D_t), using the three types of state transitions described in Section 2.3. Last, the *output differ* detects and shows the differences between the output of each test case from D_o and the DUT. A difference indicates a possible fault in the DUT.

Both the static analyzer and test generator modules are configurable so that we can, for example, use a variant of combinatorial test generation, or perhaps random test generation instead. We can also completely disable the static

Figure 3: VDTTest architecture

analyzer and generate test specifications manually in situations when the source code of DUT is unavailable.

We describe each component of VDTTest in more detail next. Figure 4 (left) provides a code snippet for the PL050 device from QEMU. The PL050 is a keyboard interface device that directs communications between the CPU and external keyboards. We will use this example throughout the rest of this section.

3.1 Test Specification

VDTTest utilizes a test specification skeleton, which specifies a list of device properties needed to generate test drivers and test cases. The specification skeleton is defined only once and generic to all DUTs, so engineers do not need to manually write it for each DUT.

A specification skeleton models the basic properties for a DUT, including both mandatory and optional properties. Figure 4 (middle) is an example that presents an overview of the test specification. The device properties are defined as elements (e.g., **name**). The value of each mandatory property (indicated by “_”) and optional property (indicated by “_.”) is initially set to *null* by default in the skeleton, and can then be instantiated by the static analysis. The root element **device** refers to the DUT. There are three properties defined for this element, **name**, **base**, and **io**. The **name** property defines the device’s name, **base** property specifies the base address of the DUT, and the property **io** indicates the I/O mapping policy.

The elements at the second level refer to the device components, (i.e., registers and data buffers). The children elements under each component specify the properties of the component. In this example, the **offset** and **size** elements under the register component, and the **address** and **size** under the buffer component are mandatory properties, whereas the other elements are optional properties. Here, specifying **input** enables VDTTest to limit input space to specific values rather than an entire range of the input domain (e.g, $1 - 2^{16}$). The **read-only** indicates that the register is a read-only register. The **interrupt** indicates that writing to the register may affect interrupt status, and thereby the state of interrupt should be checked at the *output differ* component. The **depend** specifies components that may have combinatorial effects with the current component. The **output** indicates that reading/writing the current component may affect other components, such that the current component must be tested individually first.

3.1.1 Static Analysis

The specification skeleton can be instantiated by SA, with the mandatory property values and possible optional property values analyzed and specified. The only manual step is the annotation plugin, which specifies a list of device entry points. The information is provided by users who are supposed to be domain experts. This annotation strategy has been widely adopted by existing static analysis techniques for device drivers [7, 22, 24]. Figure 4 (right) illustrates a sample of annotation plugin for the PL050 code snippet.

The SA module first identifies the base address of the DUT in the system-specific source files which are specified by the **file** element under **pluginsConfig.System**. Here the base address is found in **arm/versatilepb.c** in QEMU for ARM Versatile Platform. The two properties for the root element **device** are instantiated in the same way.

The register entry points are identified by **pluginsConfig.EntryWrReg** and **pluginsConfig.EntryRdReg** annotations, indicating register write and read points. The **function** specifies the name of the entry function for all registers, the **address** indicates the variable used for a register address offset, and the **value** defines the actual data written to a register. The buffer entry points are specified by the **pluginsConfig.EntryBuf** annotations. The optional buffer property **address** is often defined by a data structure, which is obtained by analyzing disassembling files.

To identify property values for registers and buffers, VDTTest constructs a system dependency graph [10, 19] starting from the entry point of registers and buffers, as shown in Figure 5. The **offset** value for a register can be obtained by its definition. **KMIDATA** is control dependent on **offset** and its value is **0x008** defined in the header file. The **size** of input can be obtained by examining the size of **value** (i.e., **sizeof(uint64_t)**). To obtain input values, VDTTest tracks **value** propagation from the entry point to the constant values assigned. In Figure 4 (left), the possible input values of **kmiData** can be traced to the function **ps2_write_keyboard** and to the constants **0x00** and **0x05** (line 16 and line 19). The constants are mapped into the **input** element in the test specification. If the constant value can not be located, the range of the input value is $[0, 2^n - 1]$, where n is the register bit size. Since **KMICR** is control dependent on **KMIDATA**, the two components are considered to have combinatorial effects on the device state. **KMIMR** is control dependent on **KMICR**, but is written by a value. This implies that change of **KMICR** may affect the output of **KMIMR**.

<pre> 1. #define KMIDATA 0x008 2. ... 3. void pl050_write(addr offset,uint64_t value,...){ 4. switch (offset) { 5. case KMIDATA: 6. ps2_write_keyboard(s->dev, value); 7. if (KMICR == 0x3) 8. ... 9. case KMICR: 10. KMIMR = val 11. ... 12. } 13. } 14. void ps2_write_keyboard(void *opaque, int val){ 15. switch(val) { 16. case 0x00: 17. ps2_queue(&s->common, KBD_REPLY_ACK); 18. break; 19. case 0x05: 20. ps2_queue(&s->common, KBD_REPLY_RESEND); 21. break; 22. ... 23. } 24. } 25. void ps2_queue(void *opaque, int b){ 26. ... 27. qemu_set_irq(s->irq, raise); 28. } </pre>	<pre> <device name="pl050_keyboard" base=0x10006000 io="memory-mapped" > <register1> <offset> 0x008 </offset> <size> 64 </size> <input>[0x00, 0x05,...]</input> <read-only> false </read-only> <interrupt> true </interrupt> <depend> [register2,...] </depend> <affect> [register3,...] </affect> ... </register1> <buffer1> <address> 0x168 </address> <size> 16 </size> </buffer1> </device> </pre>	<pre> pluginsConfig.System = { name = "pl050_keyboard" file = "arm/versatilepb.c" io = "memory-mapped" interrupt = "qemu_set_irq" } pluginsConfig.EntryWrReg = { function = "pl050_write" address = "offset" value = "value" } pluginsConfig.EntryRdReg = { ... } pluginsConfig.EntryBuf = { function = "ps2_read_data" address = "" value = "PS2Queue.data" } </pre>
--	---	--

