VirtAV: an Agentless Runtime Antivirus System for Virtual Machines

Hongwei Tang[1,2,3,4], Shengzhong Feng[1,2,3], Xiaofang Zhao[3,4] and Yan Jin[3,4]
[1] Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences
Shenzhen, 518055 - China
[e-mail: tanghongwei@ict.ac.cn, sz.feng@siat.ac.cn]
[2] Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences
Shenzhen, 518055 – China
[3] University of Chinese Academy of Sciences
Beijing, 100049 - China
[4] Institute of Computing Technology, Chinese Academy of Sciences
Beijing, 100190 - China
[e-mail: zhaoxf@ict.ac.cn, jinyan@ncic.ac.cn]
*Corresponding author: Hongwei Tang

Received August 9, 2016; revised May 4, 2017; accepted July 10, 2017; published November 30, 2017

Abstract

Antivirus is an important issue to the security of virtual machine (VM). According to where the antivirus system resides, the existing approaches can be categorized into three classes: internal approach, external approach and hybrid approach. However, for the internal approach, it is susceptible to attacks and may cause antivirus storm and rollback vulnerability problems. On the other hand, for the external approach, the antivirus systems built upon virtual machine introspection (VMI) technology cannot find and prohibit viruses promptly. Although the hybrid approach performs virus scanning out of the virtual machine, it is still vulnerable to attacks since it completely depends on the agent and hooks to deliver events in the guest operating system. To solve the aforementioned problems, based on in-memory signature scanning, we propose an agentless runtime antivirus system VirtAV, which scans each piece of binary codes to execute in guest VMs on the VMM side to detect and prevent viruses. As an external approach, VirtAV does not rely on any hooks or agents in the guest OS, and exposes no attack surface to the outside world, so it guarantees the security of itself to the greatest extent. In addition, it solves the antivirus storm problem and the rollback vulnerability problem in virtualization environment. We implemented a prototype based on Qemu/KVM hypervisor and ClamAV antivirus engine. Experimental results demonstrate that VirtAV is able to detect both user-level and kernel-level virus programs inside Windows and Linux guest, no matter whether they are packed or not. From the performance aspect, the overhead of VirtAV on guest performance is acceptable. Especially, VirtAV has little impact on the performance of common desktop applications, such as video playing, web browsing and Microsoft Office series.

Keywords: agentless, antivirus, antivirus storm, virtual machine, virus signature

A preliminary version of this paper appeared in IEEE ICACT 2016, Feb 2-4, Korea. This version includes a detailed description on the basic idea, design and implementation of VirtAV.

https://doi.org/10.3837/tiis.2017.11.026 **ISSN : 1976-7277**

1. Introduction

Antivirus protection is necessary for modern computers. As virtualization technology has been extensively adopted to enhance the effectiveness of resource utilization, more and more applications are deployed in VMs. Currently three antivirus approaches, i.e., internal approach, external approach and hybrid approach, are provided in the industry and academia. In the **internal approach**, antivirus software is installed in the VM and it often uses signature-matching [1][2] technology to search viruses in files [3][4]. When all guest VMs on a host machine schedule virus scanning or signature database updating simultaneously, the host will be overloaded and corresponding performance of VMs will be degraded dramatically. As a result, the 'antivirus storm' problem [5] is inevitable because of a large number of concurrent resource-intensive operations stressing on both computing resource and I/O resource. On the other hand, as a useful approach that facilitates VM fault tolerance and system maintenance, snapshot & rollback also has been applied. However, when the VM rollbacks to a snapshot previously saved, the signature database installed inside may be out of date [6][7]. In this case, antivirus software may be unable to detect newer viruses timely, which leads to security vulnerabilities of antivirus peculiar to virtualization environment. We call it as a 'rollback vulnerability' problem. Moreover, antivirus software is highly susceptible to attacks when exposed in the VM under protection [8].

The **external approach** is inspired by the feature that the VMM has supervisory privilege on guest VMs. Antivirus software is totally moved out of the VM [8][9][10], and gets necessary information from the guest OS using VMI techniques [11][40][46]. The existing work following this approach can only detect virus in guest files or memory, and can't prohibit it properly.

The **hybrid approach** is usually composed of two parts: hooks on the guest OS to monitor events and capture information, and scanning engine deployed in a dedicated VM [5][12][13]. The most widely used antivirus product from VMWare and Trend Micro follows this approach [13]. Although this approach can avoid the antivirus storm problem and the rollback vulnerability problem, the hooks or agents in the guest OS are still vulnerable to attacks.

Furthermore, some research shows that, over 80% of modern viruses appear to be using packing techniques to evade detection by antivirus software. There is also evidence that more than 50% of new viruses are generated by simply packing existing ones [48][49]. However, traditional approaches widely adopted by antivirus software is using unpacking routines to recover the original virus program before scanning it [3]. The limitation of this approach is that it needs a specific unpacker for each packer, and moreover, only viruses packed with known packers can be unpacked and detected.

Motivated by the aforementioned observations, an agentless antivirus system VirtAV that is based on in-memory signature scanning is proposed in this paper. Basically, it can be classified into external approach since signature scanning and database updating are performed completely outside of guest VMs under protection. The main contributions are listed as follows.

- We designed an agentless approach for runtime antivirus protection on VMs, which actively scans guest binary codes on the VMM side, transparently performs virus detection and timely prohibits virus found. Such an approach guarantees the security of antivirus system to the greatest extent because there is no attack surface exposed to the

outside world. It avoids the antivirus storm and the rollback vulnerability problems. Moreover, it can also detect packed virus relying on the virus to unpack itself in memory.

- We proposed memory signature of virus, variant of file signature used in traditional antivirus software, which can be used to uniquely identify virus when it has been loaded or even partially loaded into memory. Such an approach helps antivirus system detect and prohibit virus in memory before it is activated.
- We implemented a prototype system based on the Qemu/KVM hypervisor and the open-source scanning engine of ClamAV. Functionality verification reveals that VirtAV is independent of guest operating systems, and is able to detect both non-packed and packed viruses running in user space or kernel space inside guest VMs. Moreover, benchmarks of typical desktop applications, such as Windows booting, video playing and synthetic office workload, are evaluated in both single-VM and multiple-VM environments. The performance results show that the overhead of VirtAV is reasonable and acceptable. From the results, we can further draw a conclusion that antivirus storm is eliminated by VirtAV in the multiple-VM environment.

The rest of the paper is organized in the following manner: Section 2 gives a brief introduction of Qemu/KVM hypervisor and ClamAV antivirus system, and reviews related works on security protection on virtual machines. Section 3 discusses the main design principles on VirtAV. Section 4 gives the implementation details of VirtAV. Section 5 presents the experimental results from the viewpoints of function and performance. Finally Section 6 concludes the paper and mentions the future work.

2. Background and Related Work

2.1 Qemu/KVM Hypervisor

Qemu/KVM hypervisor is a full virtualization solution based on Linux for X86 platform with CPU virtualization extensions (Intel VT or AMD-V). It provides isolated virtualized hardware environment for virtual machines running unmodified Linux or Windows as guest OS. Guest codes (user-level processes or kernel-level OS) run natively on CPU in non-root mode with the exception of sensitive instructions or operations, such as accessing I/O port, which are trapped to KVM, and emulated by KVM and Qemu. For each virtual machine, there is an independent memory address space that starts from physical address 0x0. On the host, EPT (from Intel, or NPT from AMD) is used by MMU to translate guest physical memory address (GPA) into host physical memory address (HPA) when CPU that is operating in non-root mode accesses guest memory. Moreover, if the corresponding host memory page has not been allocated or mapped, or there are not sufficient access rights on the page, EPT violation will be generated and captured by KVM.

2.2 ClamAV

ClamAV is the most widely used open-source antivirus system, which supports detection of both non-polymorphic viruses and polymorphic viruses. For non-polymorphic viruses, signatures are in simple string format, and the Boyer-Moore (BM) algorithm is used to detect this kind of viruses. While for polymorphic viruses, signatures could contains regular expressions and wildcards, and the Aho-Corasick (AC) algorithm [2] is adopted.

Specially, the implementation of the AC algorithm in ClamAV uses a trie to store the finite-state machine (FSM) constructed from the signatures. In addition, 3 helper functions including *goto*, *failure* and *output*, are defined accordingly. The *goto()* function on a given

state points out which state to move if the next input character matches with any predefined input of the state, otherwise, the FSM resorts to the *failure()* function. The *output()* function summarizes and outputs target patterns matched in the end. At the initialization phase, fixed string parts of each polymorphic signatures are loaded from the database, and are used to build the trie. At the scanning phase, ClamAV scans an input file byte-by-byte and detects occurrences of signatures. In general, fixed parts of signatures are scanned in the trie firstly, and after that wildcards and regular expressions are processed. Take a signature with a wildcard as an example. The fixed string is split by an asterisk wildcard into two substrings that is in the "substring1*substring2" format. The two substrings are matched individually by the AC FSM. When all the two parts are found, it will further verify the order and gap between them. ClamAV uses a 256-element array for each node with each element corresponding to an ASCII character. The integer value of the input byte is used as the index into the array, and if the element is present, a match character is found. Furthermore, ClamAV provides several common unpackers, such as MEW, Upack, Petite, yC, FSG, AsPack, WWPack, NsPack and etc., to recover the original programs from packed viruses obfuscated with the known packing algorithms.

2.3 Related Work on Intrusion Protection of Virtual Machines

Virtualization technology brings opportunities for security and is extensively used to improve the anti-attack capability of intrusion or malware detection and protection tools [8][26].

A. *VMI-based intrusion protection*

Security tools can be installed outside of the VM that need to be protected, and monitors the guest via special interfaces, such as VMI [46]. As is isolated from the VM, the security tools are immune to attacks even if the VM is compromised by attackers. However, as lacking of guest OS-level semantic (known as the semantic gap [40][46]), the accuracy and effectiveness of the security tools outside the VM have to be sacrificed. Livewire [8] is a general IDS framework adopting VMI technology, which uses the priori knowledge about the guest OS data structures to interpret the hardware-level view in OS-level semantics. Especially, to detect signatures of malicious program, it performs a full scan of all the host memory. However, this methodology cannot timely find out virus and prohibit them. VMwatcher [9] reconstructs semantic views of guest OS in a non-intrusive manner, and provides objects with the guest OS view to external malware detection tools. Just like Livewire, VMwatcher only provides malwares detection and cannot intervene execution of the guest. [12] proposed a hybrid approach enabling security tools to perform active monitoring while pertaining isolation to ensure security. It hooks the guest OS to monitor interested events and handle the events in an isolated VM. It solves the semantic gap problem in a simple manner, but it heavily depends on integrity of the guest OS. [20] described a framework that enables security monitoring applications to be placed in the untrusted guest VM without sacrificing the security guarantees. The monitoring application itself is protected by the VMM and should be self-contained to ensure its integrity. This method imposes restrictions on the monitoring applications and even requires reconstruction of commodity tools to adapt to this framework. [27] proposed a mechanism to verify the integrity of user and kernel code at runtime in page granularity. [50] described a framework that combines intrusion monitoring, evidence preservation, in-depth log analysis and decision making on suspicious events handling for guest VMs.

