Replacement Attacks Against VM-protected Applications

Sudeep Ghosh Jason Hiser Jack W. Davidson
{sudeep, hiser, jwd}@virginia.edu
Department of Computer Science, University of Virginia, Charlottesville, VA-22903, USA.

Abstract

Process-level virtualization is increasingly being used to enhance the security of software applications from reverse engineering and unauthorized modification (called *software protection*). Process-level virtual machines (PVMs) can safeguard the application code at run time and hamper the adversary's ability to launch dynamic attacks on the application. This dynamic protection, combined with its flexibility, ease in handling legacy systems and low performance overhead, has made process-level virtualization a popular approach for providing software protection. While there has been much research on using process-level virtualization to provide such protection, there has been less research on attacks against PVM-protected software. In this paper, we describe an attack on applications protected using process-level virtualization, called a replacement attack. In a replacement attack, the adversary replaces the protecting PVM with an attack VM thereby rendering the application vulnerable to analysis and modification. We present a general description of the replacement attack methodology and two attack implementations against a protected application using freely available tools. The generality and simplicity of replacement attacks demonstrates that there is a strong need to develop techniques that meld applications more tightly to the protecting PVM to prevent such attacks.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors – Run-time Environments; D.4.6 [*Operating Systems*]: Security and Protection

General Terms Security

Keywords Software protection, Process-level virtualization, Dynamic binary translation, Code obfuscation, Reverse engineering, Software tamper resistance.

1. Introduction

Process-level virtualization is a versatile and powerful technique that addresses a wide range of system challenges. It has been increasingly used to deliver solutions in the area of software protection (i.e., mechanisms that protect software applications from reverse-engineering attacks and tamper) [1, 20]. A number of commercial products have been designed to provide software protection via process-level virtualization such as VMProtect [51], Code Virtualizer [37], Themida [36]. A number of computer gaming software applications employ the StarForce virtualization system for copy protection and anti-reverse engineering [48]. Recently, malicious agents have used this protection technique to design state-of-the-art malware that can evade current detection systems [41]. There are several reasons why PVMs are popular amongst security researchers for enhancing software protection.

- Process-level virtualization provides a platform for enhanced run-time security. Attackers are increasingly using dynamic techniques to attack software (e.g., running applications under a debugger or a simulator) [5]. PVMs allow run-time monitoring and checking of the code being executed, making them an excellent tool for devising dynamic protection schemes [38]. PVMs can also mutate the application code as it is running (e.g., changing code and data locations, replacing instructions with semantically equivalent instructions, etc.), hampering iterative attacks [20].
- It is advantageous to have the protection techniques closely integrated with the application, yet keep the implementations separate. This modular approach enables easier testing and debugging of the system, and it allows legacy systems to be retrofitted with new protections without the need for modification and recompilation. PVMs can be used to provide such a flexible capability.
- Static protection schemes can be strengthened when the application is run under a PVM. For example, encryption is a useful technique that hampers static analysis of programs. Because the encrypted code cannot be run directly on commodity processors, the software decryption of the application code becomes a point of vulnerability. For example, schemes which decrypt the application in bulk are susceptible to dynamic analysis techniques [5], whereas decryption at a lower granularity (e.g., functions) can suffer from high overhead [9, 29]. In contrast, executing encrypted applications under the control of a PVM has been shown to have a better performance-security tradeoff [27]. The PVM incurs low information leakage, with the addition of a small performance overhead [20]. Another example is the improvement of the robustness of integrity checks that are located in the application. When run under a PVM, these integrity checks never execute from their original location, instead, they can be invoked from randomized locations in memory [20]. This randomization makes it harder for the attacker to locate and disable the checks.

Considering their increasing use in program protection, it is important to assess the security of process-level virtualization as a whole. The existence of any weaknesses in the PVM or its interaction with the application can seriously undermine any protection mechanism that relies on process-level virtualization.

VEE'12, March 3–4, 2012, London, England, UK.

This work presents a generic attack methodology, called a replacement attack, that targets applications protected using process-level virtualization. The basic premise behind this attack strategy is that the PVM is not anchored sufficiently to the execution environment, and can be replaced by the attacker at run time. Using a replacement attack, the attacker can effectively remove any dynamic protection technique actuated by process-level virtualization and proceed to analyze the application at run time. Some of the contributions of this work are listed below.

- This paper describes replacement attacks, a novel attack methodology targeted towards virtualized applications (i.e., applications that run under the mediation of a process-level virtual machine), that seeks to render the protective PVM ineffective. A replacement attack can be used against any application that is run under the mediation of a process-level virtual machine. The goal of a replacement attack is the circumvention of the dynamic protections driven by PVMs, thereby making dynamic analysis easier.
- Our analysis shows that existing protection schemes, such as software checksumming guards (proposed by Chang et al. [10]), fail to adequately protect virtualized applications from this class of attack.
- We present a comprehensive two-part case study describing the replacement attack methodology. The first part of this study describes the creation of a protected, virtualized application, and then how an attacker can easily replace the protective PVM. We describe two prototypes of the replacement attack using easily available, free-to-use tools. The first involves replacing the protective PVM with an attack PVM (i.e., a PVM without any protections). The second prototype involves running the application on a modified simulator which circumvents the protective PVM and simulates the application directly. These examples demonstrate the feasibility and effectiveness of replacement attacks on non-trivial applications.
- We then discuss the implications of the replacement attack in the second part of our case study. It involves examining dynamic attacks on unprotected applications, PVM-protected applications, and applications subjected to the replacement attack. Our results show that the replacement attack renders the application completely vulnerable to run-time analysis and subsequent tamper.
- This research demonstrates that process-level virtualization currently fails to provide adequate protection to applications. Techniques that tightly bind PVMs with the protected applications are needed to thoroughly realize the protective capabilities of PVMs.

The remainder of the paper is organized as follows. Section 2 provides background on process-level virtualization. The protection model for a virtualized application is described in Section 3. Section 4 describes the attacker's capabilities and goals. Section 5 gives a high-level overview of the dynamic replacement attack. Section 6 uses a case study to describe the details of the protection mechanisms and how a dynamic attack circumvents them to leave an application vulnerable. Section 7 discusses the impact of this methodology with respect to reverse engineering. Section 8 explores some of the requirements to mount a successful attack. Section 9 describes some of the past work in the field of software tamper resistance, and finally, Section 10 presents the conclusions.

2. Process-level Virtualization

A software application is a sequence of instructions that execute on a particular computing system. Virtualization is a software layer

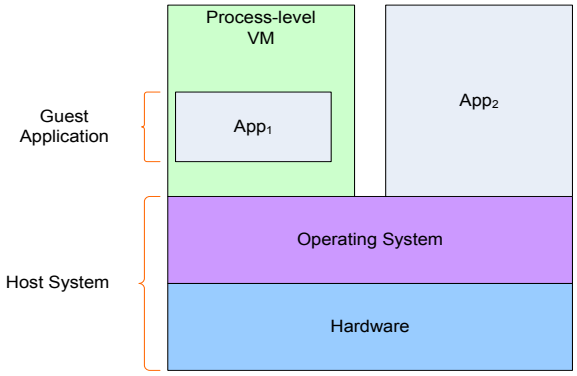


Figure 1. High-level overview of process-level virtualization. The guest application runs under the mediation of the process-level VM, giving an outward appearance of a process that is native to the underlying platform.

that encapsulates the application and its associated platform from the native computing system, allowing the application to execute on different platforms. Virtualization has been used to overcome the barriers imposed by new hardware [17, 39], or to improve security [7, 19, 30]. Formally, virtualization involves the construction of an isomorphism that maps a virtual system (called the *guest*) onto the native system (called the *host*). It is the responsibility of the virtual machine to run the application compiled for the guest (called the *guest application*) on the host system. Some of the necessary tasks include converting the guest application's instructions to run on the host, and mediating communication between the application and the host platform. Virtualization can be done at the system level (i.e., operating system), or at the process level (a single application).

