SPAD: Software Protection Through Anti-Debugging Using Hardware-Assisted Virtualization*

ZHENGWEI QI, BINGYU LI, QIAN LIN, MIAO YU,
MINGYUAN XIA AND HAIBING GUAN
Shanghai Key Laboratory of Scalable Computing and Systems
Shanghai Jiao Tong University
Shanghai, 200240 P.R. China

Debugging usually facilitates the dynamic analysis of runtime application for software development. Yet it can also be a threat to system security when adopted by malicious attackers, and hence anti-debugging becomes valuable. The major challenges of software-only anti-debugging are the compromised strategy and lack of self-protection. This paper proposes software protection through anti-debugging (SPAD), a technique that imperceptibly monitors the behavior of debuggers. Leveraging hardware virtualization, SPAD detects debugging behavior by intercepting debug events on a higher privilege level than the conventional kernel space. Our experiment shows that SPAD can effectively prohibit the debugging behavior from 8 popular debuggers while the overhead incurred is 1.14%.

Keywords: software protection, anti-debugging, hardware-assisted virtualization, self-protection, system security

1. INTRODUCTION

The requirement for software protection has gained general attention in the digital world. A great variety of copy-protection strategies have been developed to prevent cracking, tracing and reverse engineering. The majority of these mechanisms provide a reasonable level of security against static-only analysis. Nevertheless, most of them are vulnerable to dynamic or hybrid static-dynamic attacks which are commonly used by hackers [1]. Although debugging is usually employed in the software development to help find software bugs or deficiencies, it also facilitates hackers to reverse commercial software or steal private information [2].

A detailed description of the mechanism to protect common password from being cracked is presented in the book of "*Hacker Debugging Uncovered*" by Kris Kaspersky [3]. Hackers could utilize a generic debugger to set a breakpoint at the password input function, scan the specific buffer and wait for the debugger window to reveal the real password. Similar scenarios may be applied to other private and sensitive account data acquirement. Some software may be packed and protected by packers to fight against reverse engineering, but most packers suffer from the compatibility problem, resulting in erroneous effects on the protected applications.

Contemporary studies on anti-reverse-engineering and anti-dynamic-analysis emphasize on obfuscation, virtual machine detection [4], anti-disassembly and tamper-proofing,

Received May 31, 2011; accepted March 31, 2012.
Communicated by Jiman Hong, Junyoung Heo and Tei-Wei Kuo.
* This work was supported by the National Natural Science Foundation of China (Grant No. 60873209, 60970107, 60970108, 61073151), the Key Program for Basic Research of Shanghai (Grant No. 10511500100, 10DZ15-00200, 11530700500), IBM SUR Funding and IBM Research-China JP Funding.

yet seldom consider anti-debugging [2]. Anti-debugging mechanism for the whole system scope makes a worthwhile contribution to the real world software security.

Current anti-debugging methods can be divided into two categories [2]. One is to check hardware or software debug structure for the presence of debugger state such as breakpoint setting. The other is to detect human behavior such as the unexpected pause within execution. An unambiguous rule to distinguish debugger's behavior is recognized as highly significant for the effectiveness and accuracy of the anti-debugging approach. Furthermore, one drawback of the current anti-debugging mechanisms is that most of them fail to conceal themselves from other processes. If malware or debuggers perceive that they are monitored by the anti-debugging mechanism, they would attempt to circumvent such monitoring or even strike back. Privilege promotion supported by hardware architecture can enable the malicious code to access the whole system, which increases the chance of operating system (OS) being exploited [5].

The growing popularity of hardware-assisted virtualization motivates our new solution for software protection through anti-debugging [6]. Virtual machine can be an effective environment to prevent software from attacks and malicious behavior [7]. Unlike conventional methods relying on kernel space to construct anti-debugging architecture, we leverage hardware-assisted virtualization to occupy a higher privilege level so that not a single debugging behavior can escape from our protection shield.

In this paper, we make the following contributions.

- We desert the traditional anti-debugging mechanism with the limitation of kernel privilege and establish a novel anti-debugging model based on hardware virtualization to protect software. This model offers system range protection and requires no code modification to the existing OS and target applications. A lightweight and extensible prototype termed SPAD is actualized to perform the anti-debugging strategy transparently. As the experiments reveal, our approach has successfully prohibited most debugging behaviors played by the canonical debuggers.
- We present and implement a self-protection mechanism for SPAD to conceal itself and enhance its isolation. Taking advantage of hardware-assisted virtualization and memory remapping strategy, SPAD could stay in the hypervisor-aware private memory region and withdraw the view of guest OS kernel to avoid the potential external attacks.
- We classify the prevailing anti-debugging methods and compare them with our SPAD prototype. The comparison report indicates that SPAD owns the best anti-debugging capability among them, while the total induced overhead is only 0.50% in average in terms of real world application tests. Meanwhile, SPAD also shows a good compatibility with current protectors.

