Don’t Panic: Reverse Debugging of Kernel Drivers

Pavel Dovgalyuk, Denis Dmitriev, and Vladimir Makarov
Institute for System Programming, Russian Academy of Sciences
{pavel.dovgaluk, denis.dmitriev, vladimir.makarov}@ispras.ru

ABSTRACT

Debugging of device drivers’ failures is a very tough task because of kernel panics, blue screens of death, hardware volatility, long periods of time required to expose the bug, perturbation of the drivers by the debugger, and non-determinism of multi-threaded environment. This paper shows how reverse debugging reduces the influence of these factors to the process of drivers debugging. We present reverse debugger as a practical tool, which was tested for i386, x86-64, and ARM platforms, for Windows and Linux guest operating systems. We show that our tool incurs very low overhead (about 10%), which allows using it for debugging of the time sensitive applications. The paper also presents the case study which demonstrates reverse debugging of the USB kernel drivers for Linux.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging — Debugging Aids

General Terms

Performance, Reliability

Keywords

Debugging, deterministic replay, reverse debugging, kernel debugging, USB

1. INTRODUCTION

Every hardware device that is designed for connecting to computer needs drivers for operating systems. Drivers of hardware are difficult to debug. Drivers work in the privileged mode, and every invalid operation or non-catched exception leads to kernel panic in Linux or BSoD in Windows. This behavior is fatal to system, which cannot recover and continue its execution.

Dealing with operating system faults is not the only problem in debugging. Stopping the program in the debugger may cause timeout in data processing or data transfer. The behavior of the connected device may change and the bug will disappear. Each program run can expose different behavior of the program without giving a chance to examine the bugs.

Reverse debugging is the solution for the problems with drivers debugging. It focuses on recording program behavior. Reproducing the recorded buggy scenario again and again becomes quite simple.

Key benefit of reverse debugging is examining the prior program states [7], which allows tracing sources of the data values back in time. One can set a breakpoint in the program and then “execute” it backwards to see whether this breakpoint could be hit in the past as it was set before execution.

The paper presents a tool for debugging of user and kernel-level applications that interact with hardware devices. In summary, this paper makes the following contributions:

- Low-overhead method for deterministic replay of hardware communications, which supports all commodity USB devices, serial port, audio and network adapters.
- Implementation of replay debugging based on QEMU and GDB, which works with unmodified state of the art operating systems.
- Practical tool for debugging of the kernels for commodity operating systems executed on state of the art hardware platforms.

2. REVERSE DEBUGGING DESIGN

There are no available implementations of reverse debuggers that support multiple targets and capable of debugging communications with external hardware. Multi-platform support implies that debugger should be made on top of the virtual machine. We chose QEMU [1], an open source simulator, for the implementation of reverse debugging tools, because it is fast enough due to its dynamic translation engine and supports multiple targets including x86, x86-64, ARM, MIPS, and PowerPC.

There were some efforts on making reverse execution in QEMU [3, 4, 5]. But all these efforts were research projects or intermediate results in other areas. Some of them were targeted to a single platform, others were too slow or could not replay operations with peripheral devices. And none of these projects finished with a practical and publicly available tool capable of reverse debugging.

ESEC/FSE’15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803179

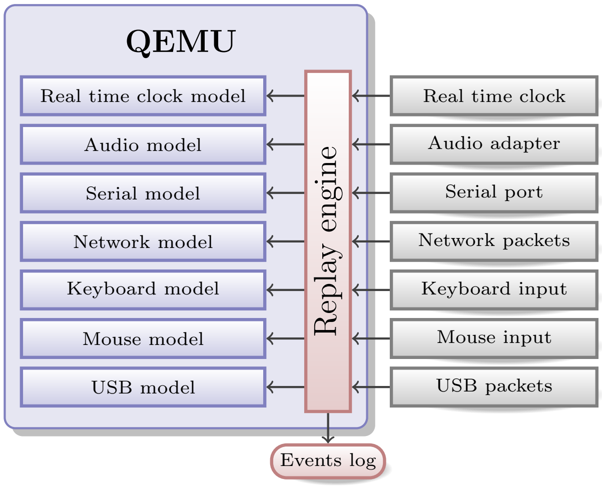

Figure 1: Sources of non-determinism in virtual machine.

2.1 Record and Replay

The first version of our reverse debugging implementation was based on QEMU v0.13, which was the latest at that moment [6]. Now we are using the version 2.3, which provides more accurate simulation and supports USB passthrough.

