Software Protection via Composable Process-level Virtual Machines

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Engineering)

by

Sudeep Ghosh

December 2013

APPROVAL SHEET

The dissertation

is submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Sudeep Kumar Ghosh

AUTHOR

The dissertation has been read and approved by the examining committee:

Jack W. Davidson

Advisor

John C. Knight

Gabriel Robins

Ronald D. Williams

Yuan Xiang Gu

Accepted for the School of Engineering and Applied Science:

James H. Aylor

Dean, School of Engineering and Applied Science

Abstract

Complex hardware/software systems are ubiquitous, affecting every aspect of daily life. Software is integral to the normal functioning of critical systems such as power plants, financial systems, communication systems, modern medical systems and devices, and transportation systems to name a few. Because of society's increasing reliance on these systems, it is of paramount importance that software perform as intended, and not be subverted for malicious purposes. Consequently, techniques that thwart reverse engineering and tamper, (called *tamper-resistance techniques*), have become increasingly important as a means to hinder malicious exploitation of software in critical systems.

Given the growing importance of preventing tampering with critical systems, research in this area has grown. Recently, software virtualization has been proposed as a suitable mechanism to impart tamper resistance to software applications. However, protections based on virtualization have not fully matured, which has led to successful attacks. This dissertation is the culmination of a detailed study examining the application of low-overhead process-level virtualization to protect software applications from reverse engineering and tamper.

This research is structured as follows: First, a formal model describing virtualization is presented. The model is useful in describing general-purpose computing systems and the applicability of virtualization in protecting applications. Then we explored several novel tamper-resistance techniques that are based on process-level virtualization. Each technique was thoroughly evaluated in terms of performance overhead and protection. During the course of our investigation, a serious vulnerability in current process-level virtual machines was discovered. We modeled this vulnerability using our formal model and describe two attack implementations that successfully exploit this vulnerability. Finally,

we conceptualize a revolutionary protection technique to compose an application with multiple virtual machines, providing robust program protection. The ideas presented in this dissertation are evaluated using current state-of-the-art attacks to gauge its effectiveness. The results of our investigation reveal that composable virtual machines are significantly more effective in thwarting reverse engineering and software tamper than current protection techniques.

Acknowledgements

There are too many people who deserve thanks for their invaluable help throughout this process, so this is not a comprehensive list. First and foremost, I would like to thank my committee, Jack Davidson, John Knight, Ronald Williams, Gabriel Robins, and Yuan Gu.

Friends and family have been invaluable to me throughout this process. This research would not have been possible without their constant support and encouragement. I am extremely grateful to my parents, Birendra and Suhasini Ghosh for inculcating a sense of curiosity and research in me. My elder brothers, Subir and Sujoy have always encouraged me to pursue my dreams. I am also grateful to Subhadeep Chatterjee, Shankar Natarajan, Alan Hostetler, Kanshukan Rajaratnam, Yvonne Baki, Dan Upton, and Misty Boos for their friendship.

Finally, I'd especially like to thank (again) Jack Davidson and members of his research group for their insights and assistance. In particular, I would like to thank Jason Hiser for his assistance towards this reseach. Michele Co, Dan Williams, Wei Hu, Clark Coleman, and Ben Rodes have always been forthcoming with their knowledge, and my work would have been much more arduous without their help.

Contents

Abstract **ii**

Contents **v**

List of Figures ix

1 Introduction **1**

1.1 Motivation 2
1.2 Problem Description 3
1.3 Threat Model 7
1.4 Limitations of Current Approaches 8
1.5 Software Virtualization 9
1.6 Thesis 9
1.7 Research Overview 10
1.8 Contributions 12
1.9 Organization 14

2 Process-level Virtualization **16**

2.1 Motivation 16
2.2 Introducing Virtualization 18
2.3 Software Dynamic Translation 22
2.4 Summary 24

3 Modeling Virtualization-based Protections **25**

3.1 Model 25
3.1.1 Software Applications 26
3.1.2 Software Interpretation 27
3.1.3 Modelling Tamper Attacks 30
3.1.4 Modeling Program Protection 33
3.1.5 An Example: Authentication 35
3.2 Summary 37

4 Creating Tamper-resistant Binaries **38**

4.1 Encryption 39
4.1.1 Implementation 41
4.1.2 Key Protection 43
4.2 Interleaving Application and PVM code 45
4.2.1 Evaluation 46
4.3 Related Work 50
4.4 Summary 51

5 Strengthening Tamper Detection Using PVMs **53**
5.1 Checksumming Code . . . 54
5.1.1 Guards . . . 55
5.1.2 Knots . . . 56
5.2 Instantiation Polymorphism . . . 57
5.3 Evaluation . . . 60
5.3.1 Run-time Protection . . . 61
5.3.2 Measuring Diversity of Checker Instances . . . 64
5.3.3 Performance . . . 65
5.4 Discussion . . . 66
5.4.1 Protection of Generated Code . . . 66
5.4.2 Software Diversity . . . 66
5.4.3 Circular Protection . . . 67
5.4.4 Effectiveness Against OS and VM attacks . . . 67
5.5 Summary . . . 68

6 Temporal Polymorphism **69**
6.1 Incorporating Temporal Polymorphism . . . 71
6.1.1 Triggering Cache Flush . . . 71
6.1.2 Randomizing Code Locations in the Cache . . . 72
6.2 Evaluation of Temporal Polymorphism . . . 73
6.3 Robustness of the Flush-trigger Mechanism . . . 76
6.4 A Use Case . . . 77
6.4.1 Analyzing the Control Flow Graph . . . 77
6.4.2 Thwarting Dynamic Analysis . . . 78
6.5 Summary . . . 82

7 Point-ISA: Binding the Application to the PVM **84**
7.1 Replacement Attack . . . 86
7.1.1 Modeling the Attack . . . 87
7.1.2 Description of the Attack . . . 90
7.2 Use Cases . . . 92
7.2.1 Attack Implementations . . . 93
7.3 Implications of the Attack . . . 95
7.4 Attack Discussion . . . 98
7.4.1 Determining PVM Entry Function . . . 99
7.4.2 Determining the ISA of the Guest Application . . . 99
7.5 Point-ISA: Using Homographic Instructions to Semantically Bind the Application and the PVM . . . 101
7.5.1 Formalizing Point-ISA . . . 103
7.6 Designing Point-ISA . . . 108
7.6.1 Identification of Homographic Instructions . . . 108
7.6.2 Insertion of Point-ISA Components . . . 109
7.6.3 Response to a Replacement Attack . . . 109
7.7 Evaluating Point-ISA . . . 110
7.7.1 Designing the Prototype . . . 111
7.7.2 Selection Criteria for Homographic Instructions . . . 113
7.7.3 Performance . . . 115
7.8 Security Discussion . . . 116
7.8.1 Effectiveness against Replacement Attacks . . . 116
7.8.2 Effectiveness against Reverse Engineering . . . 117
7.8.3 Tampering with the Protective PVM . . . 118
7.8.4 Synopsis of Overall Protection . . . 119
7.8.5 Co-designing Applications and protective PVMs . . . 119

7.9 Summary . . . 119

8 Establishing a Data Dependence between the Application and the PVM 121
8.1 Value-based Dependence Analysis (VDA) Attacks . . . 124
8.1.1 Coogan, *et al.*'s implementation of VDA . . . 127
8.2 Effectiveness of VDA on Binary Translation Systems . . . 131
8.2.1 Dynamic Transformation of Addressing Modes (DTAM) . . . 131
8.3 Modified Value-based Dependence Analysis (MVDA): An Example of White-hat Attack 135
8.3.1 Evaluation . . . 137
8.4 DataMeld: Blending Data between the Application and the PVM . . . 140
8.5 A Tool to Calculate ICV . . . 143
8.5.1 Rules Regulating ICV Calculation . . . 143
8.6 Mapping Memory References to Variables in Source Code . . . 145
8.7 Implementing DataMeld . . . 146
8.8 Evaluation . . . 148
8.8.1 ICV Tool Output . . . 148
8.8.2 Measuring Obfuscation . . . 149
8.9 Summary . . . 150

9 Composable Virtual Machines 151
9.1 Partitioning Applications . . . 153
9.2 Creating CVM Packages . . . 154
9.3 A Use-case Study . . . 155
9.3.1 Series Configuration . . . 156
9.3.2 Nested Configuration . . . 157
9.4 Performance Overhead of Nested CVMs . . . 163
9.4.1 Experimental Setup . . . 163
9.4.2 Self-modifying Code . . . 163
9.4.3 Super-patching of Trampolines . . . 164
9.5 Issues Regarding CVMs and Scope for Future Work . . . 166
9.5.1 Performance Trade-offs . . . 166
9.5.2 Security Evaluation of CVM Methodology . . . 167
9.5.3 Incorporating Protection Techniques into CVMs . . . 168
9.6 Summary . . . 168

10 Related Work 169
10.0.1 Self-aware Integrity Verifiers . . . 170
10.1 Obfuscation . . . 172
10.1.1 Static Obfuscations . . . 172
10.1.2 Dynamic Protection Techniques . . . 173
10.2 Remote Tampering of Software . . . 175
10.3 Hardware Approaches . . . 175

11 Summary 176
11.1 Conclusions . . . 176
11.2 Scope for Future Work . . . 179
11.2.1 Expanding the Scope of Homographic Instructions . . . 179
11.2.2 Alternate Threat Models . . . 180
11.2.3 Expanding the Scope of Protections . . . 180
11.2.4 Appropriate Program Partitioning . . . 180
11.2.5 Incorporating Protections into CVMs . . . 181
11.3 Summary . . . 181

Glossary 182

Bibliography **184**

List of Figures

1.1 Run-time working of *Skype*. The binary uses static encryption, self-modifying code and checksums to protect itself. 4
1.2 Split memory attack on guard systems, as devised by Wurster *et al.* The Translation Lookaside Buffer (TLB) is a structure that is used to store virtual to physical address mappings. Instruction fetches are directed by the OS to physical addresses containing pages from the modified program A' whereas data reads go to the original version A. 6
1.3 A high-level illustration of process-level virtualization 10
1.4 High-level overview of the protection process. During application creation, the application is co-located with a protective PVM. At run time, the PVM takes control of execution and runs the application in a protective environment. 11
1.5 High-level overview of the the different contributions of this dissertation. The boxes in gray represent the various stages of a traditional engineering process. The boxes in blue represent the various contributions of this dissertation. 13

2.1 Computing systems are designed using layers of abstraction. These layers typically manage different aspects of the system that pertain to the implementation. In this particular design, the system has been partitioned into the hardware, the OS kernel, the system libraries, and finally, the applications that run on the system. 19
2.2 High-level overview of the *fetch-decode-dispatch* mechanism that forms the basis for PVMs. The guest application instructions are decoded, one at a time and dispatched to appropriate handling functions, which execute according to the semantics of the instructions. 21
2.3 Strata Virtual Machine modified to apply dynamic protections. 23

4.1 Layout of the on-disk binary consisting of the guest application and the protective PVM. The guest code is encrypted. A few libraries of the PVM end up amongst the application code. 43
4.2 Layout of the code regions after the instruction blocks have been randomly permuted. 46

5.1 On-disk layout of PVM-protected binary containing guards($G1$, $G2$, $G3$, and $G4$). . 55
5.2 Flowchart depicting the run-time of a Strata-VM protected application. The code is protected by *guards* and *knots*. 57
5.3 Two examples of knots, created using a random selection of instructions. 59
5.4 Number of checkers (guards and knots) executing per second for `175.vpr` and `176.gcc`. The protected range consists of both the application code and the translated instructions. 61
5.5 Guard connectivity 62
5.6 Average time delay between checks for each byte of the program. This metric gives an indication as to how long a modification can exist undetected. 63
5.7 Percentage of unique checkers for each benchmark. This statistic indicates that a simple regular expression will fail to locate all of them. 64
5.8 Performance overhead for the protection features normalized to native application run. 65

6.1 Performance overhead due on flushing based on indirect branch count 73

6.2 Rate of plaintext application code due to flushing on indirect branch count for `176.gcc` 74

6.3 Distribution of code cache addresses for guards across 10 independent runs. The cache is flushed every second. 75

6.4 Execution frequencies for the application blocks under the two run-time scenarios (No protection, and Protected). 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 adversary to expand their search space. 81

7.1 Steps illustrating the attack methodology on virtualized applications. 92

7.2 Listing of x86 assembly code snippet preceding the entry point into Strata. 94

7.3 Execution frequencies for the application blocks under the three 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. 96

7.4 Number of opcodes in the database that can form part of I_H, for each benchmark. . 114

7.5 Performance overhead for Point-ISA normalized to native. Overhead for Strata is provided for reference. 116

7.6 The average delay between the invocations of two Point-ISA components. The error bars correspond to the standard deviation. 117

8.1 Dynamic instruction counts, normalized to the original application. The four scenarios include running the PVM-protected application, the count of the instructions obtained when VDA is applied to the PVM-protected application, the instruction count when VDA is applied to an application protected with dynamic address-mode transformation, and finally, MVDA applied to an application protected with dynamic address-mode translation. 138

8.2 The workflow for the application protection process via DataMeld. First, the memory references with most instruction coverage are calculated. Subsequently, the variables in source code that correspond to these references are identified. Finally, DataMeld is implemented using Strata 142

8.3 Dynamic instruction counts, normalized to the original application across all the five scenarios. To recap, these scenarios include running the PVM-protected application as is, the DIC obtained when VDA is applied to the PVM-protected application, the DIC when VDA is applied to an application protected with DTAM, the DIC obtained when MVDA is applied to an application protected with DTAM, and finally, the DIC when MVDA is applied to an application protected via DataMeld. 149

9.1 A conceptual overview of a CVM-protected application package. The application is partitioned (based on certain criteria) into two sections. The first section is protected by a set of two CVMs, while the second section is protected by one CVM instance. . 152

9.2 A high-level overview of the series configuration. During software creation, Diablo synthesizes the CFG of the application and inserts the entry and exit functions of the CVM instances, Strata_1 and Strata_2, by dividing the `main` function equally. At run time, first Strata_1 translates its partition, followed by Strata_2. 157

9.3 A high-level overview of the nested configuration. During software creation, Diablo synthesizes the CFG of the application and inserts the entry and exit functions of the CVM instances, $Strata_1$ and $Strata_2$. In this case, $Strata_2$ is encapsulated within $Strata_1$. At run time, $Strata_2$ translates the application to its software cache, SC_2. $Strata_1$ translates $Strata_1$ and SC_2 onto SC_1. Since the code resident in SC_2 does not execute directly, it can be encrypted. 158
9.4 This figure illustrates the software cache of $Strata_1$. This software cache contains the code from $Strata_2$, and its software cache SC_2, which is basically translated application code. As can be seen from the figure, the code from the components are interleaved, to provide better obfuscation. 160
9.5 Performance overhead for super-patching in the nested CVM configuration. The performance overhead of Strata is also provided for comparison. On average, super-patching reduces the overhead of nested CVMs to 70% over native execution. 165

Chapter 1

Introduction

In the context of computer security, the term *tamper resistance* refers to the techniques that obstruct modification of software binaries. Seminal work in the area of tamper resistance was done by David Aucsmith [1], and much of the current research in this area has been influenced by his work. Techniques in tamper resistance fall into two broad areas: *tamper detection*, which includes mechanisms that uncover any unauthorized modification of the application [2, 3, 4, 5, 6], and *obfuscation*, consisting of techniques which make a program harder to understand, thereby increasing the difficulty of reverse engineering the application [7, 8, 9, 10, 11, 12]. The research presented in this dissertation investigates the use of process-level virtual machines (PVMs) in both areas of software tamper resistance—tamper detection and code obfuscation. To evaluate the strength of these techniques, we developed several metrics that measure the effectiveness of tamper resistance in appropriate contexts. In the course of evaluating existing tamper resistance techniques and developing metrics to measure the strength of tamper resistance techniques, a serious flaw in current PVM-based protections was discovered. To address this flaw, a revolutionary new composition methodology for the creation of tamper resistant software was developed that provides a robust tamper-resistant execution environment.

