applied sciences

MDPI

Article

VMGuards: A Novel Virtual Machine Based Code Protection System with VM Security as the First Class Design Concern

Zhanyong Tang [1], Meng Li [1], Guixin Ye [1], Shuai Cao [1], Meiling Chen [1], Xiaoqing Gong [1,*], Dingyi Fang [1] and Zheng Wang [2]

1 School of Information Science and Technology, Northwest University, Xi'an 710127, China; zytang@nwu.edu.cn (Z.T.); lijmeng@stumail.nwu.edu.cn (M.L.); gxye@stumail.nwu.edu.cn (G.Y.); implementist@stumail.nwu.edu.cn (S.C.); meilingchen@stumail.nwu.edu.cn (M.C.); dyf@nwu.edu.cn (D.F.)

2 School of Computing and Communications, Lancaster University, Lancaster LA1 4YX, UK; z.wang@lancaster.ac.uk

* Correspondence: gxq@nwu.edu.cn

Received: 3 April 2018; Accepted: 8 May 2018; Published: 12 May 2018

Abstract: Process-level virtual machine (PVM) based code obfuscation is a viable means for protecting software against runtime code tampering and unauthorized code reverse engineering. PVM-based approaches rely on a VM to determine how instructions of the protected code region are scheduled and executed. Therefore, it is crucial to protect the VM against runtime code tampering that alters the instructions and behavior of the VM. This paper presents *VMGuards*, a novel PVM-based code protection system that puts the security of VM as the first class design concern. Our approach advances prior work by promoting security of the VM as the first class design constraint. We achieve this by introducing two new instruction sets to protect the internal implementations of critical code segments and the host runtime environment where the VM runs in. Our new instruction sets not only have an identical code structure as standard virtual instructions, but also provide additional information to allow the VM to check whether the critical internal implementation or the runtime environment is affected. We evaluate our approach by using a set of real-life applications. Experimental results show that our approach provides stronger and more fine-grained protection when compared to the state-of-the-art with little extra overhead.

Keywords: software tamper-proofing; process-level virtual machine protection; code obfuscation; software protection

1. Introduction

Malicious tampering and unauthorized usage of software are serious concerns for the computing industry. According to a recent study, unauthorized usage of software costs nearly $400 billion in 2015 [1]. Code obfuscation technique based on process-level virtual machines (PVMs) is becoming a viable means to protect software systems from malicious tampering. Recent studies have shown that such techniques are effective in managing digital copy rights [2], code protection [3], and tamper-proofing [4].

With PVM-based tamper-proofing techniques, the native codes of the crucial part of the software are transformed into bespoke virtual machine instructions (bytecodes). These bytecodes will then be translated into machine code by a virtual machine (VM) interpreter during runtime. The underlying principal of PVM-based protection is to force the attacker to move from a familiar computing environment to the unfamiliar and obfuscated virtual environment, which can significantly increase the cost of software tampering so that it prevents tampering and deters unauthorized reverse engineering of software.

The VM system plays a key role in any PVM-based protection scheme because it determines how the virtual instructions are scheduled and interpreted—which ultimately determines which information and software behavior will be exposed to the user and attackers. Therefore, it is crucially important to protect the VM against malicious tampering. Prior approaches on PVM-based code protection work by creating a new section in the program binary to store virtual instructions of protected code regions. However, the new section is vulnerable to advanced attacks where an adversary can use sophisticated tools (e.g., Ollydbg and IDA Pro) to locate the section so as to discover the information of VM implementation and virtual instruction sets. Holding such information helps an adversary better understand the working mechanism of the VM. As a result, the security of the protected code regions will be greatly compromised.

This paper presents *VMGuards*, a novel tampering detection system. Contrary to previous work, our approach regards the security of the VM as the first class concern in the system design. We achieve this by providing two new sets of virtual instructions to protect the internal implementation of the VM and the host runtime environment where the VM runs in from code tampering. Firstly, we design a set of Tamper Proofing Instructions (TPIs) to protect the internal code integrity. Our TPIs extend the standard virtual instruction by providing additional information for the VM Interpreter to check the starting and ending addresses of the protected code region, as well as the checksum of the code to prevent code from unauthorized modifications. Our careful design ensures that TPIs have the same structure as the standard virtual instructions so as not to cause the attackers' suspicions. Secondly, we provide a set of Anti-Debug Instructions (ADIs) to protect the host runtime environment (e.g., operating systems and libraries) where the VM runs in against code tampering. These two instruction sets, put together, allow us to build a code protection system that is stronger than any previous work we have seen so far. By carefully integrating the instructions with the VM implementation, we are able to build a code protection with little extra overhead compared to existing solutions.

We evaluated *VMGuards* using a set of real-life applications. Experimental results show that *VMGuards* provides stronger protection than the state-of-the-art code protection systems at the same overhead. This paper makes the following specific contributions:

(1) It is the first work on PVM-based code protection that provides security protection for the VM system.
(2) It introduces a novel set of tamper-proofing instructions to protect critical internal implementation from runtime code tampering.
(3) It provides a new set of anti-debugging instructions to detect the presence of code tampering in the current host environment.

The remainder of this paper is organized as follows. Section 2 discusses previous work in the area of software tamper-proofing. In Section 3, we describe the threat model for program execution. Section 4 gives an overview of *VMGuards*. The concepts of TPI and ADI are explained in Section 5 and Section 6. In Section 7, we evaluate and discuss the performance and overhead of *VMGuards*. Finally, Section 8 presents our conclusions.

2. Background

2.1. Code Obfuscation

Obfuscation [5] is a traditional method of software protection, such as control flow expansion [6], garbage code insertion [7], instruction deformation [8], binary code encryption and packaging [9], and virtualization obfuscation [10]. These code obfuscation techniques are now common in malware, which make it difficult to discover the true logic of the program and give the security analyst an incredibly hard time. Most current obfuscation methods can resist static reverse engineering. However, attackers often use debug tools to dynamically analyze the structure of the program so that the obfuscation logic can be easily removed. More and more researchers are concerned about VM-based code protection, which can effectively resist debugging attacks and prohibit memory dumps.

2.2. Software Tamper-Proofing

Software programs are increasingly used to execute critical tasks. At the same time, their protection against undesired modification is very important. Much research has been done in the field of tamper-proofing. The most classical one is based on the seminal research by Aucsmith [11]. However, in total, program protection techniques can be widely classified into two sides: hardware and software approaches.

Hardware-based approaches are somewhat monetary cost and cannot be circle used basically. However, on the other hand, these approaches are more difficult to break. Now, there are many hardware solutions that have been proposed to against physical and software attacks. Dongles is a very common device that is distributed to the copy of the software and can be attacked to a computer. The program will invoke a callback discontinuously from dongles during the execution progress of the program. If the reply has no sign of abnormal, the program will continue to run, and, if not, it ceases to run. It is more difficult to debug the dongles. That is to say, anyone who wants to copy the dongles is illegal. However, if an adversary can obtain the security function from the dongle, he can also bypass the protection using emulation. As the time goes by, modern processors possess their own protection functionality in hardware (e.g., the Cell Broadband Engine, which provides the Runtime Secure Boot feature, is used to verify an application that has not been tampered with at start up [12]). Other hardware solutions involve the use of trusted computing platforms such as TPM (TPM2.0, Infineon, Munich, Germany) and Microsoft's NGSCB (Microsoft Corporation, Redmond, WA, USA) [13]. Finally, there are customized hardware techniques to prevent tampering, such as execute-only memory [14] and systems based on secure processors [15].

On the contrary, software-based approaches are used more generally. Chang [16] proposed the original guards used to tamper-proof, which employed the idea of checksum guards to protect the integrity of an application. After that, Horne [17] et al. came up with the concept of tester, which was similar to the guards. It consists of a sequence of instructions that calculated a hash value of fixed length addresses and compared it with a corrector during the program runtime.

Obfuscation techniques are also widely used in the area of software tamper-proofing so that it is difficult for an adversary to understand and analyze the original code of the application. Collberg et al. provided a number of obfuscation techniques to obscure the code [18,19]. Linn et al. designed a novel technique to disturb the disassembly of the code [20]. Wang et al. proposed the use of indirect branches to obfuscate the control flow of a program. Encryption of the program code is another way to consolidate the program. The specific hardware can decode the code (i.e., a secure co-processor [21]). However, as we mentioned before, they cost a lot than other software-based approaches. Nevertheless, these static obfuscation techniques are vulnerable to some dynamic analysis techniques [22].

2.3. Process-Level Virtual Machine

Among many software-based tamper-proofing techniques, virtualization is a milestone that improved the protection techniques, especially in runtime protection scenarios. Currently, a number of commercial PVMs have been created, such as VMProtect [23], Code Virtualizer [24] and Themida [25]. These tools protect the program by using a mid-expression—Instruction Set Architecture (ISA) [26]. In addition, the code will be interpreted during program runtime.

Among the existing research about PVM, almost all of it integrates various protection techniques together. Ref. [4] combined the diversity with virtual machine, proposing a diversity software protection model based on virtual machine. They also designed an RSIC-based software protection virtual machine, which is the first work that merges diversity with VM. In addition, Ref. [3] introduced the concept of multi-stage binary code obfuscation to obscure a program. First, it will proceed n times control flow obfuscation at the beginning of the VM protection. Then, it will choose the key code segment for protection. Moreover, their work also adopts to the centralized and chained dispatching model. This is the first study to improve the VM's dispatching structure. A basic block as the dispatching unit combines the multiple-level obfuscation to enhance the protection strength of the VM further. Ref. [27] presented another VM protection method based on adaptive dynamic instruction set. It dug down into the virtual machine protection mechanism, enhancing the execution efficiency and security of the protected program. Ref. [28] merged the virtual junk instruction sequence and virtual instruction obscure transformation technique further. In addition, it improved the design of the virtual instruction set, making it much harder for an adversary to reverse-engineer the protected program. All in all, the research on PVM is much more valuable than it appears.