Our reverse debugging mechanism utilizes recording and replaying virtual machine behavior to travel back in time. All non-deterministic inputs of the virtual machine are recorded and replayed. Figure 1 shows the non-deterministic data from peripheral devices saved into the log. Inputs from simulated hardware, guest memory, software interrupts, and execution of instructions are not saved, because they are deterministic and can be replayed with simulation. Saving only high-level non-deterministic events makes the log file smaller and simulation faster.

Replay engine needs to know when to inject saved real world events when replaying. We specify these moments of time by counting the number of instructions executed between every pair of consecutive events. Record/replay reuses icount feature of QEMU to perform instructions counting. icount allows deterministic execution in absense of external inputs. We extended this mode to allow record and replay of the whole virtual machine execution.

2.2 Reverse Debugging Interface

QEMU supports GDB guest remote debugging protocol. This support includes breakpoints, watchpoints, single-stepping and other common debugging options. We added support of reverse debugging commands (reverse step, reverse continue) into QEMU. GDB just passes desired command through the remote interface. These commands become usable only when replaying the execution. Reverse step proceeds to the previously executed instruction. Reverse continue finds the latest breakpoint hit before the current step [2].

Both of these commands require loading of previously saved system snapshots as shown in figure 2. We save snapshots while recording to allow restoring them later. The first snapshot is created at the start of the simulation. Others are taken every N'th second (where N is the command line option).

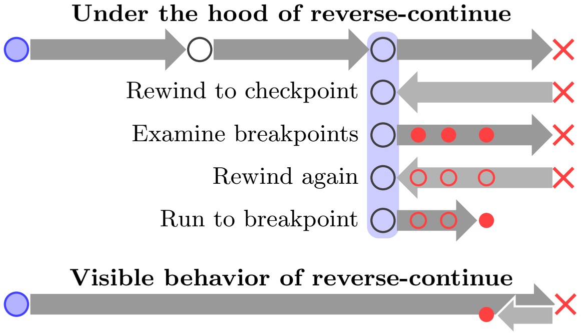
Under the hood of reverse-continue
Rewind to checkpoint
Examine breakpoints
Rewind again
Run to breakpoint
Visible behavior of reverse-continue

Figure 2: Reverse debugging through deterministic replay.

After loading the latest snapshot, simulator runs forward to find a desired point (for reverse step) or to examine all breakpoints that were hit (for reverse continue). If no breakpoints were found, simulator loads earlier snapshot and searches for breakpoints again.

2.3 Hardware Devices Passthrough

QEMU provides passthrough capability for USB devices and COM ports. It also allows connecting audio inputs and network adapters to the "real world". Recording and replaying implementation is similar for all hardware devices. We describe USB passthrough in this section as the closest to the real hardware. QEMU passes USB devices with all their parameters into the guest machine.

USB passthrough in QEMU is implemented with libusb library, which provides generic access to USB devices. Our record/replay supports all data transfer modes of the USB devices: control, interrupt, bulk, and isochronous. We do not open the connection to real USB device while replaying. All device's parameters and operations results are read from the log.

There are several types of non-deterministic inputs in communications with external devices. They may be divided into synchronous and asynchronous types.

Functions calls working with hardware (e.g. libusb calls) are synchronous events, because they are invoked by the simulator's code. Return values of these functions are non-deterministic for the virtual machine. When any of such functions is about to be called in replay mode, replay engine just reads its previously recorded return values from the log.

Incoming USB or network packets and serial interrupts occur asynchronously, because they are initiated by external devices. All these inputs are added to the queue in the record/replay module. This queue is processed and flushed to the log file at the specific phases of the simulator's execution. In replay mode the same code eventually reads the queue items from the log. All packets are injected into the virtual guest devices that processes inputs as they are coming from the real world.

3. PERFORMACE EVALUATION AND CASE STUDY

In this section we will examine buggy kernel driver using reverse debugging. We also present measurements of

```
long usbInfoIoctl(struct file *f,
    unsigned int cmd, unsigned long arg)
{
  int i;
  struct urb *urb;
  char *buf, result[24];
  struct usb_device *device;
  ...
  /* Receive data from USB */
  transmit_bulk_package(&urb, device, &buf, 36,
      usb_rcvbulkpipe(device, 0x82), 0x00000201);
  /* Some result processing */
  memcpy(result, buf, rand());
  for (i = 0; i < 24; i++)
    printk(KERN_INFO "result[%i] = %c\n", i, result[i]);
  ...
  return 0;
}
```