The rest of the paper is organized as following. Section 2 presents the background knowledge such as debugger classification and mechanism. Section 3 illustrates the design and implementation details of our anti-debugging solution. Section 4 evaluates SPAD through experiments. We conclude the paper in section 5.

2. BACKGROUND

To construct anti-debugging architecture, it is crucial to first understand that a de-

bugger is not a solitary and independent system [2]. In this section, we describe several typical debugging models to generalize the features and requirements of debugging, which enlighten our anti-debugging approach.

2.1 Debugger Classification

Debugging facilitates the dynamic analysis of computer program running on a physical or virtual processor. To be effective, dynamic program analysis requires the target to be executed with sufficient test inputs to produce meaningful behavior [8]. Debugging medium could be diverse. We classify all prevailing debuggers into four categories: *user-mode debugger*, *kernel debugger*, *system-level debugger* and *emulator debugger*. This classification is based on the difference of debugging mechanisms, as shown in Fig. 1. As the computer system has evolved to such a complex architecture, debugger developers make use of every accessible resources, both hardware and software, to implement their effective debugging mechanism.

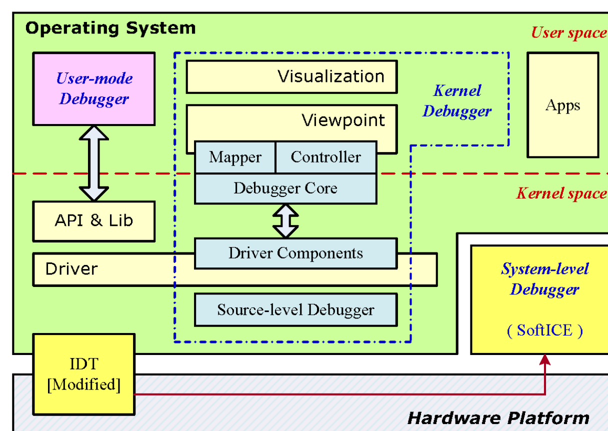


Fig. 1. Debugging model of user-mode debugger, kernel debugger and system-level debugger; The classification is based on the diversity of debugging mechanism.

2.1.1 User-mode debugger

User-mode Debugger deploys the application programming interface (API) to play debugging tricks. On starting, the user-mode debugger attaches itself to the target process or constructs debug environment by forking the debuggee process from inside. In order to trace the target, the debugger uses some wrapped APIs to set breakpoints in the target process. As long as the process hits the breakpoint, a breakpoint exception will be triggered and suspend the target process. Then it packages the relevant context and transfers the control to the debugger for post-processing. Afterwards, with respect to user's determination, the debugger will guide the kernel to deal with the target process such as stepping, resuming and canceling. The representative user-mode debuggers include OllyDbg with its plug-ins and OllyICE.

2.1.2 Kernel debugger

Kernel Debugger possesses a more privileged level than the user-mode debugger to achieve superior debugging ability. Residing in the kernel space and holding the highest execution permission, the kernel debugger can exploit kernel resources to rule the debugging procedure. Most of debugging functions and libraries exist as driver components, supplying the debugger core with common interfaces such as execution control (starting, stopping, resetting, and stepping), access to symbol data (memory addresses of methods) [8] and memory manipulation (scanning and writing). *WinDbg* is one of the typical members in this class.

2.1.3 System-level debugger

System-level Debugger is designed to run underneath OS so that OS is unaware of its presence. For the OS independency, system-level debugger exhibits a strong ability to suspend all operations in the target OS. The notable delegate is *SoftICE*. SoftICE replaces the original interruption disposal routines with its own version to obtain system control. During the installation of SoftICE, it modifies the *Interrupt Descriptor Table* (IDT) and redirects some interrupt handlers, such as INT3 (breakpoint exception) and INT31 (keyboard controller interruption), so that it can be notified immediately about debugging relevant events. Besides, *Syser* is another newcomer in this class.

2.1.4 Emulator debugger

Software-based CPU emulator deploys simulative techniques to build up emulative execution container. The target applications or even the guest OSes are present in such container and run normally. The containers are actually some memory blocks and provide “hardware” interfaces via simulation. As one of the emulator components, the internal debugger is convenient to debug programs within the container because it owns an even higher privilege level than the guest OS kernel does. *Bochs* and *QEMU* equipped with built-in debugging facility are the representative ones in this category.