2.4. The Design Goal

Reverse engineering of VM-based code protection usually follows many steps. The attacker first reverse-engineers the virtual interpreter to understand the semantics of the various bytecode instructions, such as how the bytecode is translated to native code. Then, he/she translates the bytecode back to native machine instructions or even high-level programming language to understand the program logic. Among these steps, understanding the semantics of a single bytecode instruction is often the most time-consuming process, which involves analyzing the handlers used to interpret each bytecode instruction. Many methods have been proposed to protect the VM handlers from reverse engineering. Most of them aim to increase the diversity of program behavior by obfuscating a handler or using multiple interpretation techniques to transform a single program iteratively multiple times. However, all previous work employs a fixed strategy where each bytecode is deterministically translated to a fixed set of native code. For programs that use the same obfuscation protection, these techniques are vulnerable. In particular, an attacker can reuse the knowledge of the handler implementation obtained from one program (termed analytical knowledge in this paper) to attack another program. This paper presents *VMGuards*, an enhanced VM-based code obfuscation system to address the issue of attacking the VM. We use two types of security instructions, ADI and TPI, to monitor the dynamic changes of VM in memory.

3. The Attacking Scenario

Our approach is evaluated using a strong attacking scenario of tampering described in [29]. In this paper, we assume that the attacker has set up a tampering host environment to execute the VM-protected program and that he/she has the right debugging tools and skills to simulate and modify instructions of the protected software, and can access and rewrite contents in the program memory space. This setting represents some real-world attacking scenarios. For example, it is reported that Skype, a widely used communication software (Skype 2.5.0.126, Microsoft Corporation, Redmond, WA, USA), could be cracked by using a debugger to locate and modify the key-code segment information in the program memory space during runtime [22]. Under our setup, the runtime environment including the operating systems, libraries, and hardware may also be affected so that it would provide applications with fake information. Therefore, in this work, we assume that the hardware and software used by the target application may be compromised by the attacker.

4. Overview of Our Approach

VMGuards is a PVM-based software protection system. Like traditional PVM-based protection schemes, the VM is used to virtualize a single application. Our goal is to protect the VM that is a key component of any PVM-based scheme from runtime code tampering so as to alter the behavior of the VM and the protected software. This is accomplished by introducing two new sets of virtual instructions, TPI and ADI, to prevent unauthorized internal modification of the VM and to detect any tampering performed in the host environment. VMData is the key secret of the scheduling mechanism.

It determines the order in which these handlers run, but it is completely different each time. During the running process, *VMGuards* encrypt and decrypt the VMData with the same key in order to reduce the costs of the system. In addition, every protected software version has a unique fixed key in advance.

Figure 1 depicts the overall approach of *VMGuards*. In the figure, K_C is the key code segment (e.g., the key algorithm of the authorization information) to be protected. During static compilation, the segment will be filled up with some junk or useless code. A new section N_C will be created to store the generated virtual instructions for K_C. wnen K_C is executed, the control will be given to N_C and the virtual instructions will be dynamically translated into native code by a VM.

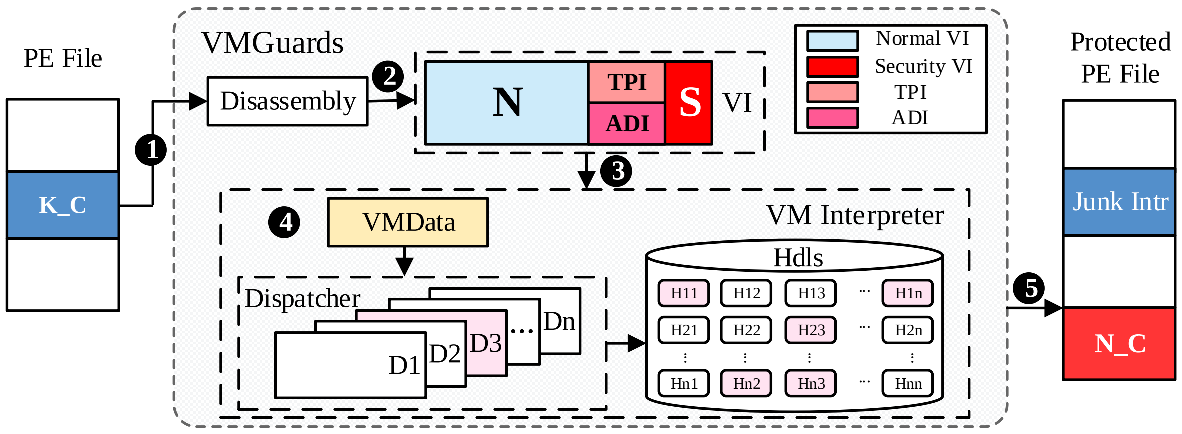


Figure 1. The framework of *VMGuards*. Tamper Proofing Instruction (TPI) and Anti-Debug Instruction (ADI) are two kinds of security instructions, which can ensure the virtual instructions' self-security.

This paper holds that all PE files can be protected in principle, so the legitimacy of the PE file inspection is not elaborated. The concrete construction of *VMGuards* proceeds as follows:

Step 1 Locate the range of the critical code segment (start and end addresses) of the PE file to be protected and disassemble the binary code of the key code segment.

Step 2 Design a set of intermediate instruction sets, virtual instruction sets (VIs), and transform these disassembled instructions into virtual instructions.

Step 3 Construct atomic function sets(Hdls), and generate handlers by the corresponding virtual instructions.

Step 4 Use the appropriate encryption algorithm to encrypt handlers generated by Step 3 and organize it into VMData.

Step 5 The design of reasonable Dispatcher.

Step 6 According to the order of execution, VMData, Dispatcher and Hdls make up the VM Interpreter, as shown in Figure 1, and it is represented in the form of a new section on the protected PE file.

Step 7 The original key code segments fill with some junk or useless code and add the correct jump instruction, and then output the protected PE file.

The specific execution flow within the VM Interpreter must first use Dispatcher to decrypt the first used Handler, while the latter execution is that the Dispatcher selects VMData and decrypts it to get the required Handler, or jumps directly from the previous Handler executed to the address of the next Handler to be executed based on the offset address. The following paper details the five parts of *VMGuards*.

4.1. Disassembly

There are two typical disassembler techniques [30]: linear disassembly and recursive disassembly. The former first disassembles the first instruction of the program. Once the instruction at an address is disassembled and determined to have a length of 1 byte, disassembly proceeds to the instruction starting at address addr+l. This process continues until the end of the program. Linear disassembly may be confused by "gaps" in code that consist of data or alignment-related padding. These gaps will be interpreted as instructions by linear disassembly and decoded, resulting in erroneous disassembly.

The latter uses different strategies. It starts with a set of code entry points specified in the binary file. For an executable file, there may be just one such entry point specified, but, for a shared library, the beginning of each exported function is also specified. Unlike linear disassembly, recursive disassembly is not confused by the gaps in the code and therefore does not produce incorrect disassembly. However, it cannot disassemble the code that can only be accessed through indirect control flow (ICF) transfers.

Our method combines linear disassembly and recursive disassembly. We use linear disassembly to ensure the integrity of the disassembled results and guarantee the correctness of disassembly results by recursive disassembly.

4.2. Virtual Instructions

After disassembling all instructions, we need to transform these disassembled instructions into a mid-expression (VI) [26] so that they can only be recognized by our *VMGuards* when the protected program is executed. Moreover, we have the freedom to design our own VI, which leaves us many choices to make. In this paper, we designed VI to simplify the correspondence between the original instructions and the Handlers, making the design of *VMGuards* more concise. While the relationships between the original instructions and virtual instructions, and virtual instructions and Handlers are many-to-many, it will increase the difficulty of reverse engineering. However, the virtual instructions are not stored in the protected program.

In our *VMGuards*, there are three main principles for designing a reasonable VI sets. First of all, completeness, that is to say, the designed VI must be able to interpret all x86 instructions. Secondly, complexity, a reasonable complexity can lead to a many-to-many relationship, or VI is redundant. Last but not least, functionality, every virtual instruction must have a well-defined meaning, and it can precisely guide an x86 instruction to a certain Handler sequence.

Tamper-proofing instructions and Anti-debug instructions are the two most designed security virtual instructions. Through these two security instructions, on the one hand, we can guarantee the internal security of the program, on the other hand, we can prevent the external execution environment of the program from being affected by malicious tools. Sections 5 and 6 below will focus on these two virtual instructions. More details will be introduced one by one.

4.3. Handler Set

Handler set is short for Hdls. Hdls consists of multiple Handlers. The Handlers are some small instruction segments. In our design, there are two types of Handlers, one with no parameters and the other with one or more parameters. In a Handler, it may contain one or more of the following operations: mathematical operations, conditional jump operations, assignment operations and special operations. It should be noted that, in this paper, the Handler is not the common handle in Windows, it is a small piece of program that can be invoked by the Dispatcher (described in the next section). It is used to interpret the original x86 instructions. In other words, every primitive x86 instruction can be interpreted by several Handlers through a mapping relationship with virtual instructions. That is to say, the original x86 instruction firstly should be mapped to virtual instructions and then be interpreted by several Handlers. Figure 2 shows the mapping relationship of the three elements.