Figure 1 illustrates a typical process-level virtual machine environment, where the guest application runs under the control of the PVM, giving it the outward appearance of a native host process. This work focuses exclusively on process-level virtualization and its weaknesses with respect to software protection.

During program startup, the PVM assumes control and starts decoding the application's instructions in program order. The decoded instructions are then used to invoke appropriate handling routines, which interpret the instructions on the host. As such, the PVM can regularly monitor the application code being executed on the host system, and serve as a platform for applying protection schemes dynamically.

3. Software Protection Model

This section describes the importance of software protection techniques and illustrates the general protection model in the context of a virtualized application.

Software applications often perform critical tasks in a wide variety of fields, such as banking, transportation and medical systems. Any unauthorized modification to such critical software systems can lead to extensive disruption of services and potential losses in terms of life and property. Therefore, it is important to protect software applications from malicious modification. We define the entity whose goal is to preserve the integrity of the software application as the *defender*. To achieve his goals, the defender uses different protection schemes to protect the critical functionality of the software. Numerous such techniques have been proposed to safeguard software, which can broadly be divided into static tech-

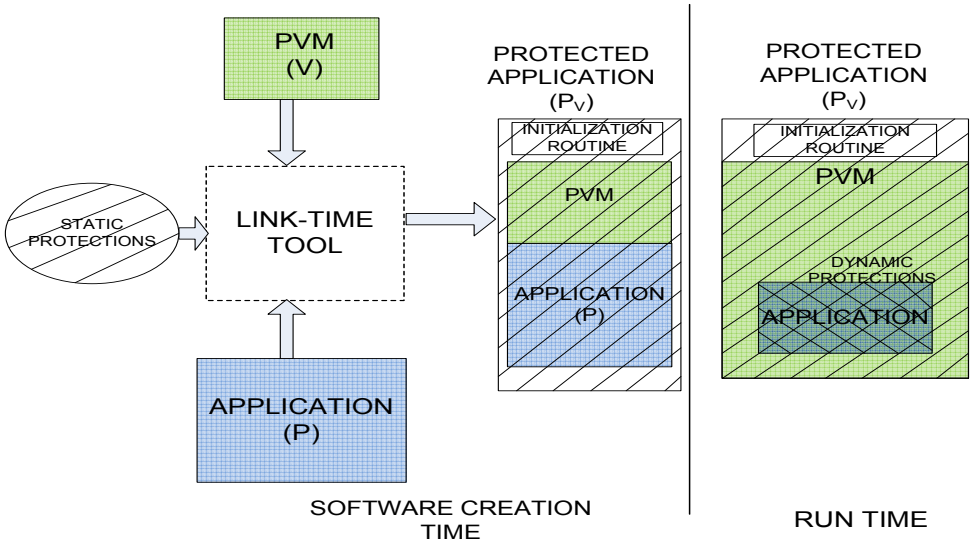


Figure 2. A high-level overview of the protected software creation process. A link-time tool is used to assemble the application code, libraries and the PVM code and apply various protection mechanisms, to create a protected binary. At run time, the application runs under the control of the protective PVM.

niques (which protect the on-disk binary from analysis), and dynamic techniques (which protect the software while it is running).

Static protections protect the on-disk binary from analysis and modification. A number of powerful static protection techniques have been proposed in recent years [13, 14, 32, 52]. For example, opaque predicates are a common technique that aid in program obfuscation [14]. An opaque predicate is a construct with *true/false* outcome. The opaqueness of these predicates is attributed to the fact that, it is hard to reverse engineer their values from the on-disk binary using static techniques. These constructs are utilized in obfuscating control-flow information. Code encryption is another useful technique against static analysis [9]. Figure 2 illustrates the creation of the virtualized application which is protected by such techniques. At software creation time, the defender packages the protective VM with the guest application P, along with its software libraries using a link-time rewriting tool. The tool then applies various protection schemes to the virtualized application P_V. Once the protection techniques are applied, the rewriting tool outputs the protected binary, as shown in Figure 2.

Static techniques alone fail to provide adequate protection. As we describe in Section 4, attackers are increasingly using run-time techniques to analyze applications. To protect against such attacks, a few solutions have been proposed which involve changing program code as it runs, to create a shifting attack target (e.g., dynamic instruction rewriting [28], and edit scripts [35]). Another solution involves packaging a process-level VM with the application, that protects the run time from analysis and generic attack [1, 20, 51], as shown in the right side of Figure 2. The application is modified such that, at program start up, the PVM is initialized and control is transfered to it. Subsequently, the application runs under the control of the PVM, rather than directly on the native platform. The code for invoking the PVM is stored in the *initialization routine* of the virtualized application. There are different techniques to create these initialization routines (e.g., in Executable and Linking Format (ELF) files, these routines can be placed in the `.init` section [54]. The MacOS binary format, Mach-O, contains the `__DATA,__mod_init_func` section for placing class constructors [2]).

A PVM's ability to monitor the guest application's code makes it a suitable platform for dynamic protection. As the PVM is virtualizing the guest application, it can check to make sure the code has not been modified, and introduce schemes to obfuscate the code. The PVM can be used to apply protection schemes on a *per-run* basis, creating a different attack surface every time the application is run [20]. Such a scheme makes it difficult to launch automated attacks on the application. PVMs have been shown to increase the effectiveness of existing protection techniques as well. For example, code encryption schemes are often vulnerable due to coarse levels of decryption at run time [5]. PVMs facilitate just-in-time, on-demand code decryption [20]. Only those instructions of P which are scheduled to be executed, get decrypted. After execution, the instructions can be re-encrypted. Therefore, P is never fully decrypted at any point during execution.

4. Attack Model

Once the software application is deployed, it may be subjected to various attacks. The attacker is defined as the entity that seeks to affect a software's functionality or misappropriate critical information from it. To achieve their goal, the attacker has to initially analyze the software and obtain a level of understanding of its operation. Using this derived knowledge, the attacker can then proceed to disable the protection scheme.

The attacker's goals have been aided by recent advances in reverse engineering technology, which have led to the availability of powerful tools for analyzing software. There are two types of analysis tools: *static analysis* tools that collect information about a program by studying its binary but without executing it (e.g., disassemblers and decompilers), and *dynamic analysis* tools that collect information from program runs (e.g., debuggers, simulators, dynamic analysis frameworks, and emulators [18, 34, 55]). In particular, the development and availability of dynamic tools has strengthened the attacker's capabilities because many static protec-

Term	Notation	Definition
Guest Application	P	The software application
Protective PVM	V	A process-level virtual machine that has been configured to apply various protection techniques at run time.
Virtualized application	P_V	Software application consisting of the P packaged together with a protective PVM.
Entry function	EP	The function in the VM code which initiates the process of application virtualization. To mount a successful attack, the attacker has to locate this function.
Attack PVM	M	A process-level VM that can aid the attacker in analyzing P.
Code Introspection Framework	CIF	An introspection framework capable of monitoring and instrumenting the code being executed.

Table 1. Glossary of the terminology used in this paper.

tion techniques are susceptible to run-time analysis [5]. Using these tools, the attacker attempts to obtain information necessary to carry out the intended attack. The attack described in the next section enables use of such tools on PVM-protected applications.

The terminology used in this paper is summarized in Table 1.

5. Replacement Attack

This section describes an attack methodology that targets software applications protected by process-level virtualization. The goal of this attack methodology involves replacing the protective PVM, rendering P_V vulnerable to analysis. The attacker can then use some of the tools described in Section 4 to obtain relevant information.

This methodology targets the surface of the application that is most vulnerable to attack (i.e., when protections are at their weakest). More specifically, this attack methodology targets the application just after start up (when static protections are not as effective), but before the PVM assumes control and begins applying protections to the application. If successful, the attack disengages the protective PVM and disables the run-time protections.