2.2 Current Anti-Debugging Methods

Table 1 summarizes thirteen anti-debugging methods that are most commonly used to against debuggers on Windows. They are grouped into three categories according to their mechanisms. As the specialty of emulator debugger, most anti-debugging methods did not take it into account, so did SPAD. Using completely software emulative environment to debug the run-time behavior of applications or ever OS is more likely to be adopted in the software development rather than the real world application deployment. Therefore, it is convincible to make the assumption that designing an anti-debugging strategy towards emulator debugger is orthogonal to the goal of this paper.

Table 1. Thirteen existing software-only anti-debugging methods.

Method	Description
Invoke *IsDebuggerPresent*()	Windows supplies several documented APIs for the debugger detection which can be effective to the conventional debuggers. Besides, feedback from some undocumented interfaces and direct intervention into debugging procedure can also imply the existence of debugger. Specifically, attempt to unload the resource of debugger will trigger certain exception, which can be employed to detect the debugging behavior.
Check remote debugger	
Test *SeDebugPrivilege*	
Check parent process	
Scan debugger window	
Detect *CloseHandle*() calling	
LastError of *OutputDebugString*()	
Check *BeingDebug* flag in PEB	Some specific and characteristic data structures in the Windows kernel could be applied to detect the debugging behavior. When a process is being debugged, certain structures are modified for conveying messages to the debugger.
Check debug port	
Check debug object handle	
Detect *INT*3 exception	When attached to the target process, debugger will overtake all its exceptions. Thus, a process can mislead the debugger by crafting a well-designed exception trap. These designs are extended and enhanced in modern packers such as Armadillo.
Detect single-step exception	
Detect *INT*2*D* exception	

3. DESIGN AND IMPLEMENTATION

In this section, we focus on SPAD's design and implementation in details, involving architecture and design principles. Several core components with their approaches are analyzed thoroughly, manifesting the composition of SPAD. These modules include *protection filter*, *anti-debugging module*, *self-protection module*, *etc*.

3.1 SPAD Design Overview

SPAD is designed for the goal of protecting unmodified software within unmodified OS, and it has proven to be challenging. Leveraging hardware-assisted virtualization, SPAD is entitled to stay in a higher privilege level than OS. Hardware-assisted virtualization extends the conventional privilege hierarchy to *root mode* and *non-root mode* for the virtual machine monitor (VMM, or hypervisor) and the guest OS, respectively [9]. SPAD employs the post-booting method to deploy the anti-debugging hypervisor, *i.e.*, it is loaded as a driver of the Windows and launched after the OS booting. Once SPAD is built up and becomes independent of the host system, it conversely places the OS into guest position. Although we assume the clean booting in our current prototype, supporting the SPAD pre-booting requires additional engineering effort [10] and is orthogonal to the current design of our framework with a focus on anti-debugging effectiveness and efficiency. Since the interception of debugging is accomplished in *root mode*, the guest OS could not detect the procedure. Thus the transparency to the guest OS is guaranteed and no modification is needed to the guest application.

We tailored *BluePill* as the base of SPAD implementation. The thin hypervisor with hardware virtualization support only monitors the sensitive behavior instead of managing the whole guest OS. The guest OS can directly access the I/O devices and memory so

that a high overhead could be eliminated [11]. Owing to the implementation of a thin hypervisor, the reliability is also attained for less complexity.

Fig. 2 illustrates SPAD's architecture. There is no extra-component running in the guest OS. And there are mainly three modules in the hypervisor:

