Software Protection for Dynamically-Generated Code

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 Virtual machines (PVMs) often play a crucial role in program protection. In particular, virtualization-based tools like VMProtect and CodeVirtualizer have been shown to provide desirable obfuscation properties (i.e., resistance to disassembly and code analysis). To be efficient, many tools cache frequently-executed code in a code cache. This code is run directly on hardware and consequently may be susceptible to unintended, malicious modification after it has been generated.

To thwart such modifications, this work presents a novel methodology that imparts tamper detection at run time to PVM-protected applications. Our scheme centers around the *run-time* creation of a network of *software knots* (an instruction sequence that checksums portions of the code) to detect tamper. These knots are used to check the integrity of cached code, although our techniques could be applied to check any software-protection properties. Used in conjunction with established static techniques, our solution provides a mechanism for protecting PVM-generated code from modification.

We have implemented a PVM system that automatically inserts code into an application to dynamically generate polymorphic software knots. Our experiments show that PVMs do indeed provide a suitable platform for extending guard protection, without the addition of high overheads to run-time performance and memory. Our evaluations demonstrate that these knots add less than 10% overhead while providing frequent integrity checks.

Keywords Process-level Virtual Machines, Tamper detection, Obfuscation, Polymorphism

1. Introduction

Today, software has become an essential component of many critical systems, e.g., transportation control systems, banking and medical devices, communications systems, etc. Such critical software systems are potential targets by adversaries equipped with advanced reverse engineering tools. Any unauthorized modification could lead to extensive disruption of services and loss of life and property.

A variety of schemes have been developed to protect critical software from unauthorized analysis [1, 2, 8, 23]. However, these obfuscation techniques suffer from a variety of weaknesses.

- Much of the previous research has targeted making the application hard to analyze statically [11, 30]. For example, an opaque predicate is a predicate that is difficult to analyze statically. However, dynamic attacks drastically reduce the effectiveness of such static schemes and provide useful information to enable reverse engineering of protected applications. [32, 46].
- The use of additional hardware is required by some solutions [29]. This extra hardware adds an additional cost that will have to be borne by the end user, and might restrict the software to a particular set of platforms.
- A number of schemes have an impractical overhead constraint or provide only a partial solution. The Proteus system involves overheads between 50X-3500X, which is too high for most applications [1]. Remote tamper-proofing techniques, although widely used among embedded devices [40], require strict Quality-of-Service guarantees [7]. A realistic solution must not incur significant overhead, otherwise developers will be unwilling to deploy such measures on a large scale.

Recently, software developers have turned to process-level virtualization to safeguard applications from analysis. Process-level virtualization involves the introduction of an extra layer of software (called the *process-level virtual machine* or PVM) between the guest application and the host hardware [38]. At run time, the PVM assumes control and mediates the execution of the guest application. The growing use of PVMs for program obfuscation can be attributed to the following reasons:

- Virtualizing a guest application makes static analysis very hard. During software creation, the encoding of the guest application binary is transformed to either a secret

PPREW '13 Jan 26, 2013 Rome, Italy

Instruction Set Architecture (ISA) or encrypted using a secret key. Only the PVM possesses the ability to execute the application. This transformation effectively renders traditional disassemblers and static analyzers useless.

- Even analyzing the run-time instruction trace is harder, because the execution of the PVM and the application is interleaved. Any adversary trying to reverse engineer the application will first have to identify and remove portions of the code belonging to the PVM.
- It is advantageous to have the obfuscation 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 schemes without the need for modification and recompilation. PVMs can be used to provide such a flexible capability.
- As mentioned previously, overhead is an important concern for security researchers. Widespread adoption of security techniques is dependent upon the associated performance and memory penalties. Extensive research has already been done that enables PVMs to incur low overheads [21, 22, 44]. Thus, the efficient implementation of PVMs makes them suitable to be co-opted for research in program obfuscation.

VMs have been used successfully in the area of code obfuscation. A number of commercial products have been designed that provide protection from disassembly and analysis via the use of PVMs, such as VMProtect [47], Code Virtualizer [34], Themida [33]. To maintain efficiency, VMs often use dynamic binary translation and software caching of frequently-executed code [43]. SecureQEMU is a software protection mechanism based on binary translation and caching [25]. It uses a system-level VM, QEMU [4], to run encrypted guest applications. During execution, the guest application instructions are copied to the host OS, decrypted and stored in a software cache. Although the application instructions never appear in plain text in the guest OS, the adversary can attack the host OS and tamper with the code in the cache. There are no protection techniques that safeguard the cached code.

This work presents a novel method that applies tamper protection to applications running under the control of a PVM that uses dynamic binary translation. The basic premise behind this methodology is that the PVM can continually inspect and monitor the guest application code and use software caching of application instructions to speed up execution [31, 39]. At randomly-selected points during execution, the PVM emits dynamic *knots*, that maintain the integrity of translated and cached code. The knots are structurally similar to the checksumming guards, first proposed by Chang and Atallah [8]. Guards are inserted statically and protect the code present in the application binary. Knots, on the other hand, are generated by the VM at run time, and focus on protecting the code that is cached by the PVM.

