Implicit Detection of Hidden Processes with a Local-Booted Virtual Machine

Yan Wen[1], Jinjing Zhao[2], Huaimin Wang[1]
[1] *School of Computer, National University of Defense Technology, Changsha, China*
wenyan@nudt.edu.cn, whm_w@163.com
[2] *Beijing Institute of System Engineering, Beijing, China*
misszhaojinjing@sina.com.cn

Abstract

Currently stealth malware is becoming a major threat to the PC computers. Process hiding is the technique commonly used by stealth malware to evade detection by anti-malware scanners. On the defensive side, previous host-based approaches will be defeated once the privileged stealth malware controls a lower reach of the system. The virtual machine (VM) based solutions gain tamper resistance at the cost of losing the OS-level process view. Moreover, existing VM-based approaches cannot introspect the preinstalled OS which is just the protecting concern for PC users. In this paper, we present a new VM-based approach called Libra which accurately reproduces the software environment of the underlying preinstalled OS within the Libra VM and provides an OS-level semantic view of the processes. With our new local-booting technology, Libra VM just boots from the underlying host OS but not a newly installed OS image. Thus, Libra provides a way to detect the existing process-hiding stealth malware in the host OS. In addition, instead of depending on the guest information which is subvertable to the privileged guest malware, Libra adopts a unique technique to implicitly construct the Trusted View of Process List (TVPL) from within the virtualized hardware layer. Our evaluation results with real-world hiding-process rootkits, which are widely used by stealth malware, demonstrate its practicality and effectiveness.

1. Introduction

On PC platforms, with more and more users are willing to download and execute freeware/shareware, potential stealth malware accompanied with the downloaded untrusted code has become a major threat to PC users [1]. The term "stealth malware" refers to the software programs that start up with the sacrificed OS and try to hide their presence from the process enumeration utilities commonly used by malware detectors [2]. *Process hiding* is the most widely used stealth technique. According to statistics released by Microsoft's widely deployed *Malicious Software Removal Tool*, a significant fraction of the malware it encounters and removes consists of stealth components with the capability of process hiding [3]. Consequently, the ability to detect and respond to malicious hidden processes is a clear advantage in the race to prevent the computers against stealth malware.

The most feasible mechanism to detect hidden processes is so-called *cross-view* validation [2]. It operates by observing the process list from two perspectives, *untrusted view* and *trusted view*, and locating the inconsistencies between these two views. Untrusted view is often retrieved from an untrusted, high-level point. Trusted view is

obtained from within a lower layer in the system that is unlikely to have been tampered by malware. If a process exists in the trusted view but does not appear in the untrusted view, a cross-view detector can conclude that it has been hidden.

In the defense point of view, cross-view validation will fail in retrieving the "true" process list if an attacker subverts the level from which the trusted view is obtained. So, the key challenge of cross-view validation is the inevitable race that develops between attackers and defenders to control the lowest reaches of a system. Referring to the layer wherein the trusted view of processes is gained, existing cross-view validation approaches can be categorized into two classes: *host-based* (so-called *"in-the-box"*) [3] and *VM-based* (so-called "*out-of-the-box*") [4-7].

The host-based technologies provide the capability of introspecting the host OS with a native, semantic-rich view, whereas they in the meantime make themselves visible and left many chances to privileged malware to dive deeper than the detectors [8-10].

Compared to host-based approaches, VM-based approaches significantly improve the tamper-resistance of detection facilities in virtue of their location in an isolated virtualization layer. However, existing VM-based approaches either suffer from the semantic gaps between the view of the VM and the view from the inside of the OS [4, 6], or depend on the guest kernel data structures which is still subvertable to the privileged stealth malware [5, 7]. Moreover, the OSes within the VMs don't reproduce the environment of the underlying preinstalled host OSes, which are just our protecting concern on PC platforms. In other words, they only deal with the OS deployed in the VM instead of our daily used host OS.

To address these limitations, we propose a new VM-based cross-view validation approach called Libra for detecting process-hiding malware. Compared to previous VM-based approaches, Libra provides two unique advantages: host OS environment reproduction, and implicit introspection of TVPL.

Host OS Environment Reproduction. To detect the potential stealth malware existing in the preinstalled host OS at the virtualization layer, Libra introduces a novel *Local-Booted System Virtual Machine* (*Libra VM*). Unlike existing VM-based approaches, Libra VM boots just from the underlying preinstalled host OS instead of booting from a newly installed OS image, viz. Libra VM loads another instance of the host OS. As a consequence, the software environment of host OS, including the possibly existing stealth malware, is faithfully reproduced within this local-booted OS.

