2013 13th International Conference on Quality Software

Symbolic Execution of Virtual Devices

Kai Cong, Fei Xie, and Li Lei
Department of Computer Science
Portland State University
Portland, OR 97207, USA
Email: {congkai, xie, leil}@cs.pdx.edu

***Abstract*—Device drivers are a principal source of failures in computer systems. Therefore, improving driver reliability greatly improves overall system reliability. However, driver development largely has to wait until a first stable version of the device becomes available. This dependency often leaves not enough time for driver validation. Recently, virtual machines and virtual devices have found their way into early driver development and validation. Virtual devices enable driver development even before real devices become available and bring complete observability and traceability that evade real devices. We present an approach to static analysis of virtual devices which is central to achieving observability and traceability. This approach exercises the device model of a virtual device by symbolic execution. Based on the result of symbolic execution, a concrete test case is generated for each path through the device model, which has been exercised. We have applied this approach to virtual devices of five network adapters. The results show that this approach is feasible, efficient, and useful.**

***Keywords*-Static analysis, virtual device, symbolic execution, test generation**

I. Introduction

New computer systems, such as smart phones, tablets, laptops and servers, are entering the marketplace at an ever-accelerating pace. This brings pressures on product development teams to shorten the time-to-market for these products. Since every computer system comes with peripheral hardware devices, a significant portion of development effort is devoted to devices and their drivers.

Device drivers are one of the most critical parts of an operating system (OS) kernel, but they are also one of the least reliable parts. A driver failure causes the OS kernel, the application, or both to crash or behave incorrectly. OS crashes and the so-called "blue screens" are common experiences for virtually every computer user. Most system failures are caused by device drivers, for example, bugs in device drivers caused 85% of Windows XP crashes [1] while Linux drivers have up to seven times the bug rate of other kernel code [2].

Therefore, driver development should start as early as possible to shorten product cycles and improve quality. However, driver development largely has to wait until a first stable version of the real device becomes available. This dependency often leaves not enough time for driver validation. Moreover, even when a real device is available, debugging the device-driver combination is still a very challenging task. A real device is a black box that can be observed solely by its input, output and transfer characteristics without knowledge of its internal workings. There are two major difficulties working with real devices. *Observability*: Real devices can only be observed through their fixed pin interfaces and their internal states are generally not observable. *Traceability*: Once a device error happens, it is difficult to trace what happens before the error due to lack of information about device execution history.

Recently, virtual machines such as VMWare [3], QEMU [4], VirtualBox [5] and Xen [6] have seen increasing use in system development, deployment, and validation. They have also found their way into device and driver development and validation. In all these virtual machines, there are capabilities for introducing virtual devices to emulate real devices needed by the guest OS executing on the virtual machines. The virtual machines usually include many virtual devices covering all popular device categories such as buses, video adapters, and network adapters. For some virtual machines such as QEMU, a new virtual device can be easily introduced following the same standard approach as those already included in the virtual machines. Before the real device becomes available, the developers can use virtual devices to shorten time-to-market in the development of device and driver. An example is how Intel used virtual devices to enable driver development for their 40G network adapter before the device became available [7]. The virtual machine used in this development is QEMU, which includes a virtual device for a previous Intel network adapter, E1000. A virtual device for the E40G was created based on the E1000 virtual device and used to test and validate the E40G driver being developed. Bugs were found in the driver using the E40G virtual device, even before the E40G real device became available.

Virtual devices not only enable early driver development. Moreover, virtual devices are no longer black boxes as real devices and bring complete observability and traceability. *Observability*: Virtual devices are software components. Developers have abilities to observe all variables, not only interface variables but also internal states in virtual devices. *Traceability*: Developers can record the initial states and all inputs of virtual devices and are able to trace what happens based on such information and debug errors. However, there lacks a systematic way for achieving such observability and traceability. Currently, developers usually observe virtual devices when they are used in virtual machines at run-time. Developers can observe paths triggered by run-time device requests. However, it is easy to miss corner cases that are seldom hit at run-time. Moreover, observing and tracing at run-

978-0-7695-5039-8/13 $26.00 © 2013 IEEE
DOI 10.1109/QSIC.2013.44

IEEE computer society

time is a time-consuming process with high overheads: usually many device requests are needed to trigger the behaviors to be observed and traced, and recording device states and requests tends to significantly slow down system operation. For the reasons above, we need a systematic way to statically analyze virtual devices to augment run-time observation and tracing.

This paper presents an approach to static analysis of virtual devices. This approach exercises the device model of a virtual device by symbolic execution to achieve better observability and traceability. Based on the results of symbolic execution, we generate concrete test cases for all paths through the device model that is exercised. Moreover, we can replay test cases to help developers better analyze and understand virtual devices. Our approach needs to address the following challenges.

- *Environment modeling.* A virtual device by itself is not a stand-alone program and can not be executed directly by a symbolic execution engine. There are two issues with the incompleteness of a virtual device. First, the virtual device needs to be properly initialized and its entry functions need to be properly exercised. Second, the virtual device invokes libraries in its environment. Therefore, we need a solution to enclose the device model so that the symbolic execution engine can consume the device model and perform accurate and efficient analysis.
- *Symbolic execution engine adaptation.* We will symbolically execute device models using an existing symbolic execution engine. This engine is not specially designed for running device models while device models have specific characteristics. Hence, we need to adapt the engine so that the engine can execute device models efficiently and provide more useful information.
- *Result presentation.* Symbolic execution can generate symbolic inputs and path constraints. However, such symbolic expressions are hard to understand and utilize by general developers. We need a user-friendly way to present the results of symbolic execution.