1.1 Motivation

Today, computer and software usage has become ubiquitous. This proliferation of software systems has primarily been fueled by the availability of inexpensive hardware and development of advanced toolsets which enable programmers to create complex and efficient software applications using high-level languages. Software's malleability has enabled vendors to continuously provide new features and repair erroneous program operation, all without expensive hardware updates. Such utility, and ease of design and flexibility have enabled the use of software applications to provide critical functionality in a wide range of systems. To give a few examples, doctors use applications on embedded devices to diagnose and treat patients, and store patient related information on standard computing systems. Transportation systems (*e.g.*, air traffic control systems, highway traffic systems, railway switching systems, *etc.*) rely on the correct operation of software applications to ensure smooth and consistent functioning. Financial institutions use software to process important financial information about their clients.

Because software provides essential and critical functions in these systems, the software has become the target of malicious entities that aim to subvert the functionality of these systems for personal gain. Such entities have been aided in their goals by the availability of tools to reverse engineer application packages and obtain a higher level of comprehension (also known as *decompilation*). Reverse engineering tools, such as IDA-Pro [13], OllyDbg [14], and LordPE [15], have made it easier to obtain a representation at a higher level of abstraction and provide increasing opportunities for modification and unauthorized use. An adversary with access to such state-of-the-art technology and tools can modify the binary file (*i.e.*, the software file that contains the instructions and data of the application), and use it to their own advantage.

Even with a superficial understanding of the software, an adversary can change the operation of a program and exploit functionalities which they are not entitled to use or cause damage to the underlying system (*e.g.*, disrupting the network by tampering with the routing software, accessing privileged information). Any unauthorized modification to critical software systems could lead to extensive disruption of services and losses in terms of life and property. The Business Software

Alliance estimated that worldwide losses due to information theft totaled $63.4 billion in 2011 alone, up from $59 billion in the previous year [16].

In addition, code tampering is commonly used by malware to exploit software systems. Worms and viruses are increasingly being programmed to first disable any protective software and then infect the system(called *retroviruses*) [17]. One example is the *Beast* Trojan, which infects different variants of Microsoft's Windows operating system, from Windows 98 to Windows XP [18]. It operates by modifying *explorer.exe* to incorporate and execute its payload. The payload then proceeds to disable any anti-virus and/or Firewall applications executing in the system and allowing the attacker to essentially obtain complete control over the host machine.

Because of the potential consequences of the such attacks along with the growing ease that they can be carried out, it has become vital to develop protection mechanisms that hamper reverse engineering and tampering of software, either on disk or while the program is running.

1.2 Problem Description

This research addresses the problem of malicious analysis and modification of software applications The prototype designed to implement our research ideas, uses application object files as input. Operating on object files enables the process of compilation and administration of protections to be kept separate which helps maintain modularity and reduces the impact on the software development processes.

Unauthorized analysis of software is a widespread problem. To illustrate the capabilities of modern adversaries, we discuss two attacks on software systems that have been described in the literature.

The first case involved researchers performing *penetration testing*, which is a method to evaluate software protections by simulating an attack scenario. Biondi et al. performed their study on the protections of the popular Voice-over-IP tool, Skype, and published its closed-source algorithm [19]. Figure 1.1 illustrates the protection mechanisms of Skype. The on-disk binary is protected via a combination of encryption (to prevent static analysis), and checksums over address ranges (to protect

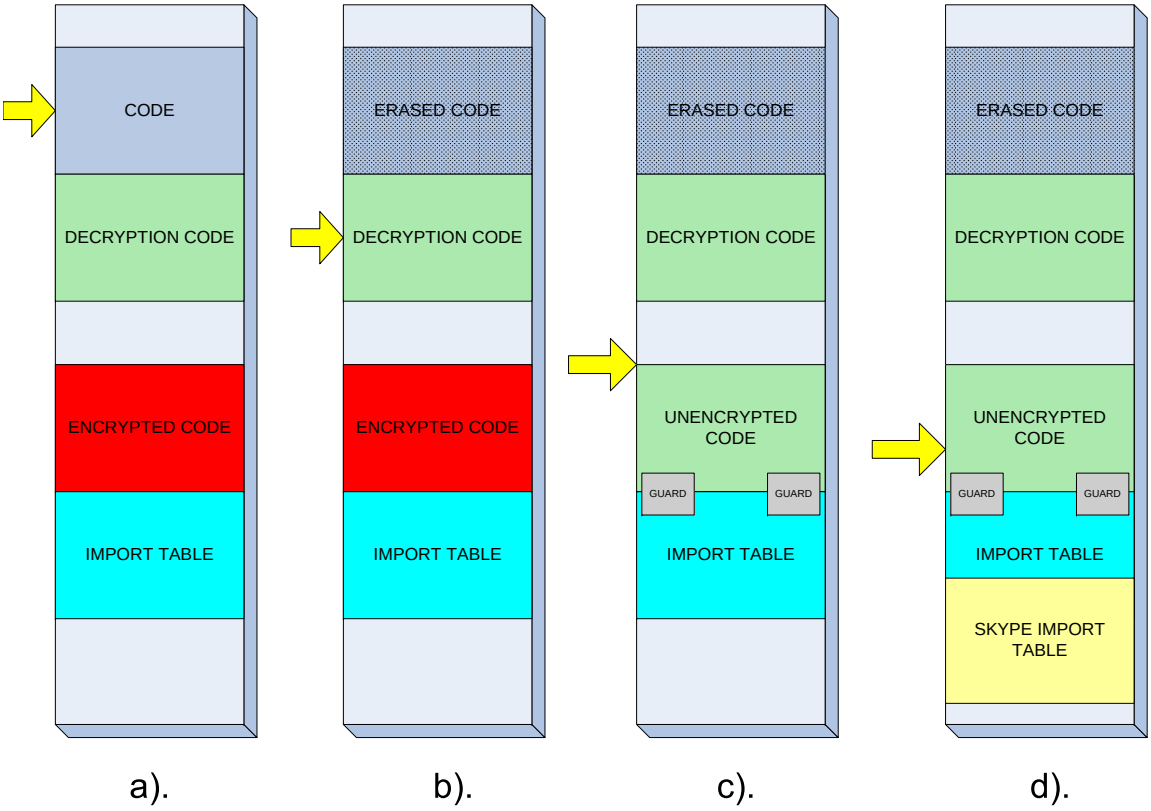


Figure 1.1: Run-time working of *Skype*. The binary uses static encryption, self-modifying code and checksums to protect itself.

against tampering). Skype uses run-time code generation, therefore the on-disk binary file does not contain the complete code.

On start up, Skype deletes some redundant code sections (Figure 1a). Then it decrypts the entire code base (Figure 1b), leaving the application fully exposed in memory. Finally, it generates the remaining code (Figure 1c).

Biondi, *et al.* were able to run Skype under a debugger and study these techniques using analysis tools. They located the checksumming code and disabled it by running two instances of the program, *I* and *I'*, in parallel. The checksums were obtained incrementally in version *I*. This value was then used to bypass the actual checking of the corresponding checksumming code in version *I'*. Because the program always executed from the same location, the attackers were able to mutually relate, and disable checks in the two versions.

The primary reason behind the failure of Skype's protections was the mutual dissonance of the techniques. The designers of Skype applied several schemes, each of which, by itself, is relatively strong and impervious to trivial attack. However, they do not reinforce each other. Consequently, Biondi and his colleagues were able to systematically remove the protections and extract useful information.

The attack on Skype's protections was achieved primarily by the use of debugging and analysis tools. Software can also be attacked by subverting hardware or software, and returning incorrect information. This approach was illustrated in a well publicized attack by Oorschot *et al.* [20], called the *split-memory attack*.

The split memory attack methodology targets protection mechanisms used on machines with the von Neumann memory architecture. The Von Neumann model specifies one type of memory in the system, which is shared by both data and code. Modern hardware processors, on the other hand, follow the Harvard memory model, which specifies separate structures for data and code. In most cases, this dichotomy on physical Harvard processors is managed by the operating system, which presents a von Neumann view to the application.

Based on the von Neumann architecture, Chang and Atallah proposed *checksumming* of code, to provide integrity. During software creation, the content of application code regions are hashed and the value is stored. At run time, small sequences of code (called *guards*) rerun the hash over the same regions and ensure the hash value is identical. A mismatch triggers a tamper response.

The functionality of guards depends on presence of the von Neumann memory model, since during the hashing process, the code is treated as input data to the hash operation. The split memory attack involves modifying the OS such that the inherent Harvard model is no longer hidden. Thus, the code that is checked by guards is not the same as the code that is executing. Any changes made to the executing code will no longer be detected.

Figure 1.2 illustrates this attack methodology in more detail. The adversary initially creates a tampered version (A') of the original protected program (A). The malicious OS maps instruction accesses to A', while data accesses are directed to unmodified pages from A in the data cache.

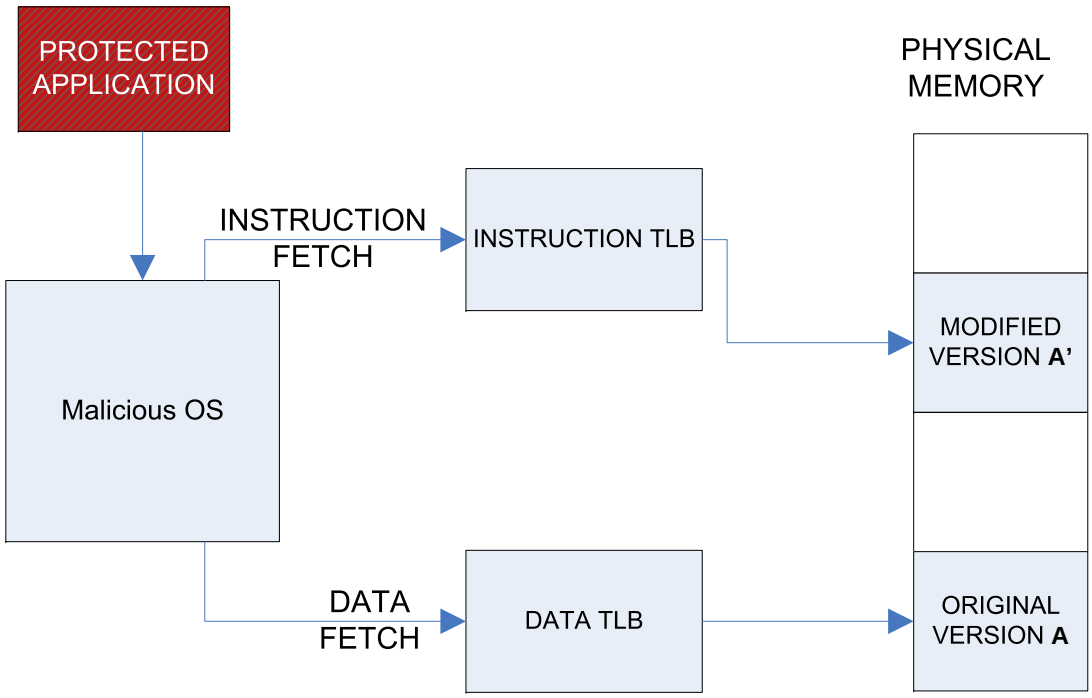


Figure 1.2: Split memory attack on guard systems, as devised by Wurster *et al.* The Translation Lookaside Buffer (TLB) is a structure that is used to store virtual to physical address mappings. Instruction fetches are directed by the OS to physical addresses containing pages from the modified program A' whereas data reads go to the original version A.

Consequently when the application validates the application code through data reads (such as when guards read their input), it accesses A, while the actual instructions that would execute on native hardware are fetched from A'. This technique also allows an adversary to quickly identify loads from the text segment and potentially locate and remove checksumming code.

To summarize, adversaries subject software applications to various analyses in their attempts to reverse engineer, and modify, them. The next section describes the capabilities and limitations, if any, of the adversary.

1.3 Threat Model

Prior to examining any security solution, it is necessary to understand the environment in which the protected software executes. Additionally, it is also necessary to recognize the set of attack capabilities of the adversary. Any protective technique can only be truly evaluated when it is tested in the appropriate environment, and against correct set of capabilities. Consequently, every security technique has an associated *threat model*, which describes its operational environment.

The threat model associated with this research is as follows: During software creation, the computing environment is completely trusted. This environment includes various tools (e.g., the compiler, linker, assembler *etc.*) as well as, the software developer. The software is created, verified, tested, and then protected using various security transformations. The research in this dissertation targets general-purpose software applications that run on standard commercial hardware, without the use of any special hardware.

Once deployed, the software is susceptible to attack. An adversary can use various tools such as debuggers, simulators, and emulators to run, modify, and observe the program in a number of ways. Even the operating system can be modified to return inaccurate information [20]. Consequently, we consider all software on the machine as potentially malicious, and the entire application created at the trusted development site (including any virtual machines distributed with said software) as potentially vulnerable to attack.

Furthermore, even hardware devices in the entire system cannot be trusted, as the application may be running within a simulator or emulator which can arbitrarily return forged results to the application. In essence, this is a white-box attack on the application [21]. The adversary can inspect, modify, or forge any information in the system.

Given enough time and resources, the adversary can always succeed in manually inspecting and making modifications to the program [22]. However, human adversaries have difficulty directly solving problems involving large data sets. As such, they rely on algorithmic solutions in an attempt to disable security features in software. Typically, adversaries use automated tools to perform various analyzes on application packages, including determining instruction locations, disassembling the

program, and capturing the control-flow or call graphs, to name a few. The goal of this research is to make such tasks harder to accomplish.

1.4 Limitations of Current Approaches

Current tamper-resistance techniques suffer from a variety of drawbacks.

- Most techniques have targeted making the application hard to analyze statically [23, 24, 10, 25]. For example, an opaque predicate is a binary expression (usually $true$ or $false$) that is hard to analyze statically, but several runs in a simulator can provide the adversary with information required to defeat the protections. Static encryption techniques have been used to prevent static disassembly of applications [12]. Unfortunately, once loaded into memory, decryption occurs at a coarse level of granularity. As we discussed in Section 1.2, such a scheme left Skype vulnerable to dynamic analysis [19]. A major goal of this research is to reduce plaintext information at run time.

- The use of additional hardware is required by some of the solutions [26, 27, 28, 29]. Unfortunately extra hardware adds an additional cost that will be borne by the end user, and it restricts the software to a particular set of platforms. Consequently, adoption of such techniques has been slow, and only used where expense is not a restriction.

- A number of software protection schemes have impractical or unreasonable resource constraints [30, 31, 32]. The Proteus system involves overheads between 50X-3500X which is too high for most applications [11]. Remote tamper-proofing techniques, in which code running on a server is used to validate client-side code, is commonly used to verify and validate software running on networked embedded devices [6, 5]. However, such solutions are incompatible with most heterogeneous, networked environments [33] due to their differing network latencies. A realistic solution must have tolerable overheads, otherwise developers will be unwilling to deploy such measures on a large scale.

- Finally, a number of schemes have been proposed recently, which attempt to improve the run-time protection of applications, primarily using virtualization [11, 34]. A major contribution of this research is presentation of strong evidence through demonstration that such schemes are not fully mature, and are susceptible to attacks. Chapter 8 discusses these attacks.

Given these limitations, there is an urgent need for new techniques that protect software. This research presents a detailed exploration of the applicability of process-level virtualization to software tamper resistance. The next section gives a brief introduction to process-level virtualization.

1.5 Software Virtualization

Software Virtualization is a versatile technique that involves the addition of a layer of software between an application (often called the *guest*) and the underlying platform (often called the *host*). This addition is done to enhance different properties of the application (*e.g.*, security or flexibility). This layer of software, called the *virtual machine*, encapsulates a single process (a process-level virtual machine (PVM), or an entire operating system (a system-level virtual machine)). This research focuses exclusively on utilizing PVMs for program protection.

