Platform-Independent Reverse Debugging of the Virtual Machines

Pavel Dovgalyuk, Denis Dmitriev, Vladimir Makarov
Novgorod State University
Velikiy Novgorod, Russia
{pavel.dovgaluk, denis.dmitriev, vladimir.makarov}@ispras.ru

Abstract—**Prototyping and debugging of operating systems and drivers are very tough tasks because of hardware volatility, kernel panics, blue screens of death, 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 the deterministic replay of the virtual machine execution can be used to reduce the impact of these factors to the process of debugging. We present an approach to reverse debugging which allows creating multi-target whole-system debugger. Using this debugger one can investigate the failures affecting behavior of virtual hardware and guest software. Our debugger is capable of replaying whole virtual machine execution with reproducing internal state of all virtual devices. Although reverse debugging was a subject of many previous researches, there is no widely available practical tool for debugging software on different platforms. We present reverse debugger as a practical tool, which was tested for i386, x86-64, MIPS, and ARM platforms, for Windows and Linux guest operating systems. One can use this debugger to debug user- and kernel-level code, deterministic functional modelling of peripheral devices and hardware platforms. We show that this tool incurs 15–40% recording overhead, which allows using our tool for debugging time-sensitive applications. We presented reverse execution implementation as a set of patches. Some of the patches were already included into mainline QEMU.**

I. Introduction

Prototyping of new operating systems and device drivers includes efforts on simulating, testing, and debugging [16]. Virtualization-aided development and validation techniques improves the experience of developers [1].

OS kernel errors, faulty data from hardware, race conditions highly affect the debugging usability. Every time critical error happens the developer needs to restart the debugging process or even reboot the system. Rebooting obviously leads to spending more time for debugging and causes data loss.

Debugging drivers and OS kernels in the interactive debugger is a tough task. Stopping the program in the debugger may cause timeout in the data processing. After breaking the data transfer the behavior of the debugged program may change and the bug may disappear.

Traditional cyclic debugging is inconvenient for debugging the kernel code. Cyclic debugging requires that every program run behaves and fails in the same way. One can rerun the program being debugged for multiple times, but changed state of the device will lead to changed behavior of the program. It means that developer will spend time for the runs that do not expose the bug and for setting up the external device to its initial state.

Heisenbugs can also cause nuisances for debugging process. Heisenbug can disappear or alter its behavior when one tries to debug it [11]. Heisenbugs can be caused by changing timings, data races, usage of uninitialized memory, and so on.

Reverse debugging is the solution for the problems with drivers and kernel debugging. Reverse debugging allows examining the prior system states including the values of variables and memory cells [9]. Instead of re-executing the programs reverse debugging focus on recording its behavior. Reproducing the bug again and again becomes quite simple when replaying previously recorded behavior.

Key benefit of reverse debugging is the ability to trace sources of the data values back in time. Usually reverse debuggers support "reverse execution". One can set a breakpoint in the program and then "execute" it backwards to see where this breakpoint could be hit in the past as it was set before execution. Another powerful feature of deterministic replay is decoupling dynamic program analysis from execution and fault detection [5].

Reverse execution differs from making the virtual machine snapshots and re-executing the program. Program rerun could observe changed environment and therefore behave differently. Deterministic replay allows reproducing communications with virtual machine environment.

Our aim is aiding in kernel code debugging.One of the possible solutions for that is creating virtual machine-based reverse debugger. Reproducing the behavior of the whole virtual machine allows to replay kernel code, state transitions of all virtual devices, and operations with attached devices.

There are few available reverse debuggers for virtual machines. All of them are targeted to one or two platforms. There is only one true multi-platform full-system reverse debugger, but it is too slow and cannot debug time-sensitive programs [10].