This work makes the following three contributions. First, we generate harnesses for virtual devices and execute device models of virtual devices symbolically. A virtual device and its corresponding harness form a complete stand-alone program. We modify an existing symbolic execution engine to adapt to the characteristics of device models. We run this program with this modified symbolic execution engine and explore as many paths of this program as possible.

Second, we generate concrete test cases for all explored paths based on the results of symbolic execution. As developers are often unfamiliar with symbolic expressions, a concrete test case can give developers better understanding and control. To utilize these test cases, our approach can execute a test case using the symbolic execution engine forward and backward, step by step. With this replay functionality, developers can also inspect values of variables at each step.

Third, we have implemented this approach and evaluated it on QEMU virtual devices for five widely-used network adapters. This evaluation demonstrates that the approach is feasible, efficient, and useful: the approach has been successfully applied to QEMU virtual devices, it can execute virtual devices symbolically and generate concrete test cases to help developers better understand behaviors of virtual devices, and the performance overhead is modest.

The remainder of this paper is structured as follows. Section 2 provides the background of this work. Section 3 presents our approach with its design and implementation. Section 4 elaborates on the five case studies we have conducted and discusses the results. Section 5 reviews related work. Section 6 concludes and discusses future work.

II. Background

In this section, we discuss three relevant concepts: virtual machine, virtual device, and symbolic execution. We illustrate what a virtual machine is and why we chose the QEMU virtual machine as our environment for virtual devices. We will introduce an example virtual device in detail so that the reader can understand the basic structure of a virtual device. Symbolic execution executes a program with symbolic values as inputs instead of concrete ones and represents the values of program variables as symbolic expressions. Our approach exercises the device model of a virtual device by symbolic execution and generates concrete test cases based on constraints obtained by symbolic execution.

A. Virtual Machine

A virtual machine is a software implementation of a machine that executes programs like a physical machine. QEMU [4] is a generic and open source machine emulator. It can load and execute a variety of unmodified target OS such as Windows and Linux and all their applications in a virtual machine. It is widely used today for many applications: system development, debugging, profiling, security analysis, etc. QEMU also provides a set of device models such as video devices, audio devices, and network devices. A new virtual device can be easily introduced into QEMU following the same standard approach as those included. We applied our approach to the QEMU virtual devices for five well-known network adapters.

B. Virtual Device

To facilitate understanding of the virtual device concept, we illustrate it with the E1000 virtual device from QEMU. As shown in Figure 1, this virtual device conforms to the PCI (Peripheral Component Interconnect) standard [8] and has the following components:

1) The PCI device state, $E1000State$, which keeps track of the PCI configuration, interface registers and internal variables, such as dev, mac_reg, $rxbuf_size$ respectively;
2) The PCI interface functions which includes a set of I/O interface functions such as "$e1000_mmio_write$" which are invoked by the QEMU virtual machine when the driver issues I/O commands;
3) The device transaction functions such as "$start_xmit$" which are invoked by the I/O interface functions to

```
typedef struct E1000State_st {
    PCIDevice dev; //PCI configuration
    ......
    uint32_t mac_reg[0x8000]; //Interface registers
    ......
    uint32_t rxbuf_size; //Internal variables
    ......
} E1000State;

// Interface register function: write register
static void e1000_mmio_write(void *opaque,
      target_phys_addr_t addr, uint32_t val)
{
    E1000State s = (E1000State *)opaque;
    ......
    if (index == TRANSMIT) {
       mac_reg[index] = val;
       start_xmit(s); //Invoking transaction function
    }
    ......
}

// Transaction function: transmit packets
static void start_xmit(E1000State *s)
{
    ......
    set_irq(s->dev.irq[0], 1); //Fire interrupt
}

// Environment function: receive packets
static ssize_t e1000_receive(VLANClientState *nc, const
      uint8_t *buf, size_t size)
{
    ......
    set_irq(s->dev.irq[0], 1); //Fire interrupt
}
```

Fig. 1. QEMU E1000 virtual device code structure

perform the I/O commands and may fire interrupts by calling "*set_irq*";

4) The environment input functions such as "*e1000_receive*" which are invoked by QEMU to pass environment inputs such as a packet received to the virtual device and may also fire interrupts by calling "*set_irq*".

Both PCI interface functions and environment input functions are device entry functions which are invoked by QEMU to trigger device functionalities.

C. Symbolic Execution and KLEE Engine

Symbolic execution [9] executes a program with symbolic values as inputs instead of concrete ones and represents the values of program variables as symbolic expressions. Consequently, the outputs computed by the program are expressed as a function of input symbolic values. The symbolic state of a program includes the symbolic values of program variables, a path condition, and a program counter. The path condition is a boolen formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy for the symbolic execution to follow the particular associated path. The program counter points to the next statement to be executed. A symbolic execution tree captures the paths explored by the symbolic execution of a program: the nodes represent the symbolic program states and the arcs represent the state transitions.