Figure 1.3 illustrates two applications running on a computing system (the host). Application App_2 is a process that runs natively on the system. App_1 runs via a PVM. On startup of App_1, the PVM assumes control and starts emulating the instructions of App_1 on the host. In Chapter 2, we provide a detailed explanation of PVM functionality, and the feasibility of its use in the field of program protections.

The next section states the main thesis of this research.

1.6 Thesis

Set in the context described in Section 1.4, it can be recognized that new techniques are required to thwart an adversary's goals of subverting critical software and exploiting them for their own

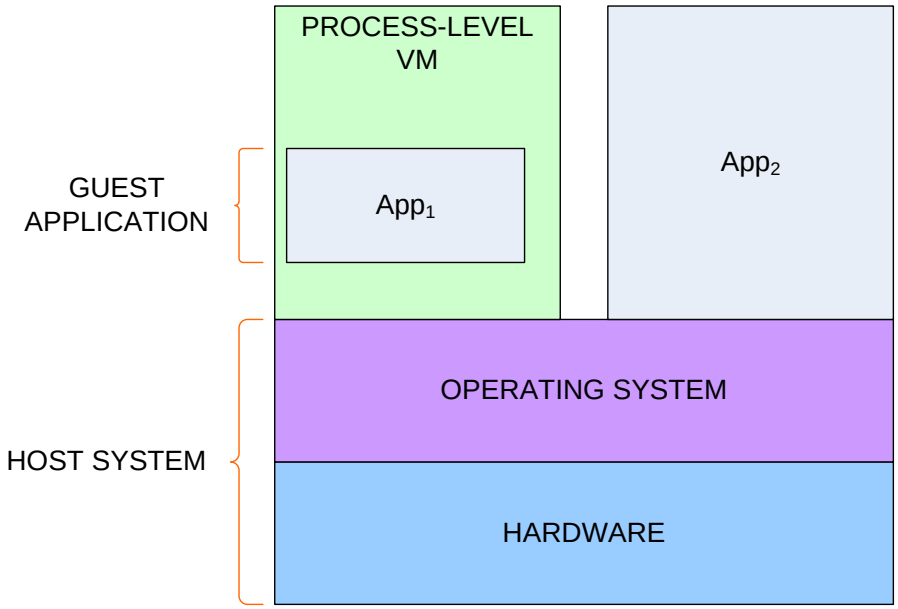


Figure 1.3: A high-level illustration of process-level virtualization

malicious goals. This research investigates the aptness of process-level virtualization for software tamper resistance.

The thesis of this dissertation is that composing applications with process-level virtual machines can effectively hamper reverse engineering and tamper attacks on software.

In the context of this research, composing an application is defined as creating a PVM-protected application in a holistic manner. Previous attempts in virtualization-based software protections have approached the design of the application and the PVM in isolation. This research demonstrates that designing the applications and the PVM in close harmony leads to a more robust execution enviroment. The protection mechanisms reinforce each other, instead of acting in and of themselves.

1.7 Research Overview

Now that the major premise behind this research has been established, we present a high-level overview of this research. Initially, we developed an equational model to describe composable virtual

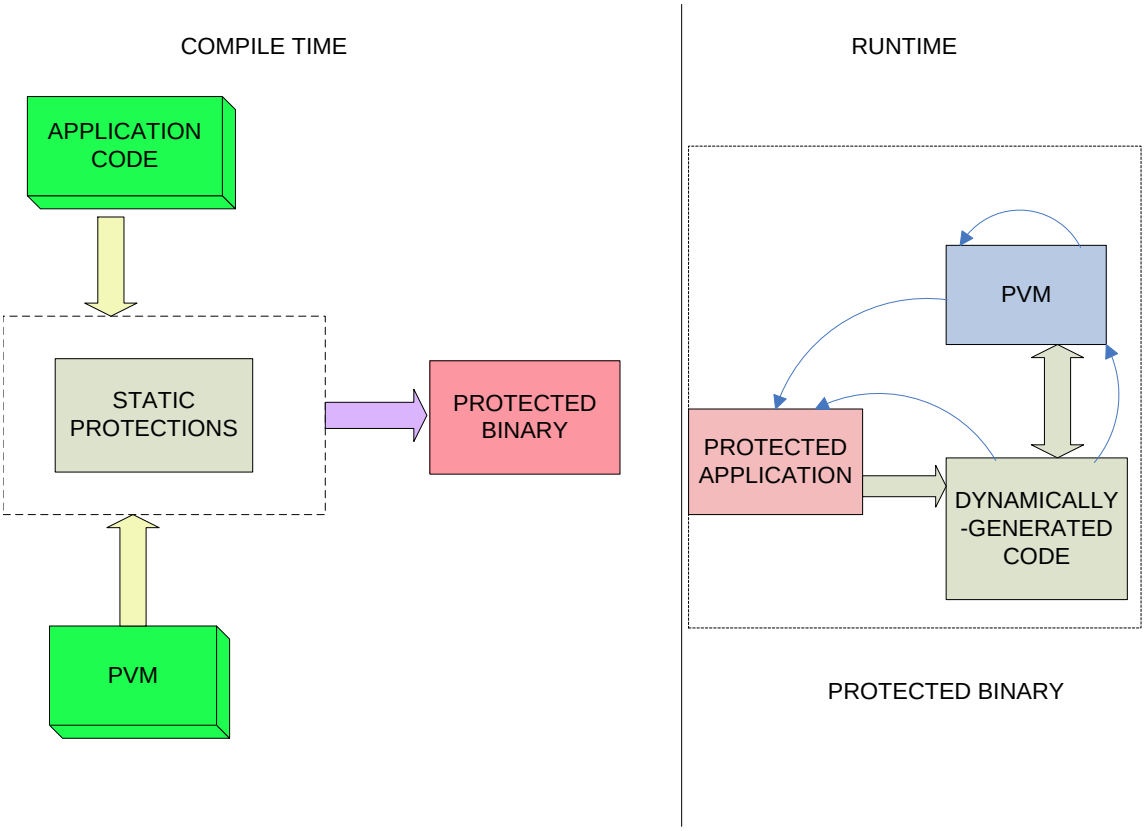


Figure 1.4: High-level overview of the protection process. During application creation, the application is co-located with a protective PVM. At run time, the PVM takes control of execution and runs the application in a protective environment.

machines. Consequently, we investigated techniques to improve the resistance of on-disk binaries to static disassembly. Finally, we proposed and evaluated mechanisms to boost the run-time protection of virtualized applications.

Figure 1.4 illustrates the creation and application of protection techniques using the proposed approach. During software creation the application is combined with a protective PVM. Then, the consolidated package is protected using static techniques. The contributions of this research in thwarting static analysis are described in Chapter 4.

At run time, the PVM gains controls and proceeds to create a protective execution environment. The execution pattern of the PVM is intertwined closely with that of the application. Throughout the execution process, several protection techniques are applied that mutually reinforce each other

and thwart dynamic attacks. Composing applications with PVMs also reduces the leakage of useful information at run time. Thus, this technique provides a robust environment for thwarting tamper attacks during application execution.

It has been proven that providing perfect protection is not possible [22]. Most protection techniques operate by increasing the effort required to obtain critical information. The goals is to have an adversary expend substantial resources (*e.g.*, monetary resources, compute cycles, memory, *etc.*), such that it would be substantially easier or cheaper to obtain the software through legitimate means, or that by the time the adversary is near achieving their goal, the software is changed thereby rendering their analysis useless. Measuring this effort provides a useful insight into the robustness of these techniques, and consequently, forms a major part of our evaluations.

In the next section, we summarize the major contributions of this research.

1.8 Contributions

The contributions of this dissertation are the direct consequence of exploring and evaluating the thesis statement in Section 1.6. These contributions can be better understood in terms of a traditional engineering framework, illustrated in Figure 1.5. In the figure, the boxes in light gray represent the different stages of a traditional engineering process. The first stage consists of formally analyzing the problem (in this case, software tamper and reverse engineering). The second stage typically involves devising solutions that address the problems identified in the previous stage. The next stage consists of prototyping the new solution schemes. This stage is followed by evaluation of the techniques, so assess their effectiveness. Often, investigation of new solutions leads to discovery of new issues. As such, engineering new solutions is often an iterative process (illustrated in the figure by the loopback from stage 5 to stage 1).

The boxes colored blue in Figure 1.5 consist the contributions of this dissertation. These contributions are listed below:

- We developed an equational model that describes applications executing via virtualization. A high-level model enables software developers to better understand the system and propose

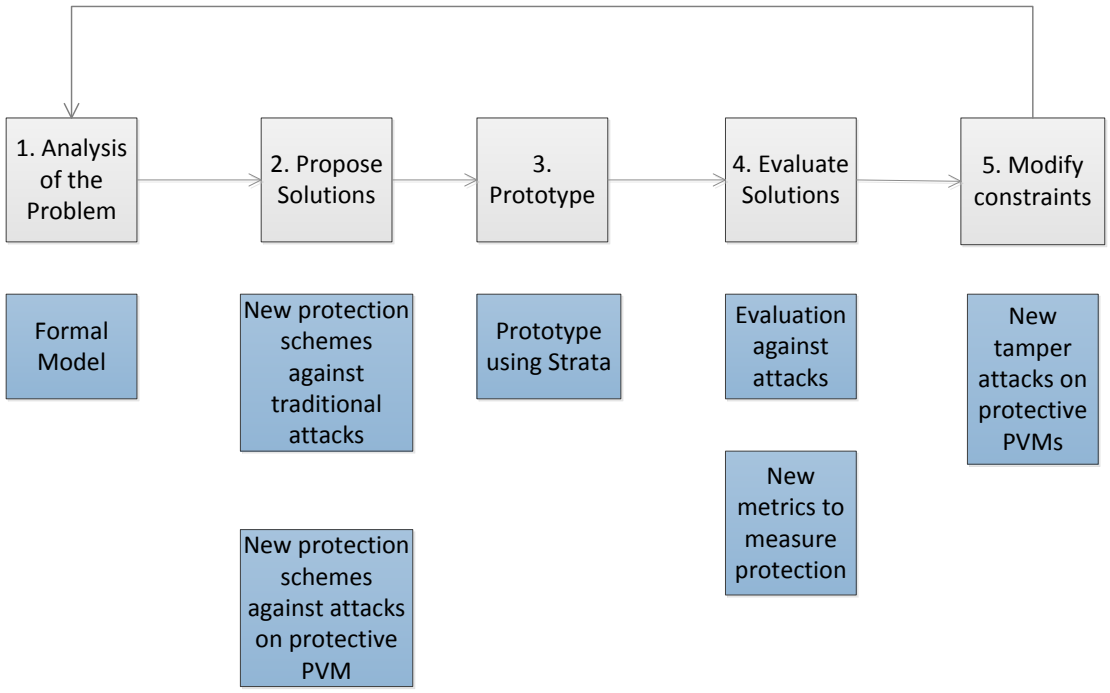


Figure 1.5: High-level overview of the the different contributions of this dissertation. The boxes in gray represent the various stages of a traditional engineering process. The boxes in blue represent the various contributions of this dissertation.

measures aimed at improving its protection. We use this model to expose a serious flaw in current virtualization systems. The model is also used to interpret various tamper-resistance techniques.

- We developed and experimentally evaluated several tamper-resistance schemes, aimed at protecting applications from reverse engineering and tamper. We examined the performance overhead of each technique, and also their robustness when subjected to known attacks. Our evaluations indicate that virtualization-based protection mechanisms provide stronger run-time protection than current techniques with acceptable overhead.

- We discovered a serious weakness in current virtualization-based program protections. This weakness arises due to the lack of data binding between the application and the PVM. Two

established use cases that exploit this vulnerability are presented.

- We developed a solution to overcome this weakness. The solution involves interleaving the data of the application and the PVM, a novel scenario in virtualization.
- Along with the design of several protection techniques, an important contribution of this research involves the design of metrics, which analytically measure different properties of the systems. These measurements provided useful insight about the strength of these protections and robustness against attacks.
- Finally, we have prepared a proof-of-concept implementation of the ideas discussed in this dissertation. Such a prototype aids in validating the practicality of our research and also, measuring the effectiveness of these mechanisms using the newly-designed metrics.

The next section describes how these contributions are presented in subsequent chapters.

1.9 Organization

This dissertation is organized as follows: This chapter describes the motivations behind this work. In particular, we describe some of the deficiencies in current research. Chapter 2 describes the concept of software virtualization, and provides a detailed description of software dynamic translation. The ideas presented in this work have been prototyped via SDTs. Chapter 3 describes a model, to facilitate better comprehension of attacks and their solutions. We utilize and extend this model to express some of the research ideas presented in this work. Chapter 4 discusses the impact of PVMs on static program protections. A novel scheme is proposed to obfuscate the application code sections by interleaving them with those of the protective PVM. Chapter 5 describes self-aware integrity checkers in the presence of software virtualization. Chapter 6 introduces the novel scheme of temporal polymorphism, which provides a continuously-shifting attack surface. This scheme makes it harder for the adversary to locate critical information, compared to current protection techniques. The focus of research then turns to weakness in the PVMs themselves. Chapter 7 discusses an attack methodology on protective PVMs, in which the adversary can dynamically replace them. A solution

is also proposed in the chapter. Chapter 8 describes an novel attack methodology on PVMs, by tracking the flow of instructions accessing critical data. This chapter also proposes a solution to thwart such dataflow-based attacks. Chapter 9 introduces the concept of composable virtualization. It provides a preliminary investigation on CVMs, and is intended to serve as a base for future research. Chapter 10 chronicles some of the past work on software protections. Finally, Chapter 11 present the conclusions of this research, and describes some avenues for further investigation.

Chapter 2

Process-level Virtualization

A brief introduction to process-level virtualization was given in Section 1.5. This chapter provides a more extensive overview of software virtualization, and lays the foundation for PVM-based tamper resistance. To begin, we provide the reasons behind the selection of PVMs as a platform for tamper resistance and code obfuscation. Next, a high-level synopsis of virtualization is provided. We then proceed to describe a PVM implementation which will be employed for prototyping purposes. Finally, we present some related works in virtualization.

2.1 Motivation

Software virtualization has been increasingly used to deliver solutions in the area of software protection [34, 11, 35]. A number of commercial products have been designed to provide software protection via process-level virtualization such as VMProtect [36], Code Virtualizer [37], and Themida [38]. A number of computer gaming software applications employ the StarForce virtualization system for copy protection and anti-reverse engineering [39]. The Terra system allows the software developer to create custom platforms where the software stack can be individually tailored to meet specific security requirements [40]. SecureQEMU uses a system-level VM to cryptographically protect the application from tamper [35]. Recently, malicious agents have used this protection technique to design state-of-the-art malware that can evade current detection systems [41].

This research focuses on the utilization of virtualization in the area of program protections. There are several reasons why virtual machines (VMs) are popular amongst security researchers.

- 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) [19]. Virtualization allows run-time monitoring and checking of the code being executed, making them an excellent tool for applying dynamic protection schemes [42]. The virtual machine can also mutate the application code as it is running (*e.g.*, changing code and data locations, replacing instructions with semantically equivalent instructions), hampering iterative attacks [34].

- 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 or to change the software development process. These properties are particularly expedient in a situation where the software developer and the software defender are not the same. The developer can create and test the application. Once assured that the application meets the required specifications, they can then transfer the binary to the defender for processing. VMs can be used to provide such a flexible capability.

- Static protection schemes can be strengthened when the application is virtualized. 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 [19], whereas decryption at a lower granularity (*e.g.*, functions) can suffer from high overhead [12, 30]. In contrast, executing encrypted applications under the control of a VM has been shown to have a better performance-security trade-off [43]. Virtualization incurs low information leakage, with the addition of a small performance overhead [34]. Another example is the improvement of the robustness of

integrity checks that are located in the application. When run under a VM, these integrity checks never execute from their original location, instead, they can be invoked from randomized locations in memory [34]. This randomization makes it harder for the attacker to locate and disable the checks.