Figure 4: Code snippet from GEM5 (left), test specification (middle), and sample Annotation plugin (right)

If a register is included in the read entry function (`pluginsConfig.EntryRdReg`), but is not included in the write entry function (`pluginsConfig.EntryWrReg`), it is a read-only register. Since read-only registers are independent from other registers, changing them does not affect the device state. Thus, a test case involving interactions between **read-only** registers and other components can be eliminated. In this example, `KMIDATA` is writable, so its **read-only** property is false.

Writing to a device may lead to changing status of interrupts. To determine whether an entry point is associated with the interrupt, we track the data and control flow from the entry point of a register/buffer to the function that can set interrupt status. In this example, writing to register `KMIDATA` causes `PL050` to raise interrupts (line 27). As such, interrupt status is considered as an observable point, which is included in the oracles.

If optional property values cannot be obtained by static analysis, developers can choose to manually add these values. For example, the `input` property may not be always presented as constants in the program, so engineers can manually select range or specific values as inputs.

3.2 Test Driver

VDTEST converts the test specification into a *test driver* (TD). VDTEST considers two classes of test drivers, which can be applied to both physical and virtual devices. The first class handles applications running on a machine with operating systems. Since such systems do not allow user-level programs to access hardware or memory, the TD was implemented as a kernel module used to communicate with the device. Specifically, VDTEST maintains an operation table that maps the elements of the test specification (e.g., elements in Figure 4 (middle)) into kernel-level system calls. These system calls are used to communicate with hardware (e.g., `ioremap`, `inb`, `outb`, `inl`, `outl`). VDTEST then parses the test specification and replaces its elements with the cor-

Figure 5: Static analysis

responding system calls by querying the operation table. For example, the attribute `io` is replaced with `ioremap` and `request_mem_region` to allocate I/O port region. Finally, a C file is generated and compiled into the test driver (kernel module). As an example, we refer to the test specification in Figure 4 (middle), which generates the following C snippet:

```

...
ret = ioremap(IO_BASE, io_size);
outl(input, IO_BASE + IFLS);
...

```

The second class of TD handles a *bare-metal* machine (i.e., a computer on which an application is running without the operating system). We distinguish it from the first class because executing test cases does not require calling kernel APIs. In this case, VDTEST converts the read and write operations with respect to specific memory addresses from the test specification into a TD source file.

3.3 Testing Approach

VDTESTS testing approach is guided by the *test generator* and *output differ* and works both with test cases and test oracles. We show its main algorithm in Figure 6. The algorithm VDTEST takes a DUT (D_t) and an oracle device (D_o) as its inputs, and outputs the faults (denoted as F) detected in the DUT. The `CHECKRESULT` function is called by the *output differ* component to compare the output states of D_t and D_o (lines 51-57). The algorithm begins by checking the initial state S_0 of D_t and D_o by the *read* state transition

Algorithm VDTTEST

```
1: Inputs:  $D_t, D_o$ 
2: Outputs:  $F$ 
3: begin
4:    $(S'_{0_t}, S'_{0_o}) = \text{Read}(S_{0_t}, S_{0_o})$ 
5:    $F = F \cup \text{CheckResult}(S'_{0_t}, S'_{0_o})$ 
6:    $S_t = S_o = S_{0_o}$ 
7:   for each read-only register  $R$  in  $D_t$ 
8:     verify  $R$  against  $D_o$ 
9:   endfor
10:  for each non-read-only register  $R'$  in  $D_t$ 
11:     $\text{UnitTest}(R')$  /*test at unit-level*/
12:  endfor
13:   $\text{IntegrationTest}(R[C])$  /*test at integration-level*/
14: end
```

Function UnitTest

```
15: Inputs:  $R$  /*single registers*/
16: begin
17:    $TC_1 = \text{computeTC}(R, 1)$  /*strength-1 tests*/
18:   for each test  $tc \in TC_1$ 
19:      $(S'_t, S'_o) = \text{Write}(tc, tc)$ 
20:      $F = F \cup \text{checkResult}(S'_t, S'_o)$ 
21:      $(S''_t, S''_o) = \text{Read}(S'_t, S'_o)$ 
22:      $F = F \cup \text{checkResult}(S''_t, S''_o)$ 
23:   endif
24: endfor
25:   while  $t > 1$  and  $t < N$ 
26:      $TC_t = \text{computeTC}(R, t)$ 
27:     for each test  $tc \in TC_t$ 
28:        $(S'_t, S'_o) = \text{Write}(tc, tc)$ 
29:        $F = F \cup \text{checkResult}(S'_t, S'_o)$ 
30:     endfor
31:   endwhile
32: end
```