Figure 3: Some buggy code in the USB driver.

```
user@debian:~/kernelModule$ sudo ./test
[938.289683] Kernel panic - not syncing; stack-protector:
Kernel stack is corrupted in: c89e93d0
[938.289741]
[938.293354] Pid: 2768, comm: test Tainted: G 0 3.2.0-4-686-pae
#1 Debian 3.2.65-l+deb7ul
[938.293853] Call Trace:
[938.296274]  [<c12c0c2a>] ? panic+0x4d/0xl41
[938.298932]  [<c1038576>] ? __stack_chk_fail+0xd/0xd
[938.301428]  [<c89e93d0>] ? usbInfoIoctl+0x217/0x21d [usb_info]
[938.301782]  [<c89e93d0>] ? usbInfoIoctl+0x217/0x21d [usb_info]
[938.302003]  [<c10291ff>] ? kmap_atomic_prot+0x2f/0xe0
[938.302583]  [<c10d9857>] ? do_vfs.ioctl+0x459/0x48f
[938.302784]  [<c12c85a7>] ? do_page_fault+0x342/0x35e
[938.302972]  [<c12c8594>] ? do_page_fault+0x32f/0x35e
[938.303173]  [<c10cIf85>] ? kmem_cache_free+0xle/0x4a
[938.303445]  [<c10cd5e7>] ? do_sys_open+0xc3/0xcd
[938.303642]  [<c10d98d1>] ? sys.ioct1+0x44/0x67
[938.303829]  [<c12c9edf>] ? sysenter_do_call+0x!2/0xl2
```

Figure 4: Stack trace shown when kernel panic happens.

the performance overhead for recording and replaying the execution.

At first, we tested record/replay engine with USB flash card, USB camera, USB notifier, and USB encryption key. These devices use different versions of USB protocol and all types of data transfer. We recorded their operations in several scenarios and successfully replayed them. We also tested record and replay for other hardware devices: microphone, COM port, and network adapter.

To check whether reverse debugging is capable to help examining bugs in drivers, we created Linux driver for USB stick. Driver code contains a bug — when request from user is processed the kernel panics and user has to reboot the machine.

Consider the code in figure 3. This code shows the fragment of the faulty ioctl function in the USB driver. This function is meant to receive the user's requests through a memory-mapped file and process them.

Received data is copied to the buffer buf and processed — we copy it to another buffer and output to the log using printk function.

When we load the module and send a request, it triggers kernel panic. Guest system shows the call trace (figure 4), which reports that stack was smashed. Other details about origin of the failure are not available.

```
(gdb) break 378
Breakpoint 1 at 0xc89e93ba: file
    /home/user/kernelModule/usb_info.c, line 378.
(gdb) continue
Continuing.

Breakpoint 1, usbInfoIoctl (f=0xc5309f0c, cmd=3349205032,
    arg=3310012224) at /home/user/kernelModule/usb_info.c:378
375     return 0;
(gdb) info frame
Stack level 0, frame at 0xc5309f34:
 eip = 0xc89e93ba in usbInfoIoctl (/home/user/kernelModule/
    usb_info.c:271); saved eip 0xc10d9857
 called by frame at 0xc5567bf0
 source language c.
 Arglist at 0xc5309eec, args: f=0xc4f09b80, cmd=2147791873,
    arg=3216111224
 Locals at 0xc5309eec, Previous frame's sp is 0xc5309f34
 Saved registers:
  esi at 0xc5309f28, edi at 0xc5309f2c, eip at 0xc5309f30
(gdb) watch  *0xc5309f30
(gdb) reverse-continue
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xc89e939e in usbInfoIoctl (f=0xc5309f0c, cmd=3349205032,
    arg=3310012224) at /home/user/kernelModule/usb_info.c:371
371     memcpy(result, buf, size);
```

Figure 5: Reverse debugging of the kernel module in GDB.

At first, we have to run QEMU in recording mode with USB stick attached. Then we have to figure out the addresses in the memory where our module resides. Address of the specific section name can be obtained with the command

```
cat /sys/module/usb_info/sections/.name
```

The next step is triggering the failure. We created test program, which uses ioctl function to send a message to the loaded kernel module. After starting that program the system immediately goes to panic.

Now we've got an recorded event log, which can be used for replaying kernel panic scenario. This event log should be passed into replay engine to start reverse debugging process. Then we connect to QEMU with GDB through remote debugging protocol and load symbol information for our module. Symbols can be loaded with the command add-symbol-file with the offsets returned by cat command before.

After examining the panic log showed in figure 4 we decided to check the corruption of the return address stored in stack, because we know that stack was smashed. We set the breakpoint to the last line of usbInfoIoctl function and continued the execution. Then we figured out the memory cell where return address resides (figure 5), set a watchpoint at that address and issued reverse-continue command.