In combination with traditional static protection mechanisms, our scheme provides the application with a robustly protected environment that is resistant to static and dynamic attacks. The major contributions of out work are as follows:

- This work proposes a novel methodology, in which a PVM introduces program protection techniques at run time. We use knots as the example technique, although the approach can be extended to include other protection mechanisms such as software watermarking.
- In many cases, the virtualization process generates code at run time, which cannot be protected by existing techniques. For example, PVMs implemented using binary translation [42] often generate and store the translated code in a software cache (e.g., Strata [39], Pin [31], HDTrans [44]). The described methodology addresses the protection of such PVM-generated code.
- Using a prototype implementation of the methodology on the Linux platform, we demonstrate that applying dynamic protections is both feasible and can provide increased protection of applications.
- The results indicate that PVMs hold great potential with regard to dynamic program protection. Preliminary results pave the way for further advancements in this area.

The rest of the paper is organized as follows. Section 2 describes the hostile environment in which an application operates. Section 3 gives an overview of process-level virtual machines. The concept of knots is explained in Section 4. The evaluation of these techniques is described in Section 5. Previous work in the area of virtual machine-based security is discussed in Section 7. Finally, our conclusions are presented in Section 8.

2. Threat Model

We begin by describing the trusted environment used to create software applications. The software developer uses numerous, trusted, tools (compilers, optimizers and assemblers) to convert high-level source code or specifications into applications. To help safeguard the application's static image, the developer applies various static code obfuscation and tamper-detection mechanisms to the application. At this point, we assume the application is safe from static-only reverse-engineering or tampering attacks. While such an assumption is likely not reasonable in practice, we make this assumption to focus on dynamic attacks against the application.

The application is then released for public use. An attacker can use various tools (debuggers, simulators and emulators) to run, observe and modify the program in a number of ways. For example, Skype, the popular VoIP tool, was reverse engineered by researchers using a debugger [5].

Even the operating system can be modified to return incorrect information, thereby thwarting any protection features that depend on the OS [50]. Furthermore, hardware can be emulated to return forged results to the application. Consequently, we consider all hardware and other software on the host machine as potentially malicious, and the entire protected application (including any virtual machine distributed with said application) is a target of attack.

In essence, the adversary is mounting a white-box attack. The adversary can inspect, modify, or forge any information in the system. Given enough time and resources, the adversary can succeed in inspecting and making modifications to the program. However, human adversaries have difficulty directly solving problems involving large data sets. As such, they rely on algorithmic solutions to perform various analyzes on the application packages (e.g., program disassembly, determining instruction locations, generating precise flow graphs, etc.) and use that information to disable protection features. Our goal is to make automatic analyses computationally more expensive.

3. Process-level Virtualization

Before describing the concept of knots, we give a brief introduction to software virtualization. A software application is a sequence of instructions that execute on a particular computing system. Virtualization is a software layer 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 [14, 38], or to improve security [6, 15, 27]. 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).

A system VM provides a complete system environment that can support an operating system and associated application programs [43]. Examples of system VMs include VMWare [48] and Xen [3]. System VMs are rarely used to provide software protection for an individual application.

On the other hand, a process-level virtual machine virtualizes a single user-level application [43], and it is more suited to providing software protection. 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. PVMs have been used to mediate the execution a wide-ranging of applications including GUI-based applications, multi-threaded applications, and high-performance server programs. This work focuses on a widely implemented form of process-level virtualization called *dynamic binary translation* (DBT) [42]. Examples of PVMs implemented using DBT include Strata [39], Pin [31] and HDTrans [44].

PVMs operate as follows: during program startup, the PVM assumes control and starts decoding the application's instructions in program order, one basic block at a time. The decoded instructions are then translated to the Instruction Set Architecture (ISA) of the host machine, and cached in software. Control is then transferred to the cached block. After the block executes, control returns to the PVM, which then proceeds to translate the next basic block of the application. The PVM can regularly monitor the application code during this translation process, and therefore, it can serve as a platform for applying protection schemes dynamically.

4. Polymorphic Knots

Knots are small sequences of instructions that checksum application instructions over a range of memory addresses. They are similar to software guards, first proposed by Chang and Atallah [8]. In that work, the authors proposed designing and placing the guards in the application at software-creation time. However, as we described in Section 3, PVMs generate code for the host machine at run-time. Consequently, checksumming guards for the translated code cannot be created post compilation and they remain vulnerable to tamper.

We propose a novel scheme of program protection, in which the PVM generates *knots* during *translation*. At randomly-selected points during the application code translation, the PVM will generate checksumming code that will safeguard translated code that is located in the software cache. The focus of these dynamic knots is to protect the code located in the software cache, but the protection domain can be extended to include the PVM and the guest application as well. At random points during translation, the PVM creates a checksum over randomly chosen range of memory addresses, using a pre-determined hash function, and stores the value. It then creates a sequence of instructions that perform the same operation over the same range of memory and places the block in the software cache. The PVM then continues to translate application instruction blocks as per norm. When control is transferred to the software cache, the knot executes and ensures no modification of the translated block has taken place.