Function IntegrationTest

```
33: Inputs:  $R[C]$ 
34: begin
35:   while  $t > 1$  and  $t < N$ 
36:      $TC_t = \text{computeTC}(R[C], t)$ 
37:     for each test  $tc \in TC_t$ 
38:        $(S'_t, S'_o) = \text{Write}(tc, tc)$ 
39:        $F = F \cup \text{CheckResult}(S'_t, S'_o)$ 
40:       if  $\text{Compare}(S'_o, S'_{0_o})$  is false
41:          $TC'_t = TC_t \cup tc$ 
42:       endif
43:     endfor
44:   endwhile
45:    $\text{SEQ} = \text{TestSequence}(TC'_t \cup TC_{p1}, l)$ 
46:   while each test sequence  $ts \in \text{SEQ}$ 
47:      $(S'_t, S''_o) = \text{writeS}(ts, ts)$ 
48:      $F = F \cup \text{CheckResult}(S'_t, S''_o)$ 
49:   endwhile
50: end
```

Function CheckResult

```
51: Inputs:  $S, S'$ 
52: begin
53:   if  $\text{Compare}(S, S')$  is true
54:     print "a fault is detected"
55:     reset to  $S_0$ 
56:   endif
57: end
```

Figure 6: VDTTEST algorithm

(lines 4-5). A fault is reported if the initial states of the two devices are different. Next, the states of both D_t and D_o are reset (line 6). The reset operation is performed throughout the testing process upon the completion of a test execution. This is important as test cases are not independent. A test

case can change the state of the hardware, so two test cases may yield different hardware states, even if both are identical. After setting the initial state, the algorithm verifies the read-only registers specified in the test specification (line 8). To do this, for each such register in the oracle device, the algorithm flips all its bits. If the state remains unchanged, the register is truly read-only, otherwise a fault is reported. Next, the algorithm begins with unit-level (Phase 1) testing for each non-read-only register (lines 10-12). It then proceeds to integration-level (Phase 2) testing for the whole device taking inputs as a set of registers that have potential combinatorial effects (line 13). Both phases use a combinatorial testing approach.

3.4 Unit-level Testing

The Phase 1 testing tests individual components of DUT (lines 15-32). The components that are not tested are left in their original state. As such this is analogous to unit testing. Since the data buffer does not contain parameters like registers, its content is randomly generated at this level. As for the registers, each t -bit combination in a register is flipped at least once, where t is the strength of testing. The test cases generated by flipping bits are done by `ComputeTC`. This is related to fault injection [25], but applied in a different context. The algorithm begins with testing using strength 1 (line 17). A strength-1 test case for register i is defined as $tc_{i,1} = S_0 \hat{\ } (1 < j)$, where $\hat{\ }$ is the exclusive OR operation and j is the position of the bit to be flipped while other bits in the register stay unchanged. This is important because the input and observable output points on a hardware device are the same, so writing values to any of these points may prevent errors contained in such points from being observed at the output. For example, on a register STAT with initial state is 0x4 (0010), there are four test cases generated for strength-1 coverage by flipping each single bit one at a time: $tc_1 = (1, -, -, -)$, $tc_2 = (-, 1, -, -)$, $tc_3 = (-, -, 0, -)$ and $tc_4 = (-, -, -, 1)$, where “-” indicates the bit is untouched.

Each test case is executed on both D_t and D_o using the *write* state transition (line 19). The algorithm then compares the resulting states of D_t and D_o (line 20). Note that the change of one device component may affect the states of other components, so the state comparison is done at the entire device-level, including interrupt controller if the *interrupt* property is set to true in the test specification. For example, flagging bits in IER register changes the value of IIR register in the same device, as well as the interrupt controller. After the *write* state transition is done, the algorithm performs a *read* state transition from S' to S'' to check whether DUT correctly simulates the self-clearance registers (line 21). If an observable difference is found, a fault is reported (line 22).

VDTTEST next enables incremental strength- t testing, where t is increased up to N , the maximum strength determined by the testing budget allocated for phase 1 (lines 25-31).

3.5 Integration-level Testing

The objective of the Phase 2 testing is to test interactions among registers and data buffers (lines 33-50). We focus on registers that have potential combinatorial effects, denoted by $R[C]$, which are specified in the test specification under the **register** element and its **depend** property. Like Phase 1, the data buffer values are randomly generated. A strength- t combinatorial testing at the integration level aims to test

value combinations of each t registers in $R[C]$. Here, the test cases generated for each register at the Phase 1 are its values. For example, the **STAT** register (example used in Phase 1) has four values. Similar to the Phase 1 algorithm, the registers not contained in the new combination remain in their original states. Suppose we have a strength-2 test suite created for a DUT that contains four registers — R1, R2, R3, and R4, where R2 is read-only (i.e., no need to be combined with others) and the **depend** property of R3 is [R1, R4]. Thus, $R[C] = \{R3, R1, R4\}$. There are three combinations to be covered — (R1, R3), (R1, R4) and (R3, R4) in Phase 2. The algorithm separates out the test cases that can yield device state change and adds them into TC'_i ; TC'_i is further used to generate stateful test cases.

Next, the algorithm generates stateful test cases (test sequences) to test *consecutive* state transitions. Given a test length l and stateless test cases from Phase 1 (TC_{p1}) and Phase 2 (TC'_i), the algorithm iteratively selects l test cases to form a stateful test case ts and adds it into SEQ (line 45). This is done by selecting only stateless cases that induce changes in the first two phases rather than the exhaustive set of permutations of all stateless tests (line 40). Note that a test sequence is an ordered event sequence, because changing the order of the state transitions may bring the device into a different state. Suppose $R1:0xb$ and $(R1:0xa, R3:0xb)$ are two test cases generated from Phase 1 and Phase 2 respectively, and both yield state changes. The two stateless test cases form two test sequences with length 2 — $R1:0xb \rightarrow (R1:0xa, R3:0xb)$ and $(R1:0xa, R3:0xb) \rightarrow R1:0xb$.