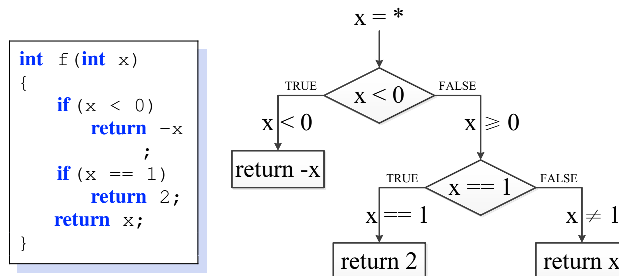

Fig. 2. An example of symbolic execution

We use the program in Figure 2 to illustrate how symbolic execution is conducted. At the entry, x has a symbolic value, i.e., any value allowed by its type (in this case, integer). At each branching point, the path condition is updated with conditions on the inputs to select between the two alternative paths. For this simple example, we can get three paths based on symbolic execution. Each path will have its own path condition, for example, $x < 0$ for the left path.

KLEE [10] is a symbolic execution engine built on the LLVM [11] infrastructure. Given a C program, KLEE executes the program symbolically and generates constraints that exactly describe the set of values possible on a given path through the program. KLEE is one of the latest tools for symbolic execution that has produced tangible results and found bugs in already heavily tested software. It can achieve very high code coverage at a fraction of the costs if it had been done manually and can potentially find serious security bugs.

III. Symbolic Execution of Virtual Devices

A. Overview

Virtual devices are software components in virtual machines. Compared to their hardware counterparts, it is easier to achieve observability and traceability on virtual devices. Instead of executing concretely as part of the virtual machine, the device model of the virtual device can be exercised by a symbolic execution engine. Such execution can be utilized to statically analyze virtual devices. The device state and all inputs to the virtual device are made symbolic. The execution paths through the virtual device are explored symbolically. For each execution path explored, a concrete test case is generated. The device model is then executed with the concrete test case to compute the relevant device state information along this path. Our approach includes four steps that are shown in Figure 3. We now discuss each step in detail.

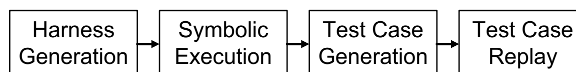

Fig. 3. Workflow for static analysis of virtual devices

B. Harness Generation

For symbolic execution of virtual devices, we adapt KLEE to handle the non-deterministic entry function calls and symbolic inputs to device models. Since the virtual device by itself is not a stand-alone program, in order for KLEE to execute a virtual device, a harness must be provided for the virtual device. A key challenge here is how to create such a harness. This harness has to be faithful so that the symbolic execution of the virtual device will not generate too many paths infeasible in the real device. On the other hand, it has to be simple enough so that KLEE can handle the symbolic execution efficiently. To an extreme, the complete QEMU with the guest OS can serve as the harness which, however, is impractical for KLEE to handle.

We have developed two approaches for harness generation as follows:

- *Manual generation for major device categories.* Since devices fall into device categories depending on their interface types, such as PCI and USB, and on their functionalities, such as network adapters and massive storage devices, we started with creating harnesses for major device categories, for instance, PCI network adapters, and improved such a harness as we experiment on a number of devices in this category. Manual harness generation involves examining how QEMU invokes the virtual device, what QEMU APIs that a virtual device invokes, and what these APIs invoke recursively, and decide what to include. At times, it may be necessary to make a function produce non-deterministic outputs by throwing away its implementation.
- *Automatic generation.* Based on our experience with manual generation, we designed an automatic algorithm for harness generation. This algorithm analyzes the call graph originated from the virtual device, follows the call graph to retrieve the functions invoked by the virtual device, and decides whether to include their implementation. We base this algorithm on two key observations: (1) the size of QEMU is relatively small compared to that of an OS; (2) it is often sufficient to include the implementations of the first level functions invoked. This algorithm also includes a refinement loop which adjusts what to include based on the symbolic execution, for instance, if making a function non-deterministic leads to an extra large number of paths, the function implementation may need to be included.

The test harness should include the following parts:

- *Declarations of the state variable and parameters of entry functions.* A virtual device is not a stand-alone program. If a virtual device is running in a virtual machine, it will register its entry functions with the virtual machine. Moreover, the virtual machine will help the virtual device manage its state variables. Every time an entry function is invoked, the state variables and necessary parameters of the function will be transferred to the function from the virtual machine. In order to exercise a virtual device, we need to handle the state variables and function parameters. Hence, we add declarations of state variables and inputs of entry functions to the harness. An example is shown in Figure 4. This example is part of the harness for the E1000 virtual device. In the first declaration, we define $svd_E1000State$ as the state variable of the E1000 virtual device. The remaining declarations define necessary parameters for the entry functions.

```
//The device state
E1000State svd_E1000State;

//Parameters of entry functions
int svd_deviceEntry;
size_t svd_size_t;
target_phys_addr_t svd_target_phys_addr_t;
uint32_t svd_uint32_t;
uint8_t svd_uint8_t_s65536[65536];
NICState nicState;
```