In addition, there are also advances in the research of VMI. VMI technique requires privilege access to the virtual machine monitor, so that it is usually not provided in public

cloud environments. CloudVMI [38] virtualizes the VMI interface and allows cloud users to introspecting their own VMs. For live VM monitoring by VMI, VMI tools and guest VMs run concurrently which might cause race conditions and data inconsistency. TxIntro [47] leverages the atomicity of hardware transactional memory to ensure concurrent and consistent VMI. To reduce the overhead of VMI and improves its generality for different OSes, [43] presents an approach that redirects the system calls of monitoring tools to the dummy process in guest VM which collects guest OS states natively in a lightweight manner. [45] presented the observation that the definition and layout of critical information in kernel data structures for process is stable as the evolution of kernel versions. The generality of VMI is improved by reconstructing the process list basing on only partial information, that is believed as sufficient for intrusion detection. [52] introduced hypervisor introspection (HVI) to detect hypercall-based attacks with hardware supports, such as nested virtualization and EPT protection.

B. *Not VMI-based intrusion protection*

To solve the antivirus storm problem, VMware & TrendTech [5][13] proposed a so-called agentless antivirus framework - vShield Endpoint. In that framework, the primary causes of storm including signature scanning and database updating are offloaded to a dedicated VM, called secure virtual appliance (SVA). Moreover, there is a lightweight agent (called 'EPsec Thin Agent') deployed in guest OS whose main duties are monitoring file operations in guest OS and forwarding files to SVA to be scanned. However, the framework is not truly agentless, and indeed, the agent is exposed to attacks from malicious users or malware. Moreover, the agent depends on the guest OS. When the agent is not running, for example in booting phase or shutting down phase of the guest OS, the guest VM loses protection of antivirus.

[24] presented an external approach to malware analysis which can hide the analyzer from the target. With the help of hardware virtualization technology such as Intel VT, it can offer both instruction level and system call level tracing of the target in guest VM. [25] described a Qemu-based system that dynamically analyses Windows kernel-level code and extract malicious behaviors from rookits. [37] introduced an event-logging based reliability and security monitoring framework for virtual machines, which relies on hardware invariant to provide an isolated root of trust, so that the events and states about guests cannot be modified by attackers and failures inside guests. [41] presented an out-of-VM user-mode process execution monitoring which supports existing user-mode process monitoring tools such as strace, ltrace and gdb. The suspect process to be monitored is moved out from the production VM to the monitor VM, where the user-mode monitoring tools run side-by-side with the process. This approach removes semantic gap and can directly intercept the process execution at the granularity of user-level function calls. CIVIC [44] creates a live replica of the production VM and the inspection or analysis operations are performed on the replica. It leaves the production VM unmodified without any impact or side-effects during monitoring. HyperCoffer [42] is a hardware-software co-designed framework that guards the privacy and integrity of tenant's VMs in cloud environment. By extending processor virtualization with memory encryption and integrity checking to secure data communication with off-chip memory, it can protect guest VMs against an untrusted hypervisor and physical attacks. [51] described a network-based computer worms detection system for virtualization environment, which runs isolated from the guest VM on the hypervisor. [6] proposed an audit based approach to protect against VM rollback attack which logs all the suspend/resume and migration operation and audits the log for checking malicious rollback behaviours. Furthermore, there are also researches [21][22][23] focusing on malware analysis with the help of virtualization techniques.

Basic Idea and Challenges

The overall goal of VirtAV is to provide antivirus protection for VMs without introducing the antivirus storm problem and the rollback vulnerability problem firstly. In addition, as the antivirus agent in the guest OS leads to vulnerabilities and weakness points on security, VirtAV is expected to implement an agentless solution. Furthermore, for the purpose of VirtAV to be a practical solution, the negative performance impacts on the guest VMs should be as low as possible.

The basic idea of VirtAV is to scan guest binary codes in memory at runtime on the VMM side for detecting and preventing viruses. As we all known, memory is the only way that must be passed for the execution of any binary codes, specially, in the virtualization environment, it is the host memory. Furthermore, the VMM can inspect any location of the host memory, so signature-based virus detection in memory is feasible. In this way, concurrent file-based virus scanning in co-locating VMs is avoided, and antivirus storm problem is solved. Based on the idea, we have to deal with the following key challenges.

A. Recognition of viruses in memory

In traditional ways, file signature is used to distinguish specific virus from other viruses and non-virus files. And in general, periodically file scanning and event-driven (such as launching a program, downloading a file from the Internet) file scanning are two common policies in commodity antivirus products. However, file signatures are no longer applicable to recognize viruses in memory on the VMM side. On the one hand, file signature based virus detection needs understanding of OS-level semantics which can't be obtained by the VMM. The VMM is unaware of when the guest is launching a program, or when it is downloading a file. On the other hand, binary codes are loaded into memory in an on-demand manner in the unit of page. As a result, we usually get partial view of the binary codes in memory even when it has already started running. Accordingly, recognition of viruses in memory is one of the key challenges. Subsection 3.2 provides the details to solve this problem.

B. Virus scanning trigger strategy

Unlike the ways of file-based antivirus products, VirtAV scans viruses after the executable is loaded into memory. The selection of a right moment to perform virus scanning is crucial to the effectiveness of antivirus. Virus scanning on a piece of binary codes should be before it is scheduled to run on the CPU. Otherwise, the execution of the binary codes or even part of that might damage the system. However, the VMM cannot intercept guest OS-level events, for example, process scheduling. Instead, it can only catch hardware-level events, such as EPT violation. Therefore, the strategy regarding when and how to trigger virus scanning is another challenge. Subsection 3.3 describes how VirtAV utilizes memory virtualization to solve this problem.

C. detection of packed viruses

To bypass virus scanning, viruses usually pack themselves and unpack at runtime. Therefore, antivirus software needs to deal with a lot of packers and be prepared for new ones every day. Nearly all antivirus products employ unpacking engines [29]. It is difficult or even impossible to unpack with only partial memory view of the executable on the VMM side. So we rely on the viruses to unpack themselves, and scan the binary codes decoded. To support for detection of packed viruses is also a problem, and the related design is in Subsection 3.2 and 3.3.

D. *Performance guarantee*

To provide runtime antivirus protection, VirtAV detects viruses on the critical path of execution of binary codes. Moreover, the execution is paused when VirtAV is scanning virus, which means the performance overhead is inevitable. To be a practical antivirus solution, VirtAV should reduce the overhead and provide acceptable performance to guest VMs. The solution is provided in subsection 3.5.

3. Design and Implementation

Basically, VirtAV is an agentless antivirus system for VMs which provides antivirus protection by extending KVM and Qemu. It identifies virus by memory signature and dynamically scans binary codes in host memory to detect viruses. It interposes synchronous virus scanning operations on the critical path of program executions in guest VM and it ensures that any code executed by vCPU has been examined. It is transparent to the guest OS and requires no modifications or hooks on the guest OS. The design and implementation is discussed in the following subsections.

3.1 Overall Architecture

The overall architecture of VirtAV is shown in **Fig. 1**. VirtAV is built upon KVM/Qemu hypervisor and is composed of four parts: 1) VirtAV-engine embedded in Qemu, 2) a centralized signature database deployed on the host Linux, 3) VirtAV-stub in KVM, and 4) VirtAV-cleaner on host Linux.

VirtAV-engine: In essence, VirtAV can be integrated with any signature-based scanning engine from different antivirus solutions. In our prototype system, we use the open-source scanning engine of ClamAV [3] and integrate it with Qemu, which is designated as VirtAV-engine. The core of the VirtAV-engine is the AC finite-state machine for signature pattern matching, which supports exact match, wildcard match and regular expression match [3]. It runs in the same context with Qemu's vCPU thread and scans host memory footprints of executables in guest VMs to search memory signatures of virus. Furthermore, the vCPU thread will be paused when it scans host memory.

Memory Signature Database: We propose memory signature to recognize virus. The definition of memory signatures are presented in subsection 3.2. A new database ('*.msdb*') for memory signatures is created in ClamAV. Moreover, various databases of ClamAVare shared by all the VirtAV-engines on the same host. Especially, it is independent of any guest VM, and can be kept up-to-date by the host, so that the rollback vulnerability and updating storm are both avoided.

VirtAV-stub: To trigger virus scanning at the right moment, we extended the memory virtualization module in KVM to trap specific memory access events. Once trapped, the vCPU is paused and VirtAV-stub transfers the binary codes in the corresponding page frame to VirtAV-engine for scanning. When there is virus found, VirtAV will take actions according to predefined policies, e.g., killing the guest process and isolating or removing the executable by default. To kill the process, it manipulates the guest page table entry of the process to revoke the execute permission on the guest page frame using VMI techniques. After the vCPU resumes, the process is automatically killed by the guest OS due to general protection fault.

VirtAV-cleaner: It is an auxiliary utility for virus treatment. It locates the executable of the virus according to information provided by VirtAV-stub, and takes actions to isolate or remove the virus. It is built upon libguestfs [14] which provides interfaces to access and modify guest file-system from outside of the VM.

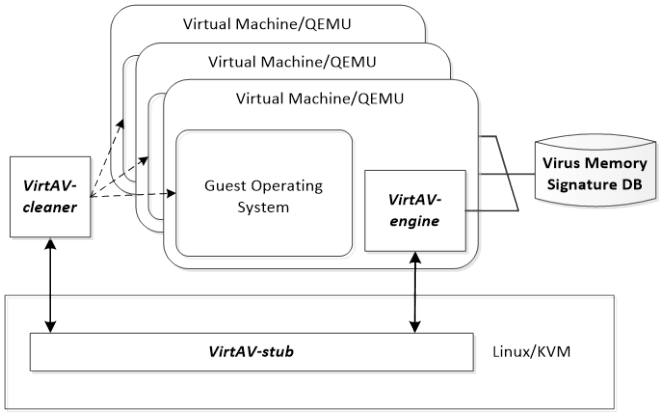

Fig. 1. Overall architecture of VirtAV

3.2 Memory Signature of Virus

To find virus accurately and timely in memory, VirtAV identifies virus using memory signature which is slightly different from file signatures. File signature is usually picked up from the text section of virus' executable. However, the way to generate memory signature is different from that of file signature. The reason can be explained in detail as follows.