4. EMPIRICAL STUDY

To assess VDTTEST we explore three research questions.

RQ1: How effective is VDTTEST at detecting faults in real-world virtual devices?

RQ2: To what extent does the use of oracle device in VDTTEST affect its effectiveness?

RQ3: What types of faults can VDTTEST detect?

The first research question evaluates whether the VDTTEST approach is cost-effective in terms of fault detection. The second research question lets us further investigate whether the use of an oracle device can improve VDTTEST's effectiveness. The third research question allows us to study the faults and classify them into different categories.

4.1 Objects of Analysis

To obtain objects of analysis, we used virtual devices from three FSSs that are widely used in both industry and academia — SIMICS [14], QEMU [5], and GEM5 [6]. SIMICS is a commercial virtual platform. with no source code provided. In this case, the test specifications were written according to the specification manuals by a graduate student who has three years experience in embedded system development. QEMU and GEM5 are two open source FSSs, containing 30K and 287K non-comment lines of code in the device codebases, respectively. To evaluate VDTTEST, we need both DUTs and oracle devices. We searched virtual devices that are contained in at least two FSSs with respect to their device models. Eight virtual devices matched this criteria. All are based on the ARM architecture. We used the eight VDs in GEM5 as DUTs, and the VDs in QEMU as oracle devices, because QEMU is more widely used. Each DUT is paired with an oracle device, resulting in eight pairs

Table 1: Objects of Analysis

VD	R E G	B U F	N_d	T_d	NC_1	NC_2	NR_1	NR_2
UART8250(S-P)	12	2	19,042	22.3	2,128	17,432	20,980	2,098,000
UART8250(G-P)	12	2	18,430	20.2	2,094	17,988	18,982	1,898,200
DEC21143(S-P)	41	2	28,456	32.8	338	25,954	29,443	2,944,300
PL011(G-Q)	14	1	17,344	20	744	15,864	18,475	1,847,500
PL031(G-Q)	8	2	10,298	10.6	360	9,542	12,006	1,200,600
PL050(G-Q)	5	2	13,840	16.9	483	12,368	14,868	1,486,800
SP804(G-Q)	7	2	15,482	16.5	377	14,682	16,902	1,690,200
A9SCU(G-Q)	3	0	4,318	3.9	256	4,028	5,088	508,800
PL110(G-Q)	13	2	20,562	24.7	721	18,058	21,873	2,187,300
i8254(G-Q)	4	0	5,082	2.2	110	3,532	3,902	390,200
MCRTC(G-Q)	4	0	5,948	2.9	156	7,488	8,560	856,000

of objects. Additionally, given the availability of physical hardware, we also selected two VDs from SIMICS and one from GEM5 as DUTs for which we had hardware, and use their corresponding physical devices as oracle devices. Together there are eleven pairs of analysis objects selected, as shown in Table 1, column 1 (VD).

Table 1 provides the characteristics of each device and pair. The notation $X-Y$ in parentheses indicates a pair of DUT (X) and its oracle device (Y), where “P” indicates a physical machine, “S” indicates SIMICS, “G” indicates GEM5, and “Q” indicates QEMU. We give the number of registers (REG, column 2) and buffers (BUF, column 3).

Other columns are described in Section 4.3. UART8250 is an integrated circuit implementing the interface for serial communications. DEC21143 is a fast Ethernet LAN controller providing a direct interface to the PCI bus. DEC21143 has 19 configuration registers, 18 command and status registers (CSRs) and 4 CardBus status changed registers; we consider only CSRs because the other registers are filled with pre-defined values. PL011 is a UART device for the ARM architecture. PL031 is a real time clock device used to provide a basic alarm function or as a long term base counter. PL050 is a Keyboard/Mouse Interface. SP804 is a Dual-Timer module that can generate interrupts on reaching zero. A9SCU is a snoop control unit that connects processors to the memory. PL110 is a Color LCD Controller that provides control signals to interface directly via a variety of color and monochrome LCD panels. i8254 is an Intel interval timer device designed to solve the common timing control problems in microcomputer system design. It is used to bring down the timing frequency to customized levels. MCRTC, short for MC146818RTC is a Real-Time clock with alarm and one hundred year calendar, a programmable periodic interrupt and square-wave generator, and a static RAM.

4.2 Variables and Measures

Independent variable. Our independent variable involves the testing techniques used. In addition to VDTTEST, to address RQ1 we enable two testing techniques — a combinatorial interaction testing approach (CIT) [11] and a random testing approach (RANDOM). Test suites generated by CIT contain every combination of t input parameters at least once in the test suites. In our context, the input parameters in Phase 1 are bits and those in Phase 2 are register and buffers. While CIT may be less expensive than the VDTTEST engine (it generates fewer test cases), by maximizing input parameter coverage in each test case, it neither eliminates masking effects nor prunes independent options. In addition, CIT does not generate stateful test cases without additional modeling.

Like VDTTEST, CIT was applied in two phases, to generate

tests for both individual registers and for the whole device. We generated two different sets of CIT test cases. For the first (denoted as CIT_1), we used the same testing strengths as used in the VDTTEST; this lets us examine the relative effectiveness of the two approaches on equivalent strengths. For the second set (CIT_2), we used the same testing time required by VDTTEST and generated multiple test suites; this lets us examine the relative effectiveness of the approaches when each is given the same amount of testing time.