The paper describes an approach which is used to design and implement multi-platform reverse debugger. This approach can be used to debug user- and kernel-level applications.In summary, this paper makes the following contributions:

- System-level execution replay method that does not depend on hardware or software platform. Idea of our method is presented in section II. In section III we show how deterministic replay may be used for implementing platform-independent reverse debugging.
- Implementation of replay debugging based on multi-platform simulator QEMU and GNU debugger. It

works with state-of-the-art operating systems like Windows and Linux, does not require any modifications of the guest OS, and supports commodity hardware platforms. Replay debugging is described in section IV.

- Set of the patches for deterministic replay that were included into upstream QEMU. These patches allow using record-replay simulator capability for other developers [8].
- Evaluation of reverse debugging. We measured performance of record-replay execution for several hardware platforms. Compared to other deterministic replay implementations, our implementation is fast enough for time-sensitive applications recording and supports many hardware platforms, as shown in sections V and VI.

II. Platform-independent reverse debugging

There are no available reverse debuggers that can satisfy our requirements: multi-target, capable of devices communications debugging, and fast enough to debug real time systems. Multi-platform support implies that debugger should be made on top of the virtual machine. We chose QEMU for creating the reverse debugging tools. QEMU is a multi-target simulator supporting commodity hardware platforms [3] like x86, x86-64, ARM, MIPS, and PowerPC. It translates guest binary code into host binary code and then executes it. Due to binary translation and caching of the translated code QEMU works faster than interpreters.

We decided to use deterministic replay for reverse debugging. The idea of deterministic replay is in recording the non-deterministic inputs of the virtual machine during failure exposition scenario. The non-deterministic inputs come from user, network, real time clock, and external devices. Hardware interrupts can also be treated as non-deterministic inputs, because they can occur at random moments.

Executed program code (read from disk) is treated as deterministic, because it does not change between simulator runs. Other inputs of guest machine usually connected to the real world and produce different data values in different executions. Therewore we have to "stabilize" these inputs to make execution repeatable.

When the program executes along the same path (i.e. it is deterministic), user can accumulate debugging experience for the same execution running it over and over again.

Our approach is based on maintaining the consistent state of CPU, memory, and all virtual devices. It means that we must capture all external inputs to these parts of simulator. All inputs are recorded (and replayed) into non-deterministic events log, as shown in Fig. 1.

The novelty of our approach is in reusing abstract hardware layer to capture non-deterministic data. We also use platform-independent instruction counter, as opposed to other approaches that use platform-specific registers for that purpose [6], [14]. These properties allow creating platform-independent implementation of the reverse debugging tool. Reverse debugging may be used with any newly added platform without its modification.

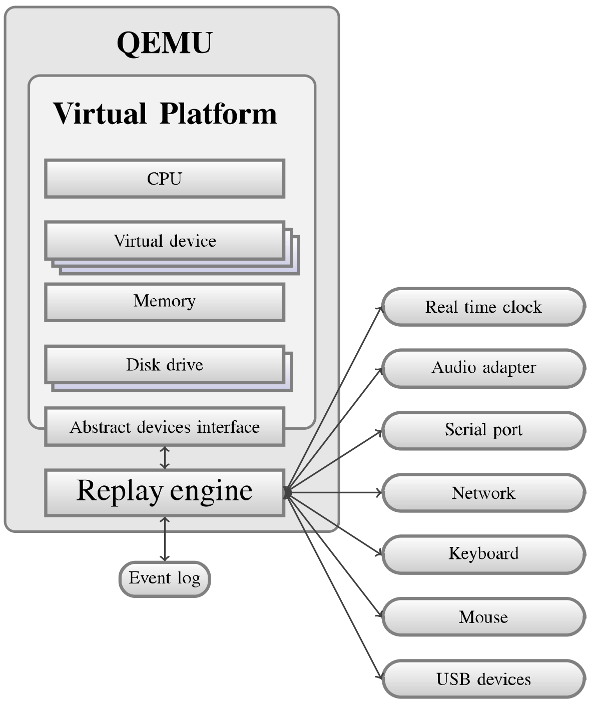

Fig. 1. Multi-platform replay in QEMU

Our approach allows implementing platform-independent record/replay and reverse debugging. All replay-specific functions work with platform-independent data and events. This design allows replaying the state of the whole virtual machine, including its devices. One can record an execution of a virtual machine and reproduce its internal devices state or external models (e.g., SystemC models [16]) for multiple times for the sake of debugging.