In modern operating systems, such as Windows and Linux, memory management is based on paging which divides the continuous system memory into fixed-sized page frames. Page frames are assigned and mapped to processes' virtual address spaces on demand. When a process is created, the OS establishes mappings between the process' virtual memory areas and sections of executable and libraries. By that time, page frames are not assigned to the process, and sections are not loaded into memory either. When the process starts to run, especially the time instance that the CPU attempts to fetch instructions from the virtual memory area mapped to the text section, page fault will be generated as page frame is not present. Then the OS will assign a page frame to the process and eventually load a block of contents from the executable. The text section may reside in more than one page frames and should be loaded multiple times. So we cannot rely on scanning the whole executable to detect virus because part of the virus could have been executed and may have infected the system before the file signature is loaded in.

There is no essential difference between memory signature and file signature of virus, which are both small pieces from the binary codes of virus and can be used to distinguish the virus from benign programs and other viruses. The only difference is that memory signature is consist of a collection of sub-signatures with each of them extracted from a corresponding page of the text section of virus, as shown in **Fig. 2**. A sub-signature can identify virus uniquely, so that VirtAV can detect virus with only one page of the text section loaded into memory. Generally, the page where the entrypoint of virus locates (as we called "the entrypoint page") is first loaded and executed, and the sub-signature from the entrypoint page is usually first detected by VirtAV. In this case, other pages are not loaded because the virus is prevented from running once found. Just like file signature, sub-signatures could contain fixed hexadecimal strings, wildcards, or even regular expression.

We need not to load the virus into memory for memory signature extraction. Instead, memory signature is calculated by statically analyzing the on-disk virus executable. Take PE-format executables in Windows as the example, which is composed of DOS header, NT header, section headers and several sections, such as code section (.text) and data section (.data) [15], as is shown in the figure. Furthermore, **Table 1** shows the most related

information in the PE headers for extracting memory signature. The raw length of the binary codes in the text section is given by *VirtualSize*, while *SizeOfRawData* indicates the section size in the executable. After loaded into memory, the length of the text section might change because the section is re-aligned to the *SectionAlignment* which is usually set to the page size (e.g. 4096 bytes) of the system. Specifically, if the size of the raw binary codes in the last page is less than 4096 bytes, the remainder of the page will be filled with zeros.

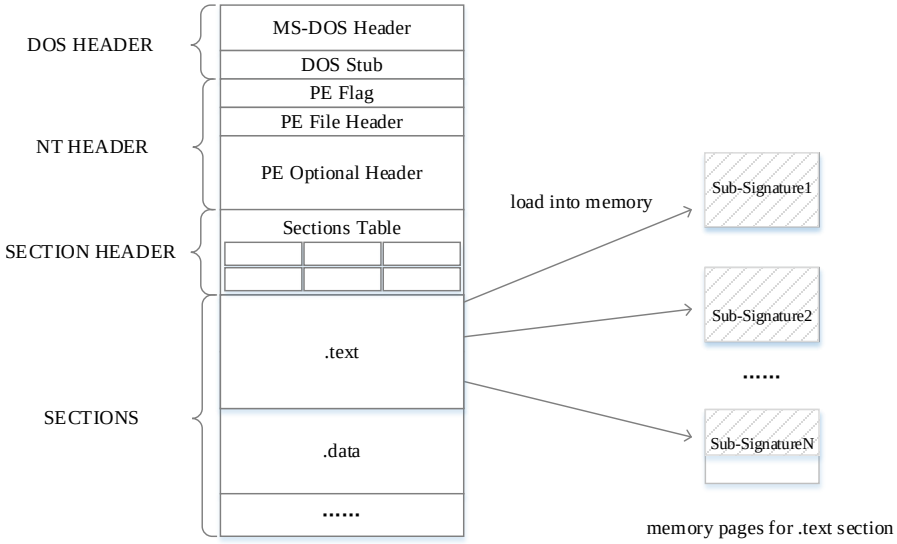

Fig. 2. Memory signature for virus in PE format

Table 1. Part of information in PE Headers [30]

Item	Meaning	PE Header
ImageBase	The preferred virtual address of the first byte of the executable when it is loaded in memory.	PE Optional Header
SectionAlignment	The alignment of sections loaded in memory, in bytes. The default value is the page size of the system.	PE Optional Header
FileAlignment	The alignment of the raw data of sections in the executable, in bytes.	PE Optional Header
Name	The name of the section, such as .text, .data, .bss, and etc.	Section Header
VirtualSize	The size of the section before aligned to FileAlignment.	Section Header
VirtualAddress	The address of the first byte of this section when loaded into memory. This value is relative to the image base.	Section Header

SizeOfRawData	The size of the section in the executable when aligned to FileAlignment. If VirtualSize < SizeOfRawData, the remainder is filled with zeroes.	Section Header
PointerToRawData	The offset of the first byte of the section from the start of the executable. The value is a multiple of FileAlignment.	Section Header

The process of calculating memory signature for virus is described as follows (shown in **Fig. 3**).

1) Locate the section header in the section table for the ".text" section, and get the start and the end offsets of the section from the start of the executable. The start offset is indicated by *PointerToRawData*, and the end offset is *PointerToRawData* + *SizeOfRawData*.
2) Iteratively copy the binary codes of the executable into 4096-byte sized temporary files, sequentially from the start offset until to the end offset. Specifically, the first byte of the text section is copied to the offset *(ImageBase + VirtualAddress) % 0x1000* of the first temporary file. The part from the start (offset 0x0) to the offset *(ImageBase + VirtualAddress) % 0x1000 -1* of the first temporary file is filled with zeros. While for the last temporary file, the size of binary codes to be copied might be less than 4096 bytes, and the remainder of the last temporary file is also filled with zeros. By this approach, the memory view of the text section is statically built from the on-disk executable.
3) Select a sub-signature from each of the above 4096-byte temporary files. The principle is minimizing both false negatives and false positives. A lot of research has been done in the field about extracting signatures from virus executables, including manually by experts and automatic solutions [1][34][35][36].
4) Store the sub-signatures into the *.msdb* database. Each memory signature is in the format of "MalwareName=subsig1, subsig2, …, subsigN".

In practice, most viruses are small in size, and even the size of text section is usually smaller than page size (typically 4096Bytes). For these viruses, we can take the file signatures as the memory signatures.

We adopt the open-source scanning engine of ClamAV which adopts AC algorithm to detect memory signatures in memory page frames. In the engine initialization phase, memory signatures are loaded from the specific ClamAV database (*.msdb*), and a trie is built from the memory signatures in host memory. Furthermore, the *goto/failure/output* functions for the AC FSM are constructed accordingly. An example trie is shown in **Fig. 4**. In the figure, boxes represent states of the FSM (or nodes of the trie), in particular, a box filled with "pat()=…" indicates a state where some pattern is matched with the input. Furthermore, the solid line with arrow represents the *goto()* function, while the dash line with arrow denotes the *failure()* function. According to the characteristic of the AC algorithm, the time complexity of the initialization phase is O(l), where l is the sum of the lengths of all the memory signatures in *.msdb*.

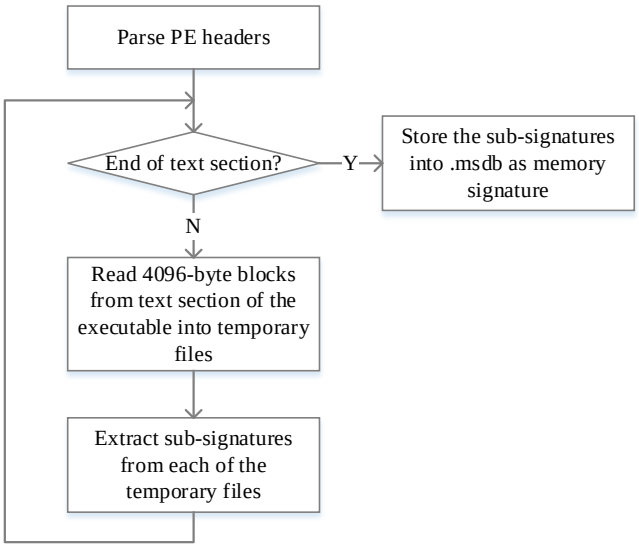

Fig. 3. Process of extracting memory signature from virus executable

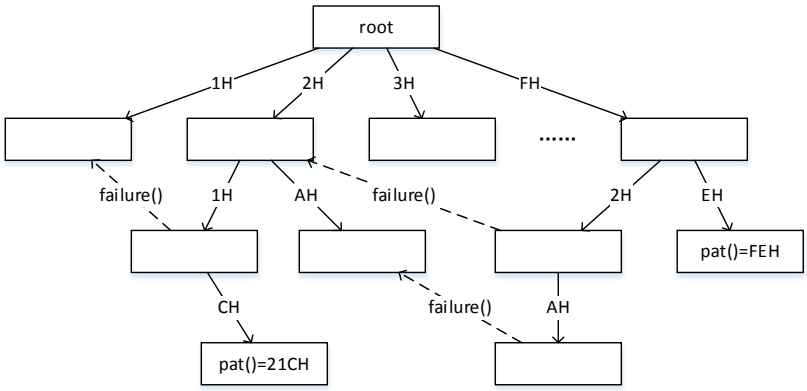

Fig. 4. An example trie built from memory signatures

When scanning a memory page, binary codes in the page are treated as a sequence of 4096 ASCII characters, which are inputted into the AC FSM, one at a time. The FSM decides which is the next state to move according to the current state, the input character and the *goto()* or *failure()* function. When a state with a matched pattern is arrived, a memory signature is found. By the AC FSM, it only needs to scan the page once. Consequently, the time complexity for scanning a memory page is O(*n*), where *n* is the length of the input sequence, typically 4096.

3.3 Virus Scanning in Memory

Generally, on-demand scanning is widely adopted in commodity antivirus products. For example, scan operations can be triggered when opening an executable to execute, downloading files from the Internet and so on. It is required that the on-demand way completely understands OS-level semantics and even hooks on system calls. However, the VMM has no knowledge of what happened in guest OS and can merely capture hardware level events, such as device I/O operations, EPT [16] violations and etc. As a consequence of the semantic gap problem [10], previously proposed external solutions have fundamental shortcomings on the instantaneity and effectiveness.

To get around this problem, VirtAV adopts a different way that scans virus in host memory page frames which are mapped to guest memory space. The idea is based on the fact that executables to be executed in the guest VM should be firstly loaded into host page frames, where they can be inspected by the VMM.

For running an executable in the guest, it mainly goes through the following steps.

Step 1. The guest OS launches a process and maps the executable into the process' virtual address space. After a collection of preparations, the control flow is transferred to the process.