We also use a random testing technique (RANDOM). RANDOM has been well studied as a baseline approach for traditional CIT techniques [32]. Test cases are generated by randomly changing the bit values of registers and buffers. Like CIT, we generated two sets of RANDOM test cases. The first (denoted as $RANDOM_1$) used the same amount of testing time required by VDTTEST. The second (denoted as $RANDOM_2$) used the number of test cases in the first set multiplied by 100 (to give random a higher chance of success). This lets us examine how the effectiveness of the VDTTEST test generator compares to that of a more robust random testing process.

To address RQ2, we disabled the oracle devices and used only observable outputs as oracles (i.e., output-based oracles). This includes exceptional behavior that results in program crashes and error messages (e.g., missing functionality, invalid memory access). We compared VDTTEST with oracle devices to those with only output-based oracles, denoted by $VDTTEST_{no}$.

To address RQ3, we manually classified fault types. We classified the faults into five categories. The first category (C1) are initialization faults, in which device registers/buffers are initialized with incorrect values. The second category (C2) is when we have an incorrect property of single register bits. For example, we use C2 when a read-only bit is writable and a reserved bit changes its status during testing. The third category (C3) was used when incorrect functionality of a device component/bit was affected by the component/bit actually being written. For example, when writing to register A leads to an incorrect state of register B. The fourth category (C4) is used for missing functionality – certain features in the device are not implemented. The fifth category (C5) are faults that are interaction faults – they are triggered by at least two device components/bits.

Dependent variables. As dependent variables, we measure the *effectiveness* in terms of the *fault detection*. We compare the numbers of *unique* faults (determined by inspecting source code) detected by test cases for each technique, VDTTEST, CIT_1 , CIT_2 , and $RANDOM_1$ and $RANDOM_2$.

4.3 Study Operation

We conducted our experiment on a physical X86 machine (the physical devices used in our study) and three FSSs that can simulate both X86 and ARM machines (the virtual devices used in our study). Each X86 physical/virtual machine runs a preemptive Linux kernel version of Fedora Core 2.6.15. The ARM machine is bare-metal, which means it does not come with an operating system. We chose Linux because it runs on a wide range of known architectures, which makes it possible to, with small modifications, extend the VDTTEST to the hardware on other architectures. In addition, the popularity and complexity of the x86 and ARM architectures make it easier to port VDTTEST to other architectures. On the simulated machine, we used the pro-

gramming interfaces that FSSs provide, which allow us to directly control and observe the hardware states. However, other virtual platforms such as OVPSIM [34] can also be used to instantiate the framework. The *static analyser* is built on CodeSurfer [10] using the system dependence graph (SDG). We implemented a plugin module that takes as input the *Annotation Plugin* and *Test Template Skeleton* along with the SDG to generate test specifications. We enabled both data dependencies and control dependencies to track data and control flow from device entry points.

On the FSSs, the *execution engine* is a built-in module, which takes the test driver to execute test cases by writing and reading registers and hardware buffers. We implemented the *execution observer* as external modules attached to FSSs. The APIs provided by FSSs allow us to observe system states at arbitrary point. On the X86 physical machine, the *execution engine* is the operating system, which executes the test driver as a kernel module. We implemented the *execution observer* by using source code instrumentation (i.e., `printf`) to log device states into a file.

To implement the testing process of VDTTEST, the maximum testing strength in the first phase is set to 3 and that in the second phase is left at 2. The length of the test sequence l is set to 2. The insight behind choosing the strengths is the observation that in most cases the appearance of a failure depends on the combination of a small number of parameter values of the DUT [23].

Returning to Table 1, we show the details of our test generation. Column 5 (N_d) lists the number of test cases generated by VDTTEST. Columns 6 (T_d) reports the time required for running these test cases in minutes. Columns 7 and 8 (N_{C1} and N_{C2}) list the number of test cases generated by CIT_1 and CIT_2 respectively. We used the ACTS [1] tool to generate CIT test cases. For each of the two CIT techniques, we set the same strengths as used in VDTTEST (i.e., 3 for the first phase and 2 for the second phase). Column 9 (N_{R1}) and Column 10 (N_{R2}) list the number of test cases generated by $RANDOM_1$ and $RANDOM_2$.

All four techniques involve randomization, therefore we ran each ten times. In addition, as the buffer value is randomly set, the effectiveness of the tests depends not only on the registers but also on the actual content of the buffers. As such, for each test (register configuration), we run it ten times with different random buffer values. In total, then, we conducted 100 runs for each of the four techniques.

4.4 Threats to Validity

The primary threat to external validity involves the representativeness of our programs. Other programs may exhibit different behaviors and cost-benefit tradeoffs. The programs we investigate are from several popular FSSs and the faults we aim to detect are real. Another threat is when choosing a virtual device as the oracle, it does not always guarantee that the oracle device is correct. This threat can be controlled by selecting more robust FSSs as oracles.

The primary threat to internal validity is possible faults in the implementation of our approach. We controlled for this threat by extensively testing our tools and verifying their results against a smaller program for which we can manually determine the correct results. A second source of potential threats involves test oracles used. Any deviations from the oracle devices are reported as failures. It is possible that the oracle devices may contain faults that may lead to the

report of false positives. We controlled for this threat by using robust FSSs. We also confirmed the deviations with developers on both sides. It is possible that the deviation between DUT and physical device does not mean a real fault, but it is just because that the testing on the physical device result in nondeterministic results due to irregular state of other hardware components (e.g., cache, scheduling, lack of memory, etc). We controlled this threat by running each technique multiple times.