To craft a successful replacement attack against PVM-protected applications, certain requirements need to be met:

- The attacker must be able to locate the entry function (EP) of the protective PVM in P_V. The entry function is defined as the function of the PVM which initiates software virtualization. The entry function often takes the starting address location of P's code as an argument.
- The attacker must be aware of the guest application's instruction set architecture. The code of the guest application P, is typically obscured using a secret ISA or encryption. To analyze and run P after the protective PVM has been disabled, the attacker needs to be cognizant of the ISA, which involves either analyzing and understanding the secret ISA, or extracting the key from the binary.

Section 8 discusses these requirements in more detail, including heuristics that the attacker can employ to obtain the required information.

The attack occurs in two stages. In the first stage, the attack PVM has to be extended to decode the protected application, which involves understanding the guest ISA. If the ISA is encrypted, the decryption keys and algorithms must also be obtained and used to further extend the attack PVM by including the decrytion algorithm and keys. Details on deciphering the guest application's ISA are given in Section 8.2.

Figure 3 illustrates the second stage of the attack on P_V. In Figure 3(a), the attacker invokes P_V, under a *code introspection framework* (CIF), observing instructions as they execute. Well known examples of CIFs include Pin [33] and QEMU [3]. The attacker modifies the CIF to locate the call to the entry function of the protective PVM.

The initialization routine then proceeds to prepare the PVM's internal structures. As the entry function of the protective PVM is invoked, the CIF intercepts this call and extracts the start address, depicted in Figure 3(b). Details on identifying the PVM's entry function are given in Section 8.1

The CIF then proceeds to load and initialize the attack PVM, shown in Figure 3(c). The CIF then invokes this attack PVM with P's start address which has been extracted from the initial call.

Thus, P now runs under the mediation of the attack PVM (shown in Figure 3(d)). The protective PVM is circumvented and fails to provide dynamic protection to P. The attack PVM can be used to perform tasks that helps the attacker understand P (e.g., dump information, identify function locations, trace instructions, etc.).

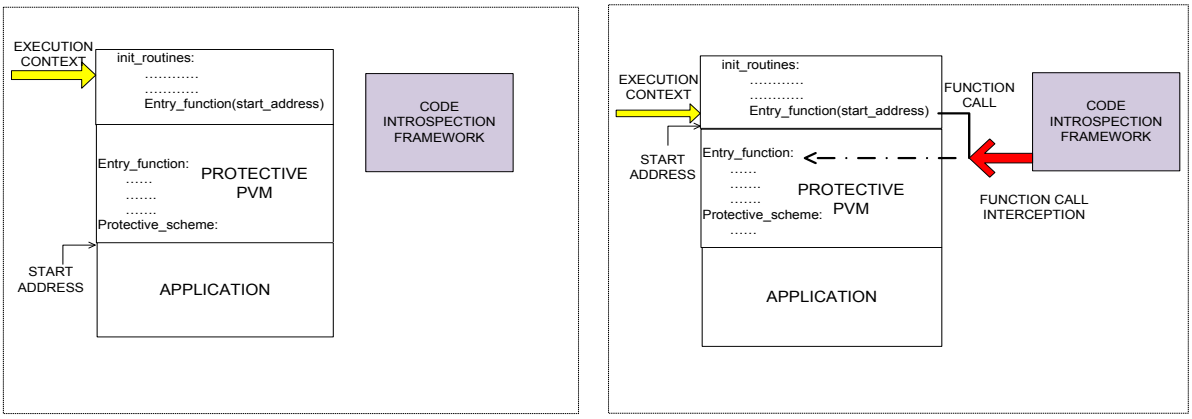
In Section 6, we describe two proof-of-concept implementations that use the approach just described. The first prototype makes use of a widely used CIF, Pin, to replace the protective PVM with an attack PVM and execute the guest application. The second uses a modified architectural simulator, which performs code introspection as well as virtualization.

6. Case Study

This section provides a comprehensive case study of the replacement attack methodology. It describes, Strata, the protective virtual machine, and the creation of the protected application, P_V. The target application was chosen from the integer benchmarks of the SPEC CPU2000 suite. The benchmarks were selected as examples of typical applications and they are commonly used to measure runtime performance. These benchmarks range from a few thousands lines of code to hundreds of thousands of lines, and perform various tasks. Thus, these benchmarks present a wide range of code size and functionality to validate our ideas. For the purposes of this discussion, we focus on the *256.bzip2* benchmark as the target guest application, P. *256.bzip2* is a modified version of the bzip compression program, designed to evaluate CPU performance. All our tests were carried out on the Intel x86 32-bit platform running Linux OS. All the components were initially compiled using *gcc*.

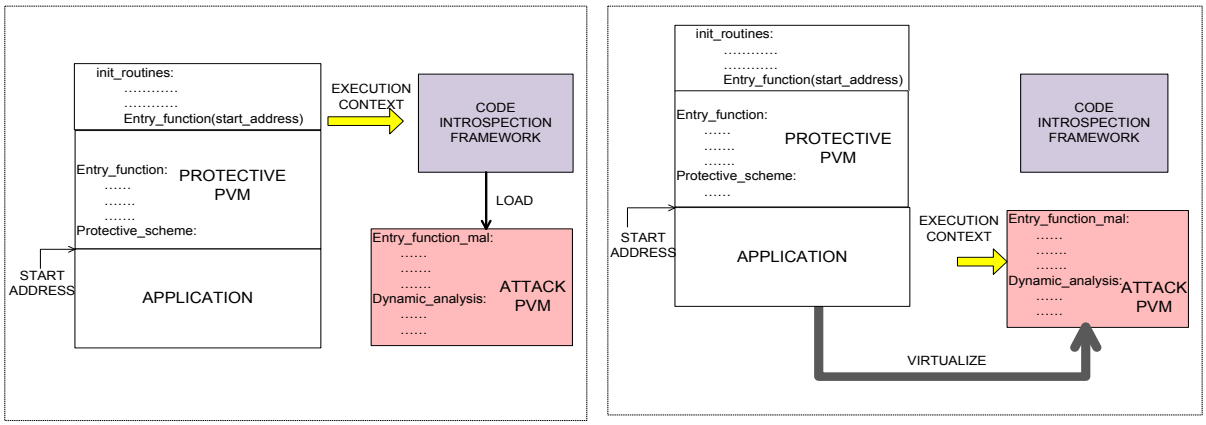
6.1 Dynamic binary translation

Section 2 gave a high-level overview of process-level virtualization. Although process-level virtualization systems are complex in nature, they are all variations of the *decode-dispatch* approach [46]. *Decode-dispatch* systems consists of a main loop that iterates through three phases for every guest application instruction: *decode, dispatch* and *execute*. The decode phase fetches the opcode for the instruction. The dispatch phase uses this information to invoke appropriate handling routines. The execute phase then fetches the operands and proceeds to invoke the corresponding



(a) The virtualized application, P_V, is run under an introspection framework.

(b) The CIF intercepts the call to the entry function of the protective PVM.



(c) The CIF proceeds to load an attack PVM.

(d) Control is then transferred to the entry function of the attack PVM, which proceeds to run P without any dynamic protections.

Figure 3. Steps illustrating the attack methodology on virtualized applications.

handling routine implemented on the host. Performance overhead can become an issue with *decode-dispatch* systems in their basic form. One of the requirements of using virtualization as a platform for security is low run-time overhead. This case study focuses on process-level virtualization based on *dynamic binary translation* [45], which involves performing just-in-time, on-demand conversion of guest application code blocks into executable instructions for the host machine and caching them for subsequent use. A number of efficient, low-overhead dynamic binary translation tools have been created, including DynamoRIO [7], HDTrans [47], Pin [33], PTLSim [55], and Strata [43].

The protective PVM in our case study is implemented using the Strata binary translator [42, 43]. Figure 4 illustrates the mechanism of Strata and the dynamic protection techniques. At program start up, Strata gains control of execution, saves the current execution context (i.e., current PC, register values, conditional codes, etc.), and starts fetching, decoding and translating instructions from the P's start address. This process continues until an end-of-translation condition is satisfied. The translator restores context and proceeds to transfer control to the newly translated block. After the block completes execution, control transfers back to the translator, and it begins translation at the next address.