Fig. 4. State variables and input parameters of entry functions

- *Code for making the state variable and parameters of entry functions symbolic.* In order to cover as many paths as possible in an entry function, we need to make all inputs of the entry function symbolic. The inputs of an entry function should contain the state variable and necessary parameters. We implement a specific function "$svd_make_symbolic$" in the engine to initialize the inputs symbolically. An example is shown in Figure 5. This example is part of the harness for the E1000 virtual device. This example shows that the state variable and parameters of entry functions are made symbolic before they are used.

```
//Make the device state symbolic
svd_make_symbolic(&svd_E1000State, sizeof(
    svd_E1000State), "svd_E1000State");

//Make parameters of entry functions symbolic
svd_make_symbolic(&svd_deviceEntry, sizeof(
    svd_deviceEntry), "svd_deviceEntry");
......
```

Fig. 5. Initializing symbolic variables

- *Non-deterministic calls to virtual device entry functions.* For a real device, there are several ways for the OS and the environment to communicate with it. Similarly, virtual devices provide all kinds of entry functions for communicating with the OS and the environment. To analyze a virtual device, we go through all entry functions with symbolic inputs. We define a variable $svd_deviceEntry$ in the harness and make it symbolic. With this symbolic variable, we make non-deterministic calls to all entry functions. An example is shown in Figure 6. This example is part of the harness of the E1000 virtual device. There are four main entry functions in the E1000 virtual device. Functions "$e1000_mmio_wirte$" and "$e1000_mmio_read$" are invoked by the E1000 driver to write to or read from the device registers.

Function "$e1000_set_link_status$" is invoked by the environment to notify the device whether there is a physical link to the device. Function "$e1000_receive$" is invoked by the environment to inform the device there is a new packet to receive.

```
//Make non-deterministic choice symbolic
svd_make_symbolic(&svd_deviceEntry, sizeof(
    svd_deviceEntry), "svd_deviceEntry");

//Non-deterministic calls to interface functions
switch(svd_deviceEntry) {
case MMIO_WRITE:
   e1000_mmio_write((void *)&svd_E1000State,
        svd_target_phys_addr_t, svd_uint32_t);
   break;
case MMIO_READ:
   e1000_mmio_read((void *)&svd_E1000State,
        svd_target_phys_addr_t);
   break;
case NIC_LINK_STATUS:
   e1000_set_link_status(&nicState.nc);
   break;
case NIC_RECEIVE:
   e1000_receive(&nicState.nc, (const uint8_t *)
        svd_uint8_t_s65536, svd_size_t);
   break;
default: break;
}
```

Fig. 6. Non-deterministic calls to virtual device entry functions

- *Stub functions for virtual machine API functions invoked by virtual devices.* Virtual devices often invoke API functions of virtual machines to achieve certain functionalities. Stubs for these functions need to be provided to complete the harness and are created manually or automatically as discussed above. The stub functions for the E1000 virtual device are not shown due to space limitation.

C. Symbolic Execution

Before we discuss the details of generating test cases using symbolic execution, we first introduce the definition of a test case.

Definition 1: A ***test case*** is denoted as ***tc*** = $\langle$***state, para, choice***$\rangle$, where ***state*** is a concrete device state, ***para*** includes a set of concrete values for parameters of entry functions, and ***choice*** saves the concrete value for non-deterministic choice.

As shown in Figure 6, the device state $svd_E1000State$, the parameters of entry functions and the choice $svd_deviceEntry$ are all made symbolic in the harness. We then execute virtual devices with these symbolic variables. For each explored path, we can get symbolic constraints for these variables. Then a concrete case is generated based on the constraints with the help of STP constraint solver [12]. Therefore, a test case includes three parts: a concrete device state, concrete parameters of entry functions and a concrete choice.

We use KLEE as our symbolic execution engine to exercise virtual devices symbolically. To improve efficiency of symbolic execution and realize the device specific characteristics, we modify KLEE to address four technical challenges for static analysis of virtual devices.

1) Path Explosion Problem: Path explosion is a major limitation for symbolic execution to thoroughly test software programs. The number of paths through a program is roughly exponential in program size. This problem limits the extent to which large software can be thoroughly tested. The problem also exists with executing virtual devices symbolically.

```
int x=10, i=0;

while(i++ < x){
  ......
}
```

```
int x, i=0;
svd_make_symbolic(&x, sizeof(x), "x");
while(i++ < x) {
  ......
}
```

Fig. 7. An example of symbolic loop condition

We can apply two constraints when executing the virtual device to combat the path explosion problem. First, we add a loop bound to each loop whose loop condition is a symbolic expression. As shown in Figure 7, the loop condition "$i++<x$" on the left side is always a concrete value. However, the loop condition "$i++<x$" on the right side is a symbolic expression. We add a loop bound to limit the execution iterations of the loop. With the loop bounds, the user controls the depth of each loop explored. Currently, we add the loop bounds manually in virtual devices. This is practical since there are only a few loops in our analysis of five virtual devices. Second, we can add a time bound to ensure that symbolic execution will terminate in a given amount of time. If the symbolic execution does not completely finish within the given time bound, there may be unfinished paths. For such paths, we still generate test cases with the path constraints obtained so far.

2) Similar Trace Problem: Given a sample program shown in Figure 8, obviously there are two paths in the program and each path constraint is as follows:

```
int f(int x)
{
   if(x == 1 || x == 2)
      return 0;
   return 1;
}
```

Fig. 8. Sample program for traces covering the same statements

$$Path\ 1:\ x == 1\ ||\ x == 2. \tag{1}$$

$$Path\ 2:\ x \neq 1\ \&\&\ x \neq 2. \tag{2}$$

To execute a program with KLEE, we first compile it to LLVM bitcode. However, LLVM compiler will decompose the "if" conditional expression into two seperate branches.