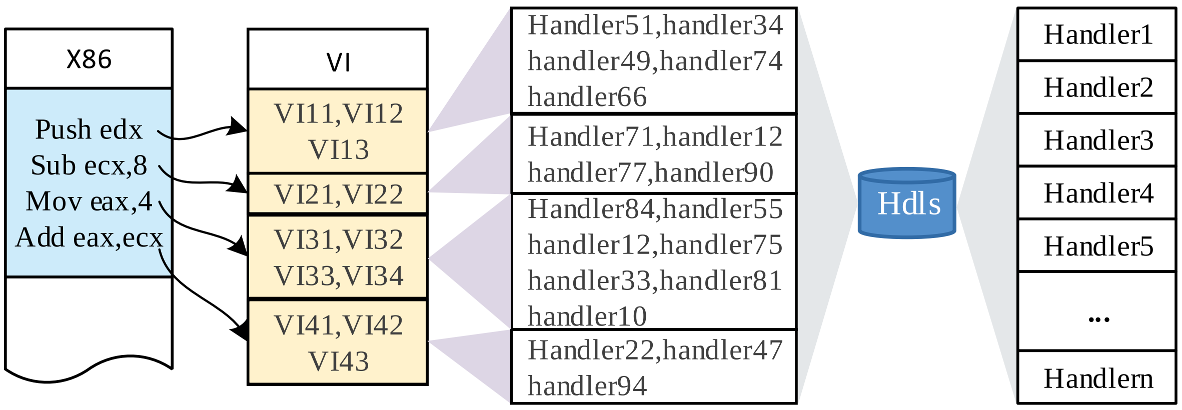


Figure 2. Every original x86 instruction maps to several sets of Handlers.

4.4. Dispatcher

Just like buses have dispatching centers, our *VMGuards* has Dispatchers for reading and decrypting VMData (described in the next section). After decryption, the plaintext will be used to calculate the corresponding Handlers' addresses by using a certain algorithm, and then the program will jump to the accurate address to be executed. From this point, we can see that the Dispatcher is an important component of the VM interpreter and it is responsible for controlling the Handler's scheduling process.

So far, there has been a lot of research on improving the structure of Dispatcher about PVM. At present, we can divide the existing structure into three categories: centralized model [4,31], Chained model [32] and Hybrid model [32,33]. In this paper, we prefer to use the third one. Because it is easy to reverse-engineer the centralized model to find the only Dispatcher, further locate all the Handlers invoked by the Dispatcher. For the Chained model, it is one way in and one way out. Handles in chained model Dispatcher are called by an offset from one to another, as long as the adversary finds the jmp instruction behind every Handler, he can get all Handlers, and furthermore reverse-engineer the whole program.

We choose to combine the Centralized model and the Chained model together to improve the security of Dispatcher's structure. There are the following advantages. Firstly, an adversary cannot gather the whole dispatching procedure through only one fixed Dispatcher's address. Secondly, we designed all these Dispatchers to be similar to the normal Handlers, and after executing the Handler, it randomly returns to one of these Dispatchers or Handlers, so as an adversary, he cannot easily figure out which one is the Dispatcher and which is the Dispatcher handler. Figure 3 gives a schematic of the structure.

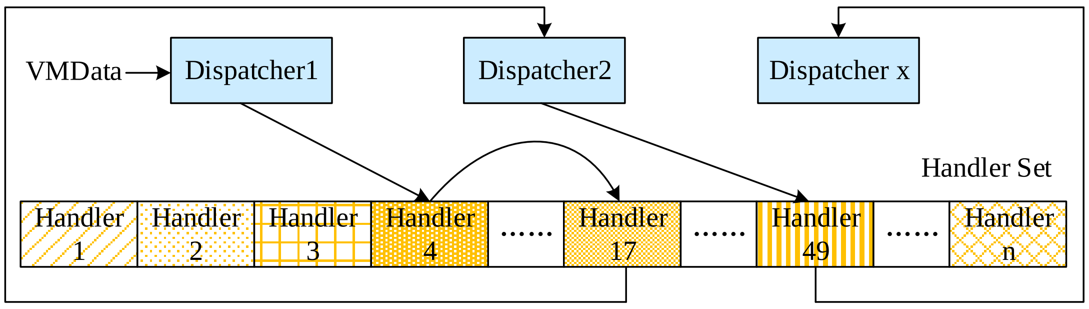


Figure 3. The structure of hybrid model Dispatcher.

4.5. VMData

As we can see from Figure 3, VMData is an encrypted data used to drive the Handler execute, and it is an encrypted Handler sequence that is executed. More specifically, the VM Interpreter chooses Handler to execute based on the VMData read in the Dispatcher. Looking at Section 4.3, there are two kinds of handlers, and one is a handler that contains one or more parameters. VMData also provides encrypted data to the parameters to further obfuscate the Handler numbers and parameters.

The VM Interpreter will use a lot of crucial information, including VMData, which is why we usually use encryption mechanisms to protect them. Correctly generating the VMData is the key to accurately dispatching the Handlers. Therefore, the specific meaning of the VMData is closely related to the Dispatcher.

5. Tamper-Proofing Instruction

The concept of guards was first proposed by Chang and Atallah [16] as a tamper-proofing mechanism. Guard is a dynamic tamper-proofing technology whose software tamper-proofing technology includes the detection and response functions. In general, for the detection function, the basis of code integrity check is to calculate the hash value H of the protected code segment in advance and store it in the code gap (the PE file alignment strategy leaves a lot of unused space, which is called code gap). The same checksum algorithm is designed, and each time the protected program is executed, the hash value H′ of the same protected code segment is calculated again and compared with the value H. If the hash value H is equal to H′, the code segment has not been tampered with during execution. For the response function, once tampering is detected, two types of response methods can be selected: first, the explicit response, such as a pop-up reminder, direct termination of the program execution and so on; second, the implicit response, such as forged results, the introduction of infinite loop and so on.

Existing research on software guards have gained so much success [27,34–36]. However, these implementations are either simple structure or high overhead. In Chang's scheme, it is said that the guard tends to be removed as an atypical operation by reading the code segment during execution; then, the adversary can usually pinpoint the checks by setting breakpoints or examining the code. The main drawback of guards net is that the complexity will result in higher overhead and longer execution time, which is detrimental to software developers debugging and optimizing programs. Moreover, the function of every guard is the same, and all guards are run in the host process. Thus, an adversary can locate one guard by debugging the process; then, he can disable them all.

In this paper, we introduced Tamper-Proofing Instruction, or TPI for short, consisting of existing Handlers. The TPI implements the same functions as the guards but is presented as a normal Handler. With this design, on the one hand, the key code segments of the program are detected and protected by the TPI; on the other hand, the structure of the TPI Handler is confused with normal Handlers, which makes it troublesome to identify which Handler determines the function of TPI.

We choose VMData, Handler and Dispatcher as tamper detection targets. VMData, Handler and Dispatcher do not change during the execution of the software, so, if any changes are detected in these modules during execution, it is sufficient to prove that the software has been tampered with. In view of the security instruction itself affecting the performance of the program execution, the tamper-proofing algorithm should not be too complicated, while the hash function, parity algorithm and CRC32 are more complex, so we use a relatively integrity check algorithm as a tamper-proofing algorithm.

In addition, we have designed a variety of TPIs to defend against the attack that the adversary can remove all checkpoints just from one, and every two TPIs are crossed with each other to make sure that when an adversary can locate and remove one TPI, the other can detect the tamper behavior in time. Figure 4 shows the basic organization structure of the TPI; it is like a ring, so we call it TPI-ring. In the figure, TPI1 and TPI2 are interleaved, TPI2 and TPI3 are interleaved, and their protection ranges are a fixed length. If an adversary can locate TPI1 successfully and remove it from the TPI-ring, TPI2 and the last TPI can detect the behavior in time to make sure they cannot be tampered with.

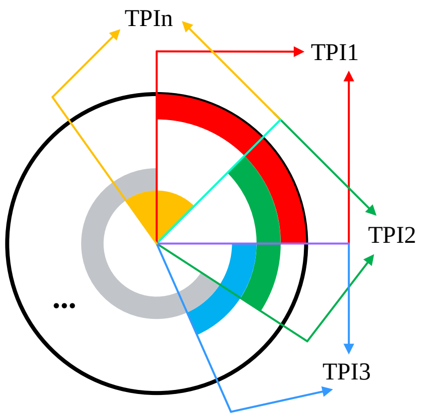


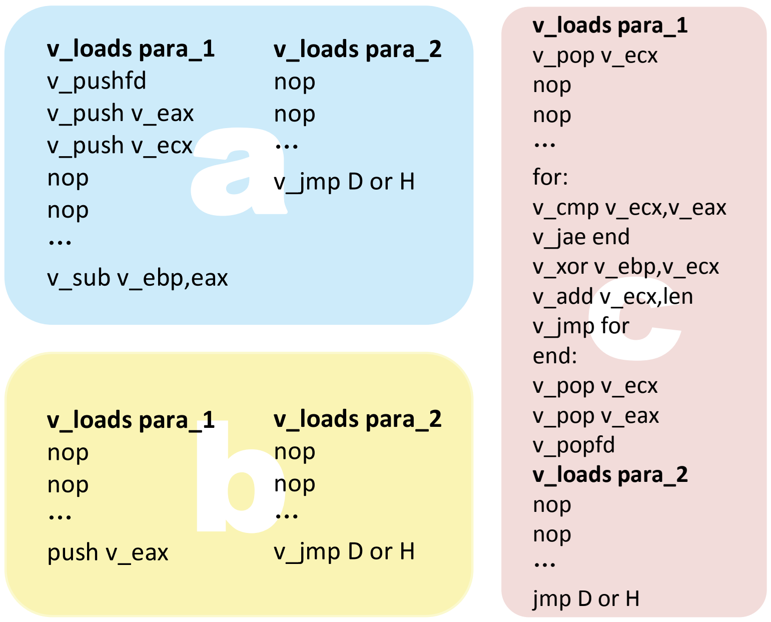
Figure 4. The organization of TPI-Ring. To make sure every sensitive area at least has three TPI Handlers protecting it.