The translated code blocks, instead of being disposed of after execution, are cached in memory (called the code cache) [7]. Strata initially searches the cache before attempting translation. If the block is found, control will be directed to the cached block. This process of caching significantly reduces the cost of translation. Numerous other techniques have also been proposed for reducing performance overhead of binary translators [24, 25].

Strata is configured to enhance run-time security. For example, during translation, it introduces dead code of random size in the virtualized code blocks. Thus, the layout of the code cache will be different for each program run. This technique makes it harder to launch automated attacks, since the actual location of the translated code varies from run-to-run.

The code cache is also reinforced to hamper analysis [20]. One such reinforcement technique is periodic cache flushing, in which the translated code is deleted from the cache at regular intervals. All the code cache snapshots must be collated to reconstruct the whole application. Strata can use code metamorphosis [6] so that the same application blocks are translated into different forms after each flush. Flushing also ensures that translated code reappears at different locations during execution. Such dynamism makes it harder to locate and remove protections.

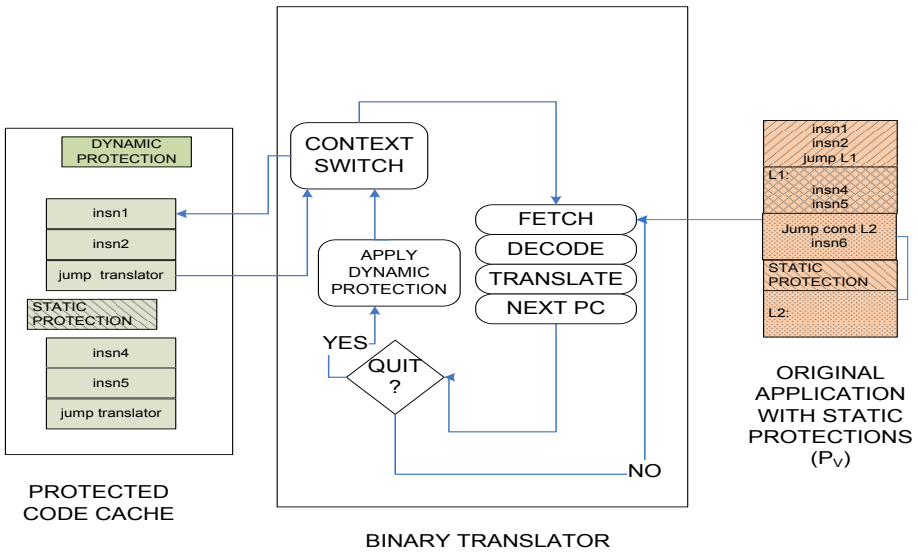


Figure 4. Strata Virtual Machine modified to apply dynamic protections.

6.2 Protected software creation

The protected virtualized application was created using a link-time binary rewriting tool, *Diablo* [16]. Diablo reads in all the object files and libraries constituting the application and creates an internal representation. It provides an API to modify and augment this representation. The modified representation can then be written out as an executable file. We modified Diablo to apply several protection techniques to the application. For example, *checksumming guards*[1] are placed in code regions for both *256.bzip2* and Strata. At run time, the guards present in *256.bzip2* safeguard Strata and vice versa, creating a cyclical level of verification. Diablo was also adapted to interleave the code of the PVM and the guest application, making static analysis harder. Figure 5(a) shows the layout of the virtualized application's binary after normal compilation and linking. A knowledgeable adversary could possibly analyze and identify the individual code regions and extract useful information. Therefore, during linking, Diablo shuffles code blocks of *256.bzip2* with Strata's code blocks using random permutation, resulting in a layout depicted in Figure 5(b). Dissecting the permuted code using static mechanisms is equivalent to the problem of separating code from data, which is unsolvable in the general case [26]. The application code blocks were encrypted using AES.

6.3 Attack implementations

This section describes two implementations of the attack methodology that renders the application, P, vulnerable to analysis and subsequent attack based on information obtained from that analysis. The first proof-of-concept uses a dynamic instrumentation framework (Pin) to replace Strata with an attack PVM (built using HDTrans that we extended to perform AES decryption). The second implementation uses an architectural simulator, PTLsim [55], as both the code introspection framework and the attack PVM. While we use these particular tools to demonstrate the methodology, any similar tools would suffice.

[1] A checksumming guard is a small sequence of instructions that verifies at run time that the checksum over a range of the application's instructions matches the checksum calculated during software creation [10]

6.3.1 Attack using a dynamic binary translator

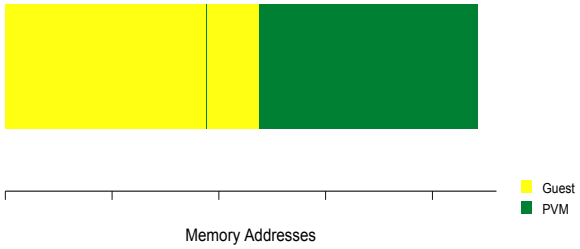
This prototype uses Intel's run-time binary instrumentation framework, Pin [33], to replace Strata with another binary translator, HDTrans [47]. Pin offers a rich API to dynamically inspect and modify the instrumented application's original instructions. The instrumentation functionality is implemented in a module called a Pintool. At run time, the Pin framework takes as input the Pintool and the target software, and performs the necessary instrumentation.

Because the protected application is encrypted, we must first locate the decryption routine in the protective PVM and extend the attack VM to use the same algorithm. The cryptographic primitives are located in the PVM which is not as strongly protected as the application, enabling easier analysis. Techniques have been proposed which can automatically infer these cryptographic primitives from binary code [8, 21, 31]. These schemes involve profiling the virtualized application and analyzing the trace to locate the cryptographic primitives. We successfully used Gröbert's technique to identify the underlying algorithm (AES) and extracted the key [21]. HDTrans was subsequently modified to use AES decryption on the application code blocks prior to translation. Section 8.2 describes techniques that can be used to obtain cryptographic information in greater detail.

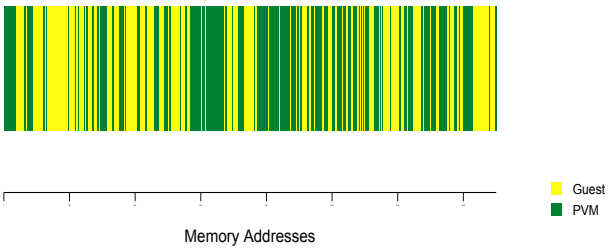
The Pintool, which implements the attack, operates as follows: It starts by loading and starting execution of the protected application. As execution proceeds, the Pintool watches for the entry point function of the protective PVM. In the case of Strata, the call to entry point function is preceded by the following code sequence:

```
pop %eax
sub 0x1c, %esp
pusha
pushf
push %eax ; contains application start address
push <address>  ; return address
jmp <address>   ; jump to entry point function
```

When the entry point function is invoked, the Pintool extracts the application's start address from its argument list. It then dynamically loads the extended HDTrans, proceeds to initialize HDTrans,



(a) Layout of the code regions of the virtualized application, as produced by standard compilation process. The PVM and P's code regions are clearly distinct.



(b) Layout of the code regions after the instruction blocks have been shuffled.

Figure 5. Layout of the text section of the virtualized *256.bzip2* application. The code blocks from the VM and the application have been randomly interleaved, making static separation harder.

and transfers the application start address to HDTrans. Thus, HDTrans takes control of the protected application and Strata never executes. The application can now be analyzed in any number of ways. We modified HDTrans to dump all the executed instructions to disk.

The attack essentially disables checksumming guards from verifying code integrity. Guards located in Strata are never invoked, whereas guards present in P continue to verify the integrity of P and Strata which remain unchanged. At this point, P's code is available for analysis.