KLEE operates on LLVM bitcode and finds three traces based on the LLVM bitcode. Each path constraint is as follows:

$$Path\ 1:\ x == 1. \quad (3)$$

$$Path\ 2:\ x == 2. \quad (4)$$

$$Path\ 3:\ x \neq 1\ \&\&\ x \neq 2. \quad (5)$$

The first two traces are very similar, covering the same sequence of C code statements. Since such decomposition is very common, there are many such similar traces. Sometimes, it is desired to merge such traces to simplify the trace results. We modified KLEE to implement an algorithm for simplifying trace results, which shown in Algorithm 1.

Algorithm 1 TRACE_RESULT_SIMPLIFICATION (s_{ut}, seq, k)

```
1:  C ← ∅, P ← ∅, R_s ← ∅, R_r ← ∅;
2:  S ← Make_Symbolic_State ( );
3:  V ← Make_Symbolic_Parameters ( );
4:  C ← Symbolic_Execution (S, V); //First round
5:  for each path c ∈ C do
6:      tc ← Generate_Concrete_Test_Case (c);
7:      p ← Concrete_Execution (tc); //Second round
8:      if (p ∈ P) then
9:          R_r ← R_r ∪ {⟨p, tc⟩};
10:     else
11:         P ← P ∪ {p};
12:         R_s ← R_s ∪ {⟨p, tc⟩};
13:         R_r ← R_r ∪ {⟨p, tc⟩};
14:     end if
15: end for
16: return R_s, R_r;
```

We run a device model in two rounds. In Algorithm 1, C is a temporary set for saving all constraints for each path computed by symbolic execution, P is a temporary set for saving all unique path trace information, R_s is a set for saving all simplified analysis results, and R_s is a set for saving all regular analysis results, In the first round, we run the device model with symbolic state S and parameters V using symbolic execution. We save all the path constraints in the set C. Based on the constraints of each path c generated by symbolic execution, we produce a concrete test case tc for each path. In the second round, we run the device model concretely with tc to collect the path trace p. If p covers the same code statements that has already existed in the path trace set P, we save $\langle p, tc\rangle$ in the regular analysis results R_r; otherwise, save p in P and save $\langle p, tc\rangle$ in both R_r and R_s. Hence, there are no two traces covering the same sequence of C code statements in R_s. The users can check the regular and simplified analysis results for different purposes.

Moreover, this algorithm collects complete path traces for unfinished paths. If the analysis does not terminate in the given time bound, there are some unfinished paths. Based on the path constraints obtained for the unfinished paths, we generate test cases for these paths in the first round. In the second round of concrete execution, we run these test cases and generate the complete path trace for each test case.

3) Environment Interaction Problem: A virtual device is a software component and may invoke outside functions to interact with its environment. We divide such interactions into two categories based on whether this function call affects the values of variables in virtual devices. Then we use different mechanisms to handle them respectively.

- If the function call does not affect the values of variables in virtual devices, we ignore it. We modified the symbolic execution engine to realize this. When such a function call happens, the engine ignores it and gives a warning message.
- If the function call may affect the values of variables in virtual devices, we implement this function in our stubs. Since there are not many such function calls for a category of virtual devices, such manual effort is acceptable.

With the above three problems addressed, we can now execute the device model of a virtual device symbolically. After the symbolic execution, we can obtain path constraints for each path. For the small example shown in Figure 2, there are three possible paths and constraints of each path are shown below:

$$Path\ 1:\ x < 0. \quad (6)$$

$$Path\ 2:\ x == 1. \quad (7)$$

$$Path\ 3:\ x \geq 0\ and\ x \neq 1. \quad (8)$$

4) Device-specific Characteristics Adaption Problem: Virtual devices have some specific characteristics which are different from common software programs. First, devices have interface registers which are accessed by drivers to control and operate the devices. It's the generic way for developers to debug the device by checking device register access when there is a bug. So it's important to collect such information to help developers better understand the virtual device execution. Second, a hardware interrupt is an alerting signal which notifies the driver that some work has been done by the device, such as finishing transmitting a network packet. It's critical for developers to know what state and inputs can trigger the interrupt, and check whether such interrupt conforms to the specification.

We modify the symbolic execution engine to capture the above two kinds of device-specific information. First, we modify the engine to collect register access information. For each explored path, the engine summarizes what registers are read or written, saves the values of accessed registers, and shows what registers are condition variables. Second, we modify the engine to hook the interrupt function "set_irq" to collect interrupt information. Then the engine reports what state and inputs trigger or clear the interrupt. The user can check the state and inputs to see whether the interrupt is expected according to the specification.

D. Test Case Generation

The symbolic execution generates path constraints. However, these symbolic expressions are hard to understand and utilize by general developers. We generate concrete test cases for each explored path based on the symbolic path constraints of symbolic execution using STP constraint solver [12]. These concrete test cases can help developers easily understand initial inputs and path constraints of each path.