Implicit Introspection of TVPL. To decouple from the information subvertable to the privileged guest malware, Libra adopts a novel technique that enables Libra to implicitly discover the "true" process list from within the virtualized hardware layer. By monitoring low-level interactions between OS and the processor memory management unit (MMU), Libra proposes a unique process awareness technology which is capable of accurately determining when an OS creates processes, destroys them, or context-switches between them. With this technology, Libra can detect more process-hiding malware than existing stealth malware detectors.

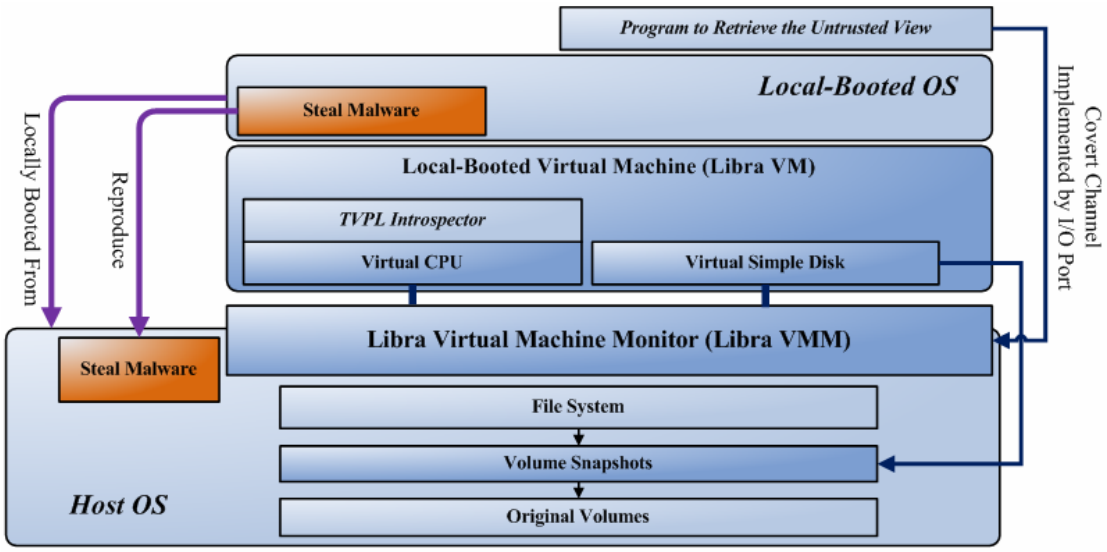

Figure 1. The Architecture of Libra.

Libra has been firstly implemented in Windows with Intel x86 processors. Our experimental results with real-world rootkits which are widely used to hide processes demonstrate Libra's unique detection capability.

This paper will discuss the architecture of Libra and focus on the implementation of implicit introspection of TVPL. The rest of the paper is organized as follows. Section 2 describes the design of Libra, followed by the implementation details in Section 3. Section 4 provides the evaluation results of the functionality of hidden process detection. Section 5 reviews previous related works. We summarize the main features of Libra in the last section.

2. Architecture of Libra

As stated in the previous section, Libra achieves two characteristics: *implicit introspection of TVPL* and *host OS environment reproduction*.

There are two classes of virtual machine monitors (VMM): Type I and Type II [11]. Each of them can serve as the virtualization layer wherein the guest OS is introspected. According to the definition of Goldberg [11], a Type I VMM just runs above a bare computer hardware platform. It tends to be implemented as a lightweight OS with the virtualization capabilities. A Type II VMM is executed as an application. The OS that manages the real computer hardware is called the "host OS". Every OS that runs in the Type II virtual machine is called a "guest OS". In a Type II VMM, the host OS provides process allocation and a standard execution environment to each guest OS.

Unlike the mainframes that are configured and managed by experienced administrators, desktop PC's are often preinstalled with a standard OS and managed by the end-user. Ignoring the difficulty of proposing a practical and seamless migration approach for PCs, it will maybe take several years to migrate all of them to the Type I VMM. It also might be unacceptable for a PC user to completely replace an existing OS

with a Type I VMM. In contrast, Type II VMM allows co-existing with the preinstalled host OS and programs.