An appropriate strategy for creation and placement of knots is essential to low performance overhead as well as good protection coverage. A trivial strategy for placing knots is to randomly insert them during application execution. However, if the knot is placed in a *hot loop* (i.e., a sequence of code that executes numerous times), run-time performance can be negatively affected. Another strategy involves assigning each block a probability inverse to it's execution count (obtained by profiling the application). When

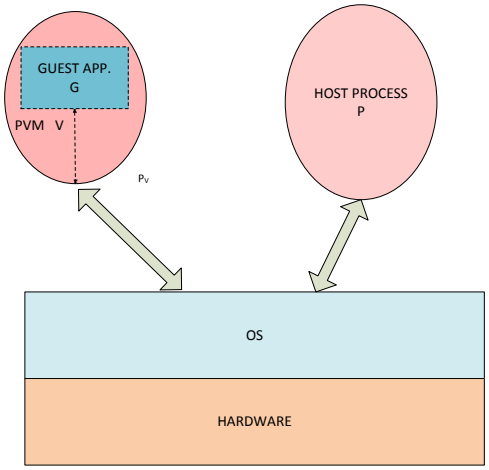

Figure 1. High-level overview of process-level virtualization. The guest application runs under the mediation of the process-level VM, giving an outward appearance of a process that is native to the underlying platform.

implemented, we observed that this strategy led to inconsistent rates of knot execution (i.e., periods of high knot execution followed by periods of little or no knot execution).

Consequently, we decided to use *predicated* knots. Predicated knots can be placed randomly during program execution. However the checking mechanism is only triggered when a predicate is satisfied. If the predicate is not true, the knot does not trigger and normal program execution continues.

An important aspect of the security provided of this scheme is the knot network. Each address range is protected by multiple knots, which in turn are protected by other knots. The connectivity, K, is defined as the number of distinct knots checking a memory location. Associated with the connectivity is the *RangeSize*; the size of the block of code checked by each knot. To increase K without changing the frequency of knot insertion, *RangeSize* is increased to achieve the specified connectivity.

K provides a knob to control the trade-off between security and performance overhead. As K is increased, each byte in the program is protected by more knots, but each knot takes longer to execute because of the larger address range checked by each knot. Section 5 provides measurements of the run-time overhead when K is 7.

4.1 Knot Polymorphism

A standard knot consists of four main components: a preamble, a loop, a checker, and tamper response. To thwart automatic scanners, the PVM uses a database of instructions to construct these components. During knot creation, the PVM chooses random instructions from this database to form the knot. Figure 2 shows code for two such knots created using a random selection of instructions. To further increase the level of obfuscation, the instruction database is placed along with other program data variable.

The knots also do not immediately report a tampering attack. Instead, whenever an attack is detected, control transfers do a different location and the tamper response is delayed till a later time when it is unclear as to the reasons behind its occurrence.

4.2 Fluctuating Protection Network

PVMs can provide a fluctuating execution environment for the application. For example, Ghosh et al. have shown that PVMs can be configured such that the translated code executes from random locations in memory, which cannot be determined *a-priori* [17]. This shifting environment is primarily achieved through the use of periodic software cache *flushing* [20]. As mentioned in Section 3, the PVM stores the translated host machine instructions in a software cache. Over periodic intervals, the PVM is configured to delete this cache and re-translate and store the instructions. We now show that this fluctuation can be used to strengthen knots as well.

Chang et al. postulated that statically inserting a collection of static guards that mutually reinforce each other as well as application code is more resilient to attacks than protection from a single guard [8]. However, once a guard network has been created statically, it cannot be changed and an adversary could be successful in disabling the network. Our work utilizes periodic flushing to create a dynamic network of knots. Each time the cache is flushed, a new network of knots that cyclically reinforce each other will be created. The networks would differ in number of of knots, their locations, and their protection range. They could also differ in their formations, ranging from simple trees to directed graphs with cycles. Compared to current static tamper de-

```
preamble:
        mov edx, -checksum
        mov eax, range_start


loop:
        cmp ebx, range_end
        jg checker
        add edx, dword[ebx]
        add eax, 4
        jmp loop
        ....
checker:
        cmp edx, 0
        je  app_code
        jmp tamper_response
        ....
tamper_response:
        ...
```

(a) Assembly listing of a knot. The checksum is calculated using the `add` operation.

```
preamble:
        push checksum
        pop  ecx
        mov eax, range_start

loop:
        cmp eax, range_end
        jg checker
        xor ecx, dword[eax]
        lea eax, [eax + 4]
        jmp loop
        ....
checker:
        jecxz  app_code
        sub esp, 4
        mov esp, tamper_response
        ret
        ....
tamper_response:
        ...
```

(b) Assembly listing of another knot. The checksum is calculated using the `xor` operation.