Based on path constraints shown in Section III-C, we can generate concrete test cases. Three corresponding test cases generated are as follows:

$$Path\ 1:\ x = -1. \tag{9}$$

$$Path\ 2:\ x = 1. \tag{10}$$

$$Path\ 3:\ x = 2. \tag{11}$$

Certainly a test case generated for a path of a virtual device will be much more complex than this example. However, the idea is similar.

E. Test Case Replay

Finding paths using the symbolic execution engine and generating test cases is only half the story. We can also replay the concrete test case for an exercised path so that developers can get better understanding of this path. A generated test case provide the information that can make the engine follow the exact same code path that the engine did while executing the code symbolically. This is achieved by instantiating symbolic variables to concrete values that satisfy the constraints for the path.

Our approach enables a developer to navigate backward and forward, step by step through the execution based on a concrete test case. We employ KLEE to help replay test cases using the algorithm shown in Algorithm 2.

In the algorithm 2, S is a temporary stack for saving all states along the replay process, and s is a temporary variable for saving the current state of the virtual device. The algorithm takes a concrete test case tc as the input, and then extracts the device state s, the choice c and parameters V of entry functions from tc. First, the current state s is pushed into S. Then the program waits for the user input. If the user input is $forward$, we execute one instruction, save the new state as s and push s into S. If the user input is $backward$, we pop out one state from S and save it as s. At any point, a user can inspect the value of any variable including both interface registers and internal variables. This algorithm is sufficiently responsive to support interactive replay. This interactive replay can help developers better understand which path is executed, what variables are changed in the path, and what inputs and initial state are used to trigger the path, and inspect values of variables at any step.

Algorithm 2 TEST_CASE_REPLAY (tc)

```
1:  S ← Initialize_State_Stack ( );
2:  s ← Extract_Concrete_State (tc);
3:  c ← Extract_Concrete_Choice (tc);
4:  V ← Extract_Concrete_Parameters (tc);
5:  S.push (s);
6:  input ← null;
7:  while (true) do
8:    if !(input = Check_user_input ( )) then
9:      Sleep ( ); continue;
10:   else
11:     input ← null;
12:     switch (input) do
13:     case forward :
14:       s ← Execute_instruction (s, c, V);
15:       S.push (s); break;
16:     case backward :
17:       s ← S.pop ( ); break;
18:     case inspectvariables :
19:       Inspect_variables (s); break;
20:     case exit : return;
21:     case default : continue;
22:     end switch
23:   end if
24: end while
```

IV. EXPERIMENTAL RESULTS

In this section, we evaluate our approach from the following three key aspects:

1) *Feasibility.* Can virtual devices from popular virtual machines and for real world devices be analyzed using our approach? How much extra effort is needed to apply our approach to analyze virtual devices?
2) *Efficiency.* How many paths can be explored using our approach in a given amount of time? What is the performance overhead such as memory usage?
3) *Usefulness.* Can our approach help developers achieve better observability and traceability? Can we provide a user-friendly tool for our approach?

Based on analyzing the QEMU virtual devices of five popular network adapters, our results demonstrate that our approach can:

1) Be readily applied to the device models of all five virtual devices. To execute the device models, we only need to create a small harness for each virtual device and implement a common stub for all virtual devices in the network adapter category.
2) Explore numerous paths in a small amount of time, e.g., five minutes, and generate the corresponding concrete test cases. Our approach also has modest performance overhead.
3) Provide a user-friendly interface to assist developers in achieving better understanding of virtual device behaviors.

A. Feasibility

QEMU includes many virtual devices, which provides a broad range of test cases for our approach. We applied our approach to five virtual devices for popular network adapters, which are released with QEMU, as shown in Table I.

TABLE I
FIVE VIRTUAL DEVICES FOR NETWORK ADAPTERS ANALYZED

Device	Vendor	Description
E1000	Intel	Pro/1000 Gigabit Ethernet Adapter
EEPro100	Intel	Pro/100 Ethernet Adapter
PCNet	AMD	PCNet32 10/100 Ethernet Adapter
RTL8139	Realtek	PCI Fast Ethernet Adapter
Tigon3	Broadcom	BCM57xx-based Gigabit Ethernet Adapter

To execute virtual devices symbolically, we manually created a simple harness for each virtual device. We also created a common library of stub functions for all five virtual devices. The stub library has 481 lines of C code. More details about the device models and their harnesses are given in Table II. All device models are non-trivial in size ranging from 2099 lines to 4648 lines of C code. All harnesses are relatively easy to create, having about 100 lines only. Only several hours are needed to create and fine-tune each harness and the stub library.

TABLE II
SUMMARY OF FIVE DEVICE MODELS

Device	Virtual Device		Harness	
	Lines of Code	Number of Functions	Lines of Code	Number of Entry Functions
E1000	2099	53	74	4
EEPro100	2178	70	85	7
RTL8139	3528	110	111	13
PCNet	2139	50	112	13
Tigon3	4648	34	80	4

The experiments were performed on a laptop with an 8-core Intel(R) Core(TM)2 i7 CPU, 8 GB of RAM, 320GB and 7200RPM IDE disk drive and running the Ubuntu Linux OS with 64-bit kernel version 2.6.38.