Virtualization at both the process level [37, 36], and the system level [35], has been shown to be effective at thwarting reverse engineering. In fact, many protection techniques are oblivious to the underlying implementation (*e.g.*, both [35] and [34] use on-demand decryption of code to thwart reverse engineering). However, prototyping a protection scheme based on system-level virtualization is more complex, since the VM has to virtualize an entire operating system. Consequently, this research exclusively focuses on utilizing the PVM to thwart reverse engineering and code tamper.

The next section gives a more detailed description of virtualization.

2.2 Introducing Virtualization

Before presenting the concept of virtualization, it is useful to present the concept of computing systems and the mutual interaction with software applications that are designed to run on them. A computing system is a collection of several resources, including (but not limited to) an ALU, registers, memory and I/O devices. Each computing system has a specific Instruction Set Architecture (ISA) that controls operations. Software applications are sequences of these instructions which facilitates the computation of complex problems on these computing systems.

Computing systems have evolved in complexity considerably since the Intel 8086 microprocessors. Currently, multiple ALUs interconnect and combine with several I/O devices and networking infrastructure to provide a base platform. Consequently, even simple software applications that run on such platforms consist of millions of instructions. The key to managing such complexity, and to increasing flexibility, involves dividing the system into levels of abstraction. Traditionally, each abstraction layer encapsulates and addresses different parts of the requirements of the system, reducing overall complexity compared to monolithic architectures. The layers of abstraction are

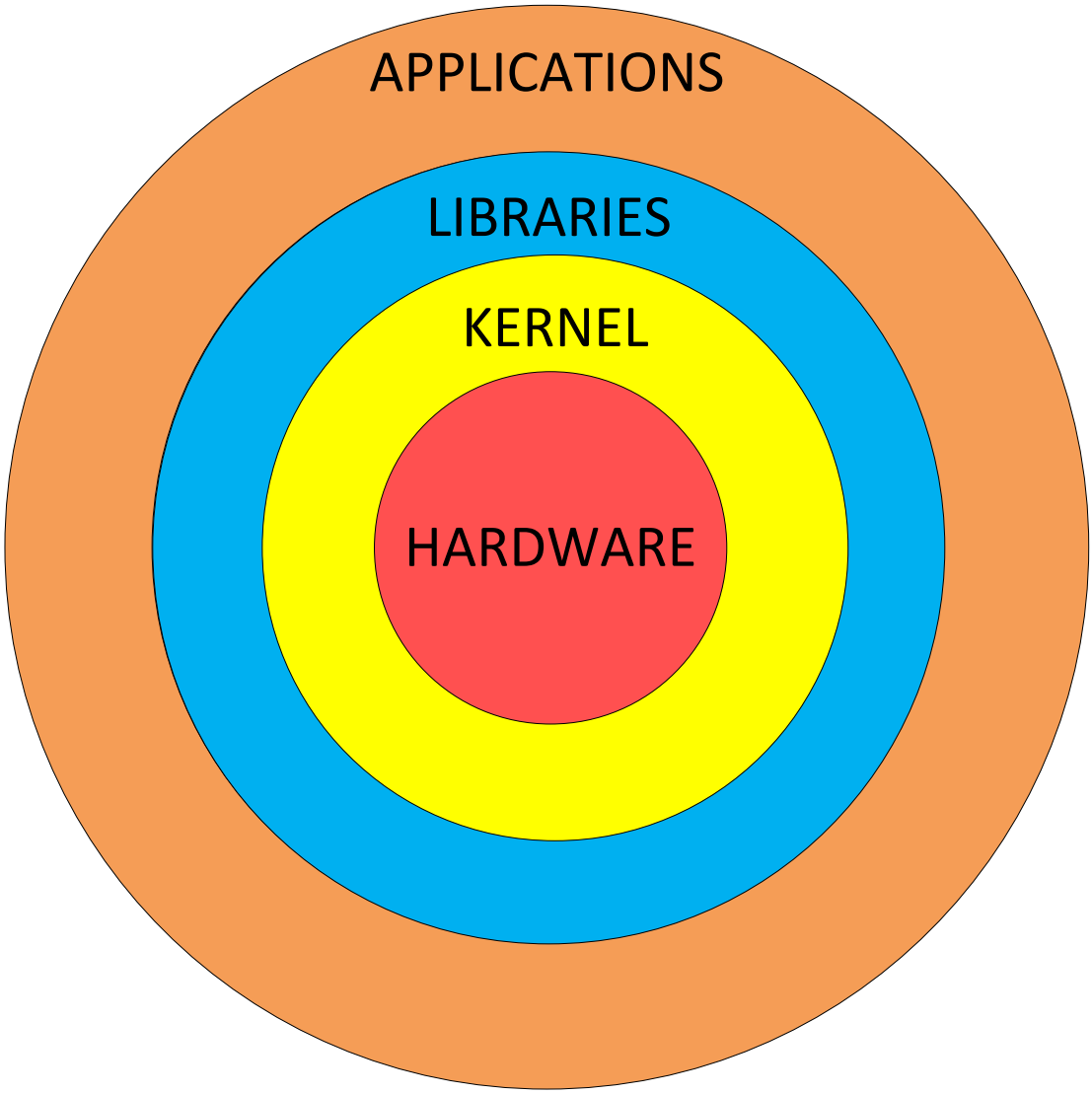


Figure 2.1: Computing systems are designed using layers of abstraction. These layers typically manage different aspects of the system that pertain to the implementation. In this particular design, the system has been partitioned into the hardware, the OS kernel, the system libraries, and finally, the applications that run on the system.

arranged in a hierarchy, with the lower layers implemented in hardware, and the upper layers in software. Each layer has a well-defined interface, allowing design tasks to be decoupled. The top of this hierarchy provides the interface through which software applications interact with the system. The operating system (OS) kernel software, application libraries and networking software implement various aspects of these layers. Figure 2.1 give a high-level overview of the design.

Virtualization involves the addition of an extra software layer, between the application and the

underlying platform. It has been used to overcome the barriers imposed by new hardware [44, 45], or to improve security [46, 47, 40]. Formally, Popek and Goldberg defined virtualization as the construction of an isomorphism that maps the resources of a virtual system (called the *guest*) onto the resources of the native system (called the *host*) [44]. It is the responsibility of the virtual machine to run the application compiled for the guest (called the *guest application*) on the host system, as the resources of the guest might not be the same as the host. 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 implemented at the system level, or at the process level. A *system-level virtual machine* (SVM) provides a complete system environment. This environment enables a single hardware platform to support multiple OSes, each running several user processes. Each guest OS can access the underlying hardware resources, including network interfaces, disk interfaces, displays, *etc.* A system-level virtual machine is often referred to as a *virtual machine monitor* (VMM in literature.

SVMs have been used to provide tamper resistance to applications. For example, SecureQEMU is an application based on the QEMU VMM that cryptographically protects the application from tamper [35]. The *Terra* system uses a trusted VMM to partition tamper-resistant hardware platform into several, isolated virtual machines, which can be configured to provide custom protection [40]. Finally, the *Overshadow* system obfuscates the memory of the guest OS from the host, preserving the privacy and integrity of the guest application even in the event of host compromise. Although such solutions are effective, they have a high overhead in terms of resources. Associating an entire OS with a single application might not be feasible in many cases, specially in the domain of embedded systems. As such, this research focuses on process-level virtualization.

Process-level virtual machine (PVM) refers to the software layer around a solitary guest application. The guest application runs under the mediation of the PVM, giving the appearance of a native process to the underlying host. Although PVMs are complex systems, they are all variations of the *fetch-decode-dispatch* method [48], which forms the basis for a standard multiple-stage pipelined processor [49]. Figure 2.2 gives a high-level overview of this scheme. PVMs based on this method

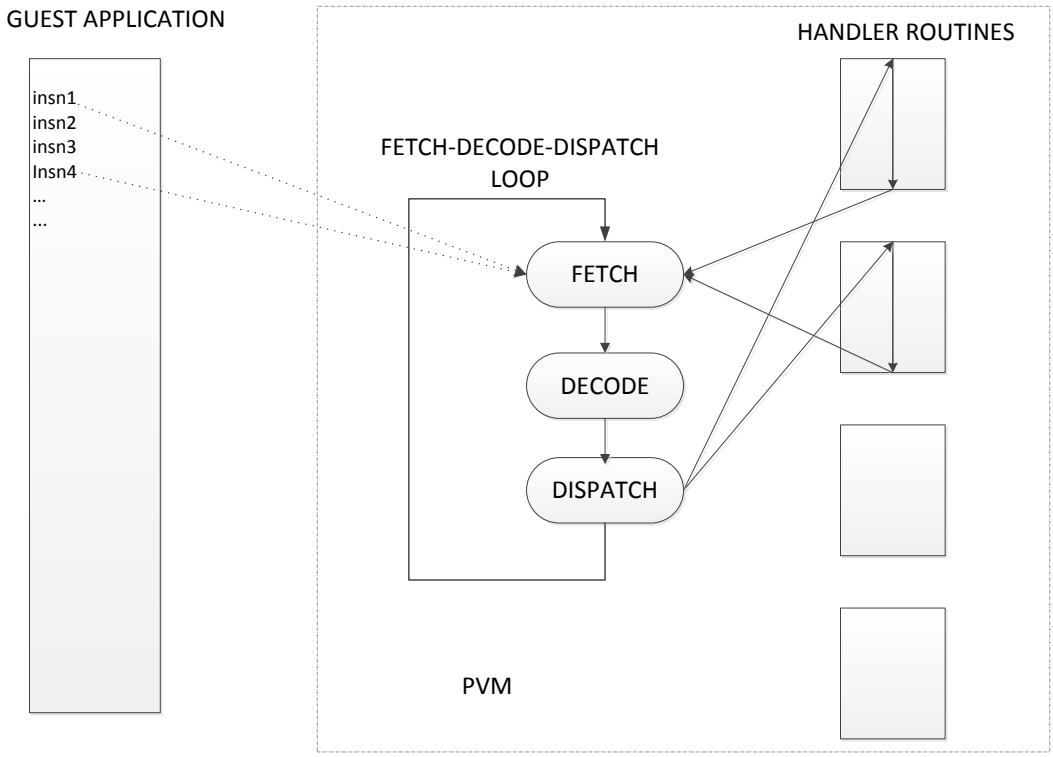


Figure 2.2: High-level overview of the *fetch-decode-dispatch* mechanism that forms the basis for PVMs. The guest application instructions are decoded, one at a time and dispatched to appropriate handling functions, which execute according to the semantics of the instructions.

consist of a central loop, which steps through the guest application, one instruction at a time, and modifies resources on the host according to the instruction. The loop consists of three main phases: the *decode* phase, which fetches one instruction, based on program order, and decodes that instruction. Based on the decoding, the *dispatch* phase invokes an appropriate handling routine. Finally, the *execute* phase utilizes the relevant operands to execute the semantics of the instruction on the host platform.

In the next section, we discuss some of the techniques that can be used to implement virtualization. We will be using one technique in particular, software dynamic translation, for prototyping our ideas.

2.3 Software Dynamic Translation

In Section 2.2, we stated that all PVMs use some variation of the *decode-dispatch-execute* paradigm [48]. In and of themselves, PVMs based on *decode-dispatch-execute* are inefficient. Such PVMs execute each instruction of the guest application using several instructions on the host machine. To enable adoption of program protection techniques based on PVMs, performance overhead needs to be reduced. As such, several optimization techniques have been designed to improve the run-time performance of PVMs.

The *threaded approach* improves performance by removing the central decode-dispatch loop [50]. The decoding and dispatching logic is appended to the execution semantics by duplicating code to the end of each handling routine. Although this technique increases code size, it improves performance on processors that have branch prediction.

Binary translation is one of the most efficient methods of virtualization. This techniques involves conversion of the guest application code block into instructions that can be directly executed on the host machines (called *translation*). The translated blocks can also cached in software to amortize the overhead of virtualization over the entire course of execution. This technique is known as Software Dynamic Translation, and PVMs implementing this technique are referred to as software dynamic translators (SDTs). The PVM only translates those code blocks that are scheduled to be executed [51]. Numerous other techniques have also been proposed for reducing performance overhead of binary translators [52, 53, 54, 55].

The protective PVM in our case study is implemented using the Strata binary translator [51, 56]. Figure 2.3 illustrates the mechanism of Strata, modified to apply 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 application's start address. This process continues until an end-of-translation condition is satisfied. At this point, Strata appends a code sequence to the block that will return control back to Strata. This code sequence is termed as a trampoline. Each trampoline is associated with an application address, containing the instruction to be executed. Consequently, Strata restores context

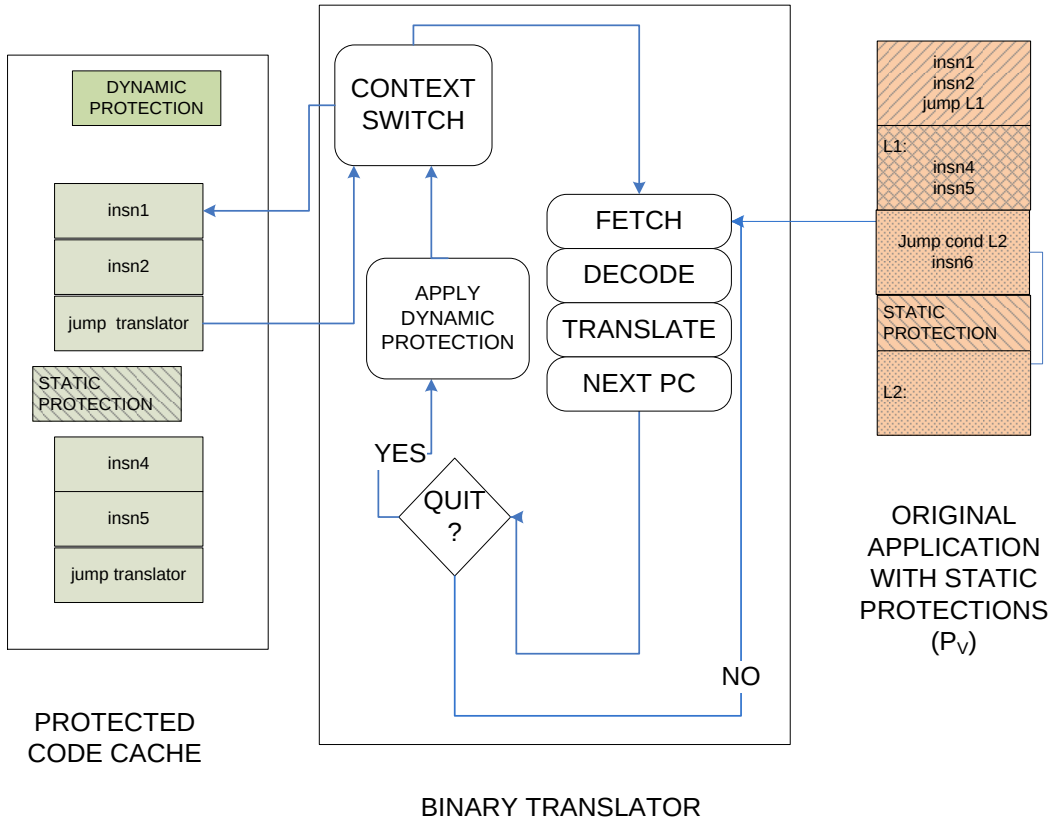


Figure 2.3: Strata Virtual Machine modified to apply dynamic protections.

and proceeds to transfer control to the newly translated block. After the block completes execution, control transfers back to Strata via the trampoline, and the next instruction to be scheduled for execution is translated.