Step 2. Driven by guest page faults or EPT violations, guest physical memory and further host physical memory for the executable is allocated respectively. From the VMM's view, the host page frames where guest binary codes reside should be given execute permission. By default, the VMM gives each page the full permissions including read, write and execute, to reduce interventions on the guest.

Step 3. The guest OS loads binary codes from the executable into memory one page at a time.

Step 4. The vCPU fetches instructions from the page and runs the executable.

From the above observation, virus detection should be inserted between **Step 3** and **Step 4** on the VMM side. To this end, we make a minor change on the memory virtualization module that the VMM only grants read and write permissions on page frames provisioned to guest VMs. Moreover, the basic principle is that the write permission and the execute permission on a host page frame should not be granted at the same time. For guest executable, it can be loaded into the page frames with write permission, but cannot be executed there. Therefore, when the vCPU fetches instructions from the page frame, it will be trapped to the VMM due to permission violation. That is the right moment when the VirtAV-engine carries out virus inspection on the page frame. If there is no virus found during scanning, the host page frame will be granted execute permission and the execution of the guest executable is resumed.

When the guest attempts to write to a host page frame with read and execute permissions, it will also be trapped by the VMM. In this case, the VMM just simply grants the write permission and revokes the execute permission on the page frame. This means that the content of host page frame that has been scanned and verified by VirtAV might change. For one thing, the host page frame may be reallocated by guest OS, for example, to store other pieces of binary code. For another, packed viruses might decode binary codes into the code segment at runtime. As for the above cases, the updated host page frames should be re-checked by VirtAV. In summary, it need to inspect each newly updated page frame before fetching instructions from it by vCPU, and the trigger condition for virus detection is defined as the **first instruction-fetch operation on a newly updated host memory page frame**, abbreviated to the "1st-IF" event. To handle the event, VirtAV will scan the page frame to detect memory signatures, and prevent the virus from executing if found. If no virus is found, the vCPU will be resumed and allowed to fetch instructions from the page frame. The flow path from loading a program to executing it in the guest VM is depicted in **Fig. 5**.

As for a host page frame, it may be in one of the following four states before loading binary codes of the guest program into it.

State I: The host page frame has not been allocated and mapped to the guest yet.

State II: The host page frame has been allocated and mapped, and the access permission specified in EPT entry is read-only.

State III: The host page frame has been allocated and mapped, and the access permission specified in EPT entry is execute-enable but not write-enable.

State IV: The host page frame has been allocated and mapped, and the access permission specified in EPT entry is write-enable but not execute-enable.

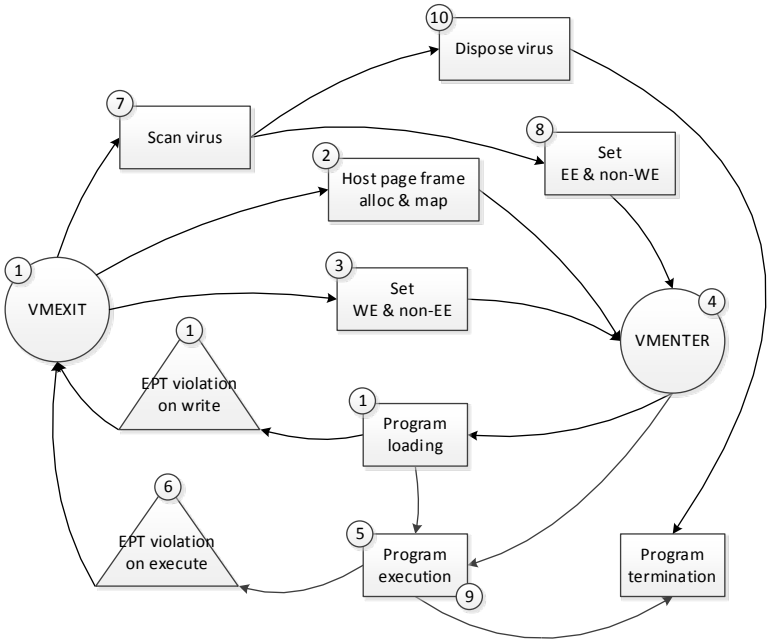

Fig. 5. Flow path of program execution in guest VM

①When loading binary codes of the guest program, VMEXIT is generated as EPT violation occurs if the host page frame is in state I, II or III. ②For state I, the VMM will allocate a host page frame, establish the EPT mapping and set the write-enable flag bit in the EPT entry. ③ For state II and III, the VMM will simply set the write-enable bit and clear the execute-enable bit in the corresponding EPT entry. ④After the EPT violation is handled, the VMM will resume the vCPU by generating VMENTER. ⑤In the guest VM, after successfully loading the binary codes, the vCPU will attempt to fetch instructions from the host page frame. ⑥Then, another VMEXIT will be generated as the execute-enable flag bit of the host page frame is cleared. That is the very "1st-IF" event defined above. ⑦By having captured the event, the VMM invokes virus scanning. ⑧If no virus is found, the VMM will simply set the execute-enable bit and clear the write-enable bit in the EPT entry and resume the vCPU. ⑨ Finally, the program successfully executes in the guest VM. ⑩If any virus is found, actions will be taken to deal with that virus.

VirtAV can also protect the guest OS kernel. From the VMM's point of view, the execution of guest kernel code will also trigger the "1st-IF" events. The difference from user-level executables is that the guest kernel is loaded into memory by boot loader during bootstrap phase. In general, host page frames for guest kernel code will be scanned only once, because they would not be reclaimed or reallocated during the lifecycle of the guest VM and the content of these page frames will be left unchanged after the first scan. Once there is virus code injected into the guest kernel, VirtAV can also be aware of the change and make inspection on the page frame.

3.4 Detection of Packed Virus

The detection of packed virus in state-of-the-art antivirus solutions is complicated. Many kinds of unpackers have been developed to extract compressed or encrypted original codes from packed executable files [31][33]. In general, a packed virus is composed of an encrypted body and a built-in decryption routine and decryption key. The entrypoint of the virus points to the decryption routine. When the virus is loaded into memory and activated by some event, the decryption routine executes firstly and decrypts the encrypted body in memory, and then jumps to the entrypoint of the original codes to run. Many virus variants are generated from the same plain codes by adopting different packers or encryption keys.

Memory signature for packed virus is taken from the original binary codes instead of its encrypted versions, so we need to unpack it first. There are extensive researches on the topic about unpacking packed malware [31][33][39][53]. For example, one can first locates the original entrypoint (OEP) [53], then monitors jump operations to the OEP that means the completion of the unpacking process (e.g. using breakpoint of gdb), and finally, dumps the memory pages for code section with original codes into files (e.g. using ProcDump in [19]). With this, hexadecimal strings are extracted from each page for memory signature, just like in the non-packed virus case.

In the virus scanning process of VirtAV, there is no need to differentiate packed ones and non-packed ones. Furthermore, it depends on the virus to unpack itself in memory so that no unpackers are needed in VirtAV, which significantly simplifies the detection of packed viruses. As is shown in **Fig. 6**, when a packed virus is executed, ①the operation of fetching instruction from the entrypoint of the decrypter routine will trigger the "1st-IF" event, and then ②VirtAV will perform virus scanning on the page frame(s) storing the decryption codes. As the memory signature is not in these pages, the decryption codes successfully pass the checking, and ③ unpack the virus in memory as usual. After finishing the unpacking process, ④the decrypter tries to jump to the original entrypoint of the virus, this triggers the "1st-IF" event again and then ⑤VirtAV detects the virus memory signature in the page frame.

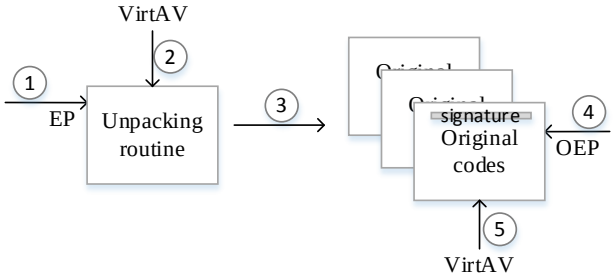

Fig. 6. virus scan for packed virus

3.5 Performance Considerations and Optimizations

As is discussed in previous subsections, virus scanning is on the critical path of program execution in guest VMs, which means performance and efficiency are very important for VirtAV to be a practical solution. In this subsection, we will introduce some fundamental design considerations and optimizations focusing on performance and efficiency.

1) Scan text section of the executable only

Generally speaking, the overhead of virus detection is proportional to the size of the data to be scanned. Commodity antivirus products scan all sections of executables on disk, or all of the system memory pages, which is time-consuming undoubtedly. VirtAV only focuses on in-memory text sections to be executed, which are small portions of the system memory, and are also small parts of the executables on disk. For this, initial access permissions on host page frames allocated to guest VMs are set to write-enable but not execute-enable, which allow loading codes or data into the page inside the guest VM. When the guest VM tries to execute codes in the page, the "1st-IF" event happens on the page and VirtAV will scan it. In this way, only the pages for text sections to be executed are scanned. From the view of the whole system, resource consumed by virus scan is significantly reduced.

2) Scan each host page frame loaded with binary codes once

VirtAV may degrade the performance of guest VMs due to the overhead introduced in the path of program execution. The overhead mainly comes from page scanning and context switches between the VMM and guest VMs. To reduce the overhead, VirtAV scans each host page frame loaded with binary codes only once, provided that the content of the page has not be changed since it is scanned. As the write-enable flag bit in EPT entry for the page frame is cleared, the page content can't be modified, so that there is no need to rescan the page frame despite the binary codes in the page frame may execute multiple times.

Moreover, in mainstream operating systems, executables are usually cached in system memory, such as system file cache in Windows and page cache in Linux, and shared by different address spaces. VirtAV detects virus on the view of host memory, which corresponds to system memory of the guest VM, so the host page frames loaded with binary codes need to be scanned only once.

3) Scan for each vCPU independently

VirtAV-stub monitors and traps the "1st-IF" events on each vCPU. When the event is captured, the vCPU generating the event is paused and trapped into VMM by VMEXIT. The other vCPUs in the same guest VM keep on running without any disturbances. If multiple events on the same page frame concurrently happens on more than one vCPUs, VirtAV will scan the page frame only once just for the vCPU on which the event firstly occurs. In this case, VirtAV-stub sets the flag of the page frame to PG_SCANNING to avoid duplicated in-flight scans, and just simply makes other vCPUs to wait for the result.