Figure 2. Two examples of knots, created using a random selection of instructions.

tection techniques, this dynamism makes it harder for the adversary to locate and disable the protection network.

5. Evaluation

We have designed a prototype that implements knots, using a process-level virtual machine, called Strata [39]. Strata runs as a co-routine with the application, making modifications as it is running. Figure 3 displays a flowchart describe the mechanism of Strata. At program startup, Strata seizes control and begins to mediate execution by dynamically examining and translating application instructions, one basic block at a time. The translated blocks are placed in an area of memory, called the *fragment cache*. Strata then cedes control to this translated block. This block executes and then transfer control back to Strata, which then proceeds to translate the next application block. To ensure a changing attack surface, Strata periodically flushed the software cache and retranslated the application's instructions. Ghosh et al. have done extensive studies on the effectiveness of periodic flushing in providing an altering environment [17]. We decided on a rate of one flush every two seconds, as this rate gave a good trade-off in terms of low performance overhead and good fluctuation.

5.1 Experimental Setup

We tested the proposed approach by building a proof-of-concept implementation that targets the 32-bit Intel x86 platform, although the concepts described in this work apply to any binary translation system. Overhead and protection were evaluated using the C-language applications of the *SPEC CPU2000* benchmark suite, running on a Linux system. Our preliminary investigation revealed that predicated knots inserted at the rate of 2 per 100 dynamic application blocks (2%) offer a good trade-off between performance overhead and protection. As such, for our analysis, we chose this rate for predicate guards. Finally, for the purposes of our proof-of-concept, we set the connectivity, K, to 7.

5.2 Dynamic Tamper Resistance

Low performance overhead is an encouraging attribute, but we need to examine the protection offered by these knots closely. Low overhead could be explained by infrequent execution of knots, which will hamper the tamper resistance of the application. In this section, we analyze some of the run-time properties.

First, we investigate the frequency of knots execution. Figure 4 displays the number of knots executing per second for two of the benchmarks, *175.vpr* and *176.gcc*, which displayed the lowest and highest rate of knots invocation, respectively. The plots indicate that knots are executing consistently for both benchmarks. Thus, predicated knots achieve a good trade-off between performance overhead and knots execution rate. We observed a similar trade-off in the other benchmarks as well.

Figure 5 displays the average checks on each application byte occurring every second. This statistic gives an idea on the duration that a malicious modification can persist. The graph shows that predicated knots give a good trade-off between protection and overhead. For most benchmarks, code cache bytes are checked multiple times every second. Some benchmarks like *256.bzip2* have a higher rate of checking. *256.bzip2* is a compression application, with a few blocks that execute very frequently, and other blocks which execute infrequently. Some of the knots were placed inside the frequently executing blocks. Such knots contributed to the high checking rate. The high standard deviation indicates that the

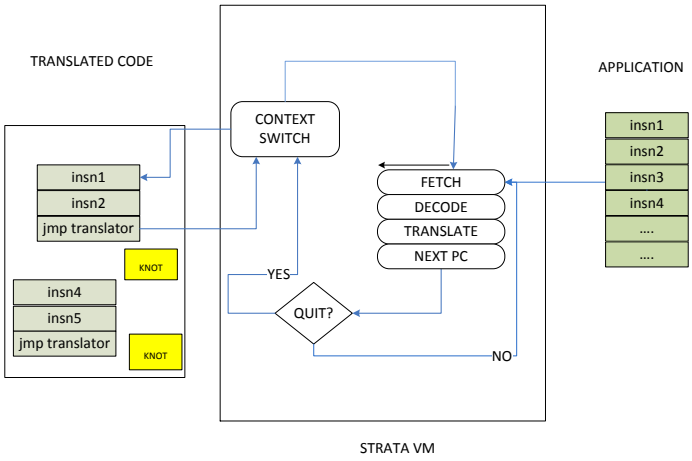

Figure 3. Flowchart displaying the translation mechanism of Strata VM. Strata translates application instructions as it is running. The translated instructions are stored in a code cache. At random points, it inserts knots that protect these translated instructions.

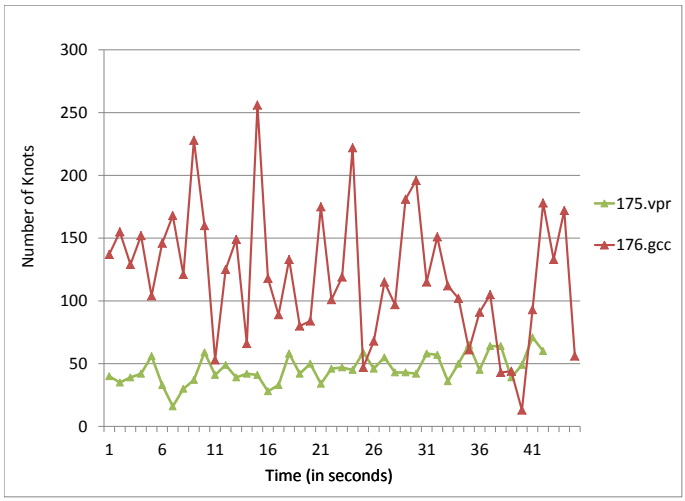

Figure 4. Number of knots invoked per second for *175.vpr* and *176.gcc*, which have the lowest and highest rate of knots activity, respectively.

checking was not uniform for all the bytes. We can draw similar conclusions for *181.mcf* and *177.mesa*.

