QEMU-Based Framework for Non-intrusive Virtual Machine Instrumentation and Introspection

Pavel Dovgalyuk
Novgorod State University
Velikiy Novgorod, Russia
pavel.dovgaluk@ispras.ru

Natalia Fursova
Novgorod State University
Velikiy Novgorod, Russia
natalia.fursova@ispras.ru

Ivan Vasiliev
Novgorod State University
Velikiy Novgorod, Russia
ivan.vasiliev@ispras.ru

Vladimir Makarov
Novgorod State University
Velikiy Novgorod, Russia
vladimir.makarov@ispras.ru

ABSTRACT

This paper presents the framework based on the emulator QEMU. Our framework provides set of multi-platform analysis tools for the virtual machines and mechanism for creating instrumentation and analysis tools. Our framework is based on a lightweight approach to dynamic analysis of binary code executed in virtual machines. This approach is non-intrusive and provides system-wide analysis capabilities. It does not require loading any guest agents and source code of the OS. Therefore it may be applied to ROM-based guest systems and enables using of record/replay of the system execution. We use application binary interface (ABI) of the platform to be analyzed for creating introspection tools. These tools recover the part of kernel-level information related to the system calls executed on the guest machine.

CCS CONCEPTS

• **Hardware → Simulation and emulation**; • **Software and its engineering → Software testing and debugging**; **Software reverse engineering**; *Dynamic compilers*;

KEYWORDS

Software instrumentation; Dynamic analysis; Virtual machine; Introspection; ABI; QEMU

ACM Reference Format:
Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. 2017. QEMU-Based Framework for Non-intrusive Virtual Machine Instrumentation and Introspection. In *Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 4–8, 2017 (ESEC/FSE'17),* 5 pages.
https://doi.org/10.1145/3106237.3122817

1 INTRODUCTION

Dynamic software analysis of the binary code is used for profiling, malware analysis, intrusion detection, protocol reverse engineering, software testing, and many other activities [4, 7, 9, 12]. Semantic gap between guest binary code (which can be obtained) and source data structures (needed for debugging) hampers the analysis. Virtual machine introspection (VMI) is the most used approach to bridge the semantic gap between binary and source representations [12].

State-of-the-art approaches to VMI use source code or SDK of the kernels to make the tools that recover information about data structures directly from the guest memory [4, 9, 10, 15].

Source code or in-guest build tools are used to create the agents embedded into the guest system. There are basically two kind of agents: setup and runtime. Setup agents are only used to create introspection profile which includes guest addresses and offsets of data that should be extracted from the OS core. Runtime agents execute within the guest system all the time and transfer the data to the host. WinDbg in Windows and gdbserver in Android are the examples of the runtime agents. They operate while system is running and collect the information for the remote debugger.

Sometimes guest agents cannot be embedded into the system. When ROM-based OS is analyzed, there is no way to embed runtime or setup user agent. Source code of such ROM kernel may be available, but without build options it may be impossible to recover the internal offsets and addresses.

Runtime guest agent also cannot be embedded into the system when its execution is recorded and replayed for the analysis [6]. Therefore the main limitations for the user agents are read only OS image and read only (replayed) execution scenario.

We present a non-intrusive introspection approach which exploits the application binary interface (ABI) of the guest platforms. ABI is much more stable and smaller than kernel source code for manual analysis and creation of the introspection tools. Our approach is completely non-intrusive, because it doesn't need any guest agents. Hooking ABI-related events like system calls is used to recover part of the kernel data. We also track CPU-level events like TLB miss or interrupt requests to recover the memory structure of the processes and parse executables to monitor API function calls. Using ABI reduces the maintainance efforts required for supporting the analysis algorithms and applying them to other versions of guest operating systems.

ESEC/FSE'17, September 4–8, 2017, Paderborn, Germany
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5105-8/17/09…$15.00
https://doi.org/10.1145/3106237.3122817

Our work is based on the open source simulator QEMU [2]. QEMU is capable of running virtual machines based on the commodity platforms (e.g., i386, ARM, MIPS, PowerPC). We added new plugin subsystem and subsystem for dynamic instrumentation into QEMU, that allow easy migration to newer simulator versions. We also created plugins for monitoring system calls, file operations, API functions, and process operations.