Assuming that the start address of TPI1 is the beginning address of the key code segment, the main challenge here is to deal with the protection range of the last TPI. We can clearly see that the last protection range is divided into two parts, the start address is in the middle of the TPI (n−1) and the end address is in the middle of the TPI1. Thus, how do we deal with this situation? Here is our solution. First, we calculate a checksum1 from the start address of the last TPI to the end of key code segment and then calculate a checksum2 from the start of the key code segment to the end address of the last TPI. After that, using the sum of checksum1 and checksum2 as the checksum of the last TPI, either checksum1 or checksum2 is tampered with, and TPI (n−1) or TPI1 can find it.

Figure 5 shows the TPI handlers. In our implementation, there are three different Handlers used to achieve the goal of TPI—to detect and respond to the tamper behavior when an adversary is tampering with a protected program. During the program creation phase, they will be placed at random locations. Furthermore, during execution time, they will not be closely executed one by one; in other words, the execution of TPI_b may not immediately follow TPI_a and TPI_c's execution may not immediately follow TPI_b, but the relative execution order of TPI_a, TPI_b and TPI_c is fixed. When TPI_c is actually executed, these three Handlers do play the role of guards. The 'nop' among the virtual instructions will be filled up with the exact decryption virtual instructions during program creation time. For each para_1 in every TPI Handler, the function of them are:

- Load the pre_checksum that we calculated before the program execution and stored it in a register (v_ebp) after some math operations.
- Load the start address of the protection range and push it into a register (v_eax).
- Load the end address of the protection range and push it into a register (v_ecx); during the real execution of the program, the program will dynamically recalculate the checksum from the relative start and end address one more time.

In our implementation, during the running of the program, the value of v_ebp will be recovered by a relative mathematical operation. In addition, this checksum is compared with the stored pre_checksum. If they are equal, no tampering occurs; otherwise, the program will take some necessary measures to prevent tampering. For example, use a MessageBox to alert the user that his program is being tampered with, or just terminate the execution of the program. It is worth noting here that there is a para_2 after every TPI Handler; we have designed para_2 to implement the centralized and chained dispatching models. That is to say, after executing the TPI handler, it will jump to the next handler, which will be executed to continue execution or jump back to one of the Dispatchers. In fact, every normal Handler has at least one parameter to determine that the executing handler will jump to the next Handler or Dispatcher.



```
v_loads para_1
v_pushfd
v_push v_eax
v_push v_ecx
nop
nop
...
v_sub v_ebp,eax

v_loads para_2
nop
nop
...
v_jmp D or H
```

```
v_loads para_1
nop
nop
...
push v_eax

v_loads para_2
nop
nop
...
v_jmp D or H
```

```
v_loads para_1
v_pop v_ecx
nop
nop
...
for:
v_cmp v_ecx,v_eax
v_jae end
v_xor v_ebp,v_ecx
v_add v_ecx,len
v_jmp for
end:
v_pop v_ecx
v_pop v_eax
v_popfd
v_loads para_2
nop
nop
...
jmp D or H
```

Figure 5. A TPI consists of three different handlers. TPI_a, to load the checksum and operate it with EBP. TPI_b, to load the start address. TPI_c, to load the end address and restore EBP.

6. Anti-Debug Protection

When we try to debug some programs, anti-debug techniques will be used to thwart our behavior. In other words, the program to be debugged can detect whether there is a debugger attached to itself once the anti-debug techniques are applied. Assuming that the program can "feel" that it is being debugged, it can make sure that someone is trying to crash itself using a debugger or disassembler.

Debugger is a software that can be used to debug the operating system kernel. It runs between the CPU and operating system and works on layer Ring0. For example, SoftICE is one debugger of NuMega (SoftICE v4.3.2.2485, Compuware NuMega, Nashua, NH, USA). In addition, OllyDbg (OllyDbg 2.01) and TRW2000 (TRW2000 1.23) are also popular debuggers. Such a debugger can load PE files into the memory, run the PE files at single step, or set breakpoints in PE files. Assuming that there is no anti-debug mechanism, the adversary can have the ability such as step tracing, breakpoints setting to the program. Furthermore, he can use the result of a tracing analysis-registration algorithm to program a register.

The basic idea of ADI design is to abstract anti-debug technologies into security virtual instructions and implement it by the corresponding Handlers. During the process of virtual machine protection, ADI is randomly placed in the VM Interpreter memory space to ensure that the software is not disturbed by the debugger during execution. The specific process is that the protected program with the anti-debug mechanism will first check for the presence of debugging during execution. The detection method can use a variety of anti-debug technologies as theoretical support and record the checking results. After that, according to the check results, it can be determined whether the program is being debugged or in a debugger environment. If debugging behavior is found, the program will make a response, and the response method can learn from TPI response method. Otherwise, the program will normally execute.

In this paper, we designed five monitoring ADHandlers and two processing ADHandlers together to achieve the function of anti-debug. In fact, these anti-debug techniques are all from "tangjiutan", a member of forum Kanxue. These ADHandlers will be introduced one by one as follows:

Five monitoring ADHandlers:

(1) BeingDebugged flag: IsDebuggerPresent is a function stated in kernel32.dll, by which we can detect whether the current process is being debugged or not. If the process is being debugged, the BeingDebugged flag will be set to '1', otherwise, it will be '0'.
(2) FindWindow: The function is used to retrieve and process the strings specified by class name of top-level window and window name, but it does not retrieve the child window.
(3) CheckRemoteDebuggerPresent: This is another Win32 Debugging API function; it can be used to check if a remote process is being debugged. However, we can also use this as another method for checking whether our own process is being debugged. CheckRemoteDebuggerPresent calls the NTDLL to export NtQueryInformationProcess and sets the SYSTEM_INFORMA-TION_CLASS to 7(ProcessDebugPort). Additionally, the "Remote" here does not imply that the debugger must be on different computers; instead, it indicates that the debugger resides in a separate and parallel process. The IsDbuggerPresent function is used to detect whether the calling process is running under the debugger. At last, if the function succeeds, the return value is nonzero; otherwise, the value will be zero. To get extended error information, call GetLastError.
(4) Execution Time Difference: When the process is being debugged, the CPU cycle will be occupied by debugger events, such as handling codes and skipping instructions. If the time cost for executing two adjacent instructions is hugely different than that for executing normal ones, it indicates that the process may be currently being debugged.
(5) OllyDbg Detection: So much like the "Detect Debugger Process", OllyDbg Detection just focuses on the character "OLLYDBG", which exists in the process list.

Two processing ADHandlers: **ExitProcess** and **MessageBox**. We only designed these two simple example response Handlers to handle the monitoring ADHandlers. Literally, ExitProcess is designed to exit the process when it detects that there is a debugging process, and the MessageBox is a little "mild". When it detects the debugger, it will pop up a warning MessageBox to the user and the user will decide what will be done next, restart the process or terminate the process directly.

In order to better balance performance and overhead, we apply the idea of diversity. In each run of the protected program, it will randomly choose one of the five monitoring ADHandlers to detect whether there is a debugger attached, and, if found, the program will randomly choose one of the two processing ADHandlers to respond, such as a friendly reminder MessageBox or simply terminate the program. Otherwise, the program continues to execute.

7. Evaluation and Discussion

We designed a Proof-of-Concept prototype called *VMGuards* for implementing TPI and Anti-debug, which was built on a process-level virtual machine. *VMGuards* runs with the protected program as a common program that will be firstly loaded and checked. Then, *VMGuards* will perform some preprocessing tasks, for example, disassembling key code segments of the program marked previously by hand, generating multiple handler sets. These information will be stored in a new section. During program execution, *VMGuards* will take over execution control when it reaches the critical code segment. Then, the new section executes and transfers control back to the original program. The program will continue executing to the end.

7.1. Experimental Setup

7.1.1. Benchmarks

We evaluated *VMGuards* with six real-life applications running on the Windows operating system (Windows 7, Microsoft Corporation, Redmond, WA, USA). Table 1 describes the key information of these applications. The code segments to be protected in these applications represent some commonly used algorithms and operations, including mathematical calculations, compression, location positioning, recursion, message sending and receiving, and web browsing. The length of the protected code segment varies from four instructions to 363 instructions, with runtime ranging from 0.34 µs to 39 ms.

Table 1. Test cases used in *VMGuards*.

APP_Name	Key-Code Segment Description	Length of Key-Code Segment	File Size (Byte)	Execution Time of Key-Code Segment (µs)
Calculator.exe	Multiply operation algorithm in Windows Calculator.	48	2,265,190	32
Compress.exe	File Compression algorithm.	110	229,447	1970
Csnake.exe	Positioning algorithm in game snake.	60	233,563	0.34
Hanio.exe	Hanio algorithm, and the plate number is 5.	82	548,942	39,750
Hviewer.exe	Response message algorithm in simple Browser.	4	2,195,556	86
Ipmsg.exe	Message sending algorithm in IP Messenger.	363	417,869	4139

7.1.2. Evaluation Platform

In principle, the idea of *VMGuards* applies to most architectures. Our prototype implementation targets the 32-bit Intel x86 platform (Santa Clara, CA, USA). Our experimental platform is a Dell OptiPlex 960PC (Round Rock, TX, USA) with a 3 GHz Intel Core Duo processor and 4 GB of RAM. It runs on the 32-bit Windows 7 operating system.