We also investigate the structure of the knot created. If all the knots possess the same form (i.e., are composed of the same instructions), it would be trivial for the adversary to craft a regular-expression based attack to identify and disable knots. Figure 6 shows the percentage of unique knots created for each benchmark. For our proof of concept, the knots database allows for tens of thousands of unique knots. During evaluation, we observed that the PVM placed several hundreds of knots for each benchmark. For all the benchmarks, our design achieved upwards of 90% unique knots for every benchmark. The database can be tuned further to achieve even greater occurrence of unique knots. Polymorphic knots increase the difficulty of carrying out a successful regular expression-based attack.

5.3 Performance

Figure 7 displays the performance overhead of Strata and the protection features normalized to native (i.e., application

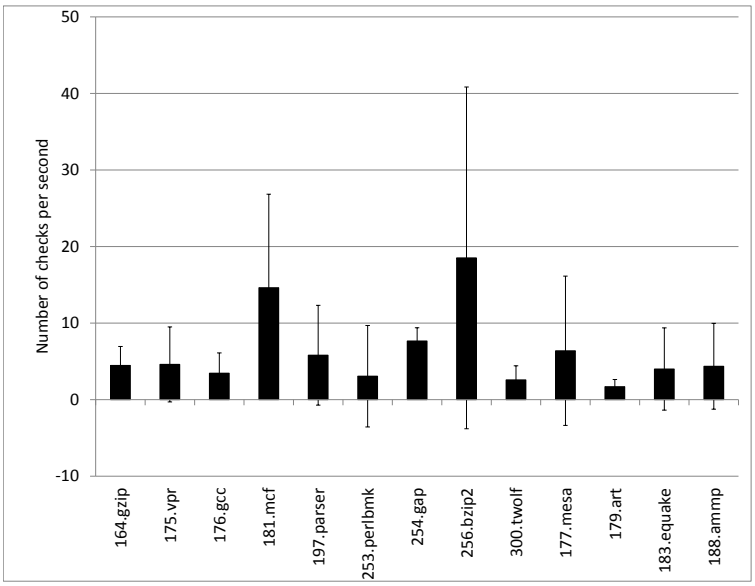

Figure 5. Average number of checks per second on each byte of the application, along with the standard deviation. This statistic gives an idea on how long a modification can exist before it is detected. The error bars correspond to the standard deviation.

running natively on the platform). Strata itself adds around 17% overhead to the run time. Previous work has investigated techniques to reduce this overhead [22]. Research in this area is still ongoing.

Inserting predicated knots dynamically into the code cache, using the calculations mentioned above, adds an additional overhead of 7% over native. The predicates that trigger knot execution are based on the exponential back-off method [12]. Initially the knot executes immediately when the predicate is triggered. Every time the knot is executed, the predicate is reconfigured such that it will have to be triggered twice more than the previous predicate value before the knot will execute, subject to a threshold value. This scheme multiplicatively reduces the rate of knot execution (and consequently, performance overhead) to gradually find an acceptable rate.

The final set of bars show the performance overhead when cache flushing is also enabled. At periodic intervals (every two seconds), all the translated code blocks are deleted and program translation proceeds at the next instruction to be schedule for execution. Flushing adds an extra 2% overhead to the run time.

6. Discussion

This section describes some of the protective aspects of the polymorphic knots, as well as vulnerabilities inherent to PVM-based security measures.

6.1 Protection of Generated Code

The primary goal of the knots involves protecting the translated code from tamper. To our knowledge, our scheme is the first attempt at tamper-proofing the software code cache. Any static techniques can only protect the guest application code and the PVM. Previously, the adversary could make modifications to the code cache with fear of detection. Flushing the code cache periodically provides some protection, as any changes made by the adversary will be removed and the correct application code translated again. However, flushing too frequently adds a serious performance overhead [17]. As such, knots add an additional layer of protection from tampering. Taken together, flushing and knots provide strong protection run-time protection against tampering.

6.2 Effectiveness against OS-based Attacks.

Modified operating systems have been used to mount successful attacks against software guards. Guard systems work on the assumption that the underlying hardware follows the von Neumann architecture (data reads and instruction fetches go to the same memory structure, called a *page*). Wurster, et al. showed a quick workaround to guards: separate data and instruction memory [50]. Each page of the application was duplicated and modifications were applied to it. The kernel was modified such that data reads would go to the unmodified application, whereas instruction fetches would bring in instructions from the tampered copy.