2 NON-INTRUSIVE VIRTUAL MACHINE INTROSPECTION

The aim of our system is debugging, analysis, and reverse engineering of the applications, firmwares, and kernels. In most cases we cannot upload guest modules into machines based on that kernels to figure out field offsets in kernel data structures (as Panda does [4]), because embedded firmwares do not include compiler and headers. When the execution is replayed, guest modules can't be installed too, because data exchange with these modules will also be recorded and won't alter to get required information [1]. However, ABI usually remains unchanged (when firmware is a tuned Linux or ABI is a part of the hardware platform) and we may use it for creating introspection tools.

ABI includes the list of system calls, calling convention, data alignment rules, execution files format, stack frame format, and registers usage pattern. ABIs are designed for the hardware or software platforms and remains mostly unchanged with platforms evolution for the sake of backward compatibility. Layout of the kernel structures may be unavailable or volatile, in contrast to system call identifiers and parameters.

System call functions in Linux are identified by an integer passed as a parameter in one of the registers. These identifiers never change. Therefore maintainance efforts for Linux introspection modules in our approach will include only adding new system calls and supporting arguments passing agreement for the new platforms added to the analysis scope. We created a subsystem for QEMU, which allows hooking system calls to capture OS-specific information from the virtual machine instead of monitoring guest memory to find kernel data structures.

To match call and return instructions for the system calls we track execution context. It consists of current process id and stack pointer. We identify processes by page directory address (e.g., CR3 for x86 or CP15.c2 for ARM) [9].

Simulator QEMU supports many commodity hardware platforms like x86, x86-64, ARM, MIPS, and PowerPC [2]. With multi-platform support one can make analysis tools that can be executed on many platforms. QEMU translates guest binary code into host binary code and then executes it.

To invoke our own code on system call execution we embed callbacks into the translated code. These callbacks recover arguments of the system calls and their return values. Parameters and return values are recovered on system call entry and exit instructions respectively.

We added instrumentation layer into QEMU and implemented VMI as a set of plugins. Existing and planned plugins in our framework are shown in Figure 1. Upper plugins in the figure use the information recovered by the lower plugins. It includes data structures and event notifications.

File monitoring is perfomed by a plugin, which is independent from executing guest OS and guest hardware platform. It operates with system call and file abstractions. File monitor maintains the list of the open files to detect close system calls that operate with files, because close operations may be used to free other system resources (e.g.,NtClose is used to close all handles in Windows).

Commodity operating systems use memory-mapped files to load executables and dynamic libraries. We hook mapping/unmapping operations in the file monitoring plugin. This information is used to detect executable image loading and parse those images to extract API functions addresses.

Monitoring of API function calls may be useful in itself (e.g., for detecting anomalies), and also for recovering more system information, than from the system calls (e.g., hooking CreateProcess in Windows is used to recover the executed processes). Formats of the executable images are well documented. Therefore we may use them to extract information of the API functions offsets and instrument its entry points with monitoring function calls. Monitoring function sends a message to the user or a higher-level plugin.

Process monitoring plugin provides a list of the currently executed guest processes to the user. For each discovered process it stores the following tuple: execution context (page directory base register, e.g. CR3 for x86); parent execution context; process id assigned by operating system (for user's convenience); name of the executable image.

For Linux we hook fork and clone functions for creation of the processes, and execve for running the programs. Windows uses NtCreateProcess for creating process without any threads and NtCreateThread for adding new threads. We use parameters and return values of these functions to reconstruct the list of the running processes. We also hook CreateProcess in Windows to match execution context and image file with the process id.

3 PERFORMANCE EVALUATION

To measure instrumentation overhead we executed QEMU on a machine with Intel Core i7 CPU with 8 cores at 3.40GHz, 8Gb RAM, 500Gb HDD, and 64-bit Ubuntu 14.04. Virtual machine on i386 platform had 128Mb of memory. We used Windows XP and Arch Linux as guest OSes.

We ran several tests: booting Windows XP, booting Linux, downloading 255Mb file under Linux, packing downloaded file with gzip, and unpacking the created archive. We measured system call instrumentation overhead and file operations logging overhead (Table 1). In most cases instrumentation of the system calls incurs very low overhead (5.1% on average). Logging of download, pack, and unpack incurs greater overhead, than booting OS, because file log includes the contents of the buffers for all read/write operations.