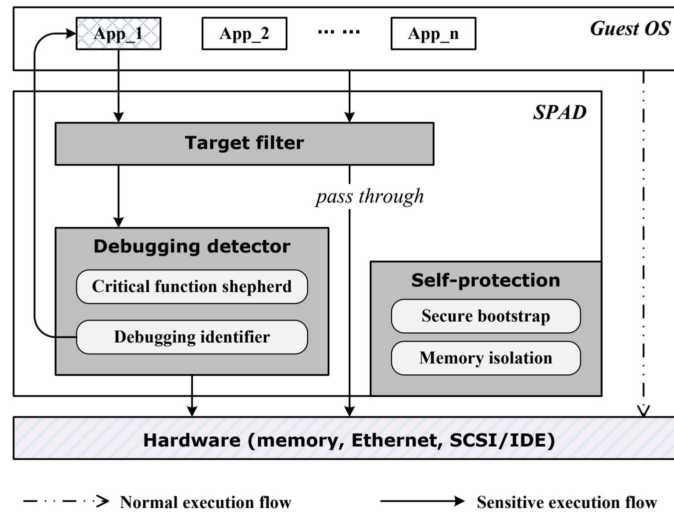


Fig. 2. SPAD architecture: All the sensitive behaviors will be trapped into the hypervisor. The protection filter identifies the target application (*e.g.*, App 1) and lets normal applications to pass through. If SPAD detects the sensitive debugging behavior with the target, the anti-debugging component will take effect and prohibit such behavior. Self-protection module entitles SPAD to conceal itself and withstand external attacks to the hypervisor.

Protection Filter SPAD manages the debugger detection through the protection filter, which is based on the process ID (PID). Certain PIDs are passed to SPAD and recorded in a target list. When a sensitive behavior occurs, SPAD takes over the control and obtains the PID of current process. If this PID matches a certain item of the target list, SPAD will enable its anti-debugging module. The identification procedure proceeds over all other SPAD modules to get rid of the influences to normal debugging.

Anti-debugging Module This module is responsible for verifying the debugging behavior. Once the debugging exceptions are identified by the protection filter, the anti-debugging component will take effect and prohibit such behavior.

Self-protection Module Since SPAD is installed dynamically, it is essential to actualize a complete transparency for SPAD. This module takes a Memory-hiding strategy to cover SPAD, which runs as a feature of SPAD hypervisor. And we will discuss this strategy in details in the next section of this paper.

3.2 Anti-Debugging Module

The design of anti-debugging module includes our anti-debugging approach and im-

plementation, sensitive behavior interception and critical function monitor.

3.2.1 Anti-debugging approach

The main challenge for anti-debugging is to verify the debugging behavior. For example, a breakpoint exception in a process may be induced by either the process itself or a debugger that injects INT3 instruction to the process. Normally, process switch will not happen if the exception is a self-raised one. But when Windows debugging mechanisms are present, the exception occurring in the target process will cause a process switching to the debugger process. Therefore, as shown in Fig. 3, we design the accurate verification of debugging behavior based on such distinctive situation.

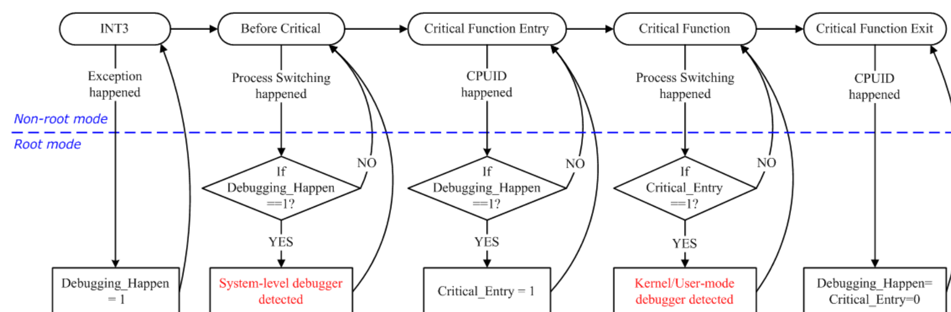


Fig. 3. Anti-debugging approach: Hypervisor monitors the sensitive events and causes mode switch when the certain events happen. These events are handled in the root mode that keeps complete transparency to the guest OS.

3.2.2 Sensitive behavior interception

By configuring VMM, we can intercept the sensitive behaviors including INT3, INT1 exceptions, CPUID instruction and CR3 register refreshing. INT1 and INT3 exceptions would cause the execution flow to exit from virtual machine and fall into VMM (named VMexit for short) depending on the exception bitmap, which is a 32-bit field that contains one bit per exception. For example, when an exception occurs, if the corresponding bit is set, a VMexit will occur; otherwise, the exception is delivered normally through the guest IDT. Thus, we configure the intercepting vector to catch every INT1 and INT3.