During this process of translation, Strata can apply various protection techniques to the newly translated blocks. This dynamic nature of code creation in SDT libraries facilitates new mechanism to dynamically thwart analysis and reverse engineering, which are missing from current state-of-the-art techniques.

The translated code blocks, instead of being disposed of after execution, are cached in a region in memory (called the software cache or software cache) [46]. Strata initially searches the cache before attempting translation. If the block is found, control will be directed to the cached block. When an application instruction is translated and cached, Strata overwrites all trampolines, that are

associated with this application instruction, with direct transfers to the translated counterpart. This overwriting is known as *patching*. Each patch connects a block, previously using a trampoline to transfer control to the SDT, to the *translated* instruction. Therefore, the context switch to and from the SDT is no longer performed, resulting in reduction in overhead.

This process of caching significantly reduces the cost of translation. The figure also depicts the ways in which protection techniques can be applied by Strata. Any mechanism that is part of the original application, get translated on to a new location in the code cache. During translation itself, Strata can apply dynamic protection schemes that thwart reverse engineering and tamper.

2.4 Summary

Virtualization was first proposed more than 40 years ago, as a technique to introduce more flexibility in application execution. Recently, virtualization has proved to be a feasible platform for imparting security to unprotected applications. Improved dynamic checking and modular implementation are some of the advantages of using virtualization for imparting protections. Virtualization can be done at the system level, or at the process level. This research focuses on process-level virtual machines (PVMs), and their applicability in improving tamper resistance. This section gave a high-level overview of virtualization, and described an equational model. In the following chapters, we make use of this model to illustrate defenses and attacks on software systems.

Chapter 3

Modeling Virtualization-based Protections

During the course of our preliminary investigations, we concluded that a model was needed to concisely describe software systems and the protection mechanisms devised to protect such systems from attacks. The model would be used to describe attacks methodologies targeted at software, and provide insights that would be useful for devising techniques to thwart such attacks. To that end, this chapter presents a model of software applications and their interaction with the underlying platform. We extend the model to include virtualization, and utilize it to illustrate attacks and protections schemes. The use of a model can also expedite the understanding of any weaknesses and techniques to strengthen software. For example in Chapter 8, the model enables an elegant explanation of the dataflow-based attack on PVM-protected applications implemented by Coogan, *et al.* [57].

3.1 Model

In this section, we specify in detail our equational model. First, software applications are described. Then, the model is extended to represent tampering and reverse-engineering attacks, and their solutions, with an emphasis on software virtualization.

3.1.1 Software Applications

A software application, P_A, represents a sequence of instructions to a computing system. During application execution, the system processes these instructions along with certain inputs. Based on these instructions, the system transforms data stored in memory locations and generates outputs. The software developer identifies critical pieces of information, called assets. These assets can either aid in computation (*e.g.*, intellectual property), or serve as data to the application (*e.g.*, monetary information in a banking application). A software application, P_A can be modeled as:

$$P_A =< I_A^H, IN_A, OUT_A, ASSETS_A > \tag{3.1}$$

where I_A^H is the instruction sequence $(i_1, i_2, ..., i_n)$ for the application P_A, which are instructions in the instruction set architecture (ISA) of machine H. IN_A represents the set of valid inputs to the application, including any information that is provided by the operating system (*e.g.*, the values returned by system call invocations). OUT_A represents the set of valid outputs generated by the application, and $ASSETS_A$ are the assets of the application which the adversary is attempting to exploit.

Software applications are inputs to a computing system. During execution of the application, the Central Processing Unit (CPU) interprets the instruction sequence comprising the application in execution order to compute the outputs. During this interpretation, several partial products are generated and consumed. Each CPU has memory structures to store these values. This process of interpretation on H can be represented notationally as

$$\phi_H(P_A, in_A, m1_H) \longrightarrow < out_A, m2_H > \tag{3.2}$$

where $in_A \in IN_A$, $out_A \in OUT_A$, and $m1_H, m2_H \in M_H$, where M_H represents the memory state on the machine, H.

3.1.2 Software Interpretation

When an application executes natively, it is interpreted directly by the hardware, as represented in Equation 3.2. Interpreters can also be implemented in software, in which additional software layers between the application and the native hardware platform (the innermost layer has to be implemented in hardware). These software layers can mediate the application as it is executing, providing a dynamic execution environment. The main focus of this research is to leverage this dynamism to improve program protection.

Next, we model software interpretation. A software interpreter is a specialized type of application that is used to execute other software applications. It takes as input the instruction sequence of an application, any of the application's inputs, and its configuration settings. The instructions are then processed to generate the corresponding output. Notationally, an interpreter program, hosted on the machine H, can be represented as:

$$P_{interp} =< I^{H}_{interp}, IN_{interp}, OUT_{interp}, ASSETS_{interp} > \tag{3.3}$$

In Equation 3.3, the instructions of the interpreter application is represented by I^{H}_{interp}. The input set, IN_{interp}, consists of the combination of all applications and specific configuration setting of the interpreter, usually in the form of disk files; denoted by $\rho \times\ IN \times\ C_{interp}$ and the output set, denoted by OUT_{interp}, consists of all the applications' outputs, as well as any outputs generated by the interpreter.

To the software application, it makes no difference whether it is interpreted by a software interpreter or on hardware. At run time, its instruction sequence is executed in program order. Periodically, it issues requests for information from the execution environment (in the form of system call invocations). On a hardware platform, these requests are typically handled by associated resources (disks, CD players, *etc.*), and the information is returned to the application.

In the case of the software interpreter, it itself runs on hardware. As such, any requests it receives from the application, are transferred to the underlying platform. We can describe this scenario as a

mapping, between the resources on the virtual machine (as referenced by the application), and the actual resources of the hardware platform. In our model, when the application is interpreted by a software VM, all information related to the environment (*e.g.*, information returned by system calls) is conveyed by the set, C_{interp}.

We proceed to model the functioning of an actual interpreter. For our example, the host machine H, is any generic Intel x86 processor. In such a machine, the interpretation occurs in hardware. The machine accepts applications written in the x86 ISA. One such application is the Java Virtual Machine (JVM). The JVM performs software interpretation, accepting applications written in Java. Equation 3.7 models the interpretation of a Java application, P_Java, on a JVM.

$$P_{Java} =< I_{Java}^{JVM}, IN_{Java}, OUT_{Java}, ASSETS_{Java} > \tag{3.4}$$

Equation 3.4 represents an application written in Java byte codes, P_{Java}. It consists of the instruction sequence, I_{Java}^{JVM}. It operates on an set of inputs, denoted by IN_{Java}, and produces OUT_{Java}. The assets that enable the application to generate outputs is denoted by $ASSETS_{Java}$.

Similarly, Equation 3.5 represents a JVM interpreter, P_{JVM}.

$$P_{JVM} =< I_{JVM}^{x86}, IN_{JVM}, OUT_{JVM}, ASSETS_{JVM} > \tag{3.5}$$

Its instruction sequence is represented by I_{JVM}^{x86}. The input set of the JVM, IN_{JVM}, consists of the set $(P_{Java} \times IN_{Java} \times C_{JVM})$, where P_{Java} refers to the set of Java applications, IN_{Java} represents the inputs to the Java applications, and C_{JVM} refers to the configuration settings of the JVM. The output set of the JVM, OUT_{JVM}, consists of the $OUT_{Java} \times O_{JVM}$, where OUT_{Java} represents the output of the Java application, and O_{JVM} represents any output specific to the JVM (*e.g.*, log or error messages). The assets of the JVM are represented by $ASSETS_{JVM}$.

Thus, expanding Equation 3.5 with the new terms, we have

$$P_{JVM} =< I_{JVM}^{x86}, (P_{Java} \times IN_{Java} \times C_{JVM}), OUT_{Java} \times O_{JVM}, ASSETS_{JVM} > \tag{3.6}$$

Next, we describe the interpretation of these two applications. Equation 3.7 represents the interpretation of a Java application on the JVM:

$$\phi_{JVM}(P_{Java}, in_{Java}, m1_{JVM}) \longrightarrow < out_{Java}, m2_{JVM}) \tag{3.7}$$

The Java interpreter (the JVM) operates over the Java application P_{Java}, a particular input $in_{Java} \in IN_{Java}$, and the initial memory state, denoted by $m1_{JVM}$. This operation results in the generation of a particular output, $out_{Java} \in OUT_{Java}$, and the memory state is transformed to $m2_{JVM}$.

Next, we consider the interpretation of the JVM on the x86 hardware, represented by Equation 3.8.

$$\phi_{x86}(P_{JVM}, in_{JVM}, m1_{x86}) \longrightarrow < out_{JVM}, m2_{x86} > \tag{3.8}$$

The x86 interpreter (the CPU) operates on the JVM application P_{JVM}, with the input to the JVM represented by in_{JVM}. This input consists of a tuple $< P_{Java}, in_{Java}, c_{JVM} >$, where P_{Java} is a Java application, in_{Java} is its input, and c_{Java} is the configuration for the JVM. The initial memory state on the x86 machine $m1_{x86}$. This interpretation results in an output sequence out_{JVM} (consisting of a tuple $< out_{Java}, o_{JVM} >$, where out_{Java} refers to the output of the Java application corresponding to input in_{Java}, and o_{JVM} refers to any output from the JVM). The memory state at the end of interpretation is represented by $m2_{x86}$.

The previous equations describe nested interpretation (*i.e.*, the x86 hardware interpreting an application that is interpreting another application). We now present this nesting using a single equation. Since the x86 is the physical platform in this example, with actual memory storage, the memory locations being accessed by P_{Java} exists on that platform. Equation 3.9 illustrates the resultant equation.

$$\phi_{x86}(P_{JVM}, < P_{Java}, in_{Java}, c_{JVM} >, m1_{x86}) \longrightarrow << out_{Java}, o_{JVM} >, m2_{x86} > \tag{3.9}$$

The x86 interpreter operates on the JVM application P_{JVM}, and takes as input a 3-tuple, consisting of the Java application P_{java}, one of its inputs in_{Java}, and the initial configuration settings,

c_{JVM}. The initial memory state on the x86 machine is represented by $m1_{x86}$. The results of this interpretation consist of out_{Java}, the output of the Java application corresponding to in_{Java}, and o_{JVM}, which refers to any output specific to the JVM. The final memory state is set to $m2_{x86}$.

This example illustrates that multiple layers of software virtualization can be modeled in our framework. This feature plays a pivotal part in laying the foundation to describe composable virtualization in Chapter 9.

We now proceed to describe attacks on software using our model.

3.1.3 Modelling Tamper Attacks

Software applications often perform critical tasks. In many cases, such critical applications must conform to previously-established rules about the execution environment, or their domain of operation. For example, software controlling missile defense systems should only be operational on selected computing systems, and corporate accounting software should only run within the private network of the specific corporation, to name a few. Critical software systems also possess critical assets that are used in computation and it is often desirable to protect those assets from malicious adversaries. The goal of software tampering involves subverting these two scenarios, *i.e.*, the adversary aims to break any rules related to application execution, or to steal the valuable assets of the application. Such tasks can be achieved by reverse engineering the application and modifying the application's instruction, or providing false information to the application such that the application generates outputs under abnormal execution conditions.

Equation 3.10 represents a critical application.

$$P_{critical} =< I^{H}_{critical}, IN_{critical}, OUT_{critical}, ASSETS_{critical} > \tag{3.10}$$

The application consists of a tuple, consisting of $I^{H}_{critical}$, the instruction sequence in the ISA of the machine, H. $IN_{critical}$ refers to any inputs to this application, including any information provided by the OS. $OUT_{critical}$ refers to all the outputs that can be generated by this application, including any

error messages. Such error messages can be referred to by $ERR_{critical}$ ($ERR_{critical} \in OUT_{critical}$). Finally, the assets of the critical application are $ASSETS_{critical}$.

Attack scenarios on this application can be modeled within our framework. First, we consider attacks in which the adversary provides false information to the application, to circumvent checks that verify the execution environment. For example, an application that confirms the MAC address of the hardware platform prior to execution, can be misled by running the application on an emulator that can spoof any MAC address, and pass it to the application. To model this scenario in our framework, we first specify the interpretation of the application on the correct platform. Equation 3.11 represents the interpretation of critical application, $P_{critical}$, on a machine with MAC address $MAC1$.

$$\phi_H(P_{critical}, in_{critical}, m1_H) \longrightarrow < out_{critical}, m2_H > \mid MAC1 \in in_{critical} \tag{3.11}$$

The interpreter accepts an input sequence, represented by $in_{critical}$. The memory state changes from $m1_H$ to $m2_H$.

Running this application on a machine with a different MAC address, $MAC2$, that accepts the same ISA, will result in an error message. This outcome is represented by Equation 3.12.

$$\begin{aligned} \phi_H(P_{critical}, in_{critical}, m1_H) \longrightarrow < out_{critical}, m3_H > \mid \\ MAC2 \in in_{critical},\ out_{critical} \in ERR_{critical} \end{aligned} \tag{3.12}$$

In Equation 3.12, the hardware interprets the application, with input $in_{critical}$. In this case, $MAC2$ is the MAC address provided by the hardware, via the application input set. The application checks the MAC address to verify whether it should run successfully on this machine. In this case, the check fails, resulting in an error message ($out_{critical} \in ERR_{critical}$). The memory state is changed from $m1_H$ to $m3_H$.

To circumvent this check, the adversary can run this application on an emulator that spoofs the appropriate MAC address (in this case, $MAC1$). This spoofed information can be provided as a

configuration (*i.e.*, as part of the set C introduced in Equation 3.6). Since the emulator is itself a software, Equation 3.13 gives its representation in our framework.

$$P_{emulator} =< I^{H}_{emulator}, IN_{emulator}, OUT_{emulator}, ASSETS_{emulator} > \tag{3.13}$$

$I^{H}_{emulator}$ is the instruction sequence of the emulator, in the ISA of H. $IN_{emulator}$ refers to the inputs to this emulator, which can consist of all the applications written in the ISA of H, P_{APP}, their inputs, IN_{APP} , and configurations for the emulator, C_{APP}. The outputs of this emulator, $OUT_{emulator}$, consists of the outputs of the application, OUT_{APP}, and any outputs generated by the emulator $O_{emulator}$.

We now model the interpretation of the critical application on the emulator, with spoofed information. The interpretation is represented in Equation 3.14.

$$\phi_H(P_{emulator}, < P_{critical}, in_{critical}, c_{emulator} >, m1_H) \longrightarrow << out_{critical}, o_{emulator} >, m2_H > \ | \\ MAC1 \in in_{critical},\ MAC1 \in c_{emulator} \tag{3.14}$$

The emulator software runs on the native hardware interpreter. The input to the emulator consists of the critical application, $P_{critical}$,its inputs $in_{critical}$, and the configuration, $c_{emulator}$. The adversary can modify the emulator to pass any forged information (in this case, the MAC address) to the critical application. This forgery can be done primarily, by modifying system call returns. Thus, the application runs successfully on the machine, generating output $out_{critical}$, and output from the emulator, $o_{emulator}$. The adversary can modify the emulator to leak essential information about the critical application, such as the trace of the instructions executed. Such leakage of information forms part of the output of the emulator.

Tampering can also be achieved by modifying part of the application's instructions. Given an application P_A, tampering results in a modified application, $P_{T(A)}$ defined as follows:

$$P_{T(A)} =< I^H_{T(A)}, IN_{T(A)}, OUT_{T(A)}, ASSETS_A, > \tag{3.15}$$

Equation 3.15 models the tampered application. Potentially, any of the application's components, such as its instruction sequence, set of inputs, or set of outputs can be modified. In our model, the tampered application consists of the tampered instruction sequence ($I^H_{T(A)}$), the tampered input set, $IN_{T(A)}$ (which includes the original input set, IN_A), and the tampered output set, $OUT_{T(A)}$ (which also includes the output set of the original application, OUT_A).