6.3.2 Attack using an architectural simulator

The second prototype for the attack uses the PTLSim architectural simulator [55]. In this implementation, PTLSim acts as the instrumentation framework as well as the attack PVM. PTLsim models a modern superscalar Intel x86 compatible processor core along with the complete cache hierarchy, memory subsystem and supporting hardware devices. It models all the major components of a modern out-of-order processor, including the various pipeline stages, functional units and register set. PTLsim supports the full x86-64 instruction set along with all the extensions (SSE, SSSE, etc.). More details of the simulator can be found in the user's manual.

The Intel x86 ISA is a two-operand CISC ISA, however PTLSim does not simulate these instructions directly. Instead, each x86 instruction is first translated into a series of RISC-like micro-operations (`uops`). To further improve efficiency, PTLSim maintains a local cache containing the program ordered translated `uop` sequence for the previously decoded basic blocks in the program.

The attack proceeds as follows: The cryptographic primitives are obtained as in Section 6.3.1, and PTLSim is extended to decrypt instructions after fetching them from memory. At load time, PTLSim initializes its internal data structures and reads in P_V's binary file. The fetch stage of PTLSim accesses instructions from the memory address pointed to by its program counter, called the Virtual Program Counter (VPC). We modified the fetch stage to check for the instruction sequence (illustrated previously in Section 6.3.1) denoting Strata's entry function. Once the fetch stage recognizes the entry function, the simulator retrieves its arguments, which contain the start address of the application code. The simulator then discards its current instruction, waits for the pipeline to empty, and then proceeds to fetch instructions from the retrieved application start address. The simulator decrypts the instruction using the extracted key before decoding it into its constituent `uops`. In this way, Strata never executes and PTLSim is able to fetch and simulate P's instructions directly.

As with the previous prototype, checksumming guards fail to provide adequate protection. The guards only check the original code, which is never executed. We can analyze P in PTLSim's local cache and modify it, if desired.

7. Implications of the Attack

This section discusses how the VM-replacement methodology facilitates analysis and reverse engineering of PVM-protected applications. The first step of any attack involves obtaining a basic understanding of the application. Useful information in program understanding and analysis is the control flow graph (CFG). The CFG obtained from the on-disk binary contains the superset of all possible paths through the program. Obtaining the CFG from the binary in the presence of static protections can be computationally very expensive [49]. As such, attackers have increasingly focused on run-time techniques to obtain the CFG. Although CFGs obtained dynamically can be incomplete, they still provide the attacker with useful information about the application. PVMs provide protection by making dynamic CFG construction and analysis highly resource- and time-intensive tasks. Therefore, to successfully reverse engineer the application, the protective PVM must be replaced using the methodology discussed in this work.

To demonstrate this point, we studied dynamic reverse engineering schemes that have been shown to be successful in attacking software [34, 50], and compared their effectiveness in the presence of a protective PVM. Typically, these techniques involve instrumenting the protected application to obtain the instruction trace. The trace is analyzed to identify individual basic blocks. Consequently, control flow analysis is performed to obtain the dynamic CFG of the application. The attacker then performs profiling of various structures, such as basic blocks and procedure calls, to isolate relevant portions of the code. For example, Madou et al. used basic block execution frequency and in-degree of functions to identify a watermarking function [34]. Similarly, Udupa et al. used edge profiling to identify and remove unnecessary edges from the static CFG [50].

To show that the protective PVM must be replaced, we explored the applicability of such profiling techniques on three different runtime scenarios:

- The application executing without any protections (No protection).
- The application executing in the presence of a protective PVM (Protected).

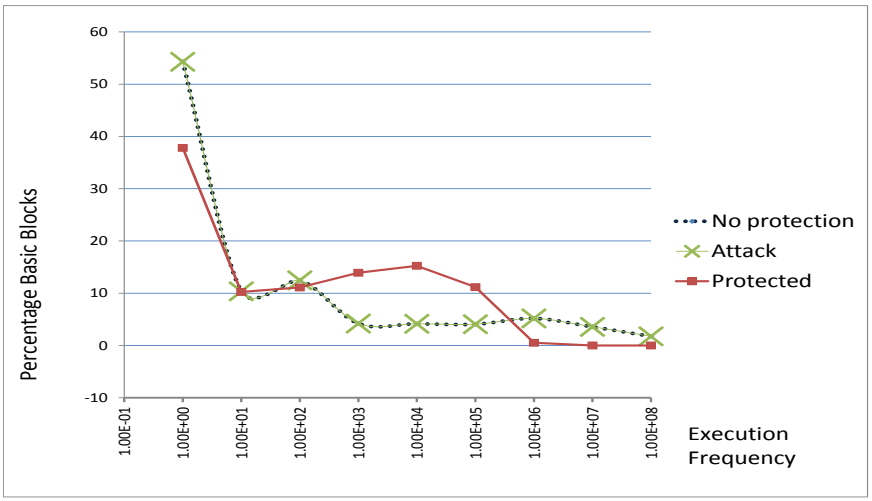


Figure 6. Execution frequencies for the application blocks under the 3 run-time scenarios (No protection, Protected, and Attack). The periodic flushing and retranslation of the application's code blocks by the protective VM drastically changes the execution frequency characteristics. Under the control of the VM, blocks no longer execute at very high frequencies (10^7), instead substantially more blocks execute at intermediate rates (10^3 and 10^5), forcing the attacker to expand their search space. Replacing the protective VM restores the original execution characteristics.

- The protected application that has been subjected to a PVM-replacement attack, i.e., the application is running under the control of a compromised PVM (Attack).

To facilitate collection of application code blocks that have been virtualized, the application was run under an instrumentation framework in all three scenarios. In the following discussion, we refer to such blocks as *dynamic blocks*. The dynamic blocks are identified based on their starting virtual address. The protective PVM was also modified to generate the mapping between on-disk application code blocks and translated blocks. This modification was performed only for the purposes of this study and would not typically be available to the attacker. The goal of this study was to demonstrate that the replacement methodology exposes the original run-time characteristics of the application that have been obfuscated by the PVM, facilitating program analysis and understanding.

We began by comparing instruction trace generation and block analysis across the three scenarios. On comparison, we observed that packaging a protective PVM with the application makes analysis of the dynamic trace and CFG generation much harder. First, the periodic flushing and retranslation of application code increased the number of individual basic blocks manifold. In the case study involving *256.bzip2*, the number of dynamic code blocks increased from around 3.7K for the unprotected run, to more than 160K when the application was subjected to PVM protection. Similarly, the number of distinct CFG edges rose from 6.4K to 290K. Although a large number of these dynamic blocks originate from the same application blocks, the PVM can employ techniques such as code polymorphism, instruction rescheduling and dead-code insertion to make code blocks appear diverse [6]. Thus, the attacker would have to perform control and data flow analysis on a much larger instruction trace. Employing the replacement attack enables the attacker to obtain the original instructions from the application and reduces the search space significantly.

The periodic flushing and randomization of the VM's code cache alter a number of dynamic characteristics of the application, such as block execution frequency, and in and out degrees of the CFG nodes. Figure 6 shows the execution frequency of the dynamic blocks in the three scenarios. When the application was run with no protections, we observed that there were a few code blocks which execute very frequently (of the order 10^7). An attacker would initially focus on reverse engineering these blocks, as they are on the hot paths of the application. Madou et al. used this heuristic to locate the watermarking function [34]. Running the application under the control of a protective PVM obfuscates such blocks due to periodically flushing and retranslation to different locations. The execution frequency for the protected application show that there are no longer blocks which execute as frequently (i.e., no blocks with an execution frequency over 10^7). Instead, there are now more code blocks executing at a lower frequency (e.g., between 10^2 and 10^5). For example, 15% of the blocks execute at least 10^4 times when running under the control of a protective PVM, as compared to just 4% in the unprotected run. Thus, there are no obvious candidate blocks where the attacker could initiate analysis. The attacker will have to increase the search space to find the hot paths for the application.