3.2.3 Critical function monitor

Figs. 4 and 5 demonstrate how SPAD monitors the kernel's critical function, *e.g. KiDispatchException*(), by hooking its entering and exit, respectively. This is done by replacing the function entry with a trampoline, which executes CPUID instruction before the original entry. Besides, the trampoline also hooks the function's exit by modifying caller's return address in the stack. Although SPAD's modification to kernel integrity may reveal its existence and incur possible attack, formal transparent integrity verification as SecVisor [12] can be employed for improvement.

```
// Hook the KiDispatchException once
TraceKiDispatchException {
    if (!hooked) {
        // Preserve the original entry for backtracking
        PrepareProxy(ProxyKiDispatchException, KiDispatchExceptionEntry);
        // Enter the critical section for kernel modification
        Lock KernelModification();
        // Modify the entry to redirect to the trampoline
        DecorateEntry(KiDispatchExceptionEntry);
        UnlockKernelModification();
        Hooked = true;
    }
}

// The trampoline
KiDispatchExceptionEntryMonitor {
    // Save and change the return address
    OrigRetAddr = ChangeRetAddr(KiDispatchExceptionExitMonitor);
    // Notify the hypervisor of the behavior
    Hypercall(TRACE_ENTER);
    // Jump to the preserved entry and never return
```

Fig. 4. Pseudo code for SPAD tracing *KiDispatchExecution*() entering.

```
KiDispatchExceptionExitMonitor {
    // Notify hypervisor of the behavior
    Hypercall (TRACE_EXIT);
    // Jump to the original return address saved
    // when entering KiDispatchExceptionEntryMonitor
    Jump(OrigRetAddr);
}
```

Fig. 5. Pseudo code for SPAD tracing *KiDispatchExecution*() exiting.

```
InterruptMonitor {
    InterruptHappen = true;
}

KiDispatchExceptionEntryMonitor {
    EnterKiDispatchException = true;
}

DebuggerMoniter {
    if (InterruptHappen && EnterKiDispatchException)
        Output("Kernel/User-mode_Dbg_Detected");
    else if (InterruptHappen)
        Output("System-level_Dbg_Detected");
}

KiDispatchExceptionExitMonitor {
    EnterKiDispatchException = false;
```

Fig. 6. Pseudo code for anti-debugging module.

3.2.4 Anti-debugging implementation

The prototype of SPAD is implemented on Windows. By analyzing the debugging mechanism on Windows, we find it possible to detect the existence of both user mode and kernel debugger by checking whether process switch to debugger happens in *KiDispatchException*() function. Fig. 6 demonstrates the procedures of detecting debuggers.

3.3 Self-Protection

Aside from the effectiveness of anti-debugging accomplishment, the self-protection of SPAD is also of great importance; otherwise, a malicious attacker could detect and remove SPAD easily to disable the anti-debugging mechanism, as shown in Fig. 7.

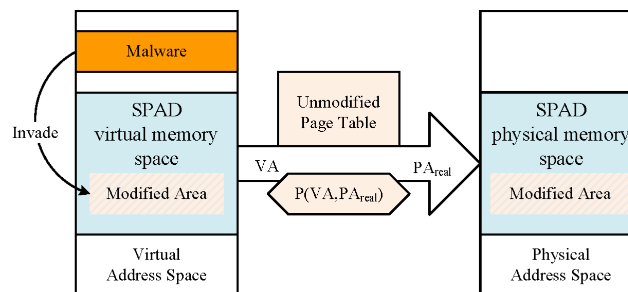


Fig. 7. SPAD without memory-hiding strategy: A malware can invade SPAD's memory space and modify the real physical space by accessing the kernel page table.

With current computer architecture, a typical operating system needs to build and manage process page tables for address translation. When the hypervisor is created and loaded dynamically, the guest OS will build an address mapping for it. It means that SPAD's memory mapping is recorded in the kernel page table, as shown in Fig. 8. For loading all data and code of the hypervisor by the guest OS, a mapping from the hypervisor's virtual address to real physical address is created. Hence, the whole hypervisor is so vulnerable that it can be easily modified and unloaded in the face of a malicious kernel. If a malware or rootkit attempts to invade SPAD's memory space, it just tampers with SPAD's virtual memory space to deprive the hypervisor of the ability to provide software protection. Furthermore, any SPAD's memory operation can be detected by guest OS, *i.e.*, SPAD owns no transparency and the protection strategy introduced by SPAD becomes vain efforts.

Therefore, the hypervisor should attach great importance to maintaining the transparency to the guest applications and guest OS. SPAD takes advantage of hardware virtualization and memory remapping techniques to conceal itself so that the threat of external attack could be eliminated. In order to implement self-protection, we modify the corresponding address mapping of guest OS's page table. Fig. 8 depicts the steps of realizing this strategy, involving copying a new private page table for SPAD, creating a pseudo memory space and modifying the mapping of kernel page table.