Where construct validity is concerned, numbers of faults detected are just two variables of interest. Other metrics such as the cost of manual effort could be valuable.

5. RESULTS

Table 2 reports the results observed (the cumulative faults detected) in our study; we use this table to address our first two research questions.² The numbers in parentheses indicate the number of faults detected in Phase 2. The numbers in brackets are the standard deviations across runs.

RQ1: Effectiveness of VDTTEST. Column 2 of Table 2 reports the number of faults detected by VDTTEST. VDTTEST detected 64 real faults. We reported the deviations between GEM5 and QEMU to the GEM5 developers and the faults were confirmed although priorities and fixes have not yet been implemented. We also reported the faults in SIMICS, but because we are using an academic version, there is limited support and were not able to get a confirmation.

Columns 3-4 in Table 2 report the numbers of faults detected by the two sets of CIT techniques (CIT_1 , CIT_2). As the data shows, CIT_1 detected 42 faults and CIT_2 detected three more faults. All 45 faults detected by CIT techniques were also detected by VDTTEST. VDTTEST detected 19 additional faults. On eight pairs (out of eleven), VDTTEST detected more faults than CITs, with improvements ranging from 33.3% to 100%. These results show that the VDTTEST is more effective at detecting faults than traditional CIT.

As shown in Column 5 and Column 6 of Table 2, the two RANDOM test suites together detected only 34 faults. Compared to VDTTEST, VDTTEST was more effective in nine out of eleven subject pairs (all except A9SCU(G-Q) and i8254(G-Q)), with fault detection improvement ranging from 50% to 133.3%. These results show that the VDTTEST is substantially more effective than RANDOM.

RQ2: Effectiveness of Device-based Oracles. Column 7 in Table 2 reports the numbers of faults detected without an internal oracle $VDTest_{no}$. Of the 64 faults detected by VDTTEST, $VDTest_{no}$ revealed only 23 faults. Clearly, device-based oracles substantially improved the fault detection effectiveness, compared to observable output-based oracles. We further examined the data and discovered that all 23 faults were related to missing features that led to output errors or crashes (i.e., observable faults). For example, when the test case tried to write to an unimplemented register in GEM5, an error “writing to invalid memory” was generated, followed by program termination.

RQ3: Fault Categorization. 11 faults (17%) belong to the initialization ($C1:Init$ category). For example, the initial values of two registers (RIS and FR) in the PL011 are inconsistent with those in golden devices. Six out of 64 (10%)

faults, belong to the $C2:bits$ error, these stem from incorrect implementations of specific bits (i.e., read-only, write-only and reserved bits). For example, on UART8250 in Simics, the values of two read-only registers, IIR and MSR, were changed unexpectedly by some test cases. The write-only register THR is expected to return zero, whereas the device returns a non-zero value. A few reserved bits³ in DEC21143 do not respond when they are written. The writes to the reserved bits changed the register values while these bits were not modified in the physical device.

16 faults (25%) belong to the $C3: incorrect\ functionality$ category, and are caused by incorrect implementation in the virtual devices. For example, the writes to the IER register of the UART8250 returned different values than the physical device did. The reason is that the IIR register in the UART8250 did not react to the change of the interrupt bits. The reason is because the automatic self-clearance mechanism was not implemented for the IIR and LSR registers. On the device PL031, when LR register is written, MR register is updated to the same value as LR as, but the golden device did not update MR. On device DEC21143, in a few cases, the device failed to enable interrupt bits when the receiving and transmitting processes were stopped, but the golden device did. On device SP804, when the 7th bit is set to 1, the CD register was not changed but the golden device did.

Nine faults (14%) are caused by $C4:missing\ functionality$. For example, the FIFO buffer in the UART8250 on GEM5 is missing. Three registers (RSR_ECR, DMACR, and LPR) in PL011 are not implemented.

Finally, there are 22 faults (34%) that we classified as interaction faults $C5: Interaction$ category, of which 9 faults were detected at the register level, and 13 at the device level. For example, on Simics UART8250, the interaction fault occurred in the FIFO data buffer. This buffer was not enabled when the associated registers (IIR and FCR) were set. A few interesting faults occurred due to interactions among registers and interrupts. On DEC21143, the device can raise early interrupts – right after a frame has been put to the internal transmit FIFO buffer. The device failed to trigger the interrupt on these early interrupts.

Of the 22 interaction faults, ten faults were detected by the test sequences (i.e., stateful test cases) from VDTTEST, and only 5 faults were detected by CIT (also detected by VDTTEST). This indicates that exercising consecutive state transitions contributes to enhancing the effectiveness for testing virtual devices. We also observed the differences when changing the order of stateless test cases in a test sequence. For example, a fault occurred in PL011 when executing a stateful test case (DR:0x40, IMSC:0x20) → ICR:0x20. The first stateless test case (DR:0x40, IMSC:0x20) changed the values of the RIS and MIS registers and the status of the interrupt controller PIC. A fault occurred because executing second stateless test case ICR:0x20 did not revert the bit values in MIS and PIC to their original values as the golden device did. This fault was not detected when the order of the two stateless test cases were flipped in the sequence. In fact, interrupts are sensitive to the order of state transitions.

6. DISCUSSION

In this section, we examine the influence of several tunable

² Artifacts and experimental data are available at <http://cs.uky.edu/~tyu/research/vdtest>

³ reserved bits or registers are reserved for future special use and do not perform any function.