4) Zero-copy for kernel-to-user data transfer

VirtAV-stub captures the events in KVM context in kernel level, while VirtAV-engine works in Qemu context in user level. So it needs to transfer page content from kernel level to user level for scanning. To improve performance, we use zero-copy for cross-address-space data transfer. Firstly, the GPA for the guest page is got by VirtAV-stub from the *GUEST_PHYSICAL_ADDRESS* field of the VMCS structure when the "1st-IF" event happens. Then, VirtAV-stub searches the guest memory slot for the page, and computes the host virtual address (HVA) by simply adding up the base HVA of the memory slot and the offset of the page into the memory slot. At last, the HVA is directly used by VirtAV-engine to scan the guest page.

5) Exempting scan on checked same pages

There usually exist memory pages with same content across different virtual machines on a host, which are called same pages. A typical reason is the existence of common executables, runtime libraries, OS kernels and data files, in different virtual machines cloned or derived from common images. Kernel same page merging (KSM) [28] technique is adopted on KVM to merge same pages in a COW (copy-on-write) manner, which helps to increase memory

density. The merging algorithm uses two red-black trees, a stable one and an unstable one to optimize the process. The stable tree stores pages that have been merged, while the unstable tree stores candidate pages to be merged. The *ksmd* kernel thread periodically scans pages of virtual machines for merging with the pages in the two trees. For each page being scanned, *ksmd* first searches in the stable tree, using *memcmp()* function to compare the content of the given page with that of the pages in the tree. If the same page is found, the given page is freed and references to it are redirected to the same page in the tree. Otherwise, *ksmd* then searches in the unstable tree. In addtion, *ksmd* checks whether the content of the page in the unstable tree has been changed by recalculating and comparing the current checksum with that is last recorded. If the content has been changed, the page is removed from the unstable tree and not considered to be a candidate for merging. If a page to be scanned by VirtAV is as same as some pages in other virtual machines which have passed antivirus check, the scan on the page can be exempted. Based on this idea, we integrate the antivirus process with the page sharing mechanism. For this purpose, the pages in the stable and unstable trees are further divided into two parts respectively, according to whether the page has passed the check by VirtAV. The resulting four trees are checked stable tree, unchecked stable tree, checked unstable tree and unchecked unstable tree, as shown in **Fig. 7**.

- Checked stable tree: The tree for pages which have been merged and passed the check by VirtAV. Typically, pages storing common binary codes that have been executed in different virtual machines are put in this tree.
- Unchecked stable tree: The tree for pages which have been merged but not yet been checked by VirtAV. For example, common data pages of different virtual machines belong to this kind.
- Checked unstable tree: The tree for pages which have been checked by VirtAV but have not been merged yet.
- Unchecked unstable tree: The tree for pages which have not been checked by VirtAV and not been merged.

For each node of the trees, a 128-bit MD5 digest of the content of the corresponding page is stored. To search the same page for a given page in the trees, it firstly compares the MD5 digest of the given page with that in the node using Intel SSE 128-bit comparison instruction. If a match is found, it will further compares the content of the them using *memcmp()*. This is because there exists MD5 collision though the probability is very small.

In this case, when the "1st-IF" event is captured, VirtAV first searches down the checked stable tree. If the same page is found, virus scanning is exempted and the guest CPU is resumed immediately. To accelerate this process, same pages are merged asynchronously by *ksmd* in the background. Especially, a fixed-length ring array is employed to record the same pages found by VirtAV, as shown in **Fig. 8**. The "PFN" field specifies the page frame number of the page to be merged, while the "Tree" and "Node" fields indicate the page to be merged with. When the count of pages in the ring array exceeds a predefined threshold, for example, 75% of the length of the ring array, VirtAV will explicitly notifies *ksmd* to work immediately. In our prototype implementation, the length of the array is set to 2048.

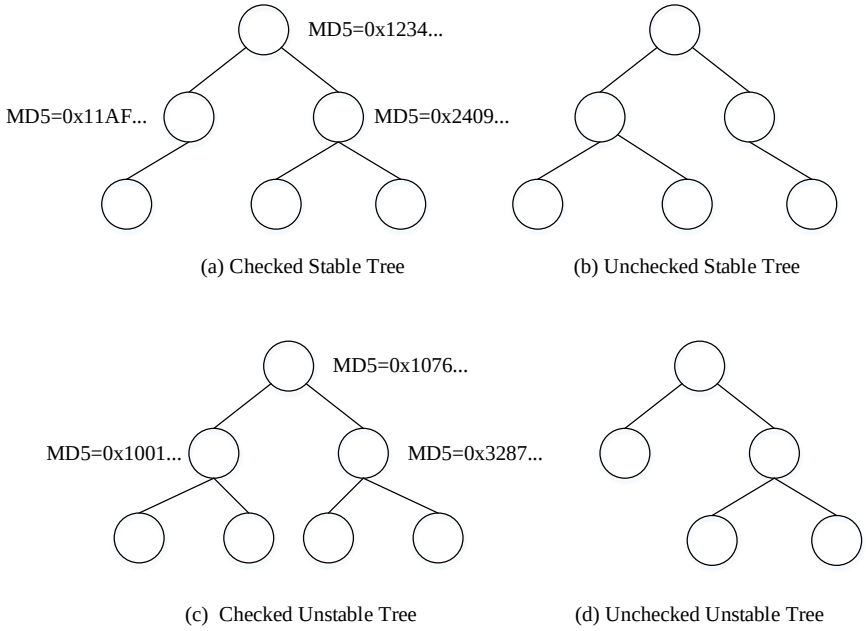

Fig. 7. 4 red-black trees for page sharing and scan exemption

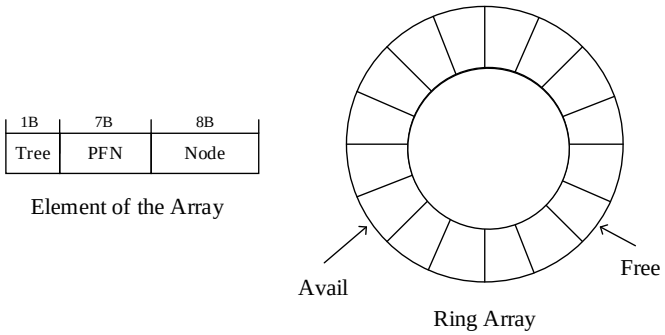

Fig. 8. Ring array for same pages to be merged

If there is no same page found in the checked stable tree, VirtAV will then search in the checked unstable tree. If found, it will also exempt virus scan on the page and merge them asynchronously. Otherwise, VirtAV performs virus scanning on the page as usual, and finally insert the page into the checked unstable tree if it passes the check. Moreover, when the memory signature database is updated, new signature patterns are added to the AC FSM. And pages in the checked trees are re-scanned in the background to detect new signatures.

The worst case overhead on comparisons is computed as **Eq.1**, where the unit is 128-bit SSE comparison. In the equation, N is the number of pages in the checked stable tree, while M is the number of pages in the checked unstable tree. For a 4096-byte page, it needs 256 comparisons by the SSE comparison instruction. So the worst-case time complexity for searching same page is $O(\log N + \log M)$.

$$\log N + \log M + 256 \qquad \textbf{Eq.1}$$

In general, binary codes occupy a small part of system memory, so the heights of the checked trees are small. Take a system with 512MB binary codes in memory, the number of pages for the binary codes is 131072, the upper limit of the height of checked trees is 11.7. And at most, 280 128-bit SSE comparisons are needed. Compared to signature matching operations on a page, where 8192 comparisons are needed, the overhead of searching in the checked trees is negligible. The space consumption of this optimization is 16 * L, where L is the length of the ring array. In the prototype implementation, L is 2048, and the space consumption is 32K bytes.

There is little change on the workflow of *ksmd*. It is woken up by the timer with fixed period or by VirtAV when the number of the elements in the ring array exceeds the threshold. When it works, it first merges the same pages recorded in the ring array, and then tries to merge other pages of virtual machines with pages in the 4 trees. When a page is merged with some page in unstable tree (checked or unchecked), the node will be moved to the corresponding stable tree. Furthermore, it also verifies whether content of pages in the unstable trees (checked and unchecked) has been changed, and removes changed pages from the unstable trees. Overall, this optimization can reduce overhead on viruses scan in the host wide.

3.6 Virus Treatment

If there is matched memory signature found in host page frame, VirtAV should dispose the virus according to predefined policies, such as killing the corresponding process and removing the virus file by default. The sketchy procedure is described as follows.

In general, VirtAV-engine, VirtAV-stub and VirtAV-cleaner cooperate to deal with virus. Firstly, VirtAV-engine notifies VirtAV-stub that it has found virus in the host page frame. Under the help of VMI tools, VirtAV-stub locates the page global directory for the current process running on the vCPU in guest VM, and walk through the guest page tables to find the entry referencing the guest physical page frame which holds the virus. Then, VirtAV-stub updates the flag bit of the entry to reclaim execute permission on the guest page frame, resumes the vCPU and lets the current process to go. The current process will be killed immediately by the guest OS due to general protection fault. In the meanwhile, VirtAV-stub also notifies VirtAV-cleaner the location of the executable corresponding to the virus. Finally, VirtAV-cleaner isolates or removes the executable accordingly.

4. Experimental Evaluation

4.1 Environment Setup

The hardware platform is composed of a virtualization host and a NFS server backed by a fiber channel disk array. The virtualization host includes dual Intel Xeon E5620 processors with 4 CPU cores (@2.40GHz) each, 24GB system memory and a 300GB local disk. The NFS server includes dual Intel Xeon E5620 processors, 16GB system memory and a 300GB local disk. The fiber channel disk array is organized in RAID5 using ten disks (Seagate ST3300657SS, SAS, 300GB). As VirtAV is built upon the KVM module in Linux kernel version 3.17.4, Qemu version 2.1.2 and ClamAV version 0.98.6, we run 64-bit CentOS release 6.5 with 3.17.4 kernel and Qemu-2.1.2 on the host hardware platform to facilitate performance comparison. Guest VMs are configured as 4 vCPUs (2 sockets with dual-core), 3GB memory, and 50GB disk space. We run 32-bit Windows 7 SP1 in guest VMs.

4.2 Functional verification

From the functional view, VirtAV has the general ability to detect both non-packed and packed viruses. Moreover, VirtAV is independent of guest operating system, and can provide antivirus protection for different types of operating systems, such as Windows and Linux. Besides viruses running in user space of the guest OS, VirtAV can also detect viruses in kernel space, such as in kernel modules. So the functional verification focuses on the validity of our approach on detecting the following types of viruses:

- Windows viruses running in user mode including non-packed viruses and packed ones.
- Windows virus running in kernel mode, which is used to test the validity on kernel-level virus.
- Linux viruses running in user mode, which are used to verify the applicability of VirtAV to different guest OSes.

We setup 2 testing guest VMs (one with Windows 7 SP1 as the guest OS, and the other with CentOS 6.5) with clean state and take snapshots for them respectively. Before performing test on a sample virus, the guest VM is rolled back to the clean snapshot to avoid influences among different viruses.