B. Efficiency

To evaluate the performance overhead of our approach, we execute these virtual devices under five configurations which are shown in Table III. The five configurations are divided into two groups based on different loop bound and time bound. The first group contains configurations 1, 2 and 3, for which we select the same loop bound and different time bounds. The second group contains configurations 1, 4 and 5, for which we select different loop bounds and also different time bounds.

Table III illustrates the different numbers of paths explored in simplified results and memory usages for running the five virtual device under the five configurations. With the given loop bounds and time bounds, our approach is able to explore numerous paths with modest memory usages. With the same

TABLE III
SUMMARY OF EVALUATION RESULTS
(NUMBER OF PATHS EXPLORED / MEMORY USAGE (MB))

Device	Config 1		Config 2		Config 3		Config 4		Config 5	
	loop bound: 1 time: 150 sec		loop bound: 1 time: 300 sec		loop bound: 1 time: 600 sec		loop bound: 2 time: 300 sec		loop bound: 3 time: 600 sec	
	Paths	Mem.	Paths	Mem.	Paths	Mem.	Paths	Mem.	Paths	Mem.
E1000	318	216	545	1006	601	1229	427	626	669	1891
EEPro100	207	41	534	115	1087	328	469	82	590	126
RTL8139	457	76	487	82	503	86	493	82	508	87
PCNet	279	74	424	139	646	262	417	139	601	262
Tigon3	150	172	150	172	150	172	315	817	366	1541

loop bound, more paths can be explored with a larger time bound. With a larger loop bound, we can cover new paths with multiple loop iterations. However, each path may take different amount time to explore. A path containing multiple loop iterations often takes more time to explore. Therefore, in some test runs, for instance, PCNet, it is possible to get few paths with the same time bound but a larger loop bound.

Moreover, the analysis terminates before the time limit in some test cases. For example, it is possible to explore all paths in the Tigon3 virtual device when the loop bound is 1. The actual time needed is 78 seconds to cover 150 paths.

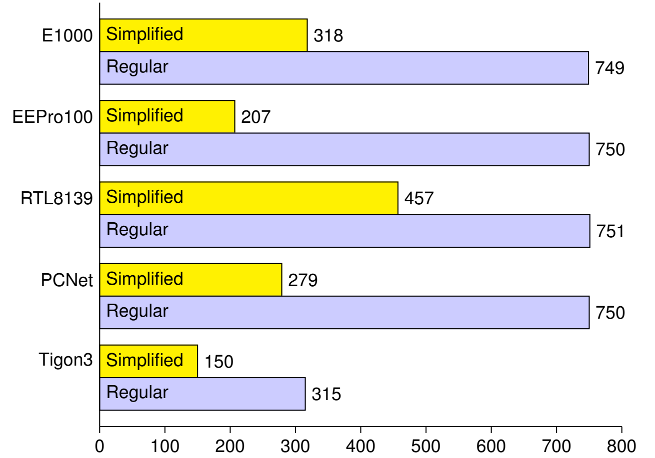

Fig. 9. Number of paths in both simplified and regular results

To evaluate the trace result simplification algorithm, we compare numbers of traces in both the simplified and regular results. The data is collected under Configuration 1 and shown in Figure 9. It can be observed that with the simplification algorithm, we can significantly reduce the number of traces included; therefore, making it easier for developers to exam the traces.

C. Usefulness

Our approach assists developers in analyzing a virtual device and generating test cases. A test case contains the concrete values for the device state and inputs that can be used to replay the corresponding path symbolically explored. Replaying the test case enables developers better observe and