Table 2: Fault Detection Effectiveness. Total Cumulative Faults, (Phase 2 Faults) and [standard deviation]

Virtual Device	VD	CIT ₁	CIT ₂	Ran ₁	Ran ₂	VD Test	Test _{no}
UART8250(S-P)	8 (3) [0]	5 (1) [0]	5 (1) [0]	4 (1) [0.32]	4 (1) [0.32]	4 (1) [0]	
UART8250(G-P)	5 (1) [0.32]	3 (0) [0.32]	3 (0) [0.32]	3 (0) [0]	3 (0) [0.32]	0 (0) [0]	
DEC21143(S-P)	21 (4) [0.32]	13 (2) [0.32]	15 (2) [0.32]	10 (1) [0.52]	11 (1) [0.52]	12 (2) [0]	
PL011(G-Q)	7 (2) [0]	5 (1) [0.48]	5 (1) [0.48]	3 (0) [0]	3 (0) [0]	3 (0) [0]	
PL031(G-Q)	2 (1) [0]	1 (1) [0]	1 (1) [0]	1 (0) [0.32]	1 (0) [0]	0 (0) [0]	
PL050(G-Q)	4 (2) [0]	3 (1) [0]	3 (1) [0]	2 (1) [0.48]	2 (1) [0.32]	0 (0) [0]	
SP804(G-Q)	8 (2) [0]	4 (1) [0]	5 (1) [0.32]	4 (1) [0.52]	4 (1) [0.48]	0 (0) [0]	
A9SCU(G-Q)	1 (0) [0]	1 (0) [0]	1 (0) [0]	1 (0) [0.52]	0 (0) [0.48]	0 (0) [0]	
PL110(G-Q)	4 (2) [0.32]	3 (1) [0.32]	3 (1) [0]	2 (1) [0.32]	2 (1) [0.52]	2 (0) [0]	
I8254(G-Q)	1 (0) [0]	1 (0) [0]	1 (0) [0.32]	1 (0) [0.48]	1 (0) [0.48]	0 (0) [0]	
MCRTC(G-Q)	3 (1) [0]	3 (1) [0]	3 (1) [0]	2 (0) [0.32]	2 (0) [0.48]	2 (0) [0]	
TOTAL	64 (18) [0.12]	42 (9) [0.15]	45 (9) [0.16]	33 (5) [0.34]	34 (5) [0.32]	23 (3) [0]	

parameters on the effectiveness of VDTTEST.

Combinatorial testing strength. We further examined our data to assess the effects of combinatorial testing strengths. For each DUT, we increased the strength of Phase 1 testing to 4 and that of Phase 2 to 3. With the new testing strengths, one more interaction fault was revealed at Phase 2 for devices of PL110 and DEC21143. Specifically, on PL110, the BGP register was not updated but the golden device did when certain values in three registers were set. On DEC21143, the transmit process did not function correctly when *three* registers were set at the same time. An implication of this discovery is that strength matters more to programs with larger number of registers. Thus, higher testing strength may be recommended to test such programs.

Test sequence length. To investigate whether the length of a test sequence can affect the effectiveness, we increased the length l from 2 to 3. No additional faults were detected. We further examined the data and discovered that all ten faults detected by the test sequences involve less than three stateless test cases. While additional studies may be needed to generalize the results, based on our current discovery, length 2 is sufficient.

False identification of register properties. There are two registers in the SIMICS FSS – one in each of UART8250 and DEC21143 devices, that were incorrectly specified as read-only registers. Because we did not have source code, we were unable to identify the read-only registers statically and during testing we prematurely assumed these registers were read-only in the oracle device (line 8 in the algorithm of Figure 6). For example, when multiple registers share the same I/O port (e.g., UART chip), the read-only register can become writable when certain bits are set in other control registers. Without exhaustively permuting register bits, it is impossible to precisely determine read-only registers.

7. RELATED WORK

There has been work on testing embedded systems using simulators [18, 20, 35]. However, these techniques take advantage of FSSs rather than test FSSs. While specification-based testing techniques [17] can be used to find cases in which the virtual device does not behave consistently with what has been defined in its hardware specifications, obtaining well documented, reusable, and accurate specifications can be difficult. By using the *golden oracle*, our approach does not rely on hardware specifications.

Cong et al. presented a technique to analyze behaviors of virtual device models [12] and detect the differences between virtual and physical devices. In later work, they extend the technique to a commercial software tool [26, 37]. Their ap-

proach leverages symbolic execution to explore states of virtual devices in QEMU. However, their approach is limited as all symbolic execution engines are, and does not generate integration test cases or stateful test cases. As shown in our study, a large portion of faults are interaction faults that can only be detected by combinatorial test cases. In addition, their technique does not generate test specifications. Engineers need to annotate and instrument source code for each virtual device. Ormandy et al. [33] use random testing to detect security vulnerabilities in the implementation of virtual machines. The author also shows several interesting examples on security faults caused by incorrect implementation of virtual machines. However, this work does not have a systematic testing approach for virtual devices such as test drivers, unit and integration testing, and test oracles.

The idea of differential testing [30] has been used in a variety of contexts, including flash file systems [16] and CPU emulators [29]. Martignoni et al. [28, 29] utilize the physical CPU to analyze CPU emulators/virtualizes to find defects in their implementation, however their work focus on the CPU virtualization. In the hypervisor-based VMs they do not take into account the peripheral virtual devices, which are essential to an FSS. Our work instead considers the distinct characteristics of peripheral devices.