III. Deterministic replay

Our record/replay system is based on saving and replaying non-deterministic events (e.g., keyboard input) and simulating deterministic ones (e.g., reading from HDD or guest memory) [2], [7]. Saving only high-level non-deterministic events makes log file smaller, simulation faster, and allows using reverse debugging even for time-sensitive applications.

The following non-deterministic data from peripheral devices is saved into the log: mouse and keyboard input, network packets, audio controller input, USB packets, serial port input, and hardware clocks. Clocks are non-deterministic too because their values are taken from the host machine. Inputs from simulated hardware, guest memory, software interrupts, and execution of instructions are not saved into the log because they are deterministic and can be replayed by simulating the behavior of virtual machine starting from the initial state.

We had to solve three tasks to implement deterministic replay: recording non-deterministic events, replaying non-deterministic events, and checking that there is no divergence between record and replay modes.

We changed several parts of QEMU to implement event log recording and replaying. Devices' models that have non-deterministic inputs from the real world were changed to write every external event into the execution log immediately, e.g., network packets are written into the log when they arrive into the virtual network adapter.

Replay engine needs to know when to inject real world events while replaying. We specify these moments of time by counting the number of instructions executed between every pair of consecutive events.

To make sure that replay process is deterministic, our implementation checks that expected events (e.g., reads of the timer, software interrupts) of the QEMU execution are coming from the log in the correct order. If one of these events is not expected in the current state of the simulator, replaying will stop immediately.

A. *Instruction counting*

Record replay reuses icount feature of QEMU to perform instruction counting. icount was designed to allow deterministic execution in absense of external inputs of the virtual machine. We extended this mode to allow record and replay of the whole virtual machine execution. Recording enables deterministic execution even when external data inputs affect the virtual machine behavior.

QEMU uses dynamic translation to execute the guest code. Every guest instruction is transformed into a sequence of host instructions that simulate behavior of the guest one. After translation QEMU joins instructions into translation blocks. Translation block is a continuous sequence of instructions. It means that execution of the block always starts at the first instruction and ends at the last one.

The icount allows to perform accurate counting of executed instructions. But the counter of the instructions is not incremented after every instruction. The counter is updated only at the beginning of the translation blocks, because translation block always executed from the beginning to the end. Hardware exceptions (such as MMU fault or division by zero) break this rule and there is special mechanism to correct icount in case of exception.

We use icount to control the occurrence of the non-deterministic events. Number of instructions executed after the last event is written to the log. In replay mode we can use instruction counter to predict when to inject that event.

Using icount is a very convenient way to control the non-deterministic events, but it has a performance tradeoff. Instructions counting code is added to every translation block, as shown in Fig. 2. "+" mark in the figure indicates the operations that perform instructions counting. We present evaluation of the performance overhead in section V.

Translation block to be executed

```
0x000fd1fe:  mov    %eax,%gs
0x000fd200:  mov    %ecx,%eax
0x000fd202:  jmp    *%edx
```

Intermediate representation of the translation block

```
  ld_i32 tmp12,env,$0xfffffff4
  movi_i32 tmp13,$0x0
  brcond_i32 tmp12,tmp13,ne,$0x0
+ ld_i32 loc14,env,$0xffffffe8
+ movi_i32 tmp12,$0x3
+ sub_i32 loc14,loc14,tmp12
+ movi_i32 tmp12,$0x0
+ brcond_i32 loc14,tmp12,lt,$0x1
+ st16_i32 loc14,env,$0xffffffe8
  ---- 0xfd1fe
  mov_i32 tmp0,eax
  movi_i32 tmp3,$0xfd1fe
  st_i32 tmp3,env,$0x20
  mov_i32 tmp6,tmp0
  movi_i32 tmp12,$0x5
  call load_seg,$0x0,$0,env,tmp12,tmp6
  ---- 0xfd200
  mov_i32 tmp0,ecx
  mov_i32 eax,tmp0
  ---- 0xfd202
  mov_i32 tmp0,edx
  st_i32 tmp0,env,$0x20
  ----
  exit_tb $0x0
  set_label $0x0
  exit_tb $0x5c8033b
+ set_label $0x1
+ exit_tb $0x5c0045a
```