Next, we model the interpretation of this tampered application on a machine, H.

$$\phi_H(P_{T(A)}, in_{T(A)}, m1_H) \longrightarrow < out_{T(A)}, m2_H > \tag{3.16}$$

Equation 3.16 illustrates the interpretation of the tampered application, $P_{T(A)}$ on host machine H, with application input in_A, and initial memory state $m1_H$. Interpretation results in the generation of output $out_{T(A)}$, and the final memory state $m2_H$. The tampered output, $out_{T(A)}$ can also contain information about a critical asset of the application (*e.g.*, the location of a password file).

Equations 3.14 and 3.16 describe software tampering at a high level. As this dissertation progresses, the equations will be modified and extended to reveal finer details about attacks and the use of process-level virtualization to improve program protection.

3.1.4 Modeling Program Protection

Section 3.1.3 described tampering of software applications. This section models the use of software virtualization to protect applications. Software virtualization is primarily achieved by the addition of a software interpretation layer between the application and the underlying platform. As we mentioned in Chapter 1, the extra layers of software are configured to apply protections during program run time.

Prior to discussing the use of virtualization in program protection, we present some basic definitions. Researchers have long been pursuing techniques to prevent successful tampering of software systems. It has been established that it is not possible to completely tamperproof an arbitrary application [22], therefore, the goal of software tamper resistance is to devise mechanisms which force the adversary to expend a large amount of effort both in terms of time and resources. These techniques usually involve applying various program transformations to the application that make it difficult for the adversary to analyze and understand the application code.

Tamper-resistance (TR) techniques are program transformations applied to application instructions that hamper the adversary from exploiting the application, while ensuring that the program's semantics remain intact. Notationally, such techniques can be described as:

$$P_{TR(A)} =< I^{H}_{TR(A)}, IN_A, OUT_A, ASSETS_{T(A)} > \tag{3.17}$$

In Equation 3.17, $P_{TR(A)}$ refers to the protected application. It comprises of an instruction sequence, $I_{TR(A)^H}$, a set of inputs, IN_A. Its execution can generate a set of outputs, OUT_A, utilizing assets $ASSETS_{T(A)}$. The adversary can modify assets by inroducing new properties and dependencies. The original assets can also be modified and removed.

Collberg and Nagra have previously designed a framework for describing tamper-resistance transformations [58, 59]. According to the authors, tamper-resistance techniques should possess the following properties:

- Correctness: dictates that the functionality of the program, P, is not affected by tamper-resistance techniques(*i.e.*, the transformations should be *semantics preserving*.

$$\left.\begin{aligned} \phi_H(P_A, in_A, m1_H) &\longrightarrow < out_A, m2_H > \\ \phi_H(P_{TR(A)}, in_A, m1_H &\longrightarrow < out_A, m2_H > \end{aligned}\right\} \forall\, in_A \in\ IN_A \tag{3.18}$$

  Equation 3.18 models the interpretation of a standard application P_A, and a protected application $P_{TR(A)}$ on the same platform, H. In both cases, input in_A is provided. The output

should be identical in both cases (represented by out_A). The memory state changes from $m1_H$ to $m2_H$.

- Soundness: implies that the probability to successfully tamper with a protected application, $P_{TR(A)}$, in polynomial time is close to zero.

$$\underset{Time=O(n^k)}{Prob}\left\{\phi_{T(H)}(P_{T(TR(A))}, in_A, m1_{T(H)}) \longrightarrow < out_A, m2_{T(H)} > \right\} < \epsilon \ \forall in_A \in IN_A \quad (3.19)$$

Equation (3.19) considers a protected application that has been subjected to tamper, represented by $P_{T(TR(A))}$. The probability that interpretation of such an application with valid inputs in_A will result in a valid output, out_A in polynomial time $O(n^k)$, is less than ϵ). For a tamper-resistance technique to be considered strong, ϵ should be close to zero.

This model can be used to describe concepts in tamper resistance, including vulnerabilities and their solutions. As an example, we use this model to describe Authentication.

3.1.5 An Example: Authentication

Authentication is the one of the key components of many software software systems, *e.g.*, email, banking, shopping. It is defined as the binding of an identity to an entity. This identity controls the entity's actions in the system [60]. In this particular example, the software application possesses encrypted data. It can take as input any combination of characters ($.*$). If the correct pass phrase is provided (in this case, $pass$), the application decrypts its data and provides it to the user (denoted by D). Otherwise, the applications terminates with an error message (err).

The system can be represented as:

$$IN_A = \{pass, .*\}$$

$$OUT_A = \{D, \ err\}$$

$$ASSETS_A = \{pass\}$$

where the input set IN_A consists of the phrase *pass*, and all other string combinations. The output set OUT_A consists of the decrypted output D, and an error message, err. The assets of the application in this scenario consist of the pass phrase *pass*. In other scenarios, the assets could also include the encryption algorithm. We ignore the algorithm as an asset in this example.

We assume that the application runs on the Intel x86 platform. One way to represent the application is:

$$P_A =< I_A^{x86},\ \{pass,\ .*\},\ \{D,\ err\},\ \{pass\} >$$

We adapt Equation 3.2 to describe the execution of this application, when the correct pass phrase is provided.

$$\phi_{x86}(P_A, pass, m1_{x86}) \longrightarrow < D, m2_{x86} > \tag{3.20}$$

In Equation 3.20, the application P_A is interpreted by the x86 platform. The input in this case is *pass*. In this case, the application successfully outputs the decrypted data, D, and the final memory state is $m2_{x86}$.

Next, we present the equation representing application interpretation when an incorrect input is provided.

$$\phi_{x86}(P_A, abcd, m1_{x86}) \longrightarrow < err, m3_{x86} > \tag{3.21}$$

The interpreter runs the application, with input *abcd*. In this case, the application generates the error message, err and terminates.

Next, the adversary subjects the application to a tampering transformation, resulting in $P_{T(A)}$. The goal of the adversary is to generate output D, without providing the correct input. Assuming the adversary is successful, such a scenario can be represented by:

$$\phi_{x86}(P_{T(A)}, abcd, m1_{x86}) \longrightarrow < D, m4_{x86} > \tag{3.22}$$

In this case, the interpreter executes the tampered application, $P_{T(A)}$. The application takes as input, *abcd*. On successful interpretation, the tampered application outputs D.

Finally, the software defender applies tamper-resistance transformations to the application, which results in $P_{TR(A)}$. Assuming the protections are successful, the execution of such an application when the adversary subjects in to tamper can be represented as:

$$\phi_{x86}(P_{T(TR(A))}, abcd, m1_{x86}) \longrightarrow < err, m5_{x86} > \tag{3.23}$$

In Equation 3.23, $P_{T(TR(A))}$ refers to a protected application that has been subjected to tamper. Since the protection techniques are successful, the tampering is ineffective. Thus, the input *abcd* generates the error message, err.

This example illustrates the applicability of this model to describe tamper in software applications. Over the next few chapters, we will be utilizing this model to describe tamper-resistance techniques based on PVMs. We summarize our design in the next section.

3.2 Summary

In the area of software security, formal modeling often aids in understanding the scope of protection techniques. Directly implementing such techniques can lead to ineffective and weak systems. A formal model enables researchers to analyze the techniques and eliminate any weaknesses in the design, if found. This chapter introduced the framework for an equational model that describes tamper resistance. The model can be used to portray software applications, virtualization and program protections. As this dissertation proceeds, this model will be extended to accommodate new attack methodologies and protections against such attacks. The goal of this model is to facilitate the synthesis of a strong foundation for PVM-based program protection techniques.

Chapter 4

Creating Tamper-resistant Binaries

In Section 1.3, we stated that adversaries are increasingly targeting the application at run time, to exploit critical information and break software protections. Consequently, the focus of this research is providing dynamic protections to the application. However, protecting the on-disk binary is still important, as dynamic protections are ineffective without complementary static protections.

In this chapter, we investigate composition of applications and PVMs to thwart static analysis. In particular we focus on an established technique, *code encryption*, and study the effects of virtualization on it. The main contribution of this chapter is a static obfuscation technique, that makes it algorithmically harder to extract the application code from the protected package (*i.e.*, the software package containing the application and the PVM). This static technique is based on random permutation of code blocks, and we demonstrate that technique improves the static protection of the package over current state of the art.

The remainder of this chapter is organized as follows: Section 4.1 discusses code encryption in a virtualized application. In Section 4.2, we describe and evaluate random code-block permutation between the application and the PVM. Finally, in Section 4.3, we discuss past research work that hampers static analysis of applications.

4.1 Encryption

Code encryption is an established technique to defeat static analysis [12, 19, 61, 35]. Without the decryption key, the level of effort required to obtain a disassembled version of the code for analysis is significantly increased.

Previous mechanisms often included the logic for decrypting the code within the application itself [12, 19]. A common problem with such techniques is the coarse granularity of decryption at run time. If the granularity is too coarse, the adversary can use simple run-time monitor to obtain the information needed to then perform static disassembly. However, fine-grained granularity can result in high run-time overhead. Previous techniques were not able to achieve an adequate balance between overhead and information leakage.

This research investigates techniques to reduce run-time leakage of information to the adversary at with low overhead by delegating the task of decryption to the PVM. The application is encrypted as in previous techniques, but the decryption is now controlled by the protective PVM. Such a configuration facilitates a much easier design of encrypted application packages. At run time, the PVM mediates the trade-off between performance overhead, and the rate at which plaintext code is exposed to the attacker. The bulk of our analysis of an PVM-protected application that uses encryption is described in Chapter 6. In this chapter, we focus on modeling PVM-enabled encryption to prevent static analysis.

Recalling Equation 3.1, a software application, P_{APP} can be represented by a 4-tuple, consisting of its instruction sequence in the ISA of the target platform, a set of inputs, a set of outputs, and its assets. An encrypted application, $P_{E_k(APP)}$, is produced by applying an encryption algorithm to the code section of the application, with k as the decryption key. This modified application can be described in our model, as in Equation 4.1

$$P_{E_k(APP)} =< I_{APP}^{E_k(H)}, IN_{APP}, OUT_{APP}, ASSETS_{APP} > \tag{4.1}$$

The first component of the 4-tuple consists is $I_{APP}^{E_k(H)}$, which is the instruction sequence of the

encrypted application. The ISA is represented by $E_k(H)$, *i.e.*, the encrypted form of the original ISA. The inputs, outputs and the assets of the application remain unchanged.

Compared to the application described in Equation 3.1, this transformed application requires the presence of the decryption key, k for successfull interpretation. Recalling Equation 3.2, interpretation of an encrypted application can be illustrated as:

$$\phi_H(P_{E_k(APP)}, in_{APP}, m1_H) \longrightarrow < out_{APP}, m2_H > \quad |k \in in_{APP} \tag{4.2}$$

In Equation 4.2, the interpreter operates on the encrypted application, $P_{E_k(APP)}$, with an input sequence in_{APP}. The decryption key, k can be considered part of the input. Successful interpretation of the application results in generation of output, out_{APP}. The memory state is transformed from $m1_H$ to $m2_H$.

In our research, we delegate the decryption of the application to the protective PVM. Its representation was first presented in Equation 3.3. The PVM takes as input, any application, the application's inputs, and a configuration setting. The PVM generates the output of the application, as well as its own output.

$$P_{PVM} =< I^H_{PVM}, IN_{PVM}, OUT_{PVM}, ASSETS_{PVM} > \tag{4.3}$$

Here, I^H_{PVM} refers to the instruction sequence of the PVM. The input IN_{PVM} consists of a member of the set $\rho_{APP} \times IN_{APP} \times C_{PVM}$, where ρ_{APP} refers to the class of all applications, IN_{APP} refers to their inputs, and C_{PVM} refers to the configuration settings for the PVM. the output set OUT_{PVM} is composed of the outputs of the applications, OUT_{APP}, and its own outputs, O_{PVM}. $ASSETS_{PVM}$ refers to its assets.

The PVM is interpreted by the underlying hardware, H. This nested interpretation is modeled as:

$$\phi_H(P_{PVM}, < P_{APP}, in_{APP}, c_{PVM} >, m1_H) \longrightarrow << out_{APP}, o_{PVM} >, m2_H > \tag{4.4}$$

In Equation 4.4, the hardware interpreter(ϕ_H) operates on the PVM application, P_{PVM}. The input to the PVM consists of the original application, P_{APP}, its inputs in_{APP}, and configuration settings c_{APP}. On success, the interpretation operator generates an output consisting of out_{APP} of the application, and o_{PVM} of the PVM. The memory state is transformed from $m1_H$ to $m2_H$.

Next, we consider encryption in the presence of virtualization. As we first introduced in Equation 4.1, encryption transforms the original application, P_{APP}, to $P_{E_k(APP)}$. To run this encrypted application, its associated decryption key, k is required at run time. Since the PVM is responsible for decryption, the key forms part of the configuration setting for the PVM. Thus, modifying Equation 4.4 to represent the interpretation of the encrypted application, we have:

$$\phi_H(P_{PVM}, < P_{E_k(APP)}, in_{APP}, c_{PVM} >, m1_H) \longrightarrow << out_{APP}, o_{PVM} >, m2_H > | \quad k \in c_{PVM} \tag{4.5}$$

In Equation 4.5, the hardware interpreter function, ϕ_H, operates on the PVM. The input to the PVM is a 3-tuple, consisting of the encrypted application, $P_{E_k(APP)}$, its input sequence in_{APP}, and a configuration setting for the PVM, c_{PVM}. The decryption key forms part of the configuration settings. On successful interpretation, the outputs generated consist of out_{APP}, the output of the encrypted application, and o_{PVM}, the output of the PVM. The memory state is transformed from $m1_H$ to $m2_H$.

Now that the model for PVM-enabled encryption has been established, we proceed to describe the implementation of this technique.

4.1.1 Implementation

In this section, we describe the implementation of a PVM-protected binary that uses encryption. To prepare the application binary for encryption and virtualization, we adopted the previous work done by Hu, *et al.* [43]. The basic framework for creating the protected application package consists

of Diablo, a binary rewriting tool created by researchers at the University of Ghent [62]. We use Strata,which was first introduced in Section 2.3, as the protective PVM used in our experiments, .

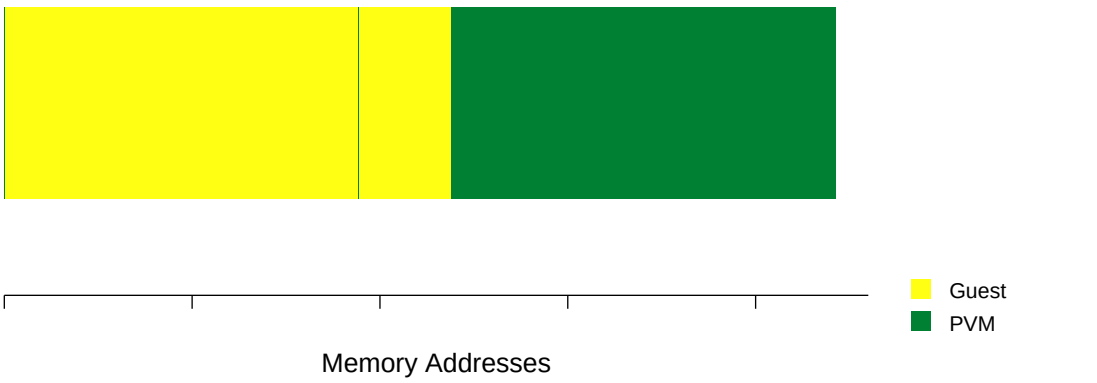
Diablo operates on object files and libraries constituting the package (Strata and the application). Diablo translates object and library code into an internal representation, and is able to generate instruction representations and the Control Flow Graph of the application. It provides a wide range of APIs, to facilitate analyzing and modifying the CFG. After the user performs custom modifications, Diablo regenerates the CFG, and produces target-machine instructions. Finally, it emits the modified binary [43].