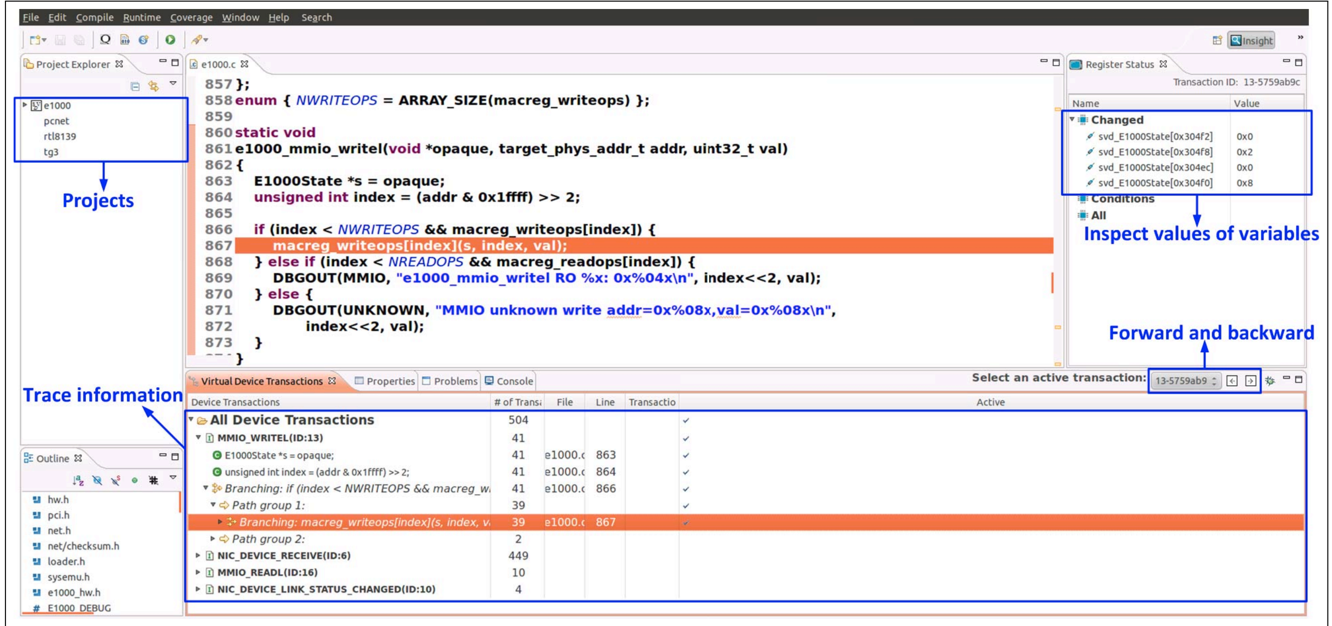

Fig. 10. Test case replay GUI

trace any variable change in the virtual device along this path. Developers can execute a device model based on a concrete test case back and forth, step by step and observe variable changes at each step. Developers can also inspect values of all virtual device variables easily at any time and check whether the state conforms to the specification. With such observability and traceability, developers can understand what paths exist and get a better understanding of the device behaviors.

We have implemented a RCP-based (Eclipse Rich Client Platform) Graphical User Interface (GUI) to assist developers in replaying test cases, which is illustrated in Figure 10. We have demonstrated our tool to industry developers. The feedbacks from these developers are that our tool is useful, can provide in-depth knowledge about virtual devices, and they are willing to trial our tool.

V. Related Work

Recently virtual devices are widely used for software validation. Intel has utilized a network virtual device to enable early driver development [7], and bugs were found in the driver using the virtual device. A wireless network virtual device was created for testing and fuzzing of wireless device drivers [13], and timing difficulties inherent to traditional 802.11 fuzzing-techniques have been solved. Virtual devices bring complete observability and traceability to support software validation, we present an approach to static analysis of virtual devices to help developers better understand virtual devices.

There has been much recent work on using symbolic execution to automatically generate test inputs, leading to software testing tools such as Java PathFinder [14], CUTE and jCUTE [15], CREST [16], BitBlaze [17], DART [18], and SAGE [19]. These tools basically follow the same approach as KLEE in solving a path's constraints to generate a test case and differ in the specifics of symbolic execution and test case generation. Mixed-model execution [15], [18], combining concrete and symbolic execution, has also been used to optimize symbolic execution efficiency. In our approach, we applied symbolic execution to a special type of programs, virtual devices, utilized characteristics of virtual devices to improve symbolic execution effectiveness, generated test cases characterizing paths through virtual devices, and provided facilities for replaying these paths with the generated test cases to assist developers in better tracing and understanding of virtual devices.

VI. Conclusions and Future Work

We have presented an approach to static analysis of virtual devices, which is central to achieving observability and traceability over virtual devices. We have evaluated our approach on five network virtual devices. The case studies and evaluations demonstrate that our approach is feasible, efficient and useful: it has been successfully applied to QEMU virtual devices, it can execute virtual devices symbolically and generate concrete test cases, it can replay concrete test cases to help developers better understand behaviors of virtual devices, and its performance overhead is modest.

Our future research will explore the following three directions. (1) We will investigate how to set loop bounds automatically without instrumenting the source code of virtual devices. (2) We will research how to better utilize the results of static analysis. Based on the analysis results, we will develop test coverage metrics that is suitable for virtual devices and automatic test generation algorithms for testing drivers with virtual devices based on these metrics. (3) We will research

how to utilize symbolic execution of virtual devices at runtime with the virtual machine for step-by-step debugging and assertion checking.

VII. Acknowledgment

This research received financial support from National Science Foundation (Grant #: 0916968). A pending patent filed on this research by Portland State University has been licensed to Virtual Device Technologies (VDTech) where Fei Xie is a partner.

References

[1] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[2] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[3] VMware, "VMware: Benefits of Virtualization," 2013. [Online]. Available: http://www.vmware.com/virtualization/

[4] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 2005.

[5] J. Watson, "VirtualBox: bits and bytes masquerading as machines," *Linux Journal*, 2008.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003.

[7] P. W. Shannon Nelson, "Virtualization: Writing (and testing) device drivers without hardware," 2011. [Online]. Available: http://www.linuxplumbersconf.org/2011/ocw/sessions/243

[8] PCI-SIG, "Conventional pci," 2013. [Online]. Available: http://www.pcisig.com/specifications/conventional/

[9] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, 1976.

[10] C. Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[11] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, 2004.

[12] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th international conference on Computer aided verification*, 2007.

[13] S. Keil and C. Kolbitsch, "Stateful fuzzing of wireless device drivers in an emulated environment," *Black Hat Japan*, 2007.

[14] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with java pathfinder," *SIGSOFT Softw. Eng. Notes*, 2004.

[15] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for c," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005.

[16] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Structural coverage of feasible code," in *Proceedings of the 5th Workshop on Automation of Software Test*, 2010.

[17] D. Bethea, R. A. Cochran, and M. K. Reiter, "Server-side verification of client behavior in online games," *ACM Transactions on Information and System Security*, 2008.

[18] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005.

[19] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2008.