Fig. 2. Translation block with icount code

B. *Real time clock*

QEMU uses host real time clock (RTC) for passing it to virtual RTC into the guest machine and for internal timers. Timers are used to execute callbacks from different subsystems of QEMU at the specified moments of time. There are several kinds of timers:

- Real time clock. Based on host time and used only for callbacks that do not change the virtual machine state. Therefore real time clock and timers does not affect deterministic replay at all;
- Virtual clock. These timers run only during the emulation. In icount mode virtual clock value is calculated using executed instructions counter. As a result, it is completely deterministic and does not have to be recorded;
- Host clock. This clock is used by device models that simulate real time sources (e.g., real time clock chip). Host clock is the one of the sources of non-determinism. Host clock read operations should be logged to make the execution deterministic;
- Real time clock for icount. This clock is similar to real time clock but it is used only for increasing virtual clock while virtual machine is sleeping. Due to its nature it is non-deterministic and has to be logged.

Real time clock used for guest RTC and timers have to be recorded because they affect the execution of virtual machine. We created wrappers for clock reading functions. These wrappers write the values to the log when recording and read them back when replaying the execution.

C. *Simulator execution checkpoints*

Replaying of the virtual machine execution is bound by sources of non-determinism. These are inputs from clock and peripheral devices, and QEMU thread scheduling. Thread scheduling affect processing events from timers, asynchronous input-output, and bottom halves.

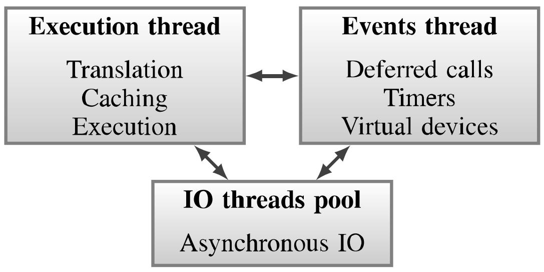

Fig. 3. Logic blocks of QEMU

The simulator consists of several threads: execution thread, events thread, translation module, execution module, and models of peripheral devices, as shown in Fig. 3.

Execution thread runs in loop and invokes execution module. Execution module calls translation functions if simulated code was not translated before, or it takes translation result from the cache. Then the translated code has to be executed. Sometimes events thread breaks the execution and invokes drivers of emulated (HDD) or external (keyboard) devices. Execution loop continues after processing events from these devices.

Such an asynchnonous architecture brings in difficulties in reverse debugging implementation. Non-deterministic inputs from peripheral devices are supplemented with non-deterministic inter-thread communication.

Invocations of timers are coupled with clock reads and changing the state of the virtual machine. As we want to keep the virtual devices in the consistent stace, we should preserve order of timers invocations. Their relative order in replay mode must replicate the order of callbacks in record mode. To preserve this order we use simulator checkpoints. When specific clock is processed in record mode, we save to the log "checkpoint" event. Checkpoints here do not refer to virtual machine state snapshots. These are just record/replay events used for synchronization.

QEMU in replay mode will try to invoke timers processing as they appear in the queue and timeout expires. But we do not process the timers until the checkpoint event will be read from the log. This event synchronizes CPU execution and timer events.

Another checkpoints application in record/replay is instructions counting while the virtual machine is idle. `qemu_clock_warp` is the function responsible for this. It changes virtual machine state and must be deterministic then. That is why we added checkpoint to this function to prevent its operation in replay mode when it does not correspond to record mode.