In our experiments, the application objects files and the Strata libraries were input to Diablo. Based on the name of the module from which the code originates, Diablo is able to differentiate application and Strata code. Once all the application code is identified, Diablo proceeds to tag such code segments for encryption. Next, the target-machine instructions (in this case, x86) are generated. The tagged instructions are then encrypted using 128-bit AES. Finally, the encrypted application instructions and Strata code is written to file. Further details of this scheme can be obtained from Hu *et al.* [43].

Figure 4.1 illustrates the layout of the protected on-disk binary, consisting of the encrypted application and the Strata. As the figures demonstrates, the code segments for the two components are largely distinct. In the figure, some sections in the application segment appear to have the same color as the PVM blocks. This discrepancy is purely an implementation issue. Currently, we mark any block that is encryptable, as belonging to the application. In our framework, the entry and exit functions of Strata are inserted manually into the application. These compiler-generated code sections, such as `_startup` and `_exit`), that execute before Strata's entry function, or after its exit, can not be encrypted and as such, get tagged as belonging to Strata).

On the whole, 99% of the Strata code is contiguous, which means that most of the critical application code is concentrated in a smaller region. Therefore, the adversary can localize his analysis on that region.

As we described in Section 2.3, Strata performs just-in-time, on-demand decryption. At run



]

Figure 4.1: Layout of the on-disk binary consisting of the guest application and the protective PVM. The guest code is encrypted. A few libraries of the PVM end up amongst the application code.

time, Strata loads the application and starts decrypting and decoding application instructions one basic block at a time. The block is cached in software and scheduled for execution. After the block executes, control again enters the Strata. If next instruction to be executed already exists in the code cache, control directly transfers to that instruction. Otherwise, Strata begins decrypting and decoding instructions at the next application address. The translated block is then placed at the next available location in the code cache. We analyze the run-time performance and information leakage extensively in Chapter 6.

4.1.2 Key Protection

A pertinent issue with any encryption scheme involves the security of the decryption key. Most encryption algorithms where initially designed to be deployed in a 'black-box' environment (*i.e.*, the adversary has access to only the inputs and outputs of the cryptosystem). Such a restricted environment does not apply in the attack model for software applications. Our techniques are attempting to protect applications running on 'open' systems (*i.e.*, PCs, tablets *etc.*). We denote this context as the white-box attack context. As we described in Section 1.3, in such a context, a 'white-box' adversary has full access to the software implementation of a cryptographic algorithm.

Hence, the implementation itself is the sole line of defense.

An encryption algorithm is only as secure as its key. If an adversary is able to analyze the cryptographic implementation, and identify the keys, the encryption algorithm is vulnerable, regardless of its complexity. Shamir and Van Someren demonstrated that cryptographic keys can be extracted from poorly-designed encrypted binaries [63]. Their attack could theoretically be applied to any data container: binaries, computer memory, and so forth. Halderman, *et al.* were also able to design techniques to extract keys from the DRAM of a computing system [64]. In 2012, Zhang, *et al.* used a system-level VM to extract information about cryptographic keys from an encrypted application running on the guest OS [65]. All these examples demonstrate that this attack methodology is realistic. Thus, modern cryptosystems are incomplete in security, unless their keys are protected as well.

The presence of this weakness led to the development of *white-box cryptography* (WBC). WBC is defined as an obfuscation technique intended to implement cryptographic primitives in such a way, that even an adversary who has full access to the implementation and its execution platform, is unable to extract key information [21]. Cryptographic key information should be spread over the entire implementation, forcing an adversary to analyze the whole implementation, instead of focusing on individual parts (that could for example, be identified by an entropy study [63]). The first WBC implementations were presented by Chow, *et al.* in 2002 on the Data Encryption Standard (DES) [66], and the Advanced Encryption Standard (AES) [21] respectively. Their white-box techniques transform a cipher into a series of key-dependent look up tables. The secret key is hard-coded into the lookup tables and protected by static randomization techniques. Although the initial implementations were subjected to cryptanalysis, and subsequently broken [67, 68], researchers have been able to design more robust implementations [69]. They have been able to successfully demonstrate that extracting keys from modern WBC implementation is extremely difficult [70].

The main concerns with white-box cryptography are the performance and size penalty. The performance issues limit the utilization of white-box cryptography for high-throughput applications, while size issues constrain its use in cost-sensitive embedded systems. These concerns can be addressed

using special-purpose white-box implementations which can exploit hardware accelerators [70].

Nonetheless, white-box cryptography has become a cornerstone in the protection of cryptographic primitives in applications that run in hostile execution environments. White-box cryptography is quickly gaining momentum in commercial applications. White-box cryptography is used in products by companies such as Microsoft, Apple, Irdeto, Arxan, and many more are actively investigating white-box techniques [71]. It would be relatively straightforward to choose an established white-box cryptography algorithm, and implement it as part of our prototype. Therefore, due to the wealth of available research in this area, we defer to previously-established work in white-box cryptography, and focus our attention on PVM-enabled protections.

4.2 Interleaving Application and PVM code

On closer analysis of Figure 4.1, it can be seen that the protected binary has two discernible components, corresponding to the PVM and the application respectively. Even if the guest application is encrypted, such a distinct layout reduces the search space for the adversary and divulges the region to attack. The knowledgeable adversary could concentrate their static analyses on just the encrypted portion of the binary and extract useful information.

Composing an application with a PVM facilitates techniques to increase the search space for the adversary and thwart static analysis. During software preparation, code blocks from the application and the PVM can be permuted in random manner. In such a case, the adversary will have to search the entire binary in an attempt to extract relevant information. The effectiveness of this scheme depends on the relative number of code blocks of the application and the PVM. As the relative PVM code blocks decrease in number, so does the strength of this protection scheme. We address this issue further in our section on evaluation.

We now proceed with the description of our implementation, and the evaluation of its strength against static analysis.

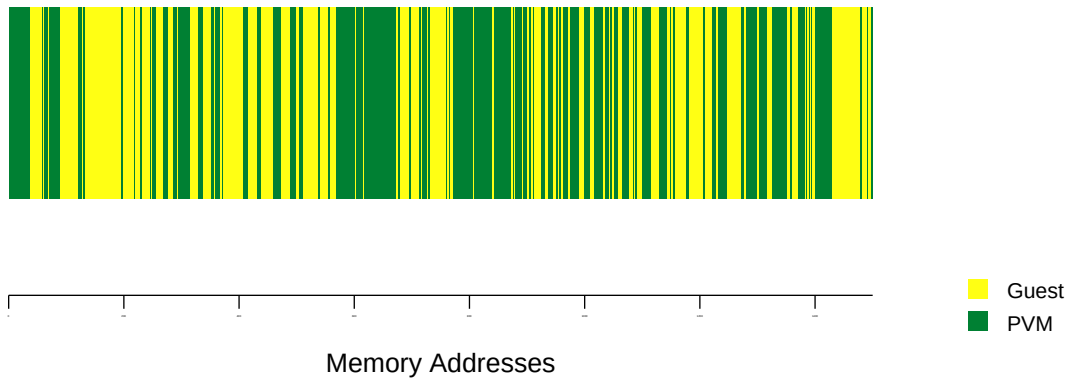


Figure 4.2: Layout of the code regions after the instruction blocks have been randomly permuted.

4.2.1 Evaluation

To obtain the maximum amount of obfuscation via interleaving, the granularity of interleaving should be at the basic block level. In this case, each block is likely to be permuted with another block. However, such a scheme could lead to high run-time performance overhead. Traditional compilers lay out blocks that are likely to execute together in close proximity in the binary file (called *chained blocks* in Diablo terminology [62]). Changing the layout of such connected blocks changes the alignment, and leads to a high overhead. Thus, in our experiments, the granularity of interleaving is a chain of basic blocks.

There are several algorithms for generating a random permutation of a finite sequence. In our experiments, we utilized the Fisher-Yates shuffling algorithm [72]. During the protected package creation, Diablo creates chains of basic blocks that are likely to be executed together. We modify Diablo, to apply the Fisher-Yates algorithm on these chains. Consequently, the blocks are converted to target-machine instructions and written to file.

Figure 4.2 illustrates the layout of a protected binary after the process of code-block permutation has been applied. There are no discernible target areas where the adversary could focus their static analyses. Thus, this scheme increases the attack surface for the adversary. In the next section, we describe our experiments to support this assertion.

Measuring Randomness

Figures 4.1 and 4.2 reveal that random permutation increases the search space for the adversary, by adding randomness to the layout. We would like to demonstrate algorithmically that random permutation indeed improves the static obfuscation. Giacobazzi have demonstrated that obfuscation is directly related to the amount of entropy in the system [73]. Thus, we need a technique to measure the relative *entropy* of the two layouts.

This question of measuring entropy has long been an area of interest for security researchers [74, 75, 76, 77]. Entropy is often directly related to the complexity of the system. The goal of security techniques often involves increasing the complexity, to make it hard for the adversary to predict its characteristics. Therefore, a metric that facilitates measurement of entropy of a security technique can be used to demonstrate its effectiveness.

After much investigation, we decided to adopt the work of Hansel, Perrin and Simon in the area of entropy measurement. In their seminal work published in 1991, the authors proved that the compression rate, and the entropy of a set of characters are closely related [78]. The main outcome of their work was to demonstrate that the compression rate of a finite string is the upper bound on the entropy of the string (*i.e.*, $\xi(X) \leq \tau(X)$ where X is a finite character string, ξ represents its entropy, and τ represents its compression ratio). They also demonstrated that the Ziv-Lempel compression algorithm designed in 1978 (LZ78) [79], provides the most optimal comparison with entropy values (*i.e.*, $\xi(X) = \tau_{LZ78}(X)$). This algorithm is implemented by the UNIX tool `compress`, further facilitating the adoption of Hansel *et al.*'s work for our analysis.

We compare the relative entropies as follows: Each basic block of the protected package is assigned a bit ID; '0' if it belongs to the PVM, and '1' if it belongs to the application. Then, a bit string is created based on the layout for the normal case, as well as the case where random permutation is employed. We then apply the `compress` tool to both these strings, and compare the sizes of the compressed outputs. Table 4.1 displays the results.

For the benchmark `181.mcf`, the original bit string size was 52983 bytes, for both cases. On compression, the bit string representing normal layout resulted in a string with 555 bytes. The bit

Benchmark	Bit-string size (bytes)	Compressed size (normal layout) (bytes)	Compressed size (random permutation) (bytes)
`181.mcf`	52983	555	4950
`253.perlbmk`	88381	714	6110
`176.gcc`	145483	889	7565

Table 4.1: Results of applying compression to bit strings describing layout for the normal case, and with random permutation applied to the blocks. This technique was applied to `181.mcf`, `253.perlbmk`, and `176.gcc`. `181.mcf` and `176.gcc` represent the opposite ends of the spectrum in terms of bit string sizes.

string representing the layout where random permutation was applied, resulted in a string of size 4950. Thus, using Hansel, *et al.*'s theorem, we can state that random permutation of basic block chains resulted in a binary that had 10 times more entropy. Thus, using random permutations on `181.mcf` results in static protections which were an order of magnitude stronger than the standard scenario. We observe similar entropies between the two cases for both `253.perlbmk`, and `176.gcc`.

A point to note is that the strength of this protection depends on the relative number of blocks for the application and the PVM. As the application code blocks increase in number, the amount of entropy obtained by random permutation decreases. This finding can be deduced by comparing entropies across benchmarks. The bit string representing the protected package for `176.gcc` is three times larger than the package for `181.mcf`, and 1.64 times larger than `253.perlbmk`. The number of blocks belonging to Strata is the same in all three benchmarks. Comparing the compressed bit strings for the benchmarks under random permutation, we observe the `176.gcc` is only 1.6 times larger than `181.mcf`, and 1.24 times larger than `253.perlbmk`. Thus, the rate of increase in entropy is much slower than the increase in number of blocks.

Confounding Static Assemblers

In this section, we investigate the effect of encrypted blocks on static disassemblers, which are a key tool for any adversary. Analyzing the output of a disassembler might aid the adversary in locating critical information (that is encrypted), and help to localize their attacks.

Our investigations revealed that several encrypted blocks of the application were erroneously identified as valid instructions. This phenomenon arises from the variable-length, CISC-like nature

Benchmark	Application size (number of ins.)	False positives (normal layout) (number of ins.)	False positives (random permutation) (number of ins.)
`181.mcf`	147070	112914 (76.71%)	113678 (77.29%)
`253.perlbmk`	285347	207247 (72.66%)	206219 (72.27%)
`176.gcc`	495013	334166 (67.50%)	335004 (67.67%)

Table 4.2: This table displays the results of running a disassembler (`objdump`) on applications. The second column displays the total number of static instructions in the binary file (not considering the PVM). The third column displays the number of instructions that were falsely identified as valid instructions (and the percentage over total count) for encrypted binaries with normal layout (labeled as false positives). The final column displays the the number of instructions that were falsely identified as valid instructions in encrypted binaries under random permutation. The code sections for the PVM are not considered.

of the ISA (in this case, the Intel x86 32-bit version). Consequently, static disassemblers misidentify encrypted byte strings as valid instructions and then attempt to use established idioms to obtain higher level information [80]. For example, on the x86 32-bit platform, several encrypted byte sequences were identified as return instructions. Table 4.2 displays the results of applying a standard disassembler (`objdump`) on three applications for the Intel x86 32-bit platform. In this analysis, we do not consider the code sections that belong to the PVM.

The second column displays the total number of static instructions that correspond to the application. The third column displays the number of instructions that were falsely identified as valid in the encrypted application. Since these values are from the binary with normal layout, all these instructions are likely to be located contiguously in the binary. On average, the number of false-positive instructions generated by the disassembler is 72% of the original application size. This number indicates that the encryption does a good job of obfuscating static analysis tools.

The final column displays the false-positive instructions for the layout generated by random permutations. In this case as well, the number of false-positive instructions generated by the disassembler is 72% of the original application size. However, these false positive instructions would be harder to identify on account of their spread in the binary (as demonstrated by Figure 4.2). For example, in the case of `176.gcc`, the average number of encrypted blocks was calculated as 700, per 1000 blocks of the protected package, in both layouts. In the normal layout, the standard deviation was approximately 400, while in the case of the layout generated by random permutations, the

standard deviation was 149.2. These numbers imply that, in the normal layout, while sampling the protected package at the rate of 1000 blocks, the number of encrypted blocks that are likely to be encountered is either very high, or very low, since the standard deviation is large (400). Therefore, for the normal layout, the encrypted blocks are likely to be localized to specific intervals. A knowledgeable adversary could potentially target those intervals with a high count of encrypted blocks, and extract valuable information. For the layout generated by random permutations, the standard deviation is lower (149.2), which implies that, while sampling this configuration, the number of encrypted blocks encountered is likely to be closer to the average, *for every sample interval*. That is, the number of encrypted blocks stays roughly similar for each interval, and the encrypted blocks are more likely to be spread throughout the package. Therefore, an adversary would require more effort to establish whether a block is valid or not, while analyzing the layout generated by random permutations.

In summary, combined together, encryption and random permutation of code blocks provide a strong resistance to static disassembly of the protected binaries. The adversary would require extra effort to obtain useful information for binaries protected using these approaches. In the next section, we discuss related work in this area.

4.3 Related Work

Obstructing static analysis of applications has been researched extensively. Primarily such techniques aim to thwart the ability of the adversary to extract useful information from the on-disk binary. Such information helps the adversary gain high-level knowledge about the constructs and functionality of the application.

Several solutions aim to break the decoding algorithms of traditional disassemblers. Code encryption forms part of that solution set. Such systems typically rely on an accompanying interpreter to execute the application. Examples of such systems include Proteus [11], in which the original application was translated to a unique instruction set, and interpreted at run time by a specific instance of an interpreter. There are several commercial tool available which use the same idea [36, 37, 39].