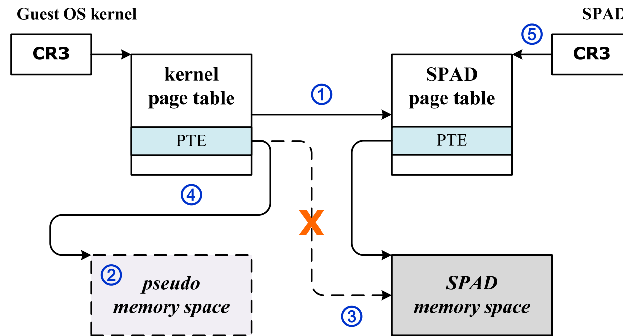


Fig. 8. Construct self-protection.

Step 1: Clone the page table of guest OS kernel as its private property.
Step 2: Reserve a spare physical memory block. Note that this pseudo memory space is allocated by the guest OS, so the kernel page table will store its mapping.
Step 3: Wipe out the page table entries (PTE) of SPAD in the kernel page table. As a result, the OS kernel cannot manage this memory space.
Step 4: Fill the erased PTE in the kernel page table with the entry of pseudo memory space created in step 2.
Step 5: Record the entry of SPAD's private memory space in VMM.

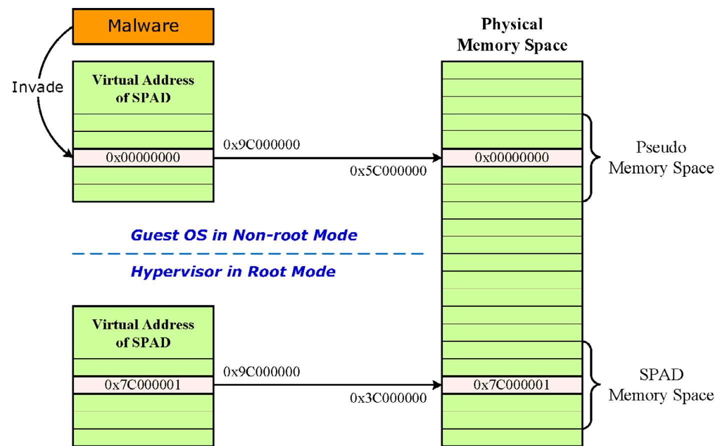


Fig. 9. SPAD with memory-hiding strategy: A malware invades SPAD's virtual memory space and modifies it by accessing the modified page table. But the real physical memory space of SPAD is not modified because there is no rule to access SPAD's physical memory space.

Consequently, when the control flow is trapped into the root mode, VMM can use the private page table for its memory space access. Furthermore, the modification to the kernel page table is done in the root mode, which shields the operations from being detected by the guest OS. Fig. 9 demonstrates a case example of SPAD leveraging self-protection strategy. Malware own no ability to invade into SPAD's memory space even if

if it owns the privilege of kernel space.

Since the effectiveness of SPAD relies on the robustness of VMM in some degree, it is essential to ensure that the hypervisor can withstand the VMM level attacks. Benefiting from the *Newbluepill* project, we introduce the *Blue Chicken* strategy to construct the anti-hypervisor-detection (AHD) module which could "launch the alarm" at any attempt to attack the hypervisor. With the help of AHD, SPAD could block the manipulation of some anti-hypervisor tools such as *RedPill*.

4. EVALUATION

In this section, we evaluate SPAD from the view of its effectiveness and performance. We check whether traditional anti-debugging methods and SPAD work properly when facing the three types of debuggers. The evaluation on the size expansion and time inflation of existing software protection tools and SPAD are also examined.

4.1 Anti-Debugging Capability

We choose thirteen traditional anti-debugging methods to compare with SPAD against generic debuggers. As mentioned in section 2, we select representative debuggers of three categories. Table 2 shows the effectiveness of selected anti-debugging approaches when facing different kinds of debuggers. For user-mode debuggers, certain hiding plugins could facilitate debuggers to conceal themselves. However, since usermode debuggers rely on APIs to communicate with system debugging server, they are highly possible to be detected by kernel modules. The result reveals that OllyDbg and OllyICE fail in more than half of the tests without hiding plugins. As for kernel debugger, WinDbg features no anti anti-debugging techniques and fails at all anti-debugging methods. System-level debuggers pervade the operating system and attain better stealth property, whereas they leave more or less debugging interfaces in system and can be compromised once exploited. For example, SoftICE performs well in all general purpose anti-debugging methods. However, more than two dozens of tricks have been found to specifically detect and attack SoftICE.