A. *User-level Windows viruses*

For non-packed viruses, we use an open virus package for Windows platform that totally contains 3546 viruses [17]. The file signatures of these viruses have been recorded in the database of ClamAV, so that they can be detected by ClamAV. There are more than 3000 viruses with text section size smaller than 4096 bytes. As for these viruses, we simply take the file signatures as the memory signatures. While for the remainder, we manually pick out memory signatures in the way described in subsection 3.2. For packed viruses, we collect 100 packed viruses from the Malfease Project dataset [32]. All the 100 packed viruses can be unpacked and detected out by ClamAV. For each of them, we first use ClamAV to unpack the viruses to recover the original programs, and then manually extract memory signatures from them. Finally, the memory signatures for the viruses are added to the *.msdb* database of VirtAV. The results show that VirtAV can find out all the 3546 non-packed and the 100 packed viruses accurately.

B. *Kernel-level Windows viruses*

We develop a simulated virus program in kernel module for Windows 7 SP1 and record the memory signature of the “virus” in the *.msdb* database. When loading the kernel module, the kernel tries to call the initialization function of the module. At this point, the memory signature is detected by VirtAV and the module is prevent to run. It indicates that VirtAV can also detect kernel-level viruses inside guest VMs.

C. *User-level Linux viruses*

We collect 14 Linux viruses[1] and extract the memory signatures manually. The way to select memory signatures from viruses in ELF format is the same to that in PE format described in subsection 3.2. The results of successfully detecting these 14 viruses reveal that VirtAV is independent of guest OS.

[1] The 14 Linux viruses include Virus.Linux.Alaeda, Virus.Linux.Diesel.962, Virus.Linux.Kagob.a, Virus.Linux.Kagob.b, Virus.Linux.Nuxbee.1403, Virus.Linux.Rike.1627, Virus.Linux.RST.a, Virus.Linux.Satyr.a, Virus.Linux.Vit.4096, Virus.Linux.Winter.341, Virus.Linux.ZipWorm, Net-Worm.Linux.Adm, Net-Worm.Linux.Cheese and Net-Worm.Linux.Mighty.

4.3 Performance Evaluation

We benchmark the performance of VirtAV on three dimensions, including 1) booting Windows guest, 2) playing video in guest VM, and 3) running synthetic workload generated by PCMark [18] test suite. We carry out the experiments in both single-VM and multi-VM environments. Furthermore, performance in 3 cases is compared and analysed:

i. VirtAV: The testing VM is protected by VirtAV, but the optimization of exempting scan on checked same pages is not applied.
ii. VirtAV-opt: The optimization is adopt for VirtAV. In the single-VM environment, performance of the testing VM might be different depending on whether there are other guest VMs with same pages currently running on the host. So we further divide it into two subcases.
    a) VirtAV-opt-cold: Before launching the testing VM, there are no other VMs with the same guest OS and applications running on the host.
    b) VirtAV-opt-warm: Before launching the testing VM, there is a VM with the same guest OS and applications running on the host. In particular, the same benchmarking applications in the VM are kept running during the tests.
iii. Native: The testing VM runs on the native implementation of Qemu/KVM hypervisor without antivirus protection by VirtAV. In this case, optimal performance can be achieved.

1) Single VM

In the single-VM environment, there is only one testing VM on the host and the host resources such as CPU, memory and I/O, are abundant.

a) Performance of booting Windows

The comparison on boot time of Windows guest VM in different environments is shown in **Table 2**. Each test runs for 10 times, and the average time is taken as the ultimate result. When the testing VM is the only guest with the specified OS image (Windows) running on the host (the VirtAV-opt-cold case), the optimization will incur a little overhead compared to VirtAV. The reason is that there is no checked same pages for binary codes which can be merged with that of the testing VM, so extra overhead is introduced by traversing the checked trees. In the VirtAV-opt-warm case, another VM with the same OS image has been launched in advance and is currently running on the host before the testing VM runs. As a consequence, many pages of the testing VM are found to be mergeable during it boots up, and the overhead on virus scanning is reduced. Overall, although it is slower with VirtAV than the Native case, the total boot time is still acceptable from user's perspective.

Table 2. Windows guest boot time (seconds)

	VirtAV	VirtAV-opt-cold	VirtAV-opt-warm	Native
Boot Time	34.8	35.6	29.7	27.5

b) Performance of playing video

We take the build-in sample video in Windows 7 for benchmarking. As a reference, the pure time for playing the video is 30 seconds with Windows Media Player (wmplayer). For the first time to play the video, VirtAV would scan virus on the page frames holding the executable of wmplayer and the libraries that it depends on. However, this may be omitted in the following invocations of wmplayer because the binary codes might be cached in the *System File Cache*.

So we compare the first and following runs of the test, and the results are shown in **Table 3**. We run 10 times for each test case and take the average time as well.

In both the VirtAV and VirtAV-opt-cold cases, virus scan on pages for binary codes of wmplayer and related libraries introduces some overhead. Furthermore, traversing the checked trees adds extra overhead in the VirtAV-opt-cold case. The optimization significantly improves the performance which is shown in the VirtAV-opt-warm case, where the time-consuming virus scan operation on pages for wmplayer and related libraries is skipped. Comparing the time of the first play and the following play, we can see that the *System File Cache* in guest OS also helps to improve the performance of VirtAV.

Table 3. Time for playing the sample video (seconds)

	VirtAV	VirtAV-opt-cold	VirtAV-opt-warm	Native
First Play	33.6	34.0	32.5	32.4
Following Play	33.0	33.2	32.0	32.0

c) PCMark tests

We take PCMark (version 8) as the synthetic benchmarking workload. PCMark is a famous tool for testing the performance of PC. The benchmark produces a score as well as detailed results for comparison with other machines. As the guest VMs in our test-bed have no physical support on GPU, some tests cannot run. So we only compare the performance results of web browsing, writing and spreadsheet. The results are shown in **Table 4**. The performance is represented as time used to complete the corresponding basic operations, so shorter is better. It is clear that the performance differences are small between the four cases which indicates VirtAV introduces little overhead on these workloads. Specially, by the optimization, the performance is very close to that under the Native case.

Table 4. PCMark tests (seconds)

Workload	VirtAV	VirtAV-opt-cold	VirtAV-opt-warm	Native
Web Browsing-JunglePin	0.400	0.410	0.390	0.390
Web Browsing-Amazonia	0.130	0.150	0.130	0.130
Writing	4.130	4.200	3.730	3.660
Spreadsheet (Libre Office Calc Conventional)	23.600	23.690	22.706	22.525

2) Multiple VMs

We create seven testing VMs from the same OS image for benchmarking concurrent performance. All the virtual disks share a common base image file which is an advanced feature supported by Qcow2. On the whole, the performance overhead on concurrent VMs is reasonable, and antivirus storm problem can be avoided in VirtAV.

a) Performance of booting Windows

We power on the 7 testing VMs at the same time, and take the time that the last one takes to complete the booting process as the overall boot time. As is shown in **Table 5**, with VirtAV, the boot time of concurrent VMs increases significantly, but still remains acceptable to user's

perspective. Moreover, the optimization significantly boosts the performance by exempting virus scan operations on checked same pages between different VMs.

Table 5. Concurrent Windows guests' boot time (seconds)

	VirtAV	VirtAV-opt	Native
Boot Time	40.2	33.6	29.3

b) Performance of playing video

We play the sample video in the 7 VMs concurrently, and take the average time as the results which are shown in **Table 6**. In the multi-VM case, the performance is stable and the overhead is low. By adopting the optimization, the performance is further improved.

Table 6. Concurrently playing the sample video in 7 VMs (seconds)

	VirtAV	VirtAV-opt	Native
First Play	34.4	33.8	33.4
Following Play	33.9	33.3	32.7

c) PCMark tests

We run PCMark tests in the 7 VMs concurrently, and the results are shown in **Table 7**. As can be seen, the penalty on the application performance remains small in the multi-VM case, and the optimization improves the performance as expected.

Table 7. Concurrent PCMark tests (seconds)

Workload	VirtAV	VirtAV-opt	Native
Web Browsing-JunglePin	0.430	0.423	0.416
Web Browsing-Amazonia	0.133	0.133	0.133
Writing	4.533	4.215	4.137
Spreadsheet (Libre Office Calc Conventional)	23.964	23.473	23.013

3) Performance of human-computer interactions

User experience is critical for desktop computer which is a typical use case of virtual machine. To evaluate the impact of VirtAV on interactive applications, we select the most commonly used softwares on Windows, including IExplorer (version 11), Microsoft Office (version 2010) Word, PowerPoint and Excel. Moreover, to quantify and measure the performance, we split the execution of these applications into two phases: launching phase and operating phase, and evaluate the two phases respectively. The launching phase starts from the time at which the user clicks on the icon to the time at which the application can accept inputs. The operating phase starts just after the launching phase ends, until to the termination of the application.

Performance evaluation of the launching phase is simple because we can take the time duration to launch the application as the indicator. The results are shown in **Fig. 9**, in which

shorter means better. As the executables can be cached in system memory, virus scan may be avoided on the following invocations of the applications. Just as seen in the figure, the time for the first launch is longer than that for the following launches. Although it is obviously slower under VirtAV than in the native case, the experience is still acceptable.

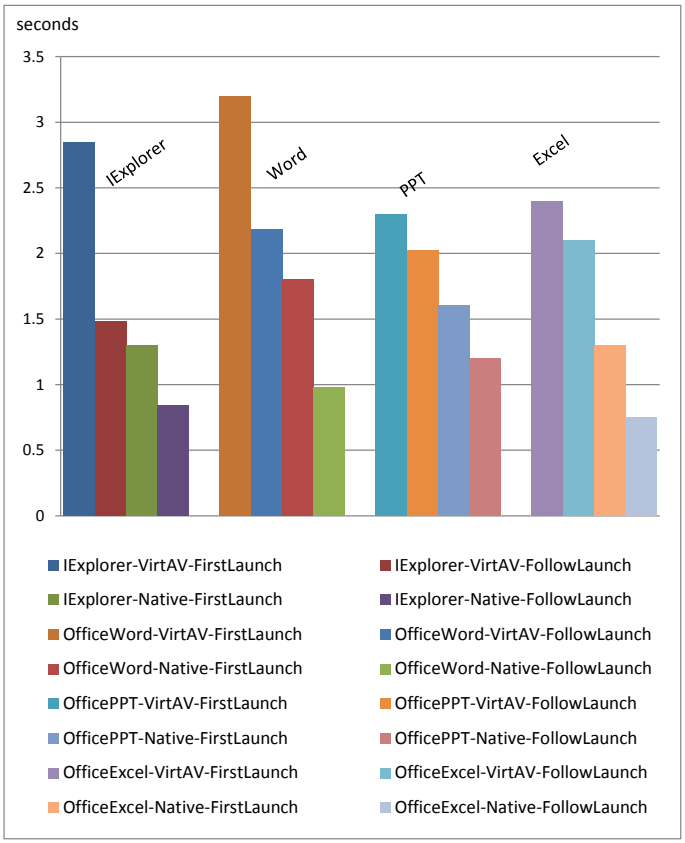

Fig. 9. Time of the launching phases

Evaluation of user experience on the operating phase is qualitative and we only focus on whether user operations are as smooth as in the native case. We perform basic operations on the applications, including typing letters in the input area, clicking on the menus, and scrolling pages. In the experiment, we found that only IExplorer can operate smoothly just as in the native case, while Microsoft Office series go through a period during which they are not responsive as in the native case. We call this period as unsmooth period. After the unsmooth period, the performance of the applications is boosted and user operations become smooth as in the native case. The lengths of the unsmooth periods are compared in **Fig. 10**.