7.1.3. Runtime Measurement

Since some of the protected code segments run for a short time, it is very important to gather accurate runtime information. For this reason, we use hardware performance counters to gather runtime information. This process is described in Table 2. To determine how many runs are needed to reduce noise during program execution, we use a 95% confidence interval to compute the confidence range after each run. When the difference between the upper and lower confidence intervals is less than 2%, the program can be stopped running.

Table 2. Execution time of key-code segments.

```
QueryPerformanceCounter(&Start_Time); //start time
VMGuards _Start //key code start flag
Key-Code Segment
VMGuards _End //key code end flag
QueryPerformanceCounter(&End_Time); //end time
//CPU counter numbers
CPUCounter_Num = End_Time.QuadPart-
Start_Time.QuadPart;
//Clock Frequency of CPU timer
QueryPerformanceFrequency(&freq);;
//The execution time of Key-Code Segment, accurate
down to µs
_i64toa(C*1000000/freq.QuadPart, szBuf, 10);
```

7.2. Experimental Results

7.2.1. Runtime Overhead

In order to accurately calculate the execution time of the TPIs of different lengths, we tested these applications four times separately using the execution time calculator mentioned earlier.

Figure 6 shows the test results. Due to different applications, different key code segment lengths, and different difficulty levels of key code segments, the execution times are obviously different from each other. Figure 6a represents all the test results, and we can see that the results of HViewer, Calculator, and Csnake are not clear. Therefore, we draw another four subfigures. Figure 6b shows the test results of Compress, and Figure 6c to HViewer, Figure 6d to Calculator and Figure 6e to Csnake. Table 3 shows the test results in detail. Take IpMsg.exe as an example, when TPI length is 0, that is to say, we don't insert any TPI Handler into the protection area, the execution time corresponds to the original time. To be clear, we collected 20 different execution times and calculated the average value, so when TPI length is 32 bytes (which means that the protection range is 32 bytes), the average execution time of the key code segment is 4462 μs. Similarly, when TPI length is 16 bytes and 8 bytes, the average execution time is 4557 μs and 4761 μs, respectively.

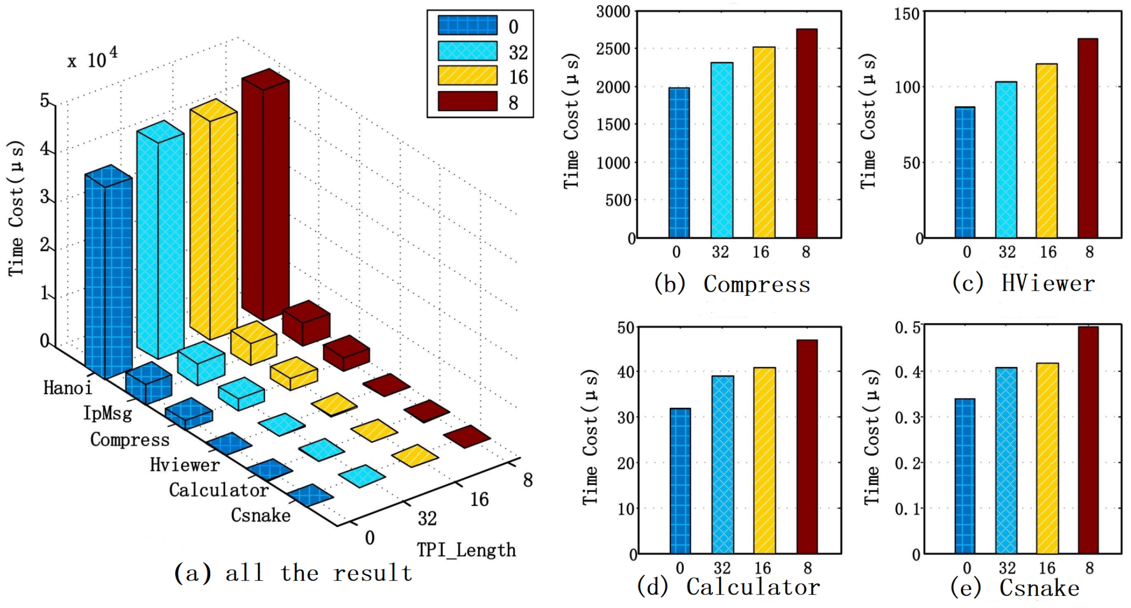


Figure 6. Runtime overhead

Table 3. Execution time (μs) of different Tamper-Proofing Instruction (TPI) length.

TPI (byte) / **APP_Name**	**0**	**32**	**16**	**8**
Hanoi	39,750	44,876	45,268	47,630
IpMsg	4139	4462	4557	4761
Compress	1970	2315	2516	2760
HViewer	86	103	115	131
Calculator	32	39	41	47
Csnake	0.34	0.41	0.44	0.5

In order to see the execution time increment transparently, we defined:

$$T\_Incre_{rate} = \Delta Time/O\_Time. \tag{1}$$

We used the formula to calculate an increment for different TPI lengths. From Figure 7, we still took IpMsg.exe as the example, when TPI length is 32 bytes, the increment is 7.8%, and when TPI lengths are 16 bytes and 8 bytes, the increments are 10.1% and 15.02% separately. Please note that the unit of time is microsecond (µs). You have not even "felt" that the application has finished executing.

7.2.2. Space Overhead

Our instrumentation adds a new section of code to the original program, which is on average 1.03 times the size of the original code. The new section contains more than one Dispatcher to assign Handlers to jump to the right addresses for execution. The Dispatchers are mostly like an address translation table for every Handler.

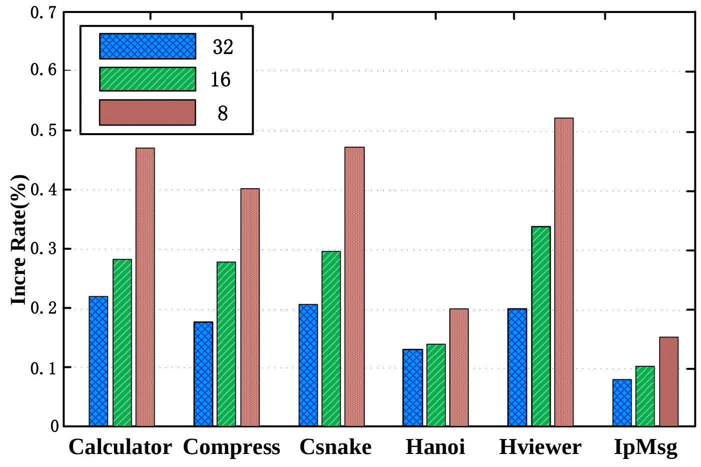


Figure 7. Increment of different lengths of TPI.

Figure 8 shows different applications' average running memory costs. The blue bar-graph represents the memory costs when the original programs are running. Correspondingly, the red bar-graph shows the memory costs when the protected programs are running. In total, the space overhead for *VMGuards* is 117% over the original program. We need to note that execution must be confined to the new code, although the file size has increased. Except for the programs that store read-only data in their code, other programs do not even access their code once. Hence, the runtime memory overhead is unaffected by the presence of the original copy of the file.

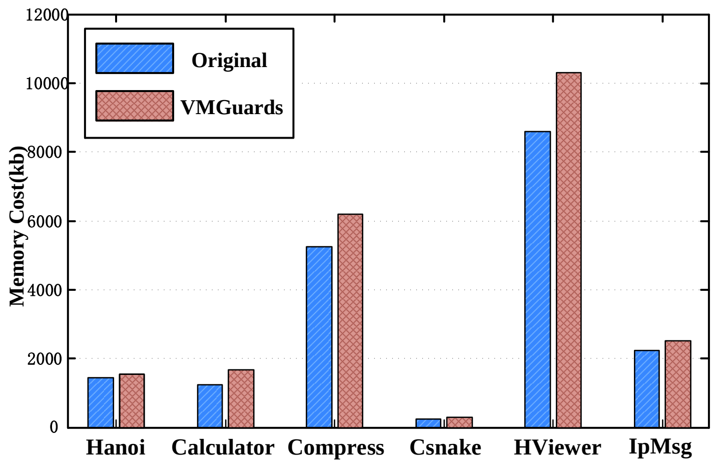


Figure 8. Memory cost of the running test cases before and after they are protected.

7.2.3. Performance of *VMGuards*

We tested the applications shown in Table 1 with commercial software Code Virtualizer1.3.1.0 (short for CV) and VMProtect1.7.0 (short for VMP) compared with our *VMGuards*. We set CV to the lowest protection strength and VMP to the fastest execution time. Figure 9 shows the change of average file size and execution time.