Accordingly, taking into account that the preinstalled OS of PC platforms is the prime protecting concern for Libra, we select Type II VMM over Type I VMM to act as the introspecting layer. As illustrated in Figure 1, Libra is composed of three key components: *Libra Virtual Machine (Libra VM)*, *Virtual Simple Disk*, and *TVPL Introspector*.

Libra Virtual Machine (Libra VM). It is a novel local-booted virtual machine built on *Libra VMM* (a Type II VMM). With our local-booting technology, the *Local-Booted OS* started in *Libra VM* reproduces accurately the software environment of *Host OS*, along with the potential stealth malware. This feature makes Libra hold predominance over previous VM-based solutions with the capability of examining the preinstalled host environment at virtualization layer.

Virtual Simple Disk Based on Volume Snapshot. A key challenge to accomplish local-booting VM is reusing the existing system volume wherein *Host OS* has been preinstalled. While *Libra VM* is running, the *Local-Booted OS* is unaware of the modifications made by *Host OS* and vice versa. So they likely crash because of the content inconsistency between the file system and disk drive. Libra resolves these conflicts by dint of the *Virtual Simple Disk* based on *Volume Snapshot*. *Volume Snapshot* introduces *Copy-on-Write* mechanism to shield the modification effects of *Host OS* from *Libra VM* and vice versa. *Virtual Simple Disk* acts as the virtual storage device to combine and export the *Volume Snapshots* to *Libra VM*. Thus, *host OS environment reproduction* is achieved.

TVPL Introspector. To retrieve the "true" process list from within the lowest reaches of a system, Libra implements four introspectors within the virtualized hardware devices layer. *Libra Process Introspector* (*LPI*) is located in the *Virtual CPU*.

Assumption on trusted Libra VM. In this paper, we make an assumption that the stealth malware in host OS will not tamper the Libra program. This is practical because the ultimate target of the stealth malware is hiding its presence and extending its lifetime as long as possible. However, subverting a closed-source program often means unstable reverse-engineering and likely crashes the sacrificed program. Examining existing popular stealth malware programs, all of them try to attack the protection mechanism used by the stealth malware detectors instead of tampering these detectors to evade from detection at the risk of being exposed. Note that this assumption is consistent with that of all of the security research efforts which focuses on a compromised computer system, such as anti-virus programs, anti-rootkits software and so on.

3. Implementation of Libra

The main title (on the first page) should begin 1 3/16 inches (7 picas) from the top edge of the page, centered, and in Times New Roman 14-point, boldface type. Capitalize the first letter of nouns, pronouns, verbs, adjectives, and adverbs; do not capitalize articles, coordinate conjunctions, or prepositions (unless the title begins with such a word). Please initially capitalize only the first word in other titles, including section titles and first, second, and third-order headings (for example, "Titles and headings" — as in these guidelines). Leave two blank lines after the title.

The Libra architecture discussed in Section 2 is OS-independent. Considering Windows and Intel processors are prevalent on PC platforms, Libra has been firstly implemented in Windows with Intel x86 processors.

To accomplish the two features of Libra, we are faced with two challenges: *local-booting technology with Intel x86 processors virtualization* and *implicit introspection of TVPL*. The detailed description about how we implement the local-booted VM for Intel x86 processors has been presented in our previous work [12]. This section focuses on the implementation of *implicit introspection of TVPL*.

Intel x86 processors use a two-level, in-memory, architecturally-defined page table. The page table is organized as a tree with a single fixed-sized, commonly 4KB, memory page called the *page directory* at its root. A single address space is active per processor at any given time. The address of the page directory is therefore characteristic of a single address space.

Consequently, we use the physical address of the page directory as the process unique identifier (*PUID*). OS informs the processor's memory management unit (MMU) that a new address space should become active by writing the physical address of the page directory for the new address space into a specific control register - *CR3 register*. Within the local-booted VM, access to the CR3 register is privileged and Libra VMM will catch this event and emulate its semantic. By tracking virtual address space creation and destruction within Libra VMM, *LPI* implicitly obtains the information about the events of process creations and exits, namely the *TVPL*.

LPI maintains TVPL using a Black-Red tree. Each node indexed by PUID refers to a "true" running process in the OS. The key to maintain the TVPL is analyzing the process creation and destruction.

Process Creation. The creation of a process always refers to a new address space represented by a new *PUID*. If we observe a PUID value being writing to CR3 that has not been seen in the TVPL, we can infer that a new address space has been created. Thus, LPI will fetch the process information and insert it along with its PUID into the TVPL.