The PVM also provides misleading information to the attacker. In the three scenarios mentioned above, we ranked all the code blocks based on their execution frequency. Rank 1 was assigned to the most frequently executing block. Table 2 shows the top-ten most frequently executing blocks when the application is run without any protections. Traditionally, an attacker would focus on analyzing these blocks first. Column 2 displays the ranking of these blocks when the application is run under the control of the PVM. For example, the most frequently executing application block in the unprotected run, appears at the 121st position when the application is run under the protective PVM. Thus, the PVM is able to reorder the blocks based on execution frequency. We observed

Application Address	Rank (No Protection)	Rank (Protected)	Rank (Attack)
0x8048830	1	121	1
0x804ac3c	2	45	2
0x804ae1b	3	13	3
0x80507d4	4	9	4
0x80507d9	5	173	5
0x80507c0	6	18	6
0x80507fa	7	29	7
0x805082c	8	351	8
0x8050810	9	139	9
0x804a750	10	779	10

Table 2. Original application addresses of the top-10 most frequently executing blocks in the unprotected run, with their corresponding rank when run under the protection of a PVM. The standard deviation for these blocks in the protected run comes to 239, indicating a very high degree of variability. Consequently, more effort will be required to locate the blocks. A successful replacement attack restores the rankings.

Application Address	Rank (No Protection)	Rank (Protected)	Rank (Attack)
0x804bec0	162	1	162
0x804ae21	17	2	17
0x804ac74	36	3	36
0x804bed3	21	4	21
0x804a7a0	42	5	42
0x804abaf	164	6	164
0x804a81a	88	7	88
0x804a99b	126	8	126
0x80507d4	4	9	4
0x804a7ca	63	10	63

Table 3. Original application addresses of the top 10 most frequently executing blocks when the application is run under the protection of the PVM, along with their corresponding rank when the application runs unprotected, and when it is subjected to the replacement attack.

similar reordering when rankings were based on the in degree of the code blocks. As indicated by column 4 in Table 2, replacing the protective PVM restores the original dynamic properties of the application. Table 3 shows the list of the ten most frequently executed blocks when the application is run under the protection of the PVM, along with their corresponding ranks in the unprotected run. Therefore, this study shows that frequency analysis is not useful to the attacker in the presence of virtualization. The critical information (in this case, the ranking based on frequency) is dispersed by the protective PVM, making it difficult for the adversary to locate and exploit it.

Finally, the constant shifting of code for the application makes it difficult to detect the execution location. We ran the PVM-protected application ten times and observed that application blocks were translated to different code cache addresses each time. Therefore, even if the adversary is able to identify critical code (i.e., the address of a relevant function) in the on-disk binary, that information is of no use at run time since there is no *a priori* knowledge about the final location of the translated code. For example, in Madou et al.'s case study, once the watermarking function was determined; the application was run under the control of a debugger and the control flow changed to circumvent the function using breakpoints. This technique will not work in the presence of a protective PVM, as code is repositioned continually. Thus, to effectively analyze the application under a debugger, the PVM has to be replaced.

Therefore, PVMs have the potential to provide strong protection against dynamic analysis on software applications. PVMs can greatly increase the search space for the attacker, provide misleading run-time information and continuously relocate critical code, making dynamic analysis exceedingly difficult to accomplish. There has been research which aims to reverse engineer PVM-protected applications, by identifying code belonging to the VM in the execution trace [15, 44]. However, such methodologies usually involve performing complex analysis on the trace information and are targeted towards applications which are typically small in size (i.e., a few hundred instructions e.g., malware). These methodologies fail to provide satisfying results when applied to VM-protected applications as they are unable to process the complex data and control flow typically associated with large applications. Thus, the replacement attack methodology is pivotal to the success of reverse engineering PVM-protected applications. Once the protective PVM has been replaced, the application can be analyzed and its true characteristics studied without any obstruction.

8. Discussion

Section 6 described two implementations of an attack that seeks to remove the protective PVM from a virtualized application, P_V. Both implementations were crafted using freely-available tools, and resulted in the guest application, P, running with the added protections disabled. Checksumming guards inserted into P_V failed to prevent the replacement. To successfully orchestrate the attack, some prior information about P_V is required. In this section, we discuss some heuristics the attacker can use to determine this information.

8.1 Determining PVM entry function

To launch a successful attack, the location of the entry function to the protective PVM must be determined. The attack PVM intercepts any calls to this function, as one of the arguments consists of the start address to P. To obtain this location, the attacker can inspect P_V for distinctive instruction sequences that indicate the location of the entry function.

For example, as we described in Section 6.1, prior to initialization, the PVM saves the current application's context. Upon initialization, the PVM restores the application's context. On 32-bit Intel x86 platforms, the instructions `pusha` and `pushf` are commonly used by dynamic translators to save state. Section 6.3.1 displayed the instruction sequence used by Strata to save state, which contains these two instructions. Dynamo-RIO and HDTrans also use these instructions to store register and flag values prior to initiating translation[2]. Investigation into the C benchmarks of the SPEC CPU2000 suite, compiled using standard flags, revealed that none of the application binaries contained these instructions. Consequently, an adversary can use the presence of these instructions to help identify potential entry points into the PVM. Because of the unique actions of the PVM, simple examination of these potential entry points can determine the actual entry point.

The attacker can also use information flow analysis to determine the entry point of a PVM. Most compilers place code and data in separate sections in the binary file. Data-read accesses into the code section are likely to be from the PVM. Thus, using taint analysis on this data and backtracing where the data location was

[2] We were not able to verify the use of these instructions in Pin as its source code is not publicly available.

determined will enable the attacker to determine the entry function of the PVM. Since the PVM initialization typically occurs very early, the attacker will only have to analyze a smaller amount of code to determine the entry function. This code is not subjected to any dynamism, making analysis easier.

8.2 Determining the ISA of the guest application

Another requirement to use a replacement attack is the identification of the ISA of the guest application, P. Traditionally, P's code has been protected by obscurity [1, 51] or encryption [20]. The attacker has to analyze the on-disk binary to obtain information required for determining the ISA, which is then used to configure the attack PVM. This section discusses some heuristics that the attacker can utilize to obtain the relevant information.

Rolles et al. have done extensive work in investigating ISAs used by obfuscation tools such as VMProtect and Themida [41]. The semantics of the ISA are not released to the public, providing security through obscurity. At the time protections are applied, P's instructions are converted to a custom ISA chosen from a set of template ISAs, which is then interpreted at run time by a PVM designed specifically for that ISA. These ISAs are RISC-like, lacking many of the complex features of traditional ISAs like Intel x86. Since these tools derive the final ISA from a template, the instruction sequences of two different protected binaries will have many similarities. This fact makes the analysis of the syntax and semantics more tractable. Further, parts of the x86 instruction set such as the SIMD instructions are not virtualized by VMProtect.

The guest ISA can also be protected via encryption. Manually analyzing and reverse-engineering cryptographic keys and routines can be an arduous task. Recently, researchers have designed techniques that facilitate automatic identification and extraction of cryptographic routines. Gröbert et al. have presented a novel approach to identify cryptographic routines and keys in an encrypted program [21]. Their techniques involves profiling the application and applying heuristics to detect the cryptographic operations. Some of the heuristics proposed by the authors include excessive use of arithmetic functions, loops and investigating the data flow between intermediate variables across multiple runs. This technique is able to identify common encryption algorithms, such as AES and DES, and extracts the key as well.

Even if the application is protected by a proprietary encryption algorithm, researchers have devised techniques to isolate and extract this information from the binary. Caballero et al. have designed a technique to automatically identify code fragments from executable files, so that they are self-contained and can be reused by external code (called *binary code reutilization*) [8]. They successfully applied this technique to identify and extract cryptographic routines from a set of malware files. Similary, Leder et al. examined data in-flow and out-flow in memory buffers, to isolate cryptographic functions [31]. Therefore, such identification techniques can be applied to the protective PVM to obtain the decryption routines, and consequently, used to configure the attack PVM.