We also reviewed the overhead of other known VMI frameworks. DECAF incurs 22% overhead for booting Windows XP on x86 [9]. TCG plugins framework provided only simple plugins for instruction counting, therefore it cannot be directly compared with our approach. Simplest instruction counting plugin incurs 3% overhead, calling empty function in every translation block — 25%. QTrace instrumentation framework incurs 90% overhead for instruction counting plugin [14]. According to this data, our tool is competetive

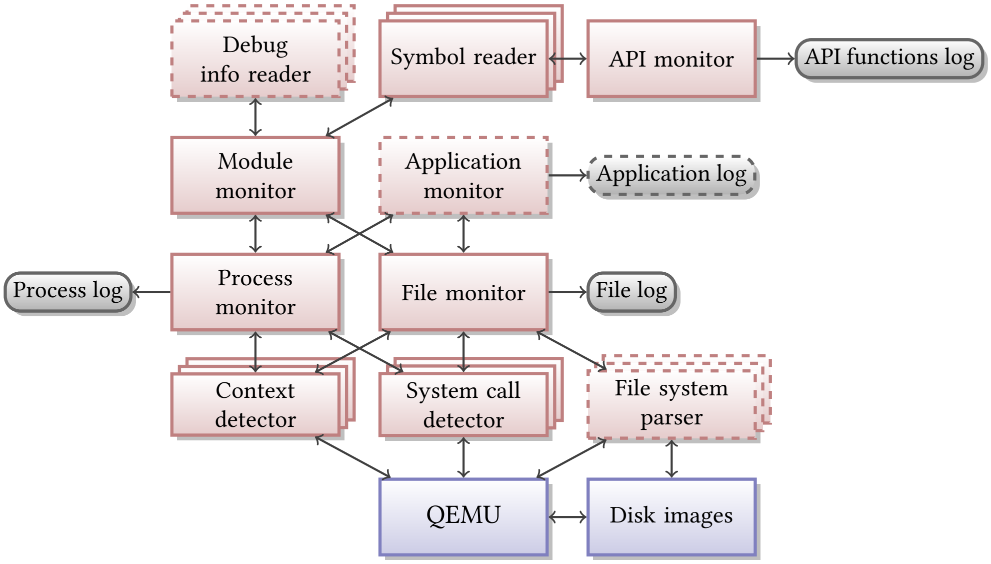

Figure 1: Plugins for virtual machine introspection. Dashed modules are not implemented yet.

Table 1: Instrumentation and file monitoring performance

Test case	Instr. overhead, %	Logging overhead, %
Loading Windows	3.2	9.5
Loading Linux	3.1	9.3
Downloading	6.7	237
Packing	6.6	168
Unpacking	5.6	56

with other system-wide instrumentation and monitoring frameworks.

4 INTROSPECTION USE CASES

4.1 Debugging Linux Kernel

Supporting new platforms in the simulator is a tough task, because complete specifications may not be available. Everything we have is the firmware and brief system description. When booting OS in the simulator fails we cannot even log in into the system and examine startup log to find out the reason. We created set of plugins that collect the following information from the running system: console output, messages sent to `dmesg` with `printk` function, programs that started with `execve` function. Fragment of such log is shown in Figure 2. We used this kind of log to understand how analyzed system behaves and what features hardware simulator must provide.

4.2 Process List in Windows

Process monitoring plugin collects information about executed processes. User may request the list of the currently running processes and select specific process to be monitored with other plugins. To ensure that produced process list is correct, we compared this output with the task manager list. Sample output is presented in Table 2.

```
<execve>/sbin/xz /sbin/xz --check=sha256
    -d /bin.tar.xz
<execve>/sbin/ftar /sbin/ftar -xf /bin.tar
</dev/console>
</dev/console>System is starting...
<syscall> mount ret=-6 source=/dev/hda1
    target=/data filesystemtype=ext2
<stderr>Can't create /etc/ssh directory
</dev/console>Abort booting!
```

Figure 2: Kernel introspection log example.