Process Destruction. Our approach to detect process destruction stems from the mechanism used by an OS to deallocate the address space. Once a process has exited, Windows (and Linux) systematically will clear the non-privileged portions of its page table pages used prior to reusing them. Privileged portions of the pages used to implement the protected kernel address space need not be cleared because they are shared between processes and map memory not accessible to user-mode programs. But the non-paged memory pages among these un-cleared privileged pages must be set un-present (a bit in a page entry in the page table).

So, to detect the address space deallocations, LPI sends a request to the memory manger of Libra VMM to remap the address of a specific non-paged kernel-mode memory page for each process in TVPL. If the remapped page is set un-present, the process should have been destructed. In this case, LPI will remove the tree node of this process from TVPL.

Picking a kernel-mode non-paged memory page for each process is an OS-dependent issue. This address of this page must satisfy three requirements: 1) being assigned as soon as a process is created; 2) being fixed among a process's lifeline; 3) being accessible among a process's lifeline; 4) being unsubvertable for privileged malware.

On Windows platforms, we select the memory page containing the *Executive Process Block (EPROCESS)* which meets all of the four requirements and consists of all the most

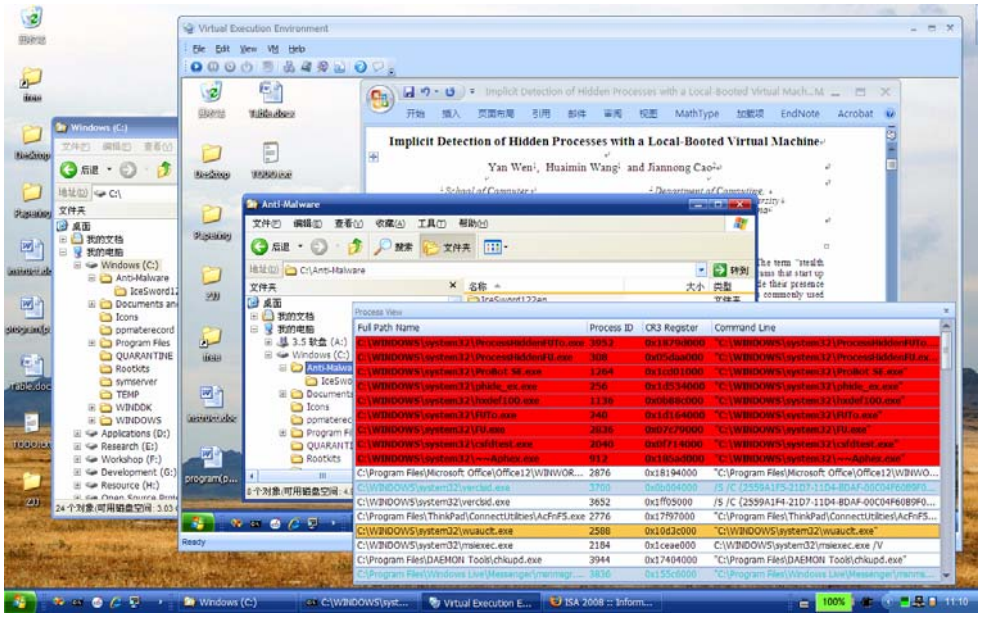

Figure 2. The Trusted Views of Processes. They are listed in the two floating tool window with the title of "*Process View*". A Running Local-Booted Libra VM is shown in the Window with a Caption of *Virtual Execution Environment*. The resolution of Local-Booted Windows is 1024x600 while the resolution of host Windows is 1280x800, so the icon arrangement within its desktop differs from that of Host Windows. As shown in this Figure, the programs of *Explorer* and *Word 2007* are running in this *Local-Booted Windows*. The topmost window is Explorer which is browsing the volume *C:*. The detected hidden processes are highlighted with red background.

important control information about a process. The address of *EPROCESS* is stored in the *Executive Thread Block (ETHREAD)* while the *ETHREAD* address of an active thread (0xFFDFF000+ 0x124) is immutable on x86 Windows platforms. Tampering the *EPROCESS* entry in *ETHREAD* structure will break up the association between a thread and its process, or even crash the Windows kernel.

Considering the peculiarity of Windows, only if the active PUID refers to ***System*** process, we are supposed to do the remapping operations because Windows will schedule **System** process to synchronize some kernel data structures after a process exits. Consequently, we significantly decrease the count of the address remapping operations.