D. Asynchronous input and output

Disk I/O events are completely deterministic in our model, because in both record and replay modes we start virtual machine from the same disk state. But callbacks that virtual disk controller uses for reading and writing the disk image may occur at different moments of time in record and replay modes.

Read and write requests are created by execution thread of QEMU. These requests proceed to the block layer, which is responsible for disk images. Block layer creates tasks for IO thread that will be processed asynchronously.

We implemented saving and replaying IO tasks synchronously to the CPU execution. When the callback of the task is about to execute, it is added to the queue in the replay module. This queue is written to the log and its callbacks are executed. In replay mode callbacks do not processed until the corresponding event is read from the events log file.

E. Initial setup

Before replaying the execution of the system it should be set into the initial state. Then execution begins from this starting point. System executes forward deterministically because starting point was the same and the executed code did not changed. Previously recorded inputs are injected into the system while replaying the execution.

F. Abstract hardware layer for external devices

QEMU is capable of passing through hardware USB devices into the virtual machine. This is a great possibility for prototyping software for newly created and commodity devices. As we added record/replay mechanism into the abstract QEMU level, it supports replaying of any USB devices that can be connected to simulator.

Abstract layer also includes functions for replaying serial port and network communications. Replaying these communications does not depend on connected endpoints. E.g., it uses the same functions for TCP and file-mapped serial communications, for Slirp- and TAP-based implementations of network connections.

IV. Reverse debugging with GDB

Reverse debugging is the ability of going through the process of execution in backward direction. With reverse debugging programmer can stop at a chosen point of execution (e.g., where invalid pointer dereferencing exception occurs), examine registers and memory, and continue execution of the program in backward direction (e.g. to the last position, where data pointer is written).

QEMU supports debugging with gdb through remote interface. This support includes breakpoints, watchpoints, single-stepping and other common debugging commands. However, QEMU did not have support for reverse debugging through GDB remote protocol. We added support of reverse debugging commands (reverse step, reverse continue) into QEMU. 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.

Both of these commands require loading of previously saved system snapshots. Therefore we had to improve QEMU for checkpointing — saving virtual machine state to allow restoring it later. The first checkpoint is created at start of the simulation. Other checkpoints are taken every N'th second (where N is the command line option). These checkpoints are used for going back through the execution process.

TABLE I. PERFORMANCE OF REVERSE EXECUTION

Execution mode	Execution time, sec	Slowdown against regular	Slowdown against icount
Loading Windows (x86)			
regular	75	1.00	—
icount	89	1.19	1.00
icount+record	98	1.31	1.10
icount+replay	228	3.04	2.56
Loading Debian Wheezy (x86)			
regular	130	1.00	—
icount	177	1.36	1.00
icount+record	183	1.41	1.03
icount+replay	418	3.22	2.36
Loading Debian Wheezy (ARM)			
regular	124	1.00	—
icount	157	1.27	1.00
icount+record	164	1.32	1.04
icount+replay	457	3.69	2.91
Loading Debian Wheezy (MIPS)			
regular	160	1.00	—
icount	179	1.12	1.00
icount+record	182	1.14	1.02
icount+replay	428	2.68	2.39

After loading one of the previous states, simulator runs forward to find a desired point (for reverse step) or to examine all breakpoints that were hit (for reverse continue). Reverse continue then makes an additional pass. After reaching the point where user stopped the execution, reverse continue re-runs execution again to seek the latest of the hit breakpoints.

To debug in reverse direction user has just to issue the commands to the debugger. We altered QEMU to perform all the required actions. QEMU automatically loads previously saved state and makes two passes for reverse continue.

V. EVALUATION

In this section we present measurements of performance overhead for the record and replay compared to the normal simulator execution.

We used several tests for evaluation. The first one was loading Windows XP 32 bit on x86 and others were loading Debian Wheezy on x86, ARM, and MIPS platforms. We set up five virtual machines for these tests. All of them used 128 Mb of virtual memory.