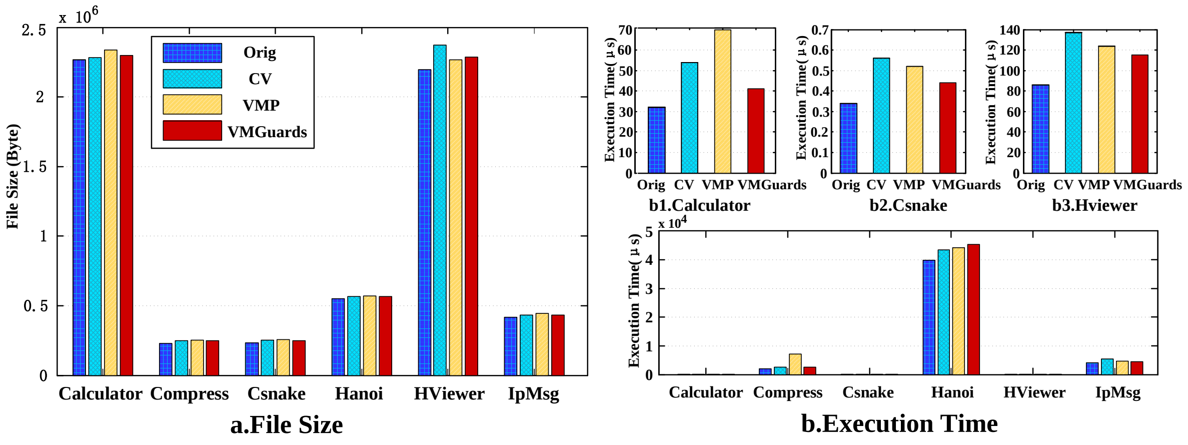


Figure 9. File size and execution time differences compared with different commercial PVMs with our *VMGuards*.

As we can see from Figure 9, comparing with commercial software, our *VMGuards* does not increase the space overhead. However, as for runtime overhead, our *VMGuards* shows some unstable results.

There are three reasons: (a) whether the x86 instructions are different types or different addressing modes, the sequence of the Handler used to interpret the VI is different from each other when the aforementioned *VMGuards* interprets the execution (the x86 instructions); (b) there are still some x86 instructions that cannot be interpreted by existing Handlers. The program needs to retreat from *VMGuards*, executing the x86 instruction by the host machine then back to *VMGuards* and continue to interpret execution to the left VI. Thus, the runtime overhead can be huge; (c) owing to the design of multiple Dispatchers, execution time to different instances can be short or long. In addition, we need to do further study on it.

7.3. Case Study

The design goal of *VMGuards* is to improve the security of PVM through two security instructions: TPI and ADI. In this section, we evaluate the security guarantee of TPI and ADI through both experiments and theoretical analysis.

7.3.1. Security of Tamper-Proofing Instruction

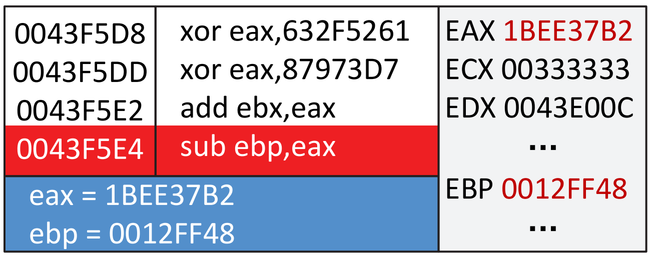
As described in Section 5, we organized our Tamper-Proofing Instruction into a ring. Comparing with guards net, TPI-Ring reduced the space overhead brought by complex network theoretically. However, TPI Handlers in TPI-ring did not lower the self-security of TPI Handlers. We can also see from Figure 4 that every TPI Handler is crossed with each other. In other words, there are at least three TPI Handlers protecting one address segment when the protected program is running. Standing by the adversary's side, when he tries to break the TPI-Ring, he would not get what he wants unless he can remove all the TPI Handlers at one time.

In order to see the TPI-Ring clearly, we did some modification to re-organize and centralize the whole function TPI's three handlers together. With some reverse-engineering experiences, we found three sets of TPIs that are deployed in the virtual instructions (see Table 4).

Table 4. TPI protected area.

0043E2E4	AD 33 C3 05	DB 04 20 5F
0043E2EC	2D 23 6E A0	6F 2B D8 66
0043E2F4	8B 14 24 52	8B D4 83 C2
0043E2FC	04 83 C2 02	87 14 24 8B

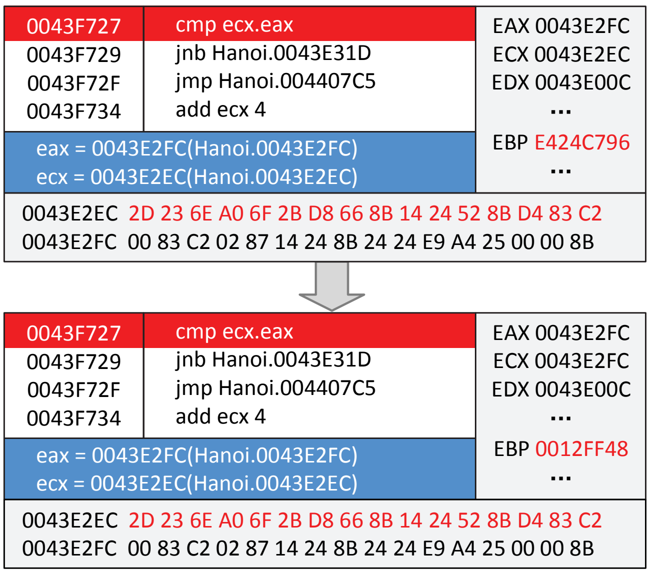
As described in Section 7.1, we set the length of TPI to 16 bytes. Based on the idea of cross protection, the start addresses and the end addresses of the three sets TPI are 0043E2E4-0043E2F4, 00E3E2EC-0043E2FC, and 0043E2F4-0043E304. We employed a simple checksum algorithm and added up the data of the protection range with four bytes length together for demonstration. During program execution, it will loop to re-compute a checksum1 from the start address to the end address. We assume that we choose the second set of TPI to test the functionality. Firstly, TPI_a will decrypt out a checksum (1BEE37B2). Then, TPI_b and TPI_c will find the start address (00E3E2EC) and the end address (0043E2FC). Figure 10 is a part of the original TPI handler.



```
0043F5D8   xor eax,632F5261      EAX 1BEE37B2
0043F5DD   xor eax,87973D7       ECX 00333333
0043F5E2   add ebx,eax           EDX 0043E00C
0043F5E4   sub ebp,eax           ...
eax = 1BEE37B2                   EBP 0012FF48
ebp = 0012FF48                   ...
```

Figure 10. Part of the original TPI handler.

Here is our starting point comparing the normal checksum algorithm. The value of a register will be frequently invoked during program execution. We did not just simply compare the pre_checksum with the checksum re-calculated during the program execution. We subtract the pre_checksum from the value of a register (EBP), with loop accumulating to restore the value of EBP (see Figure 11). If the value of EBP is not changed, there is no tampering occurring. Otherwise, the program will be crashed when it comes to invoke EBP.



```
0043F727   cmp ecx.eax                 EAX 0043E2FC
0043F729   jnb Hanoi.0043E31D          ECX 0043E2EC
0043F72F   jmp Hanoi.004407C5          EDX 0043E00C
0043F734   add ecx 4                   ...
  eax = 0043E2FC(Hanoi.0043E2FC)       EBP E424C796
  ecx = 0043E2EC(Hanoi.0043E2EC)       ...
0043E2EC  2D 23 6E A0 6F 2B D8 66 8B 14 24 52 8B D4 83 C2
0043E2FC  00 83 C2 02 87 14 24 8B 24 24 E9 A4 25 00 00 8B
```

```
0043F727   cmp ecx.eax                 EAX 0043E2FC
0043F729   jnb Hanoi.0043E31D          ECX 0043E2FC
0043F72F   jmp Hanoi.004407C5          EDX 0043E00C
0043F734   add ecx 4                   ...
  eax = 0043E2FC(Hanoi.0043E2FC)       EBP 0012FF48
  ecx = 0043E2EC(Hanoi.0043E2EC)       ...
0043E2EC  2D 23 6E A0 6F 2B D8 66 8B 14 24 52 8B D4 83 C2
0043E2FC  00 83 C2 02 87 14 24 8B 24 24 E9 A4 25 00 00 8B
```

Figure 11. Sample of tamper-proofing instructions.

Theoretically, if we modify some data within the second TPI's protection range, we believe that both the three TPIs and the whole TPI-ring can be aware of the tampering behavior.

We invite a number of students to perform a practical reverse analysis of VM protection and evaluate the security of *VMGuards* by comparing the difficulty of extracting key data and structure. These students are computer-related majors and have research experience in software security. They play the role of an attacker and target the following three points, then perform reverse analysis on *VMGuards* and Normal VM protection and compare the results.

- Target 1: Find the entrance to the virtual interpreter through analysis and debugging.
- Target 2: Through analysis and debugging, locate the Dispatcher and analyze its structure and function.
- Target 3: Modify the structure of the Dispatcher and output the Handler's scheduling sequence.

The setting of these targets is mainly based on the classical VM reverse analysis method, which first locates the key data and structures of VM interpreter, and then tracks and fetches the execution sequence of the Handler, finally extracting the target code through optimization and simplification.

The experimental results show that most participants are able to complete target 1 in the face of *VMGuards* and Normal VM protection. This is because the program jumps to the virtual interpreter when calling the critical code, so there is clear control of the transfer behavior. In the following analysis process, most attackers can quickly locate the position of the Dispatcher of a normal VM protection program through the debugging analysis (target 2). In addition, some experienced attackers can modify the structure of Dispatcher with tools (like Pin tools) and extract the execution sequence of Handlers further (target 3). However, when they reverse the analysis of *VMGuards* protected program, they encountered obstacles. ADI and TPI make it difficult to continue the dynamic debugging process for attackers. Only some experienced attackers found the Dispatcher structure through static analysis. They also cannot extract Handler execution sequences in batches by tampering with the code structure. These task targets are steps that an attacker must go through to reverse understanding of code virtualization protection, so the fewer successful attackers, the better the method's advantage. The experimental results show that our *VMGuards* protection method has better security.