Giffin et al. designed an effective solution to this split-memory attack, which involves the use of self-modifying code [18]. Self-modifying code is possible only on systems

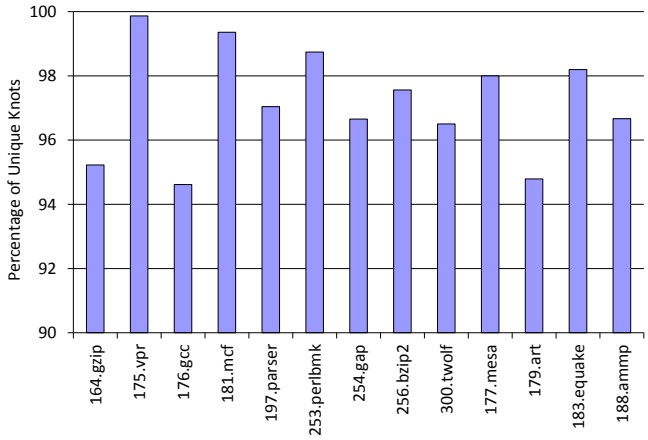

Figure 6. Percentage of unique knots for each benchmark. This statistic indicates that a simple regular expression will fail to locate all the knots in the cache.

implementing the Von Neumann architecture. This solution is inherently present in PVM systems implementing binary translation. The PVM continually translates, stores and then executes the application code. As such, the split memory attack will not succeed against knots.

The protection schemes described in this work rely heavily on randomness. For example, in our proof of concept, the placement of the knots in the code cache, and the structure of the knots themselves are decided by a random number generator (RNG) that is part of the standard library (*libc*). An adversary could modify the OS such that this RNG generates a predictable sequence. This modification could then allow the adversary to anticipate the locations and structures of the knots, and launch regular expression-based attacks. To thwart this type of attack, we chose a custom RNG algorithm that was part of the PVM itself. All decisions that are dependent on a randomizer, are based on its generated output sequence.

Finally, the altering nature of the PVM is driven by code cache flushing. This flushing is triggered by a periodic signal from the underlying OS. Again, an adversary could modify the OS such that this signal is never generated. Ghosh et al. have shown that flushing can also be triggered by using program properties, e.g., number of indirect jumps, or number of function calls [17]. The success of PVM-based protections depends heavily on relying as little as possible on entities external to the PVM and the application.

6.3 Effectiveness against PVM Attacks

Knots thwart tamper attacks on code that is resident in the internal cache of the PVM. An adversary can attempt to alter key functionalities of the PVM itself, thereby invalidating protection features such as knots. Ghosh et al. proposed a *replacement* attack on PVMs, which involves replacing a PVM applying protections with a benign VM [16]. Knots do not protect the PVM from such attacks. As such, additional security measures need to be established that protect the PVM from tamper.

Other classes of attacks have attempted to reverse engineer the application and remove code that belongs to the PVM. Coogan et al. designed a data analysis-based attack on the trace of a PVM-protected application, which designates the instructions belonging to the application[**?**]. Subsequent analyses on the marked instructions can reveal the relevant portions of the application (i.e., the critical algorithms). Sharif et al. devised an attack which attempts to identify PVM structures and remove them from the trace. J. Kinder presented encouraging preliminary results of a static analysis of VM-protected applications [26]. These types of attacks are focused more towards reverse engineering, rather than tamper-detection avoidance. Therefore, our solution is orthogonal to these classes of attack as well.

6.4 Side-channel Attacks

Recently, adversaries have started exploring indirect techniques, or side-channel attacks to break protections and tamper with software (e.g., memory access patterns, timing attacks etc.). The advent of sophisticated analysis techniques along with the increased complexity of the underlying microarchitecture, which has exposed more side channels to attack [28], has increased the incidence of such attacks. In general, all software systems are vulnerable to such attacks,

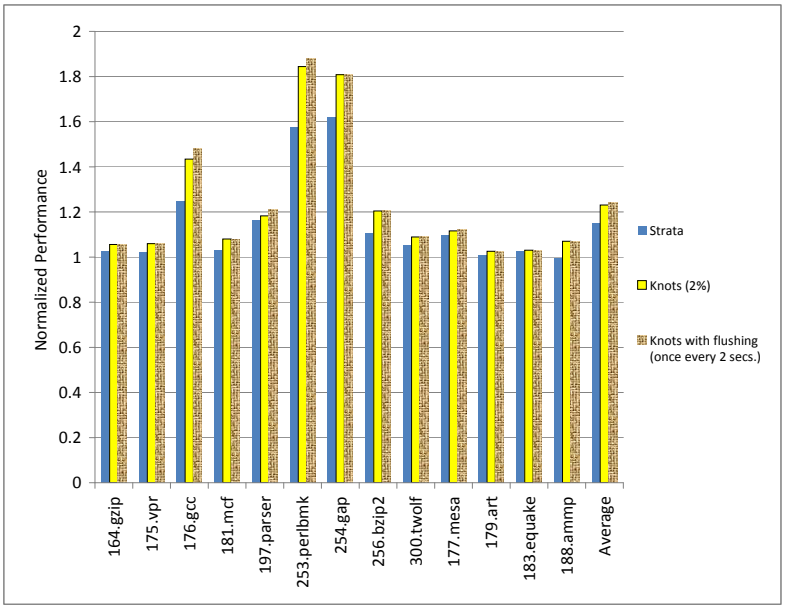

Figure 7. Performance overhead for the protection features, normalized to native application run. Flushing once every 2 seconds adds negligible overhead to the run time.