Debugger "continued execution" in backward direction and stopped at the following line of usbInfoIoctl function:

```
memcpy(result, buf, rand());
```

Stopping at this function call obviously means buffer overrun with stack overwriting and corruption of return address by the unsafe version of memcpy function. rand function call here does not allow examining of this failure with traditional cyclic debugging.

We showed that our reverse debugging tool can be used for elimination of the tough bugs that are caused by memory corruption. Buggy program can corrupt memory by buffer overflow, invalid pointer operations, attack to printf function, and so on. Recording program execution pushes aside non-deterministic obstacles as in our example.

We also measured the time and space overhead that deterministic execution and events logging incur. There were several testing environments. The first one was executing the programs on Windows XP (for i386) and the second one — on Debian Wheezy (for i386 and ARM). Our tool incurs recording overhead from 3% to 10%. Replaying overhead ranges from 130% to 190% that is quite reasonable for using interactive reverse debugging.

We also measured disk space used for the execution log. It was in range from 10 Kb/sec when OS is idle to 715 Kb/sec when there are many interrupts and disk operations. Log growth rate is small enough to allow using execution recording for long periods of time.

4. RELATED WORK

Reverse debugging was a subject of different research projects. Authors of [4, 11] used VMWare to implement reverse debugging. There was a publicly available VMWare-based reverse debugger, but now support of this debugger is discontinued and it cannot be downloaded anymore.

Another project dedicated to kernel reverse debugging is Time-Travelling Virtual Machine (TTVM) [9]. It works only with modified version of User Mode Linux on x86 platform. Other hardware and software platforms are not supported. Bugs in recompiled drivers will probably work differently due to changed environment. Our solution is better than TTVM, because it supports execution of commodity operating systems and supports replaying communications with real hardware.

Simics is a multi-target simulator from Wind River [8]. Reverse debugging in Simics supports full-system and even multi-system debugging. All commodity operating systems can be executed by the virtual machine. Wind River didn't provide an academic license for evaluation, but Rittinghaus et al. reported that Simics works up to 40 times slower than open source QEMU [10]. Our QEMU-based solution is fast enough to debug time sensitive applications.

5. CONCLUSION AND DISCUSSION

This paper demonstrates how virtual machine with reverse debugging support can be used to debug kernel drivers that communicate with real hardware. We presented record/replay engine for QEMU that allows low-overhead recording and replaying of the system execution. We tested record/replay for i386, x86-64, and ARM.

Reverse debugging allows examining of the whole system behavior with returning back in time. Replay engine supports "real world" network cards, audio adapters, serial port, and USB devices. Reverse debugging may be performed in offline mode when the device, which was used to record the execution, is unavailable.

Our approach has several limitations. We have not tested our tool for other platforms supported by QEMU. And it cannot replay operations with real world devices connected to system bus, because they cannot be passed through to the guest machine.

We submitted reverse debugging patches into the QEMU development mailing list and published it on github[1]. Every deverloper can apply these patches and get his/her own experience in reverse debugging.

[1] https://github.com/Dovgalyuk/qemu/tree/rr-15

6. ACKNOWLEDGMENTS

The work was partially supported by the Russian Ministry of Education and Science within the framework of the base part of state task, and RFBR, research project No. 14-07-00411 a.

7. REFERENCES

[1] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[2] B. Boothe. Efficient algorithms for bidirectional debugging. *SIGPLAN Not.*, 35(5):299–310, May 2000.

[3] S. S. Chia-Wei Hsu. Free: A fine-grain replaying executions by using emulation. The 20th Cryptology and Information Security Conference (CISC 2010), 2010.

[4] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[5] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. Repeatable reverse engineering for the greater good with panda. Oct. 2014.

[6] P. Dovgalyuk. Deterministic replay of system's execution with multi-target qemu simulator for dynamic analysis and reverse debugging. In *Proceedings of the 2012 16th European Conference on Software Maintenance and Reengineering*, CSMR '12, pages 553–556, Washington, DC, USA, 2012. IEEE Computer Society.

[7] J. Engblom. A review of reverse debugging. In *in S4D*, 2012.

[8] J. Engblom, D. Aarno, and B. Werner. Full-system simulation from embedded to high-performance systems. In R. Leupers and O. Temam, editors, *Processor and System-on-Chip Simulation*, pages 25–45. Springer US, 2010.

[9] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[10] M. Rittinghaus, K. Miller, M. Hillenbrand, and F. Bellosa. Simuboost: Scalable parallelization of functional system simulation. In *Proceedings of the 11th International Workshop on Dynamic Analysis (WODA 2013)*, Houston, Texas, Mar. 16 2013.

[11] M. Xu, V. Malyugin, J. Sheldon, G. Venkitachalam, B. Weissman, and V. Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation*, 2007.