7.3.2. Effectiveness of Anti-Debug Instructions

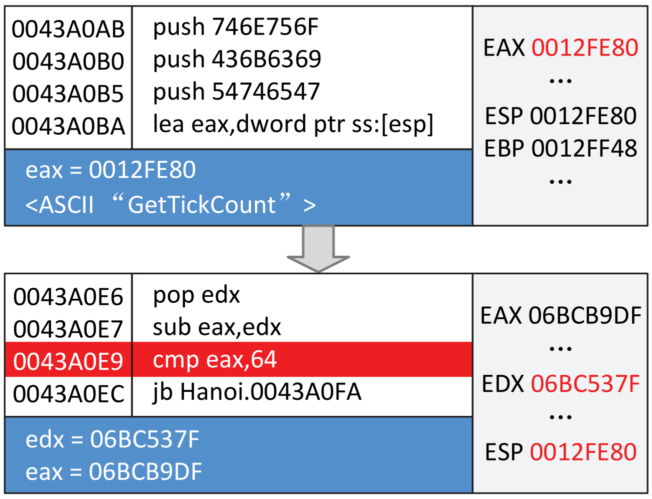
Assuming that Tamper-Proofing Instructions prevent the program itself from modifying, we applied Anti-Debug Instructions to guarantee that the execution environment of the program is harmless. As described in Section 6, we designed seven anti-debug security instructions. That is to say, we designed seven different Handlers to achieve the goal of all these anti-debug instructions.

Experimentally, *VMGuards* randomly chooses the Execution_Time_Difference monitor Handler to detect whether the program is being debugged during protection procedure, as we can see from Figure 12.

Processing Handler will be chosen to respond to the monitored debugging behavior, seen from Figure 13. The MessageBox Handler was chosen to respond to the Execution_Time_Difference Handler.

In every execution, the program will randomly choose one of the five monitor ADI Handlers to detect—for example, whether the program is under attack by the debugger or the difference in execution time is very different. If the chosen executed monitor ADI happens to be detected, then the program will randomly choose one of the two processing ADIs to alert users that the program is being debugged with a MessageBox or just terminates the program roughly.

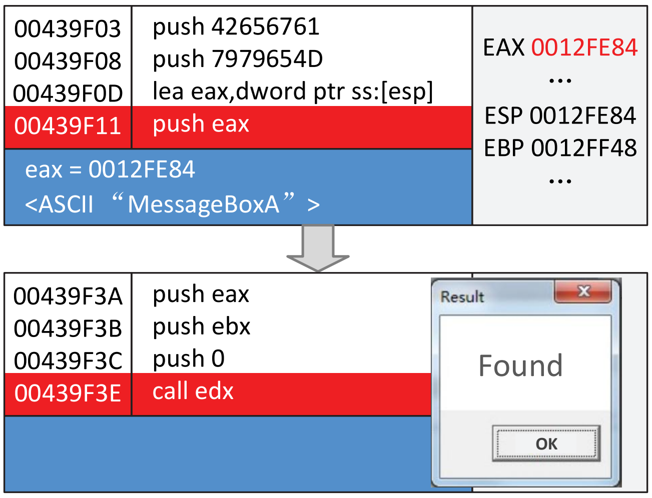
Figure 14 shows 100 times the execution results of the protected program. As mentioned before, which monitor will be executed by the Handler and Processing Handlers depends on the random number. As a result, the statistical results are not an inexorable law to follow.



```
0043A0AB  push 746E756F                 EAX 0012FE80
0043A0B0  push 436B6369                 ...
0043A0B5  push 54746547                 ESP 0012FE80
0043A0BA  lea eax,dword ptr ss:[esp]    EBP 0012FF48
eax = 0012FE80                          ...
<ASCII “GetTickCount” >

0043A0E6  pop edx                       EAX 06BCB9DF
0043A0E7  sub eax,edx                   ...
0043A0E9  cmp eax,64                    EDX 06BC537F
0043A0EC  jb Hanoi.0043A0FA             ...
edx = 06BC537F                          ESP 0012FE80
eax = 06BCB9DF
```

Figure 12. Sample of Monitoring ADHandler.



```
00439F03  push 42656761                 EAX 0012FE84
00439F08  push 7979654D                 ...
00439F0D  lea eax,dword ptr ss:[esp]    ESP 0012FE84
00439F11  push eax                      EBP 0012FF48
eax = 0012FE84                          ...
<ASCII “MessageBoxA” >

00439F3A  push eax
00439F3B  push ebx
00439F3C  push 0
00439F3E  call edx
```

Figure 13. Sample of Processing ADHandler.

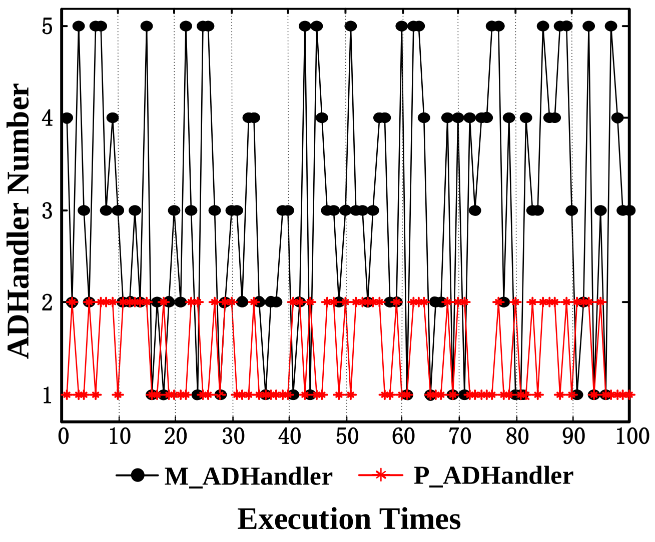


Figure 14. Statistics results from 100 times of executions, M is for Monitor Handler, and P is for Process Handler.

7.4. Discussion

This section states some of the protective aspects of the *VMGuards* and vulnerabilities inherent to PVM-based security measures.

7.4.1. Protection of Virtual Instruction

The primary goal of Tamper-Proofing Instruction is to protect the virtual instruction from tampering. To our knowledge, *VMGuards* is the first PVM-based scheme attempt at tamper-proofing software code. Previously, the adversary could, for instance, locate the critical component-Dispatcher, collect information of all Handlers, and make modifications to the virtual instructions. TPI-Ring can guard all the virtual instructions from tampering, as any changes made by adversaries will be detected and responded to appropriately. However, too much stress of the intensity of TPI adds serious performance to both runtime and space. In addition, the structure TPI-Ring itself has already added an additional layer of protection from tampering. With a better balance between TPI protection range and TPI-Ring, *VMGuards* still can be used to protect the program from tampering effectively.

7.4.2. Effectiveness against Malicious Hosts

Anti-Debug Instruction is designed to make sure that the execution environment of the program is unharmed. Currently, anti-debug techniques can be divided into two categories. One is to check hardware or software debug structure for the presence of debugger state such as breakpoint set. The other is to detect human behavior such as the undesired pause during program execution. An unambiguous rule for distinguishing the behavior of debugger is recognized as being highly significant for the effectiveness and accuracy of the anti-debug technique. However, one drawback of the current anti-debug mechanisms is that most of them fail to conceal themselves from other processes. If malware or debuggers perceive that they are monitored by anti-debug schemes, they would try their best to bypass such monitoring or even strike back using anti-anti-debug techniques. Anyhow, applying anti-debug technique into PVM-based software protection still can thwart the malicious behavior of an adversary as long as the host machine is harmless.

7.4.3. VM-Based Protection and Signature Comparison

The traditional code signature algorithm is an alphanumeric string obtained by processing the information transmitted through a one-way function to authenticate the information source and verify whether the information changes during the transmission.

There are three differences compared here to VM-based protection:

(1) The purpose of the design is different. The former is to protect the application from reverse engineering. It protects the code and its execution logic. The latter is used to verify whether the information has been tampered within the transmission process. It does not have the ability of protecting the application program.
(2) The idea of the algorithm is different. The former replaces the machine code of the application and interprets it through the redesigned Handler. The latter uses a selected one-way function (such as the hash function) to process the information transmitted to get the processing result. Then, the information is sent together with the processing result. The receiver uses the same one-way function to process the received information, and compares the obtained processing result with the result from the sender to judge whether the received information is tampered with.
(3) Based on the above two points, the former needs to make necessary changes to the application program, while the latter only uses the application program or information as an input to the processing function and does not need to change the program.

8. Conclusions

This paper has presented *VMGuards*, a novel process level virtual machine based on a code protection system. *VMGuards* advances prior work on VM-based code protection by promoting the VM security as the first class design concern. This is achieved through introducing two new sets of virtual instructions to maintain the integrity of the internal implementation of the VM and to detect any tampering attempts to the environment (e.g., the operating system, libraries and hardware) where the protected applications run. Our approach is evaluated by using six real-world applications. The experimental results show that *VMGuards* offers stronger protection with comparable overhead when compared to the state of the art. We demonstrate that the VM is key for defending runtime tampering and its protection can be achieved with no extra cost. In the future work, we intend to explore further refinement to VM protection and integrate our approach with other code protection techniques.