as such, we do not address this class of attacks in our work. Prior research has investigated solutions to side-channel attacks [19, 35].

7. Related work

Software applications are increasingly being used to perform critical tasks. As such, their protection against malicious modifications is of paramount important. Much research has been done in the area of tamper resistance. Significant work in this area is based on the seminal research by Aucsmith [2]. Program protection techniques can broadly be classified into hardware and software approaches.

Software-based program-protection approaches typically have a lower monetary cost and can be prototyped rapidly. Chang and Atallah proposed one such software-based solution, which explored the idea of a network of checksumming guards protecting the integrity of the application [8]. Subsequently, Horne et al. introduced the concept of *testers*, which consisted of sequences of code that computed a hash over a range of addresses, and compared the hash value to a *corrector* [23]. Any miscompare would trigger an appropriate response.

Oblivious hashing is a tamper-detection technique that computes hash values based on the actual execution path of the application [24]. Since static checksums are vulnerable to the split-memory attack [50], care must be taken to ensure that the assumptions about the underlying platform hold true during application execution.

A number of tamper-resistance technique utilize software obfuscation to make it harder for the adversary to analyze and understand the code. Linn et al. devised novel techniques to hamper disassembly of code [30]. Wang et al. proposed the use of indirect branches to obfuscate the control flow for the application [49]. Collberg et al. have also devised several techniques to obscure code [10, 11]. Code encryption is also a useful technique against static analysis. The code can be decrypted by hardware (i.e., a secure co-processor [51], but this method can add substantially to the cost of the product. Unfortunately, a number of these static techniques can be vulnerable to dynamic analysis [5].

Researchers have started using virtualization to improve program protections, specially at run time. A number of commercial PVMs have been created, such as VMProtect [47] and Themida [33]. These tools obscure the application using a secret Instruction Set Architecture and at run time, execute the code using the just-in-time interpretation model [43].

System-level virtual machines have also been utilized to provide tamper resistance. The Terra system implements a trusted virtual machine monitor with can be used to create closed-box platforms where the developer can tailor the software stack to meet security requirements [15]. However, this scheme requires hardware support to validate the software stack. Chen et al. discuss Overshadow, a system that cryptographically isolates the application insides a VMM from the guest OS it is running on. This system offers another layer of tamper resistance, even in the case of OS compromise [9].

Hardware-based approaches typically involve a higher cost in terms of resources, but are harder to break than software-only solutions. Many hardware-based solutions have been proposed that protect systems against physical and software attacks. One class of hardware solution involves

using dongles [37]. A dongle is a hardwired token that is distributed along with the copy of the software and can be attached to a computer. At run time the program intermittently invokes a routine in the dongle and if the reply is not validated, ceases to run. The premise is that it is more difficult to copy the dongle than the software so any user possessing the dongle is legitimate. However any adversary who can extract the security function from the dongle can bypass the protection using emulation. Modern processors also provide protection functionality in hardware (e.g., the Cell Broadband Engine, which provides the Runtime Secure Boot feature, which verifies that an application has not been tampered with, at start up [41]) Other hardware solutions involve the use of trusted computing platforms such as TPM and Microsoft's NGSCB [13, 36]. Finally there are customized hardware techniques to prevent tampering, such as execute-only memory [29] and systems based on secure processors [45].

8. Conclusions

Process-level Virtual Machines are increasingly being used in the area of software protection. This work presents a novel use of PVMs in the area of tamper detection. The scheme uses a PVM to create a dynamic network of polymorphic knots (i.e., sequence of instructions created dynamically by the PVM that checksum a range of program addresses, similar in concept to guard proposed by Chang [8]) that maintain the integrity of cached code from malicious modifications. The knots are created at run time and protect the code located in the software cache of the PVM. Periodic flushing of translated code creates a fluctuating protection network. Combined with predicates, dynamic knots achieve a good balance between execution rate and overhead. When used in conjunction with techniques that protect the static binary, knots provide strong tamper resistance. These results indicate that PVMs can provide a platform for different program protection techniques.

Acknowledgements

This research is supported by National Science Foundation (NSF) grant CNS-0811689, the Army Research Office (ARO) grant W911-10-0131, and the Air Force Research Laboratory (AFRL) 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 NSF, AFRL, ARO, 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] AUCSMITH, D. Tamper-resistant software: An implementation. In *Proceedings of the 1st International Workshop on Information Hiding* (London,U.K, 1996), Springer-Verlag, pp. 317–333.