To construct the untrusted view of processes, we call *ZwSetSystemInformation*, a Windows native API, with a process information related parameter termed of *SystemProcessesAndThreadsInformation* to enumerate all the processes. This is the most common API to list the running processes.

4. Evaluation

We have evaluated the local-booting capability of Libra on several desktop PCs with different hardware equipments[12]. In this section, to demonstrate the new implicit process introspection capability, we will evaluate our prototype with various real-world

process-hiding rootkits which are widely used by stealth malware. Each of them adopts a different representative process-hiding technique. For a desktop-oriented workload, the performance evaluation results show that Libra only suffers a slowdown over native of 0.62-6.28%, with a 2.18% average slowdown for *Libra VMM* [12].

Table 1 Completeness of Process List Introspection. The three columns refer to respective test results in three Windows versions: Windows 2000 SP4, Windows XP SP2 and Windows Server 2003 SP1.

	2000	**XP**	**2003**
Process Creation	2200	2200	2200
Address Space Creation	2200	2200	2200
Detected by Libra	2200	2200	2200
Process Exit	2200	2200	2200
Address Space Destruction	2200	2200	2200
Detected by Libra	2200	2200	2200

Hidden Process Detection. A running Libra is shown in Figure 2. This sub-section will describe the test set we used to evaluate Libra at length.

Firstly, we quantify the completeness of the implicit TVPL introspection. Our evaluation uses multiple versions of uncompromised Windows OSes which have been instrumented to report process creation, exit, and context switch. Table 1 reports the process and address space event counts gathered by the OS interfaces and by Libra during an experiment utilizing a process intensive workload. The workload creates 2200 processes, each of which runs for 25 seconds then exits. We create 10 processes per second using the *CreateProcess* API. As shown in Table 1, in this tested case, all process-related events reported by our instrumented OSes are detected by the Libra. Furthermore, Libra can precisely detect address space events, indicating that there is a one-to-one match between address spaces and processes for these OSes.

We then evaluate Libra with five popular process-hiding rootkits. *Aphex* [13] modifies the Import Address Table entry for the *ZwQuerySystemInformation* API to intercept the process list queries . *Hacker Defender* [14] also hijacks the queries with another technique: tempering the first few machine codes of *NtQuerySystemInformation* with a "jmp" instruction. *FU* [15] hides a process by using so-called *Direct Kernel Object Manipulation* (*DKOM*) technique to remove its corresponding entry from the *Active Process List* (a kernel data structure in Windows). *FUTo* [8] (an improved version of the *FU* rootkit) has the added ability to manipulate the *PspCidTable* without using any function calls. *PE386* proposed another powerful

Table 2. Experimental Results for Libra Hidden Processes Detection. *D* refers to the tool has detect the hidden process successfully while *F* denotes failure of detecting.

	Aphex	*Hacker Defender*	*FU*	FUTo	*phide_ex*
Blacklight	D	D	D	**F**	**F**
DarkSpy	D	D	D	**D**	**F**
IceSword	D	D	D	**F**	**F**
RkUnhooker	D	D	D	**D**	**F**
Bitdefender Antirootkit	D	D	D	**D**	**F**
UnHackMe	D	D	D	**F**	**F**
GMER	D	D	D	**D**	**F**
KProcCheck	D	D	D	**D**	**F**
Process Hunter	D	D	D	**F**	**F**
TaskInfo	D	D	D	**F**	**F**
Libra	D	D	D	**D**	**D**

DKOM-based process-hiding rootkit called *phide_ex* [16] which have bypassed existing hidden process detectors.

We compare Libra to ten anti-rootkit programs which are recommended by Anti Rootkit Group. As summarized by Table 2, the first two rootkits, *Aphex*, *Hacker Defender*, can be found out by using the *Active Process List* as the truth. *FU* can be detected by all the above anti-rootkit programs while *Icesword* and *F-Secure Blacklight* fail to find the FUTo-hiding processes. Libra can detect all the processes hidden by the above five rootkits (shown in Figure 2). Evaluation results show that Libra is the only detector which can identify the hidden process of *phide_ex* (*phide_ex.exe*).

5. Related Work

Cross-view validation for hiding resource detection has been proposed and variously implemented in user-mode applications, within the OS kernel [2], inside a VMM [7]. The key difference of cross-view validation is the mechanism used to obtain the trusted view.