Author Contributions: Z.T. and G.Y. conceived and designed this system; Z.T. wrote this paper; M.L. and X.G. were responsible for the implementation of the system; G.Y. provided analysis tools; S.C. modified the experimental evaluation section; M.C. modified the Introduction and Background sections; X.G. revised and improved the implementation part of the system; D.F. modified the system design part; Z.W. was responsible for the overall revision of the contents of the manuscript and the final approval of the version to be published.

Acknowledgments: This work was partially supported by projects of the National Natural Science Foundation of China (No. 61672427, No. 61672428); the International Cooperation Foundation of Shaanxi Province, China (No. 2015KW-003, No. 2017KW-008); the Special Service Foundation of Shaanxi Province Department of Education (No.16JF028); the Research Project of Shaanxi Province Department of Education (No. 15JK1734); the Research Project of ANT FINANCIAL Science Foundation; the U.K. Engineering and Physical Sciences Research Council under Grants EP/M01567X/1 (SANDeRs) and EP/M015793/1 (DIVIDEND); and a Royal Society International Collaboration Grant (IE161012).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

PVM	Process-level Virtual Machine
VM	Virtual Machine
RSIC	Reduced Instruction Set Computer
PE	Portable Executable
EBP	Extended Base Pointer
TPIs	Tamper-Proofing Instructions
ADIs	Anti-Debug Instructions
VI	Virtual Instruction
ICF	Indirect Control Flow
CPU	Central Processing Unit
TPM	Trusted Platform Module
CV	Code Virtualizer1.3.1.0
VMP	VMProtect1.7.0
ISA	Instruction Set Architecture

References

1. BSA GLOBAL SOFTWARE SURVEY. Available online: http://globalstudy.bsa.org/2016 (accessed on 15 October 2017).
2. Averbuch, A.; Kiperberg, M.; Zaidenberg, N.J. An Efficient vm-based Software Protection. In Proceedings of the IEEE 5th International Conference on Network and System Security (NSS), Milan, Italy, 6–8 September 2011; pp. 121–128.
3. Fang, H.; Wu, Y.; Wang, S.; Huang, Y. Multi-stage binary code obfuscation using improved virtual machine. In *Information Security*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 168–181.
4. Wang, H.; Fang, D.; Li, G.; Yin, X.; Zhang, B.; Gu, Y. NISLVMP: Improved Virtual Machine-Based Software Protection. In Proceedings of the IEEE 9th International Conference on Computational Intelligence and Security (CIS), Emei Mountain, Sichuan, China, 14–15 December 2013; pp. 479–483.
5. Schrittwieser, S.; Katzenbeisser, S. Code Obfuscation against Static and Dynamic Reverse Engineering. In *International Conference on Information Hiding*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 270–284.
6. Zhou, N.Q.; Qi, D.Y. Parameterized Decomposition Tree-Based Obfuscation Method with Double Flattening Control Flow. *J. South China Univ. Technol.* **2015**, *28*, 80–81.
7. Sun, Y.; Huang, G. A control flow obfuscation scheme based on garbage code. *J. Theor. Appl. Inf. Technol.* **2012**, *46*, 284–288.
8. Ledoux, C.; Sharkey, M.; Primeaux, B.; Miles, C. Instruction Embedding for Improved Obfuscation. In Proceedings of the Southeast Regional Conference, St. Louis, MO, USA, 4–6 October 2012; pp. 130–135.
9. Kim, M.J.; Lee, J.Y.; Chang, H.Y.; Cho, S.J.; Park, Y.; Park, M.; Wilsey, P.A. Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering. In Proceedings of the 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), Seville, Spain, 5–6 May 2010; pp. 80–86.
10. Banescu, S.; Lucaci, C.; Pretschner, A. VOT4CS: A Virtualization Obfuscation Tool for C#. In Proceedings of the ACM Workshop on Software PROtection, Vienna, Austria, 28 October 2016; pp. 39–49.
11. Aucsmith, D. Tamper resistant software: An implementation. In *Information Hiding*; Springer: Berlin/Heidelberg, Germany, 1996; pp. 317–333.
12. Shimizu, K.; Nusser, S.; Plouffe, W.; Zbarsky, V.; Sakamoto, M.; Murase, M. Cell Broadband Engine? Processor Security Architecture and Digital Content Protection. In Proceedings of the 4th ACM International Workshop on Contents Protection and Security, Santa Barbara, CA, USA, 23–27 October 2006; pp. 13–18.
13. Peinado, M.; England, P.; Chen, Y. An overview ofNGSCB. *Trust. Comput.* **2005**, *6*, 115.
14. Lie, D.; Thekkath, C.; Mitchell, M.; Lincoln, P.; Boneh, D.; Mitchell, J.; Horowitz, M. Architectural support for copy and tamper resistant software. *ACM Sigplan Not.* **2000**, *35*, 168–177. [CrossRef]

15. Suh, G.E.; Clarke, D.; Gassend, B.; Van Dijk, M.; Devadas, S. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In Proceedings of the 17th Annual International Conference on Supercomputing, San Francisco, CA, USA, 23–26 June 2003; pp. 160–171.
16. Chang, H.; Atallah, M.J. Protecting software code by guards. In *Security and Privacy in Digital Rights Management*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 160–175.
17. Hiser, J.D.; Williams, D.; Hu, W.; Davidson, J.W.; Mars, J.; Childers, B.R. Evaluating Indirect Branch Handling Mechanisms in Software Dynamic Translation Systems. In Proceedings of the IEEE Computer Society International Symposium on Code Generation and Optimization, San Jose, CA, USA, 11–14 March 2007; pp. 61–73.
18. Collberg, C.; Thomborson, C.; Low, D. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, 19–21 January 1998; pp. 184–196.
19. Collberg, C.; Thomborson, C.; Low, D.; Low, D. *A Taxonomy of Obfuscating Transformations*; Technical Report, Technical Report 148; Department of Computer Sciences, The University of Auckland: Auckland, New Zealand, July 1997.
20. Linn, C.; Debray, S. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In Proceedings of the 10th ACM Conference on Computer and Communications Security, Washington, DC, USA, 27–30 October 2003; pp. 290–299.
21. Zambreno, J.; Choudhary, A.; Simha, R.; Narahari, B.; Memon, N. SAFE-OPS: An approach to embedded software security. *ACM Trans. Embed. Comput. Syst.* **2005**, *4*, 189–210. [CrossRef]
22. Biondi, P.; Desclaux, F. Silver Needle in the Skype. *Black Hat Eur.* **2006**, *6*, 25–47.
23. Vmprotect Software. Vmprotect. Available online: http://vmpsoft.com (accessed on 12 December 2017).
24. Code Virtualizer. Available online: http://www.oreans.com/codevirtualizer.php (accessed on 20 December 2017).
25. Themida. Available online: http://www.oreans.com/themida.php (accessed on 25 December 2017).
26. Anckaert, B.; Jakubowski, M.; Venkatesan, R. Proteus: Virtualization for Diversified Tamper-Resistance. In Proceedings of the ACM Workshop on Digital Rights Management, Alexandria, VA, USA, 30 October–3 November 2006; pp. 47–58.
27. Ghosh, S.; Hiser, J.; Davidson, J.W. Software protection for dynamically-generated code. In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop, Rome, Italy, 26 January 2013; p. 1.
28. Blietz, B.; Tyagi, A. Software tamper resistance through dynamic program monitoring. In *Digital Rights Management. Technologies, Issues, Challenges and Systems*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 146–163.
29. Seshadri, A.; Luk, M.; Shi, E.; Perrig, A.; van Doorn, L.; Khosla, P. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. *ACM Sigops Oper. Syst. Rev.* **2005**, *39*, 1–16. [CrossRef]
30. Zhang, M.; Sekar, R. Control Flow Integrity for COTS Binaries. In *Usenix Security*; Stony Brook University: Stony Brook, NY, USA, 2013; Volume 13.
31. Averbuch, A.; Kiperberg, M.; Zaidenberg, N.J. An Efficient VM-based Software Protection. In Proceedings of the International Conference on Network and System Security, Milan, Italy, 6–8 September 2011; pp. 121–128.
32. Wang, H.; Fang, D.; Li, G.; An, N.; Chen, X.; Gu, Y. TDVMP: Improved Virtual Machine-Based Software Protection with Time Diversity. In Proceedings of the ACM Sigplan on Program Protection and Reverse Engineering Workshop, San Diego, CA, USA, 25 January 2014; pp. 1–9.
33. Kuang, K.; Tang, Z.; Gong, X.; Fang, D.; Chen, X.; Xing, T.; Ye, G.; Zhang, J.; Wang, Z. Exploiting Dynamic Scheduling for VM-Based Code Obfuscation. In Proceedings of the Trustcom/BigDataSE/ISSPA, Tianjin, China, 23–26 August 2016; pp. 489–496.
34. Horne, B.; Matheson, L.; Sheehan, C.; Tarjan, R.E. Dynamic self-checking techniques for improved tamper resistance. In *Security and Privacy in Digital Rights Management*; Springer: Berlin/Heidelberg, Germany, 2001; pp. 141–159.

35. Chen, Y.; Venkatesan, R.; Cary, M.; Pang, R.; Sinha, S.; Jakubowski, M.H. Oblivious hashing: A stealthy software integrity verification primitive. In *Information Hiding*; Springer: Berlin/Heidelberg, Germany, 2002; pp. 400–414.
36. Zhang, D. Checking Primitives with Guards. Ph.D. Thesis, University of Auckland, Auckland, New Zealand, 2005.