We executed each of the tests in four modes:

- **regular** QEMU executes the test using regular dynamic translation engine without counting the instructions. This mode is non-deterministic and used for reference as a normal mode.
- **icount** Instruction counting is enabled with `-icount shift=7` command line option. This mode is required for deterministic execution of the instructions and virtual clock. External inputs remain non-deterministic. Value of `shift` option affects the virtual clock rate and does not affect the OS booting time.
- **icount+record** Deterministic execution mode. All virtual devices inputs connected to the real world are recorded in the log.
- **icount+replay** System execution is replayed deterministically. All input events are read from the log.

TABLE II. LOG FILE GROWTH RATE FOR DIFFERENT PLATFORMS

Platform	Log size, bytes	Executed instructions	Bytes per 1000 instructions
x86	139M	6346M	21.9
ARM	80M	4469M	17.9
MIPS	257M	3408M	75.4

We measured time for execution of each test in every mode. These measurements are presented in Table I. There are two normalized time values for every execution. The first one is normalized to regular execution. This shows net overhead for the user who runs virtual system in regular emulation mode. The second normalized value shows overhead of non-deterministic events log capturing and replaying.

Enabling icount makes system execution deterministic. It means that time on virtual clock depends on number of executed instructions. So the simulation speed does not affect the behavior of the system. The only way for the guest system to notice the slowdown of the execution is to measure the timings of the communications with the real world (network, USB, and so on) or to read some external source of real time (e.g., NTP server). If the overhead is too large, it can also affect the usability of the execution recording. Fortunately, overhead of the instruction counting is quite reasonable — from 12% to 36%. Such an overhead does not affect the user experience.

Recording overhead should be as small as possible to avoid its interference in the system behavior. To estimate this overhead, we normalized recording time to the execution time with enabled icount. These values are shown in the fourth column of the Table I. Recording overhead ranges from 3% to 10% for our tests.

Replay overhead does not affect the guest system behavior. It affects only user experience. In our tests the overhead is in reasonable range from 204% to 269%.

We also measured space overhead for recording the execution log. It was in range from 10 Kb/sec when OS is idle to 715 Kb/sec in the active phase of OS booting.

Log growth rate is small enough to allow using execution recording for long periods of time. Number of bytes per executed instruction for Debian loading tests are presented in Table II.

VI. RELATED WORK

There were previous efforts on making whole-system reverse execution. XenLR project used Xen to implement a prototype of deterministic replay system [14]. It supports replaying keyboard and timer events with MiniOS inside. It is not capable of replaying other events or working with other operating systems.

Authors of [5], [18] used VMWare to implement reverse debugging. This debugger supported reverse execution for systems on x86 platform. There was a publicly available VMWare-based reverse debugger, but now support of this debugger is discontinued and it cannot be downloaded anymore.

Time-Travelling Virtual Machine (TTVM) is intended to be used for reverse debugging of the kernels [13]. 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. In contrast to TTVM, our solution supports execution of commodity operating systems and replaying communications with real hardware.

FREE project is targeted to x86 record-replay [4]. It is built upon QEMU and records IO reads into the log. This log is used for replaying virtual CPU and memory state. FREE cannot recover virtual devices' states and does not replay DMA transactions.

PANDA is the open-source system that supports record and replay for several hardware architectures [6]. PANDA maintains the state of CPU and RAM during replay. It writes inputs from I/O instructions, interrupt controller, and DMA controller into a non-determinism log.

Panda uses platform-dependent way to determine when to replay the inputs. For x86 these moments are identified by the program counter, instructions count, and value of the implicit loop variable (which is `ECX` register).

Panda cannot "go live" from replay, because it replays only CPU and RAM. User cannot examine whole virtual machine state. In contrast to PANDA, our implementation can restart regular execution from any moment of replay scenario, because all virtual devices of VM are always in a consistent state. Recording process for x86 PANDA incurs 85% overhead and replaying — 257%.