To find out the reasons behind it, we profile memory access behaviors of the applications (MS Office Word, PPT and Excel) in the guest VM, and use SysInternalsSuite [19] tools to collect guest OS-level information. From the tracing logs in KVM, we found that the guest VM generates thousands of EPT violations on some specific page frames repeatedly during the tests. The violations are caused by execute faults and write faults alternately, which means that the vCPU updates and fetches instructions from the same page frame. Moreover, the tools (including Procmon, Rammap and Vmmap) show that these page frames are mapped as

private in the application process's address space, and the protection flags on the page frames are set to PAGE_EXECUTE_READWRITE.

Based on the above observation, we perform the following optimization to improve user experience. The basic idea is to detect repeated and alternant write-execute EPT violations on a specific page frame, scan virus on the first execute fault and skip subsequent execute faults. We use the triples <HPFN, GVFN, GPFN> in KVM to identify a mapping from guest virtual page to host page frame. We can get GVFN (Guest Virtual Page Frame Number) and GPFN (Guest Physical Page Frame Number) by reading from the VMCS structure [16]. When handling EPT violation caused by write fault or execute fault, we keep track of the corresponding GVFN and GPFN, and record the counts of write fault violation and execute fault violation in the host page descriptor, respectively. If the current GVFN and GPFN are not the same as the last couple recorded in the host page descriptor, we reset the count of write fault or execute fault to 1 and perform virus scanning on the page if it is a execute fault. Otherwise, the count of write fault or execute fault is increased by 1. If the count of execute fault is larger than 1, virus scanning is skipped.

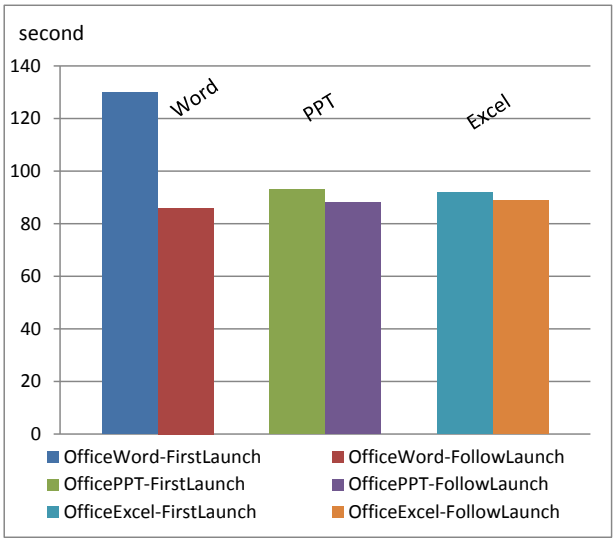

Fig. 10. Lengths of the unsmooth periods

After optimization, the unsmooth periods are eliminated, and the lengths of the launching phases are shortened slightly. **Fig. 11** a) and b) show the comparisons on the performance of the first launch and subsequent launches respectively.

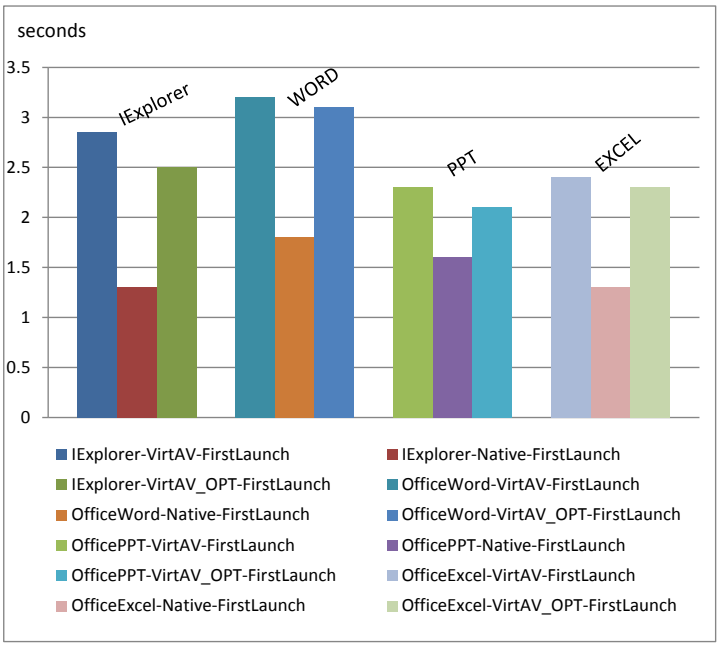

Fig. 11. a) performance comparison on the first launches

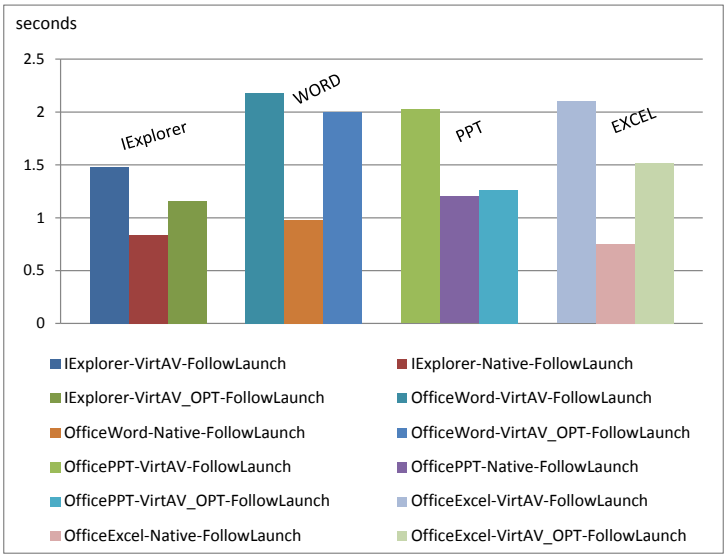

Fig. 11. b) performance comparison on the subsequent launches

5. Conclusion and Future Work

In this paper, we presented an agentless runtime antivirus solution VirtAV which relies on in-memory signature scanning. To detect and prevent virus accurately and promptly, VirtAV utilizes the memory signature to identify virus and captures the "1st-IF" event to trigger virus scanning which can be observed from the hypervisor. Event monitoring and virus detection are offloaded to the VMM, so that attacks in the guest VM cannot reach the antivirus system. Furthermore, any modification is not required on the guest OS, which can simplify

implementation of the system. Antivirus storm and rollback vulnerability issues are eliminated in VirtAV. In addition, VirtAV can provide full life cycle antivirus protection for the guest VM no matter what state it is in. Functionality experiments on 3546 non-packed viruses and 100 packed viruses show that the detection rate can achieve to 100%. Furthermore, the generality of VirtAV on detecting kernel-level virus program and Linux viruses is also verified. Performance evaluation shows that the acceptable overhead is introduced to the guest VM. For common desktop applications on Windows platform, performance degradation is negligible, and in general, the user experience on these applications is close to the native case without VirtAV.

Our ongoing and future work is to further improve the architecture and performance of VirtAV. The directions include 1) to reconstruct VirtAV-engine as a host Linux kernel module to eliminate the overhead of context switches between Qemu and KVM, 2) to support traditional file signature based virus detection so that VirtAV can be integrated with any commodity antivirus products.

Acknowledgment

We would like to thank the anonymous reviewers and editors for their valuable suggestions and comments to improve the quality of this paper.

References

[1] J. O. Kephart and W. C. Arnold, "Automatic extraction of computer virus signatures," in *Proc. of 4th Virus Bulletin Int. Conf.*, pp.179-194, 1994. Article (CrossRef Link).
[2] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp.333-340, June 1975. Article (CrossRef Link).
[3] ClamAV. http://www.clamav.net.
[4] McAfee Antivirus. http://www.mcafee.com/.
[5] Trend Micro White Paper, "Changing the game for antivirus in the virtual datacenter," September 2010. Article (CrossRef Link).
[6] Y. Xia, Y. Liu, H. Chen and B. Zang, "Defending against VM rollback attack," in *Proc. of 2012 IEEE/IFIP 42nd Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, pp.1-5, June 25-28, 2012. Article (CrossRef Link).
[7] T. Garfinkel and M. Rosenblum, "When virtual is harder than real: security challenges in virtual machine Based computing environments," in *Proc. of 10th Conf. on Hot Topics in Operating Systems*, vol. 10, pp.20-20, June 12-15, 2005. Article (CrossRef Link).
[8] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. of the 10th Annual Network and Distributed System Security Symp.*, pp.191-206, February 6-7, 2003. Article (CrossRef Link).
[9] X. Jiang, X. Wang and D. Xu, "Stealthy malware detection through VMM-based 'out-of-the-box' semantic view reconstruction," in *Proc. of 14th ACM Conf. on Computer and Communications Security*, pp.128-138, October 29-November 2, 2007. Article (CrossRef Link).
[10] P. M. Chen and B. D. Noble, "When virtual is better than real," in *Proc. of 8th Workshop on Hot Topics in Operating Systems (HOTOS'01)*, pp.133-138, May 20-2, 2001. Article (CrossRef Link).
[11] H. Xiong, Z. Liu, W. Xu and S. Jiao, "Libvmi: a library for bridging the semantic gap between guest OS and VMM," in *Proc. of 12th Int. Conf. on Computer and Information Technology (CIT)*, pp.549-556, October 27-29, 2012. Article (CrossRef Link).
[12] B. D. Payne, M. Carbone, M. Sharif and W Lee, "Lares: an architecture for secure active monitoring using virtualization," in *Proc. of 29th IEEE Symp. on Security and Privacy*, pp.233-247, May 18-22, 2008. Article (CrossRef Link).