Decryption key management is also an issue for the protective PVM. The attacker can use dynamic analysis techniques to extract the decryption key. Halderman et al. point out that modern DRAMS retain their contents for a significant amount of time and an attacker could locate and exploit these keys to analyze encrypted data [22]. Skype is a popular VoIP tool which uses encryption as a tool to hamper static analysis. Biondi et al. were able to decipher Skype's code by obtaining its decryption key from memory [5]. Techniques, such as white-box cryptography, were proposed to improve key management in encrypted systems [12]. However, researchers have since developed solutions to extract the key from such systems [4]. Once the key is available, deciphering the encrypted ISA is straightforward regardless of the the strength of the encryption algorithm.

Therefore, obscuring the ISA fails to adequately protect the guest application from analysis. Previous work has shown that the attacker can analyze such ISAs using reasonable time and effort [5, 41]. Once the the ISA is known, the attacker can successfully replace the protective PVM.

9. Related Work

Software systems are increasingly being used to control critical systems. As such, much research has been performed in the area of software security. A number of techniques have been devised to hamper static analysis of applications. Linn et al. described novel techniques to defeat disassembly of code [32]. Wang et al. proposed using indirect branches to obfuscate the control flow of the application [53]. Chang et al. explored the idea of using a network of self-verifying checksumming code, called guards, to maintain the integrity of code [10]. Collberg et al. studied various techniques to obscure code [13, 14]. However, most of these techniques can be overcome using dynamic analysis. Researchers have also investigated using run-time techniques to obscure code, including using self-modifying code to obfuscate instructions [28]. Code encryption is also useful technique to hamper analysis. The code can be decrypted by hardware (i.e., a secure co-processor) [56], which can add substantially to the costs. Software decryption is less expensive solution [9], but is vulnerable to dynamic analysis.

Software virtualization is an active area of research with implications in software security, both at the system level and at the process level. Process-level virtual machines, such as Pin [33], Strata [43] and HDTrans [47], have been used code optimization [7], instrumentation [33], and in particular, security [20, 23, 40].

Researchers have used process-level virtualization to improve the resistance of code to analysis, both to improve the security of programs [1, 20] and more recently, to obfuscate malicious software, like viruses, trojans, etc. (called *malware*). Commercial PVMs have been created recently, such as VMProtect [51], Themida [36], with a special focus on program security. and Code Virtualizer [37]. These tools follow the conventional interpretation model [46] and use obscure ISAs to protect applications from analysis.

There has been recent work aimed at deobfuscating malware protected by virtual machines. Coogan et al. have developed a scheme which focuses on identifying the flow of values to system call instructions used by the malware, thereby obviating the need to analyze VM instructions [15]. However, their scheme does not produce encouraging results when applied to complex applications. Sharif et al. have also devised techniques to automatically reverse engineer malware that is obfuscated by such PVMs [44]. Although their goals are somewhat similar to this work, there are marked differences. The authors target their work towards software (typically malware) that is transformed to random ISA's and then interpreted on-the-fly at run time. In their model, the PVM does not generate native code but dispatches application instructions to appropriate handler routines (e.g., using a switch table). In particular, their work is targeting applications which do not have a performance constraint, which is often true of malware. The authors state that their techniques do not apply to programs protected by binary translation systems due to the complex run-time behavior of such systems. Their work is based on the assumption that the interpreters do not perform any dynamic transformations or generate new code, making control and data flow analysis easier to perform on the generated trace. Finally, their approach is focused towards reverse engineering malware code, which is typically smaller in size (of the order of a few thousands lines of code). Our attack focuses on real applications that are protected by PVMs. The proof-of-concept

prototypes have been tested on real-life applications, consisting of hundreds of thousands of lines of code.

System-level virtual machines have been utilized to provide secure solutions as well. The Terra system implements a trusted virtual machine monitor which can be used to create closed-box platforms where the developer can tailor the software stack to meet security requirements [19]. However, it requires hardware support to validate the software stack. Chen et al. discuss Overshadow, a system that cryptographically isolates an application inside a VMM from the guest OS it is running on. This system offers another layer of tamper resistance, even in the case of total OS compromise [11].

10. Conclusions

Process-level virtual machines (PVMs) are increasingly being used to safeguard applications from reverse engineering. PVMs can continuously transform code during execution, as well as introduce new protection techniques. This scenario makes it hard for the adversary to analyze and understand the application. Traditionally, the virtualization layer has always been decoupled from the guest application. In this paper, we have presented a novel attack scheme that replaces the protective PVM packaged with an application at run time, leaving it vulnerable to analysis. This attack methodology is successful due to the lack of strong dynamic coupling between the application and the protective PVM instance. Our methodology illustrates that this lack of dynamic cohesion leads to serious shortcomings when used in software protection. We described an extensive case study analyzing the protection mechanisms in a real VM-protected application and discussed two prototypes of this attack constructed using freely available tools. The results from this work show that there is an urgent need to reassess the protection properties of PVMs, and to develop techniques that can thwart this attack methodology.

Acknowledgments

This research is supported by National Science Foundation grants CNS-00716446 and CCF-0811689 and the Air Force Research Laboratory (AFRL) under contract FA8650-10-C-7025. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL or the U.S. Government.

References

[1] ANCKAERT, B., JAKUBOWSKI, M., AND VENKATESAN, R. Proteus: virtualization for diversified tamper-resistance. In *DRM '06: Proceedings of the ACM Workshop on Digital Rights Management* (New York, NY, USA, 2006), ACM Press, pp. 47–58.

[2] APPLE. Mac OS X ABI Mach-o file format reference, 2009.

[3] BELLARD, F. QEMU, a fast and portable dynamic translator. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2005), USENIX Association, pp. 41–41.

[4] BILLET, O., GILBERT, H., AND ECH-CHATBI, C. Cryptanalysis of a white box AES implementation. In *Selected Areas in Cryptography* (Hiedelberg, 2004), Springer-Verlag, pp. 227–240.

[5] BIONDI, P., AND FABRICE, D. Silver needle in the skype. In *Black Hat Europe* (Amsterdam, the Netherlands, 2006).

[6] BORELLO, J.-M., AND MÈ, L. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology 4* (2008), 211–220. 10.1007/s11416-008-0084-2.

[7] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization* (Los Alamitos, CA, USA, 2003), IEEE Computer Society, pp. 265–275.

[8] CABALLERO, J., JOHNSON, N. M., MCCAMANT, S., AND SONG, D. Binary code extraction and interface identification for security applications. In *NDSS '10: Proceedings of the Network and Distributed System Security Symposium* (2010), The Internet Society.

[9] CAPPAERT, J., PRENEEL, B., ANCKAERT, B., MADOU, M., AND DE BOSSCHERE, K. Towards tamper resistant code encryption: practice and experience. In *ISPEC'08: Proceedings of the 4th International Conference on Information Security Practice and Experience* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 86–100.

[10] CHANG, H., AND ATALLAH, M. Protecting software code by guards. In *Proceedings of the ACM Workshop on Security and Privacy in Digital Rights Management* (2000), pp. 160–175.

[11] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In *ASPLOS XIII: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2008), ACM Press, pp. 2–13.

[12] CHOW, S., EISEN, P. A., JOHNSON, H., AND OORSCHOT, P. C. V. White-box cryptography and an AES implementation. In *SAC '02: Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography* (London, UK, 2003), Springer-Verlag, pp. 250–270.

[13] COLLBERG, C., THOMBORSON, C., AND LOW, D. A taxonomy of obfuscating transformations. *University of Auckland Technical Report* (1997), 170.

[14] COLLBERG, C., THOMBORSON, C., AND LOW, D. Manufacturing cheap, resilient and stealthy opaque constructs. In *POPL'98:Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1998), ACM Press, pp. 184–196.

[15] COOGAN, K., LU, G., AND DEBRAY, S. Deobfuscating virtualization-obfuscated software: A semantics-based approach. *CCS '11: Proceedings of the ACM Conference on Computer and Communications Security* (October 2011). To appear.