[3] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 164–177.

[4] 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.

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

[6] 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.

[7] CASTELLUCCIA, C., FRANCILLON, A., PERITO, D., AND SORIENTE, C. On the difficulty of software-based attestation of embedded devices. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2009), ACM, pp. 400–409.

[8] 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.

[9] 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.

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

[11] 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.

[12] COMER, D. E., AND STEVENS, D. L. *Internetworking with TCP/IP, Vol 2: Design, Implementation, and Internals*, 2 ed. Prentice Hall, 1994.

[13] COMPUTING GROUP, T. TCG TPM specification version 1.2 revisions 62-94.

[14] 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.

[15] 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.

[16] GHOSH, S., HISER, J., AND DAVIDSON, J. W. Replacement attacks against VM-protected applications. In *VEE '12: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments* (New York, NY, USA, 2012), VEE '12, ACM, pp. 203–214.

[17] 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.

[18] GIFFIN, J. T., CHRISTODORESCU, M., AND KRUGER, L. Strengthening software self-checksumming via self-modifying code. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference* (Washington D.C., U.S.A, 2005), IEEE Computer Society, pp. 23–32.

[19] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM 43*, 3 (May 1996), 431–473.

[20] HAZELWOOD, K., AND SMITH, M. D. Code cache management schemes for dynamic optimizers. In *Sixth Annual Workshop on Interaction between Compilers and Computer Architectures* (Boston, MA, February 2002), pp. 102–110.

[21] 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.

[22] 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.

[23] HORNE, B., MATHESON, L. R., SHEEHAN, C., AND TARJAN, R. E. Dynamic self-checking techniques for improved tamper resistance. In *Digital Rights Management Workshop* (London, U.K., 2001), pp. 141–159.

[24] JACOB, M., JAKUBOWSKI, M. H., AND VENKATESAN, R. Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *MM&Sec '07: Proceedings of the 9th Workshop on Multimedia & Security* (New York, NY, USA, 2007), ACM Press, pp. 129–140.

[25] KIMBALL, W. *SecureQEMU: Emulation-based Software Protection Providing Encrypted Code Execution and Page Granularity Code Signing*. Air Force Institute of Technology, 2008.

[26] KINDER, J. Towards static analysis of virtualization-obfuscated binaries. In *WCRE '12: Proceedings of the 19th Working Conference on Reverse Engineering* (2012), IEEE.

[27] 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.

[28] LAWSON, N. Side-channel attacks on cryptographic software. *IEEE Security and Privacy 7*, 6 (Nov. 2009), 65–68.

[29] LIE, D., THEKKATH, C., MITCHELL, M., LINCOLN, P., BONEH, D., MITCHELL, J., AND HOROWITZ, M. Architectural support for copy and tamper resistant software. In *ASPLOS '00: Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2000), vol. 35, ACM Press, pp. 168–177.

[30] 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.

[31] 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.

[32] 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.

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

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

[35] OSVIK, D. A., SHAMIR, A., AND TROMER, E. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology* (Berlin, Heidelberg, 2006), CT-RSA'06, Springer-Verlag, pp. 1–20.

[36] PEINADO, M., P.ENGLAND, AND Y.CHEN. An overview of NGSCB. *Trusted Computing, Chapter 4* (2005).

[37] PHIPPS, J. Physical protection devices. In *The protection of computer software—its technology and applications* (New York, NY, USA, 1989), Cambridge University Press, pp. 57–78.

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

[39] 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.

[40] SESHADRI, A., LUK, M., SHI, E., PERRIG, A., VAN DOORN, L., AND KHOSLA, P. Pioneer: Verifying code in-

tegrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the 20th ACM Symposium on Operating Systems Principles* (New York, NY, USA, December 2005), vol. 39, ACM Press, pp. 1–16.

[41] SHIMIZU, K., NUSSER, S., PLOUFFE, W., ZBARSKY, V., SAKAMOTO, M., AND 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* (New York, NY, USA, 2006), MCPS '06, ACM, pp. 13–18.

[42] 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.

[43] 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.

[44] 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.

[45] SUH, G. E., CLARKE, D., GASSEND, B., VAN DIJK, M., AND DEVADAS, S. AEGIS: Architecture for tamper evident and tamper resistant software. In *Proceedings of the 17th Annual International Conference on Supercomputing* (2003), ACM Press, pp. 161–171.

[46] 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.

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

[48] WALTERS, B. VMware virtual platform. *Linux Journal 1999*, 63es (July 1999).

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

[50] WURSTER, G., OORSCHOT, P. C. V., AND SOMAYAJI, A. A generic attack on checksumming-based software tamper resistance. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy* (Washington D.C., U.S.A, 2005), IEEE Computer Society, pp. 127–138.

[51] 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.