[13] VMWare vShield Endpoint. http://www.vmware.com/products/vsphere/features/endpoint.html.
[14] Libguestfs. http://libguestfs.org/.
[15] Microsoft PE and COFF Specification, https://msdn.microsoft.com/en-us/windows/hardware/gg463119.aspx.
[16] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide Part 2." Article (CrossRef Link).
[17] atozvirus.rar. http://yun.baidu.com/wap/link?uk=2852875414&shareid=3677790463&third=0.
[18] PCMark. http://cn.futuremark.com/benchmarks/pcmark.
[19] SysInternalsSuite. https://technet.microsoft.com/en-us/sysinternals/bb842062.aspx.
[20] M. Sharif, W. Lee, W. Cui and A. Lanzi, "Secure in-VM monitoring using hardware virtualization," in *Proc. of 16th ACM Conf. on Computer and Communications Security*, pp.477-487, November 9-13, 2009. Article (CrossRef Link).
[21] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai and P. M. Chen, "ReVirt: enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp.211-224, 2002. Article (CrossRef Link).
[22] J. R. Crandall, G. Wassermann, D. A. Oliveira, Z. Su, S. F. Wu and F. T. Chong, "Temporal search: detecting hidden malware timebombs with virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 5, December 2006. Article (CrossRef Link).
[23] Y. Wang, D. Beck, X. Jiang and R. Roussev, "Automated web patrol with strider HoneyMonkeys: finding web sites that exploit browser vulnerabilities," in *Proc. of 13th Network and Distributed Systems Security Symp.*, pp.1-15, February 2-3, 2006. Article (CrossRef Link).
[24] A. Dinaburg, P. Royal, M. Sharif and W. Lee, "Ether: malware analysis via hardware virtualization extensions," in *Proc. of 15th ACM Conf. on Computer and Communications Security*, pp.51-62, October 27-31, 2008. Article (CrossRef Link).
[25] M. Andreas, K. Christopher and K. Engin, "Exploring multiple execution paths for malware analysis," in *Proc. of 28th IEEE Symp. on Security and Privacy*, pp.231-245, May 20-23, 2007. Article (CrossRef Link).
[26] G. Xiang, H. Jin, D. Zou and X. Chen, "Virtualization based security monitoring," *Journal of Software*, vol. 23, no. 8, pp.2173-2187, 2012. Article (CrossRef Link).
[27] S. Wessel and F. Stumpf, "Page-based runtime integrity protection of user and kernel Code," in *Proc. of 5th European Workshop on System Security (EuroSec'12)*, April 10, 2012.
[28] A. Arcangeli, I. Eidus and C. Wright, "Increasing memory density by using KSM," in *Proc. of Linux Symp.*, pp.19-28, July 13-17, 2009. Article (CrossRef Link).
[29] T. Brosch and M. Morgenstern, "Runtime packers: the hidden problem," in *Proc. of Black Hat USA*, 2006. Article (CrossRef Link).
[30] PE Formart. https://msdn.microsoft.com/en-us/library/ms680339(v=vs.85).aspx.
[31] P. Royal, M. Halpin, D. Dagon, R. Edmonds and W. Lee, "PolyUnpack: automating the hidden-code extraction of unpack-executing malware," in *Proc. of Computer Security Applications Conf. 2006 (ACSAC '06)*, pp.289-300, December 11-15, 2006. Article (CrossRef Link).
[32] Malfease Project. http://malfease.oarci.net.
[33] M. M. K. Al-Anezi, "Generic packing detection using several complexity analysis for accurate malware detection," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol.5, no. 1, pp.7-14, 2014. Article (CrossRef Link).
[34] K. Griffin, S. Schneider, X. Hu and T. C. Chiueh, "Automatic generation of string signatures for malware detection," in *Proc. of Recent Advances in Intrusion Detection Int. Symp. (RAID 2009)*, pp.101-120, September 23-25, 2009. Article (CrossRef Link).
[35] Y. Afek, A. Bremler-Barr and S. Landau-Feibish, "Automated signature extraction for high volume attacks," in *Proc. of 2013 ACM/IEEE Symp. on Architectures for Networking and Communications Systems (ANCS)*, pp.147-156, October 21-22, 2013. Article (CrossRef Link).
[36] Z. Li, X. F. Wang, Z. Liang and M. K. Reiter, "AGIS: towards automatic generation of infection signatures," in *Proc. of IEEE Int. Conf. on Dependable Systems and Networks with FTCS and DCC*, pp.237-246, June 24-27, 2008. Article (CrossRef Link).

[37] C. Pham, Z. Estrada, P. Cao and Z. Kalbarczyk, "Reliability and security monitoring of virtual machines using hardware architectural invariants," in *Proc. of 2014 44th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN)*, pp.13-24, June 23-26, 2014. Article (CrossRef Link).
[38] H. W. Baek, A. Srivastava and d. M. J. Van, "CloudVMI: virtual machine introspection as a cloud service," in *Proc. of IEEE Int. Conf. on Cloud Engineering*, pp.153-158, March 11-14, 2014. Article (CrossRef Link).
[39] S. Mariani, L. Fontana, F. Gritti and S. D'Alessio, "PinDemonium: a DBI-based generic unpacker for Windows executables," in *Proc. of Black Hat USA*, 2016. Article (CrossRef Link).
[40] E. Bauman, G. Ayoade and Z. Lin, "A survey on hypervisor-based monitoring: approaches, applications, and evolutions," *ACM Computing Surveys*, vol. 48, no. 1, pp.1-33, September 2015. Article (CrossRef Link).
[41] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu, "Process out-grafting: an efficient "out-of-VM" approach for fine-grained process execution monitoring," in *Proc. of ACM Conf. on Computer and Communications Security (CCS 2011)*, pp.363-374, October 17-21, 2011. Article (CrossRef Link).
[42] Y. Xia, Y. Liu and H. Chen, "Architecture support for guest-transparent VM protection from untrusted hypervisor and physical attacks," in *Proc. of IEEE Int. Symp. on High Performance Computer Architecture*, pp.246-257, February 23-27, 2013. Article (CrossRef Link).
[43] R. Wu, P. Chen, P. Liu and B. Mao, "System call redirection: a practical approach to meeting real-world virtual machine introspection needs," in *Proc. of 2014 IEEE/IFIP Int. Conf. on Dependable Systems and Networks*, pp.574-585, June 23-26, 2014. Article (CrossRef Link).
[44] S. Suneja, R. Koller, C. Isci, E. de Lara, A. Hashemi, A. Bhattacharyya and et al., "Safe inspection of live virtual machines," in *Proc. of the 13th ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments*, pp.97-111, April 8-9, 2017. Article (CrossRef Link).
[45] J. Xiao, L. Lu, H. Wang and X. Zhu, "HyperLink: virtual machine introspection and memory forensic analysis without kernel source code," in *Proc. of IEEE Int. Conf. on Autonomic Computing*, pp.127-136, July 17-22, 2016. Article (CrossRef Link).
[46] A. More and S. Tapaswi, "Virtual machine introspection: towards bridging the semantic gap," *Journal of Cloud Computing: Advances, Systems and Applications*, 3:16, October 2014. Article (CrossRef Link).
[47] Y. Liu, Y. Xia, H. Guan, B. Zang and H. Chen, "Concurrent and consistent virtual machine introspection with hardware transactional memory," in *Proc. of IEEE Int. Symp. on High Performance Computer Architecture*, pp.416-427, February 15-19, 2014. Article (CrossRef Link).
[48] L. Liu, J. Ming, Z. Wang, D. Gao and C. Jia, "Denial-of-service attacks on host-based generic unpackers," in *Proc. of Int. Conf. on Information and Communications Security (ICICS 2009)*, pp.241-253, December 14-17, 2009. Article (CrossRef Link).
[49] H. Noh, "Complexity-based packed executable classification with high accuracy," *Master Thesis, School of Engineering, Information and Communications University, Korea*, 2009. Article (CrossRef Link).
[50] A. Fischer, T. Kittel, B. Kolosnjaji, T. K. Lengyel, W. Mandarawi, H. D. Meer and et al., "CloudIDEA: a malware defense architecture for cloud data centers," in *Proc. of OTM Confederated Int. Conf. "On the Move to Meaningful Internet Systems,"* pp.594-611, October 26-30, 2015. Article (CrossRef Link).
[51] S. Biedermann and S. Katzenbeisser, "Detecting computer worms in the cloud," in *Proc. of IFIP WG 11.4 Int. Workshop (iNetSec 2011)*, pp.43-54, June 9, 2011. Article (CrossRef Link).
[52] J. Shi, Y. Yang, and C. Tang, "Hardware assisted hypervisor introspection," *SpringerPlus*, 5:647, May 2016. Article (CrossRef Link).
[53] G. Jeong, E. Choo, J. Lee and M. Bat-Erdene, "Generic unpacking using entropy analysis," in *Proc. of Int. Conf. on Malicious and Unwanted Software*, pp.98-105, October 19-20, 2010. Article (CrossRef Link).

Hongwei Tang is a Ph.D. candidate at the University of Chinese Academy of Sciences. He received his B.S. degree from Nankai University (China) in 2006 and M.S. degree from University of Chinese Academy of Sciences in 2009. Currently, he is a senior member of China Computer Federation (CCF) and an associate professor at Institute of Computing Technology of Chinese Academy of Sciences (ICT, CAS). His research interests include cloud computing, virtual machine and operating system.

Shengzhong Feng received his Ph.D. degrees in computer science from Beijing Institute of Technology in 1997. He was employed by Institute of Computing Technology, Chinese Academy of Sciences. From 2005 to 2007, he investigated gene chip algorithm design as a visiting professor in University of Toronto, Canada. Currently, he is the executive director of the China Mathematics Software Association and a committee member of High Performance Computing Association. He has published over 30 research papers, most of them are indexed by SCI/EI. His research interests include high performance computing, grid computing and bioinformatics.

Xiaofang Zhao received her Ph.D. degrees in computer science from Institute of Computing Technology, Chinese Academy of Sciences. Currently, she is the director of computer application research center in Institute of Computing Technology, Chinese Academy of Sciences. She is a senior member of China Computer Federation (CCF) and a member of Engineering and Technology Committee. Her research interests include computer architecture of next generation, security of computer system and information security.

Yan Jin received the B.S., M.S., and Ph.D. degrees in computer science from Harbin Institute of Technology, Harbin, China, in 2001, 2003, and 2008, respectively. He was a research fellow in department of Electrical and Computer Engineering, University of Nevada, Las Vegas (UNLV) from 2008 to 2011. Since 2012, he has been an associate professor in Institute of Computing Technology, Chinese Academy of Sciences. His research interests include network security, cloud computing and cloud security.