Table 2: Process list fragment

Parent context	Context	PID	Image name
0x60fc000	0x28bf000	0x178	notepad.exe
0x60fc000	0xc14000	0x180	calc.exe
0x60fc000	0x7adb000	0x190	iexplore.exe
0x60fc000	0x4bf1000	0x2f4	sol.exe

4.3 Dumping Created Files

Another use case of non-intrusive VMI is extracting created files from running virtual machine. We encountered this case when loaded OS kernel extracted files with the custom unpacker. We couldn't run this program on the other machine, because it was bound to the custom kernel and libraries. OS didn't boot competely to allow logging in. Therefore to reverse engineer extracted binaries we had to extract them somehow to the host machine. We created

Table 3: Fragment of the API functions log

Module name	Function name
Linux log	
libc.so	realloc
libc.so	__libc_malloc
libc.so	__open64
libc.so	__fxstat64
Windows log	
gdi32.dll	SelectPalette
gdi32.dll	DeleteDC
user32.dll	ReleaseDC
gdi32.dll	GdiReleaseDC

the plugin which hooks file creation and write operations and saves obtained data to the host file system. This helped us to find out the hardware features that should be implemented in the simulator.

4.4 Windows and Linux API Monitoring

Two API monitoring plugins are intended to detect named function calls located in executable modules. There are two plugins because we support two most used executable formats: PE and ELF. Usually they are not used simultaneously within the same system, therefore only one of the plugins should be loaded.

These plugins detect loading of the shared modules (.dll, .so), parse its headers, and put tracepoints at named functions entries. Therefore we can trace the calls of the exported functions. Our plugins work for Windows XP/7/8 and for wide range of Linux-based systems (including the embedded ones). Fragments of Windows and Linux API logs are presented in Table 3.

5 RELATED WORK

In this section we give a revision of previous studies carried out by other researchers and related to reverse engineering. Our approach differs from prior ones by using non-volatile data sources (like executed file formats and system call interface) for the VMI. Therefore our approach does not require embedding any agents into the guest system. We also don't rely on hardware virtualization to allow using our analysis for non-x86 platforms.

RTKDSM system leverages OS analysis capabilities of Volatility computer forensics framework to simplify and automate analysis of VM internal states [10]. RTKDSM system uses Volatility for locating the OS-specific data structures and uses host-side monitoring agent to keep track of the changes in these structures. The main limitation of RTKDSM system is targeting to x86 platform, because of using Xen hypervisor for the virtual machine.

PinOS is the framework for whole-system dynamic instrumentation [3]. PinOS can use plugins developed for Pin dynamic instrumentation framework, but it can only boot Linux on x86 virtual machine. PinOS incurs significant execution slowdown (up to 120x) even without any instrumentation.

Another attempt to create Pin-compatible full-system analysis platform is PEMU [16]. It supports introspection of user level processes and OS kernels. To solve the semantic gap problem, PEMU forwards system calls to the guest. These system calls are used to retrieve guest-level information from the virtual machine. PEMU is intrusive, because the forwarded syscalls may alter guest system behavior. Therefore it cannot be used for the offline analysis of the recorded execution.

QTrace is an instrumentation extension API developed on top of QEMU [14]. QTrace incurs 90% overhead for instruction counting plugin. Proof-of-concept version is available on github[1], but it does not provide any features except register and memory access tracing.

TCG plugins is a proof of concept QEMU-based toolkit for dynamic instrumentation of the system execution [8]. It includes several profiling plugins and instrumentation layer for QEMU.

DECAF is a platform-neutral whole-system binary dynamic analysis framework built upon QEMU [9]. It reconstructs OS-level semantic view with VMI engine. DECAF supports Windows and Linux operating systems, ARM and x86 hardware platforms. The main drawbacks of DECAF are execution overhead (15% slowdown without any instrumentation), old version of used QEMU, and relying on internal OS data structures, which makes it dependent on OS builds.

Dolan-Gavitt et al. describes technique of mining memory accesses for VMI [4, 5]. They created a prototype system, Tappan Zee Bridge (TZB), which observes memory accesses in runtime to find points in the program that access security-relevant information. This system incurs significant overhead and uses guest agents for setup.