In comparison with software-only anti-debugging methods, SPAD is more effective because it holds an even higher privilege level than the system-level debugger, and intercepts the sensitive system behavior instead of interacting with system components. Thus it can silently detect the debuggers while remain risk-free from being sensed by interfaces exposed to the system. Table 2 illustrates that SPAD can detect all these debuggers. This also proves that our debugging model analysis in section 2 is reasonable and meaningful since most of the real-world debugging tools just are just designed with the mechanism we mentioned.

4.2 Compatibility

The compatibility of SPAD with the existing software protection tools is tested by checking out the running result of packed Bzip2 and Minesweeper program. Six well-

Table 2. The Anti-debugging Ability Comparison against Famous Debuggers: Thirteen general anti-debugging mechanisms are selected to compare their effectiveness with SPAD. "√" denotes that debugger can be detected by the anti-debugging method; "×" denotes that anti-debugging method fails to detect debugger's behavior.

Debugger / Anti-debugging	User-mode					Kernel	System-level	
	Phantom	OllyDbg	StrongOD	HideOD	Ollyice	WinDbg	Syser	SoftICE
SPAD	√	√	√	√	√	√	√	√
Invoke *IsDebuggerPresent*()	×	√	×	×	√	√	×	×
Check remote debugger	×	√	×	×	√	√	×	×
Test *SeDebugPrivilege*	×	×	×	×	×	√	×	×
Check parent process	×	×	×	√	×	√	√	×
Scan debugger window	×	×	×	√	√	√	×	×
Detect *CloseHandle*()	×	×	×	×	√	√	×	×
LastError of *OutputDebugString*()	√	√	×	√	√	√	×	×
Check *BeingDebug* flag in PEB	×	√	×	×	√	√	×	×
Check debug port	×	√	×	×	√	√	×	×
Check debug object handle	√	√	×	√	√	√	×	×
Detect *INT*3 exception	×	×	×	×	×	√	×	×
Detect single-step exception	×	×	×	×	×	√	×	×
Detect *INT*2*D* exception	√	√	×	√	√	√	×	×

Table 3. The compatible and performance of SPAD: SPAD is compatible with existing software protection tools and mechanisms and has an apparent performance advantage over them. "*" denotes that run time fails to be measured due to the execution interception.

Protector	Minesweeper Test			Bzip2 Test			
	SPAD Compatibility	File size (Byte)	Size expansion	SPAD Compatibility	Run time (ms)	Overhead	Delta Overhead with SPAD
none	√	119,808	–	√	2,574.06	–	–
PECompact	√	85,504	-28.63%	√	2,608.30	1.33%	0.19%
PEArmor	√	204,800	70.94%	√	2,689.73	4.49%	3.36%
Armadillo	√	790,528	559.83%	√	2,823.30	9.68%	8.55%
ASProtect	√	341,504	185.04%	√	2,635.34	2.38%	1.24%
Themida	√	2,193,920	1731.20%	√	2,933.02	13.95%	12.81%
VMProtector	√	137,728	14.96%	√	*	*	*
SPAD	–	119,808	0%	–	2,603.34	1.14%	–

known packers with varied abilities against hacking are chosen. Since all of them employ the software-only anti-debugging techniques, we can test and verify whether they could coexist with SPAD.

Table 2 indicates that all the tools work correctly with SPAD, *i.e.*, SPAD's protection mechanism is orthogonal to the existing methods. Besides, as a debugger detector, SPAD retains transparent to the OS and applications. Bzip2 cannot run after VMProtector packing. VMProtector features instruction emulation technique which may change the original logic. As SPAD makes no modification to the executable modules, it is less possible to change original application logic.

4.3 Performance

Bzip2 was applied as the protection target to evaluate anti-debugging performance and the time inflation. Minesweeper game from Windows XP was used to evaluate the size expansion after equipping the anti-debugging shields.

Table 2 illustrates that most packers cause notable size expansion to the original application due to unpacking code and injecting obfuscation code. PECompact features size reduction since it aims more at compression instead of protection. In comparison, SPAD causes no file size change to the original application. Meanwhile, since SPAD is compatible in most cases, packers that reduce file size like PECompact can also be applied for application and will cause no conflicts with SPAD.

Considering the overhead incurred by packing, the more anti-debugging techniques a packer employed, the slower application with packing will run. SPAD causes little time inflation to the target application, almost as less as the weakest packer, while providing strong anti-debugging performance. Although each boundary switching between VMM and guest OS will introduce a mass of extra CPU cycles [9], SPAD settles this problem by avoiding unnecessary mode switching via the protection filter. Other packers featuring better anti-debugging performance will inevitably lead to runtime inflation. In particular, Themida has the best anti-debugging performance but the worst size and time inflation.