In this paper we use ideas and language from combinatorial interaction testing [11, 23, 31] such as the strength of testing and incremental testing [15]. However, we are performing integration testing rather than system testing and only manipulate t parameters at a time.

8. CONCLUSIONS AND FUTURE WORK

In this paper we presented VDTTEST, a framework for testing virtual devices within an FSS. VDTTEST solves two essential aspects of software testing — test case and test oracle generation while also handling unique characteristics of real hardware devices. The approach is mostly automated and requires little knowledge of hardware specifications. Our study shows that VDTTEST is effective in detecting real faults. It found 33% more faults than the best variant of CIT and doubled fault detection over random testing. We also found that using stateful testing as well as testing for interactions improved our results. In future work we will develop oracles that detect timing faults, as well as experiment with different levels of granularity in our oracles. In addition, we will perform more extensive experiments.

9. ACKNOWLEDGMENTS

This work was supported in part by NSF grants CCF-1464032, CCF-1161767 and CNS-1205472.

10. REFERENCES

- [1] ACTS. <http://csrc.nist.gov/groups/SNS/acts/>.
- [2] Buffer Overrun on Invalid State Load. <https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-3689>.
- [3] CVE-2015-3456. <https://access.redhat.com/security/cve/CVE-2015-3456>.
- [4] VGA Driver Bug. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2013-4529.
- [5] F. Bellard. QEMU: A fast and portable dynamic translator. pages 41–41, 2005.
- [6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.
- [7] V. Chipounov, V. Kuznetsov, and G. Candea. S2e: A platform for in-vivo multi-path analysis of software systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.
- [8] S. Chiu and C. A. Papachristou. A design for testability scheme with applications to data path synthesis. In *Proceedings of the Design Automation Conference*, pages 271–277, 1991.
- [9] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the Conference on USENIX Security Symposium*, pages 22–22, 2004.
- [10] Grammar tech static analysis. <http://www.grammatech.com/research/technologies/codesurfer>, 2015.
- [11] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: an approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 1997.
- [12] K. Cong, F. Xie, and L. Lei. Symbolic execution of virtual devices. In *International Conference on Quality Software*, pages 1–10, 2013.
- [13] E. Dumlu, C. Yilmaz, M. B. Cohen, and A. Porter. Feedback driven adaptive combinatorial testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 243–253, 2011.
- [14] J. Engblom, D. Aarno, and B. Werner. *Full-system simulation from embedded to high-performance systems*, pages 25–45. 2010.
- [15] S. Fouché, M. B. Cohen, and A. Porter. Incremental covering array failure characterization in large configuration spaces. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–187, July 2009.
- [16] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the International Conference on Software Engineering*, pages 621–631, 2007.
- [17] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Formal methods and testing. chapter Testing real-time systems using UPPAAL, pages 77–117. 2008.
- [18] M. Higashi, T. Yamamoto, Y. Hayase, T. Ishio, and K. Inoue. An effective method to control interrupt handler for data race detection. In *Proceedings of the Workshop on Automation of Software Test*, pages 79–86, 2010.
- [19] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.
- [20] M. Iqbal, A. Arcuri, and L. Briand. Environment modeling and simulation for automated testing of soft real-time embedded software. *Software & Systems Modeling*, pages 1–42, 2013.
- [21] Android 4.4 emulator does not support orientation changes. <https://code.google.com/p/android/issues/detail?id=61671>, 2013.
- [22] A. Kadav and M. M. Swift. Understanding modern device drivers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 87–98, 2012.
- [23] D. R. Kuhn, D. R. Wallace, and A. M. Gallo, Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30:418–421.
- [24] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with ddt. In *USENIX Conference on USENIX Annual Technical Conference*, pages 12–12, 2010.
- [25] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri. An empirical study of injected versus actual interface errors. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 397–408, 2014.
- [26] L. Lei, F. Xie, and K. Cong. Post-silicon conformance checking with virtual prototypes. In *Proceedings of the Annual Design Automation Conference*, pages 29:1–29:6, 2013.
- [27] The pl031 model doesn’t seem to raise alarm interrupts. <https://bugs.launchpad.net/qemu-linaro/+bug/931940>, 2012.
- [28] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi. Testing system virtual machines. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 171–182, 2010.
- [29] L. Martignoni, R. Paleari, A. Reina, G. F. Roglia, and D. Bruschi. A methodology for testing CPU emulators. *ACM Transactions on Software Engineering and Methodology*, 22:29:1–29:26, 2013.
- [30] W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100–107, 1998.
- [31] C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11, 2011.
- [32] C. Nie, H. Wu, X. Niu, F.-C. Kuo, H. Leung, and C. J. Colbourn. Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures. *Information and Software Technology*, 62:198–213.
- [33] T. Ormandy. An Empirical Study into the Security Exposure to Hosts of Hostile Virtualized Environments. In *Proceeding of the CanSecWest Applied Security Conference*, 2007.

- [34] Open virtual platforms.
http://www.ovpworld.org/technology_ovpsim, 2015.
- [35] J. Regehr. Random testing of interrupt-driven software. In *Proceedings of the ACM International Conference on Embedded Software*, pages 290–298, 2005.
- [36] B. L. Titzer, D. K. Lee, and J. Palsberg. Aurora: Scalable sensor network simulation with precise timing. In *Proceedings of the International Symposium on Information Processing in Sensor Networks*, pages 477–482, 2005.
- [37] Virtual device technologies.
<http://virtualdevicetech.com/index.html>, 2013.