Nitro is a KVM-based framework for VMI [13]. It was tested with guest Windows, Linux on 32- and 64-bit platforms. Nitro is able to trace system calls on all these platforms, but it is limited to x86, because of using KVM.

VMWatcher uses non-intrusive introspection approach of the virtual machines for malware detection [11]. However, it scans the guest memory to find OS kernel data structures, making the whole approach dependent from the knowlenge about the source code and used build tools and parameters. This approach is implemented for several Linux builds and for Windows XP.

6 CONCLUSION

In this work we described a promising approach for non-intrusive VMI through monitoring the system calls and virtual hardware events. We showed that this approach is practical by creating the plugins for API, file and process operations monitoring. These plugins may be used for non-intrusive logging of Windows- and Linux-based guest systems. Our approach may be used for monitoring of read only virtual machine images and for offline analysis of the execution recordings. Instrumentation part of our framework and several plugins for it were published on github[2].

ACKNOWLEDGMENTS

The work was partially supported by the Ministry of Education and Science of Russia, research project No. 2.6146.2017/8.9.

[1] https://github.com/x-y-z/QTRACE

[2] https://github.com/ispras/qemu/tree/plugins

REFERENCES

[1] K. A. Batuzov, P. M. Dovgalyuk, V. K. Koshelev, and V. A. Padaryan. 2012. 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 (2012), 77–94. https://doi.org/10.15514/ISPRAS-2012-22-6

[2] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '05)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=1247360.1247401

[3] Prashanth P. Bungale and Chi-Keung Luk. 2007. PinOS: A Programmable Framework for Whole-system Dynamic Instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07)*. ACM, New York, NY, USA, 137–147. https://doi.org/10.1145/1254810.1254830

[4] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan. 2014. Repeatable Reverse Engineering for the Greater Good with PANDA. (Oct. 2014).

[5] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. 2013. Tappan Zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security (CCS '13)*. ACM, New York, NY, USA, 839–850. https://doi.org/10.1145/2508859.2516697

[6] Pavel Dovgalyuk. 2012. 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)*. IEEE Computer Society, Washington, DC, USA, 553–556. https://doi.org/10.1109/CSMR.2012.74

[7] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. 2008. The Evolution of System-Call Monitoring. In *Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC '08)*. IEEE Computer Society, Washington, DC, USA, 418–430. https://doi.org/10.1109/ACSAC.2008.54

[8] Christophe Guillon. 2011. Program instrumentation with qemu. In *1st International QEMU UsersâĂŽ Forum*, Vol. 1. 15–18.

[9] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. 2014. 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)*. ACM, New York, NY, USA, 248–258. https://doi.org/10.1145/2610384.2610407

[10] Jennia Hizver and Tzi-cker Chiueh. 2014. Real-time Deep Virtual Machine Introspection and Its Applications. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '14)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2576195.2576196

[11] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. 2007. Stealthy Malware Detection Through Vmm-based "Out-of-the-box" Semantic View Reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/1315245.1315262

[12] Asit More and Shashikala Tapaswi. 2014. Virtual machine introspection: towards bridging the semantic gap. *Journal of Cloud Computing* 3, 1 (2014), 1–14. https://doi.org/10.1186/s13677-014-0016-2

[13] Jonas Pfoh, Christian Schneider, and Claudia Eckert. 2011. *Advances in Information and Computer Security: 6th International Workshop, IWSEC 2011, Tokyo, Japan, November 8-10, 2011. Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Nitro: Hardware-Based System Call Tracing for Virtual Machines, 96–112. https://doi.org/10.1007/978-3-642-25141-2_7

[14] Xin Tong and A. Moshovos. 2015. QTrace: a framework for customizable full system instrumentation. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*. 245–255. https://doi.org/10.1109/ISPASS.2015.7095810

[15] Lok Kwong Yan and Heng Yin. 2012. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium (Security'12)*. USENIX Association, Berkeley, CA, USA, 29–29. http://dl.acm.org/citation.cfm?id=2362793.2362822

[16] Junyuan Zeng, Yangchun Fu, and Zhiqiang Lin. 2015. PEMU: A Pin Highly Compatible Out-of-VM Dynamic Binary Instrumentation Framework. *SIGPLAN Not.* 50, 7 (March 2015), 147–160. https://doi.org/10.1145/2817817.2731201