The VM introspection methodology is pioneered by the Livewire system [7]. VM introspection in Livewire is capable of examining low-level VM states from outside the VM. Livewire uses the explicit OS debugging information like the memory addresses of variables and the layout and semantics of compound structures to locate and interpret private kernel data types at runtime. VMwatcher [5] is another VM-based guest view introspector. It still highly depends upon the subvertable guest memory contents. As mentioned in literature [5], without the knowledge of the subverted scheduler, VMwatcher is not able to accurately identify all running processes.

VM-level services based on explicit implementation information are effective, but there are drawbacks. One drawback is that they may be just as susceptible to evasion by an attacker that has subverted the guest OS as if they were located within the guest itself. In spite of their location at the VM-layer, these services depend on guest-level information which is still open to simple guest-level manipulation. Instead, the introspectors of Libra are independent of the guest-level information.

Besides, desktop PCs, the concern of our approach, are often preinstalled with a standard OS. However, existing VM-based approaches only deal with the newly deployed OS within the VM. They have less contribution on detecting the existing stealth malware in the existing OSes. In contrast, Libra accurately reproduced the software environment of the host OS within Libra VM and provide a way to introspect it from within a virtualization layer.

6. Conclusion

Stealth malware are a current and alarming security issue. In this paper, we have presented a new VM-based approach called Libra to detect hidden processes implicitly. Like previous VM-based security services, Libra is resilient to kernel-mode guest malware attack by virtue of its location within a VMM layer.

Unlike prior VMM-layer process hiding detectors, by introducing a novel local-booting technology, Libra achieves the unique capability of introspecting the preinstalled host OS at virtualization layer. Moreover, Libra adopts a novel implicit process introspection technology which is decoupled from the subvertable guest OS

information. Our evaluation results with real-world stealth rootkits prove Libra's detection capability.

Acknowledgements

This research is supported by National Basic Research Program of China (Grant No. 2005CB321801), National Natural Science Foundation of China (Grant No. 60673169 and 60621003), National Science Fund for Outstanding Youths under Grant No. 60625203, and National Natural Science Foundation of China.

References

[1] "Zombie PCs: Silent, Growing Threat. PC World," http://www.pcworld.com/news/article/0,aid,116841,00.asp, July 2004.
[2] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski, "Detecting Stealth Software with Strider GhostBuster," in *Proceedings of 35th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'05)*, 2005, pp. 368–377.
[3] R. Naraine, "Microsoft: Stealth Rootkits Are Bombarding XP SP2 Boxes," http://www.eweek.com/article2/0,1895,1896605,00.asp, December 2005.
[4] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, "Detecting Past and Present Intrusions through Vulnerability-Specific Predicates," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, Brighton, United Kingdom, 2005, pp. 91 - 104.
[5] X. Jiang, X. Wang, and D. Xu, "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)*, 2007.
[6] 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," in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002, pp. 211-224.
[7] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proceedings of Network and Distributed System Security Symposium (NDSS'03)*, 2003.
[8] P. Silberman and C.H.A.O.S, "FUTo: Bypassing Blacklight and IceSword," https://www.rootkit.com/newsread.php?newsid=433, 2007.
[9] "Effective file hiding : Bypassing Raw File System I/O Rootkit Detector," http://www.rootkit.com/newsread.php?newsid=690.
[10] "Bypassing Klister 0.4 with No Hooks or Running a Controlled Thread Scheduler," http://hi-tech.nsys.by/33/.
[11] R. P. Goldberg, "Architectural Principles for Virtual Computer Systems, Ph.D. Thesis," Cambridge, MA: Harvard University, 1972.
[12] Y. Wen, J. Zhao, and H. Wang, "A Novel Approach for Untrusted Code Execution," in *Proceedings of 9th International Conference on Information and Communications Security (ICICS'07)*, 2007, pp. 398-411.
[13] "Aphex: AFX Windows Rootkit 2003," http://www.iamaphex.cjb.net.
[14] "Hacker Defender," http://hxdef.org/.
[15] fuzen_op, "FU Rootkit," http://www.rootkit.com/project.php?id=12.
[16] PE386, "phide_ex -untimate process hiding example," http://forum.sysinternals.com/printer_friendly_posts.asp?TID=8527.

Yan Wen

Born in 1979, Ph. D. candidate. His major research interests include virtualization technology, and information security.

Jinjing Zhao

Born in 1981, Ph. D., assistant professor. Her major research interests include computer networks, and information security.

Huaimin Wang

Born in 1962, professor, Ph. D. supervisor. His major research interests include distributed computing, computer networks, and information security.