[16] DE BUS, B., DE SUTTER, B., VAN PUT, L., CHANET, D., AND DE BOSSCHERE, K. Link-time optimization of ARM binaries. In *LCTES'04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Washington D.C., U.S.A, 7 2004), ACM Press, pp. 211–220.

[17] DEHNERT, J. C., GRANT, B. K., BANNING, J. P., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSON, J. The Transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *CGO'03: Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 15–24.

[18] EAGLE, C. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. No Starch Press, San Francisco, CA, USA, 2008.

[19] GARFINKEL, T., PFAFF, B., CHOW, J., ROSENBLUM, M., AND BONEH, D. Terra: a virtual machine-based platform for trusted computing. In *SOSP'03: Proceedings of the 19th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), ACM Press, pp. 193–206.

[20] GHOSH, S., HISER, J. D., AND DAVIDSON, J. W. A secure and robust approach to software tamper resistance. In *IH '10: Proceedings of the 12th International Conference on Information Hiding* (Berlin, Heidelberg, 2010), Springer-Verlag, pp. 33–47.

[21] GRÖBERT, F., WILLEMS, C., AND HOLZ, T. Automatic identification of cryptographic primitives in binary programs. In *RAID '11: Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection* (London, UK, 2011), Springer-Verlag, pp. 45–65.

[22] HALDERMAN, J. A., SCHOEN, S. D., HENINGER, N., CLARKSON, W., PAUL, W., CALANDRINO, J. A., FELDMAN, A. J., APPELBAUM, J., AND FELTEN, E. W. Lest we remember: cold-boot attacks on encryption keys, May 2009.

[23] HISER, J. D., COLEMAN, C. L., CO, M., AND DAVIDSON, J. W. Meds: The memory error detection system. In *ESSoS '09: Proceedings of the 1st International Symposium on Engineering Secure Software and Systems* (Berlin, Heidelberg, 2009), Springer-Verlag, pp. 164–179.

[24] HISER, J. D., WILLIAMS, D., FILIPI, A., DAVIDSON, J. W., AND CHILDERS, B. R. Evaluating fragment construction policies for SDT systems. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments* (New York, NY, USA, 2006), ACM Press, pp. 122–132.

[25] HISER, J. D., WILLIAMS, D., HU, W., DAVIDSON, J. W., MARS, J., AND CHILDERS, B. R. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO'07: Proceedings of the International Symposium on Code Generation and Optimization* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 61–73.

[26] HORSPOOL, R. N., AND MAROVAC, N. An approach to the problem of detranslation of computer programs. *Computer Journal 23*, 3 (1980), 223–229.

[27] HU, W., HISER, J. D., WILLIAMS, D., FILIPI, A., DAVIDSON, J. W., EVANS, D., KNIGHT, J. C., NGUYEN-TUONG, A., AND ROWANHILL, J. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (New York, NY, USA, 2006), ACM Press, pp. 2–12.

[28] KANZAKI, Y., MONDEN, A., NAKAMURA, M., AND MATSUMOTO, K.-I. Exploiting self-modification mechanism for program protection. In *COMPSAC'03: Proceedings of the 27th Annual International Conference on Computer Software and Applications* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 170–176.

[29] KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), ACM Press, pp. 272–280.

[30] KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. Secure execution via program shepherding. In *USENIX'02: Proceedings of the 11th USENIX Security Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 191–206.

[31] LEDER, F., MARTINI, P., AND WICHMANN, A. Finding and extracting crypto routines from malware. In *Proceedings of the IEEE 28th International Performance Computing and Communications Conference (IPCCC)* (Washington, DC,USA, December 2009), IEEE, pp. 394 – 401.

[32] LINN, C., AND DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In *CCS'03: Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)* (Washington D.C., U.S.A, 2003), ACM Press, pp. 290–299.

[33] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2005), ACM Press, pp. 190–200.

[34] MADOU, M., ANCKAERT, B., DE SUTTER, B., AND DE BOSSCHERE, K. Hybrid static-dynamic attacks against software protection mechanisms. In *DRM '05: Proceedings of the 5th ACM workshop on Digital Rights Management* (New York, NY, USA, 2005), ACM Press, pp. 75–82.

[35] MADOU, M., ANCKAERT, B., MOSELEY, P., DEBRAY, S., DE SUTTER, B., AND DE BOSSCHERE, K. Software protection through dynamic code mutation. In *The 6th International Workshop on Information Security Applications (WISA 2005)* (August 2005), vol. LNCS, Springer Verlag.

[36] OREANS TECHNOLOGIES. Themida. `http://oreans.com/themida.php`, 2009.

[37] OREONS TECHNOLOGY. Codevirtualizer. `http://oreans.com/codevirtualizer.php`, 2009.

[38] PAYER, M., AND GROSS, T. R. Fine-grained user-space security through virtualization. In *VEE'11: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2011), ACM Press, pp. 157–168.

[39] POPEK, G. J., AND GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17* (July 1974), 412–421.

[40] PORTOKALIDIS, G., AND KEROMYTIS, A. D. Fast and practical instruction-set randomization for commodity systems. In *ACSAC'10: Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACM Press, pp. 41–48.

[41] ROLLES, R. Unpacking virtualization obfuscators. In *WOOT'09: Proceedings of the 3rd USENIX Conference on Offensive Technologies* (Berkeley, CA, USA, 2009), USENIX Association, pp. 1–10.

[42] SCOTT, K., AND DAVIDSON, J. Safe virtual execution using software dynamic translation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference* (Los Alamitos, CA, USA, 2002), IEEE Computer Society, p. 209.

[43] SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J. W., AND SOFFA, M. L. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization* (Washington D.C., U.S.A, 2003), IEEE Computer Society, pp. 36–47.

[44] SHARIF, M., LANZI, A., GIFFIN, J., AND LEE, W. Automatic reverse engineering of malware emulators. In *SP'07: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 94–109.

[45] SITES, R. L., CHERNOFF, A., KIRK, M. B., MARKS, M. P., AND ROBINSON, S. G. Binary translation. *Communcations of the ACM 36* (February 1993), 69–81.

[46] SMITH, J., AND NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[47] SRIDHAR, S., SHAPIRO, J. S., NORTHUP, E., AND BUNGALE, P. P. HDTrans: an open source, low-level dynamic instrumentation system. In *VEE'06: Proceedings of the 2nd International Conference on Virtual Execution Environments* (New York, NY, USA, 2006), ACM, pp. 175–185.

[48] STARFORCE. Starforce crypto. `http://www.star-force.com/`, 2008.

[49] SZOR, P. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[50] UDUPA, S., DEBRAY, S., AND MADOU, M. Deobfuscation: reverse engineering obfuscated code. In *WCRE '05: Proceedings of the International Working Conference on Reverse Engineering* (Los Alamitos, CA, USA, Nov. 2005), vol. 0, IEEE Computer Society, pp. 45–54.

[51] VMPROTECT SOFTWARE. VMProtect. `http://vmpsoft.com/`, 2008.

[52] WANG, C., DAVIDSON, J., HILL, J., AND KNIGHT, J. Protection of software-based survivability mechanisms. In *DSN'01: Proceedings of the International Conference on Dependable Systems and Networks* (Goteborg, Sweden, 2001), IEEE Computer Society, pp. 193–202.

[53] WANG, C., HILL, J., KNIGHT, J., AND DAVIDSON, J. Software tamper resistance: Obstructing static analysis of programs. Tech. rep., Charlottesville, VA, USA, 2000.

[54] YOUNGDALE, E. Kernel korner: The ELF object file format: Introduction. *Linux Journal 1995* (April 1995).

[55] YOURST, M. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *ISPASS'07: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software* (2007), IEEE, pp. 23–34.

[56] ZAMBRENO, J., CHOUDHARY, A., SIMHA, R., NARAHARI, B., AND MEMON, N. SAFE-OPS: An approach to embedded software security. *Transactions on Embedded Computing Systems 4*, 1 (2005), 189–210.