Simics is the multi-target simulator from Wind River [10]. Reverse debugging in Simics was released in 2005. It supports full-system and even multi-system debugging. Simics simulates wide range of platforms including x86-64, ARM, MIPS, and PowerPC. All commodity operating systems can be executed by the virtual machine. All these features make Simics one of the best commercial reverse debuggers. Another side of this features set is that Simics is a very expensive tool that makes it unavailable for most of the developers.

However, Simics is too slow for convenient debugging of programs that interact with user or real hardware. Rittinghaus et al. reported that Simics was up to 40 times slower than open source QEMU [17].

VII. Conclusion

We presented platform-independent record-replay technique that allows low-overhead recording and replaying of the system execution. We implemented our method in multi-platform simulator QEMU. Core patches of deterministic replay were already included into upstream QEMU [15]. Reusing existing instructions counting allows changing the simulator in target-independent manner. These changes enable deterministic replay for all target platforms supported by QEMU. We tested record/replay for i386, x86-64, MIPS, and ARM.

Replay engine supports multiple types of input devices including network cards, audio adapters, serial port, and USB devices. Deterministic replay was tested on several types of commodity USB devices that work in low-speed and high-speed modes.

Reverse debugging in QEMU allows examining of the whole system behavior with returning back in time. Our reverse execution engine supports reverse stepping and reverse continuing to the last breakpoint. Reverse debugging may even be performed in offline mode when the device, which was used to record the execution, is unavailable.

Our record/replay engine may be used for debugging drivers, kernel, and user-level application without wasting time for reproducing the bugs and system setup.

VIII. Future work

We are currently working on upstreaming the reverse debugging patches to QEMU. Making all the reverse debugging code available to developers using QEMU will help them in their debugging work. We also will test the record/replay mechanism for other QEMU platforms.

Deterministic replay may be used for other dynamic analysis methods in addition to reverse debugging. Using deterministic replay for them is a great possibility to make analysis offline. Replay eliminates analysis disturbance of a system behavior. Henderson et al. reported that taint analysis with whole-system instrumentation implemented in QEMU incurs 600% overhead [12]. Deferring this overhead from system execution phase to replaying will make analysis more trustworthy. Replaying also can be used for speeding up the analysis through parallelizing it to multiple machines [17].

IX. Acknowledgments

The work was partially supported by RFBR, research project No. 14-07-00411 a.

References

[1] M. Baklashov. An on-line memory state validation using shadow memory cloning. In *Proceedings of the 2011 IEEE 17th International On-Line Testing Symposium*, IOLTS '11, pages 186–189, Washington, DC, USA, 2011. IEEE Computer Society.

[2] K. A. Batuzov, P. M. Dovgalyuk, V. K. Koshelev, and V. A. Padaryan. Dva sposoba organizatsii mekhanizma polnosistemnogo determinirovannogo vosproizvedeniya v simulyatore qemu [two approaches to organizing a full-system deterministic replay mechanism in qemu simulator]. *Trudy ISP RAN [The Proceedings of ISP RAS]*, 22:77–94, 2012.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] P. Dovgalyuk. Deterministic replay core. https://lists.nongnu.org/archive/html/qemu-devel/2015-09/msg04623.html, 2015.

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

[10] 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.

[11] J. Gray. Why do computers stop and what can be done about it?, 1985.

[12] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 248–258, New York, NY, USA, 2014. ACM.

[13] 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.

[14] H. Liu, H. Jin, X. Liao, and Z. Pan. Xenlr: Xen-based logging for deterministic replay. In *Proceedings of the 2008 Japan-China Joint Workshop on Frontier of Computer Science and Technology*, FCST '08, pages 149–154, Washington, DC, USA, 2008. IEEE Computer Society.

[15] P. Maydell. Qemu 2.5 changelog. http://wiki.qemu.org/ChangeLog/2.5, 2015.

[16] M. Monton, A. Portero, M. Moreno, B. Martinez, and J. Carrabina. Mixed sw/systemc soc emulation framework. In *Industrial Electronics, 2007. ISIE 2007. IEEE International Symposium on*, pages 2338–2341, June 2007.

[17] 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.

[18] 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.