5. CONCLUSION

Debugging is a widespread method for runtime observation that benefits software development, but it may be wicked if malicious attackers exploit it. This paper presents a lightweight and transparent protection strategy based on the art of virtualization. Leveraging hardware assisted virtual machine monitor, SPAD can detect and intercept debugging behavior in a higher privilege position than OS kernel, as well as make itself imperceptible to debuggers. Our experiments show that SPAD is effective to disable the debugging functionalities with a low performance overhead, and highly compatible with other orthogonal anti-debugging methods.

REFERENCES

1. M. Madou, B. Anckaert, B. D. Sutter, and K. D. Bosschere, "Hybrid static-dynamic attacks against software protection mechanisms," in *Proceedings of Digital Rights Management Workshop*, 2005, pp. 75-82.
2. M. N. Gagnon, S. Taylor, and A. K. Ghosh, "Software protection through anti-debugging," *IEEE Security and Privacy*, Vol. 5, 2007, pp. 82-84.
3. K. Kaspersky, *Hacker Debugging Uncovered*, Independent Pub Group, Wayne, PA, 2005.
4. X. Jiang, D. Xu, and Y. M. Wang, "Collapsar: A VM-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention," *Journal of Parallel and Distributed Computing*, Vol. 66, 2006, pp. 1165-1180.
5. N. L. Petroni Jr. and M. W. Hicks, "Automated detection of persistent kernel con-

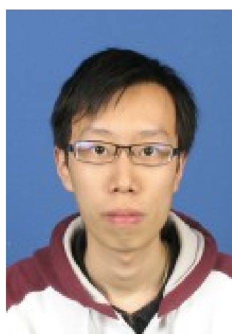
trol-flow attacks," in *Proceedings of ACM Conference on Computer and Communications Security*, 2007, pp. 103-115.
6. S. T. King and S. W. Smith, "Virtualization and security: Back to the future," *IEEE Security and Privacy*, Vol. 6, 2008, pp. 15.
7. X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proceedings of International Conference on Dependable Systems and Networks*, 2008, pp. 177-186.
8. P. Graf and K. D. Müller-Glaser, "Gaining insight into executable models during runtime: Architecture and mappings," *IEEE Distributed Systems Online*, Vol. 8, 2007, pp. 1.
9. K. Adams and O. Agesen, "A comparison of software and hardware techniques for x86 virtualization," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 2-13.
10. B. Parno, "Bootstrapping trust in a 'trusted' platform," in *Proceedings of USENIX Workshop on Hot Topics in Security*, 2008, pp. 1-6.
11. T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, and *et al*., "Bitvisor: A thin hypervisor for enforcing i/o device security," in *Proceedings of International Conference on Virtual Execution Environments*, 2009, pp. 121-130.
12. A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proceedings of ACM Symposium on Operating Systems Principles*, 2007, pp. 335-350.

Zhengwei Qi received his B.Eng. and M.Eng. degrees from Northwestern Polytechnical University, in 1999 and 2002, and Ph.D. degree from Shanghai Jiao Tong University in 2005. He is an Associate Professor at the School of Software, Shanghai Jiao Tong University. His research interests include static/dynamic program analysis, model checking, virtual machines, and distributed systems.

Bingyu Li received her B.Eng. degree from Shanghai Jiao Tong University in 2012. She is going to pursue her M.S. degree in computer science in University of California, Los Angeles. Her research interests include virtual machines and cloud computing.

Qian Lin received his M.Eng. degree from Shanghai Jiao Tong University in 2011 and B.Sc. degree from South China University of Technology in 2008. Currently he is a Ph.D. candidate in School of Computing, National University of Singapore. His research interests include operating system, cloud systems, trusted computing and distributed database.

Miao Yu received his B.S. degree and M.S. degree from Shanghai Jiao Tong University in 2009 and 2012. He will be a Ph.D candidate in Carnegie Mellon University. His research interests include virtualization, trust computing, network and operating systems.



Mingyuan Xia received his Bachelor degree from Shanghai Jiao Tong University in 2011. Now he is a PhD. student in McGill university, Canada. His research interests mainly include operating system, system security, cloud computing (virtualization) and compiler.

Haibing Guan received his Ph.D. degree from Tongji University in 1999. He is a Professor of School of Electronic, Information and Electronic Engineering, Shanghai Jiao Tong University, and the director of the Shanghai Key Laboratory of Scalable Computing and Systems. His research interests include distributed computing, network security, network storage, green IT and cloud computing.