A number of techniques attempt to conceal the control flow graph (CFG) of the application. Wang, *et al.* proposed *flattening* the CFG, i.e., replace the direct jumps in the application with a switch table [9]. The next block of instructions to be executed is determined by the value of global valuer, and the resulting flow graph loses its structure. Collberg, *et al.* suggested using spurious pointer aliasing to obstruct the disclosure of indirect jump targets. Collberg, *et al.* also proposed adding bogus control flow statements in the application. The resulting graph has branches which are never taken, branches which are always taken, and branches which are redundant (*i.e.*, their execution does not affect the semantics).

Debray, *et al.* proposed the use of *branch functions* to replace direct branches [23]. Instead of a straight jump to the target address, control flows through a special function that performs a complicated calculation to derive the target based on the caller address, and then executes a return instruction (*i.e.*, a `ret`) to the target. Conventional disassemblers expect control to flow back to the instruction immediately following the `call` instruction, but branch functions do not let control return to that instruction. Thus, using branch functions is an attempt to not only obscure the target of direct branches, but also confuse traditional disassembly.

Popov, *et al.* also proposed replacing control transfers with signals and inserting dummy code after the signals [81]. Static dissassembly will fail to extract the correct control flow of the program. However, the application will function correctly because the signal handlers are able to patch the control at run time.

Most of the techniques described in this section are complementary to PVM-based protection and can be combined with it. These techniques have been rigorously evaluated and offer robust protection against static analysis. However, most of these schemes can be subverted using dynamic means. Protecting the application at run time still needs to be addressed.

4.4 Summary

The goal of PVMs is to provide dynamic protection to applications. However, PVMs can be utilized to thwart static analysis as well. In this chapter, we discussed ways in which the PVM can improve

the protection offered by encryption of application binaries. Code encryption hinders the adversary from using traditional disassemblers on the binary. Secondly, random permutation of code blocks increases the adversary's attack surface. We employ an established algorithm to prove this assertion. Our evaluations indicate that these techniques make static analysis harder.

Chapter 5

Strengthening Tamper Detection Using PVMs

The previous chapter described the protection of the PVM-application package on disk. From this point forward, we focus our attention to the run-time protection of the application, the major capability of PVMs. This chapter investigates tamper detection of code in the presence of virtualization. Tamper detection consists of techniques that ensure the application is unchanged after creation. Such techniques have proven to be effective in protecting traditional application execution from tamper [2, 3]. Composing applications with PVMs changes the run-time environment significantly. This chapter describes the impact of virtualizing applications on traditional tamper-detection techniques. Our research also demonstrates that virtualization opens up new opportunities for protecting code from tamper.

This research focuses on code integrity as the foundation to detect unauthorized tamper. Whenever a code segment is generated (at software creation time or during application execution), a checksum is created over it. Later, when that segment is scheduled for execution, the checksum is recalculated and compared with the previous value. Any mismatch indicates that the code has been tampered and triggers an appropriate response mechanism.

The major contributions of this chapter are:

- Development of novel checksumming techniques via PVMs that are stronger than previous checksumming techniques. With the dynamism provided by PVMs, checksumming mechanisms can be relocated continuously making their identification and removal more difficult.

- Design of a multi-dimensional checking capability that provides a cyclical protection scheme. The checks are located in three regions: the PVM, the application, and the software code cache. The protection domain can include any of these regions. Previous attacks have been successfully disabled tamper-detection features in part because such schemes operated on a singular dimension (*i.e.*, the application) [19].

- Conception of novel schemes that protect code generated by the PVM at run time. These schemes can also protect code generated by the application itself *i.e.*, self-modifying code. Such code cannot be protected by existing techniques.

The remainder of this chapter is organized as follows. Section 5.1 introduces the concept of software checksumming, and illustrates two techniques: a previously-established techniques called *guards* [2], and a new technique called *knots*. Section 5.2 describes the use of code polymorphism to increase the robustness of such checksumming techniques. The techniques described in this chapter have been implemented in a prototype and Section 5.3 presents the results of a security and performance evaluation of the techniques, using the prototype. The evaluation aids in understanding the trade-off involved between performance and strength of program protection. Finally, we discuss impact of these techniques on software protection in Section 5.4. We summarize this chapter in Section 5.5.

5.1 Checksumming Code

The problem of unauthorized code tamper can be thwarted by the use of checksumming [2, 3, 1]. During the software creation process, hash values are calculated over several code ranges of the application. These values are stored at random locations in the binary file. Next, a sequence of instructions is generated corresponding to each range. Prior to the execution of a particular code

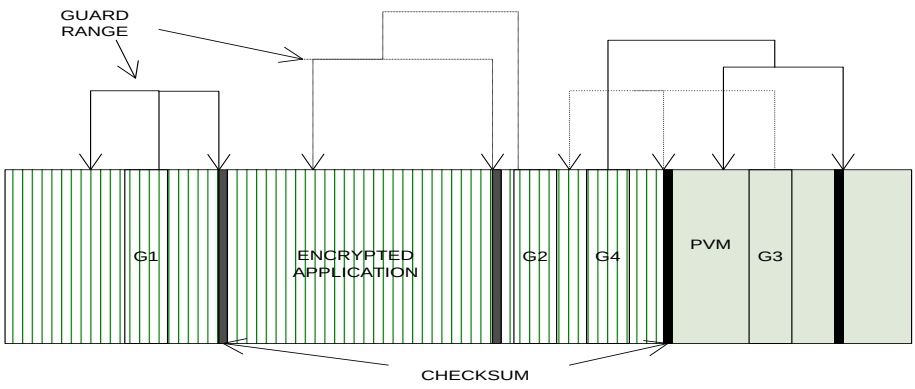


Figure 5.1: On-disk layout of PVM-protected binary containing guards($G1$, $G2$, $G3$, and $G4$).

range, the corresponding sequence checks that the hash value is consistent with the stored value. Any mismatch indicates that the code has been tampered with, and an appropriate response mechanism can be triggered. Such sequences of instructions that check integrity are referred to as checkers.

In PVM-protected applications, checking code is generated in two different ways, by the traditional tool chain (called *static* code), and by the PVM at run time. *Guards* protect static code, whereas *knots* protect any code generated by the PVM.

5.1.1 Guards

Guards are self-introspective code sequences that protect statically-generated code. This concept was initially proposed by Chang and Atallah [2]. The technique works as follows. The application is compiled and linked with associated libraries as per normal software development practices. Next, hash values are calculated over different sections of the code. Then guards are created and placed at different locations in the code. At run time, the guards are executed and check that the hash values are consistent. Any mismatch triggers an appropriate response. Guards are inserted into the application, as well as the PVM.

In our approach, guards protect the encrypted application instructions. Figure 5.1 illustrates the layout of a protected binary, with guards ($G1$, $G2$, $G3$, and $G4$) and associated checksums inserted in both the encrypted application and the VM. For example, the guard $G3$, located in the VM, protects encrypted application code (shown in hatching in the figure), whereas the encrypted

guard $G4$, located in the application, protects the VM. Initially, guard code templates are inserted probabilistically throughout the application and the VM. The checksums are only calculated after the entire application has been encrypted. This technique offers stronger protection than guarding plaintext code in the case where the adversary is able to locate and adjust the checksums to reflect any malicious modifications. The adversary would now have to encrypt the modifications as well and update the checksums accordingly. Although the guards located in the VM are unencrypted, the cyclic nature of guard protection ensures that such guards are safeguarded from tampering.

5.1.2 Knots

A number of SDT systems that use binary translation, translate code for the host machine at run time [52, 82, 83]. Statically implemented guards cannot protect such code. A dynamic scheme is required to safeguard dynamically-generated code.

We propose a novel scheme of program protection, in which the PVM generates *knots* during *translation* [84]. 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 goal 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 a randomly chosen range of memory addresses using a pre-determined hash function, and stores the hash value. It then creates a sequence of instructions that perform the same operation over the same range of memory and places the instruction sequence in the software cache. The PVM then continues to translate application instruction blocks normally. When control is transferred to the software cache, the knot executes and ensures no modification of the translated block has occurred.

Knots provide an added protection feature, in that they can also protect code generated by the application as well (*self-modifying code*). If the PVM-protected application attempts to execute dynamically-generated code at run time, the PVM clears its software cache, and translates that code sequences to its software cache. At this point, it can also insert knots that check the integrity of

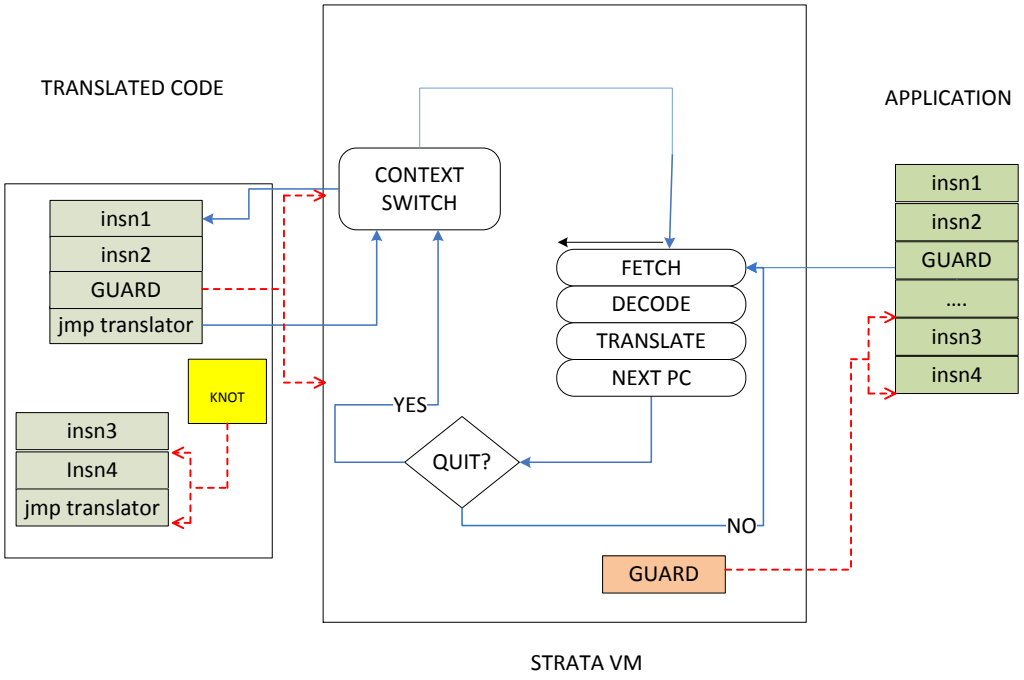


Figure 5.2: Flowchart depicting the run-time of a Strata-VM protected application. The code is protected by *guards* and *knots*.

the translated code. To the best of our knowledge, no current scheme affords such protection to self-modifying code.

Figure 5.2 depicts the flow of an application that is protected by guards and knots. Guards that are placed in the application do not execute *in-situ*. Instead, they are translated and placed in the software cache. Taken together, the guards and the knots protect the main components of the virtualized application: the original application, the PVM, and the software cache.

5.2 Instantiation Polymorphism

Traditional code-integrity checkers have a very structured construction. Such checkers usually consist of a sequence of instructions grouped together in memory, consisting of initialization (which usually loads the start address and size of the region to be checked), a loop (to iterate over the address range), a test (to check whether the checksum matches the statically-calculated value), and a response block

(to decide what to do based on the outcome of the check). Often, applications have several checker instances located in them, that are constructed based on this basic design, and use the same set of instructions. For example, in their investigation of Skype, Biondi, *et al.* discovered that the binary had over 300 guards protecting the code base, all of them designed using few templates [19]. As a result, a majority of them were identical to each other.

This compactness of locality, regularity of structure, and instruction reuse facilitate the use of automated attacks against checksumming systems. Once the structure of a few guards has been determined, the adversary can create an automated method to parse the binary, and locate each checker. These automated techniques can be easily crafted using regular expressions (RE). These REs can be created to locate code constructs that are likely to be part of the checker. In the attack on Skype, the authors created REs for the initialization part of the checker, and were successfully able to identify all the instances. Once these instances have been identified, the adversary can replace the verifier code with harmless instructions (*e.g.*, `no-ops`). After this change, the adversary is free to modify the application at will. Biondi *et al.* successfully disabled protections in Skype using this technique [19].

In general, automated attacks are an issue in most software security systems. As we mentioned in Section 1.3, one of the goals of security techniques involves increasing the effort required by the adversary to obtain the assets from the application. Since automated attacks are easier to craft than customized attacks, the vulnerability of any security scheme to automated attacks significantly reduces its effectiveness. To combat such attacks, many systems employ *software diversity*, which consists of customizing every instance of a particular security scheme [85, 86] . The basic premise behind software diversity is that each security implementation is different from another instance, such that an automatic attack has a reduced chance of success. This diversity of implementation is typically driven by randomization. Diversity has been demonstrably successful against several types of automated exploits [87, 88, 89].

We propose using diversity of structure to thwart automated attacks on checkers as well. In our case, diversity is provided by the use of polymorphic code in the construction of checkers, as well as

```
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 checker. 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 checker. The checksum is calculated using the `xor` operation.

Figure 5.3: Two examples of knots, created using a random selection of instructions.

the location of the checker components across application invocation. Whenever a checker needs to be created, the constituent instructions are chosen randomly. The basic premise is to reduce the similarity between any two checkers. Used effectively, polymorphism can prove to be a powerful deterrent against simple regular expression-based attacks [90, 85].

We utilize polymorphism as follows: A standard checker still consists of the four main components: a preamble, a loop, a verifier, and tamper response. To expand the locality, these four components are not grouped together, but are distributed in different regions in the binary. Furthermore, a database of instructions is used to construct these components. During checker creation, random instructions are chosen from this database to form its structure. Figure 5.3 shows code for two such checkers created using a random selection of instructions. To further increase the level of obfuscation, the instruction database is placed with other program data. This scheme is similar in concept to that employed by polymorphic viruses [61]. Typically, polymorphic viruses consist of a malicious payload that is encrypted, to prevent static analysis. These viruses also possess a mutation engine that generates a new decryptor each time a virus infects a new program. In many cases, even the decryption routine itself is randomized in each new copy [91]. As a result, not only is the virus payload randomized, but the virus decryption routine also varies from infection to infection. With

no fixed code to scan, the virus scanner is thwarted in its search for specific virus signatures [92]. In our system, the different permutations allow for several thousands of different checker instances. Our prototype supports the expansion of checkers generation by a few orders of magnitude, enabling the defender to create millions of checkers. Also, in case of tamper, the checkers do not immediately report an attack. Instead, whenever an attack is detected, control transfers to a different location and the tamper response is delayed till a later time. This delay has been demonstrated to be effective in thwarting the identification of checkers [93].

To summarize, previous checker systems have been vulnerable to attack primarily due to their compactness, and uniform structure. In this section, we have described our design, which can be used to generate checkers with little or no predictability in construction and layout. Thus, regular expression attack will no longer be as effective at dismantling checkers. We present the evaluation of our techniques in the next section.

5.3 Evaluation

Using a combination of binary rewriting and Strata [51], a prototype has been implemented that uses previously-described techniques. The proof-of-concept implementation targets the Intel x86 platform, but the concepts described in this work are platform-independent. First, the application and Strata are compiled using a traditional compiler. Next, a link-time optimizer, called *Diablo*, inserts guards into the package. At run time, the application runs under control of Strata, and both components are protected by static guards. Also at run time, Strata protects its software code cache using knots. We applied these techniques on several benchmarks of the SPEC CPU2000 suite and analyzed the results. All the performance related plots are normalized to native execution (*i.e.*, no protection techniques) and averaged over five runs.

For guards, the number to be inserted into the application and the PVM is determined by a heuristic. The heuristic takes into account the expected connectivity (number of guards protecting a guard on average), and the size of the address range, given by $N = \frac{K*P}{S}$ where K is the approximate coverage desired by the user, P is the program text size, S is the average guard range size set by the

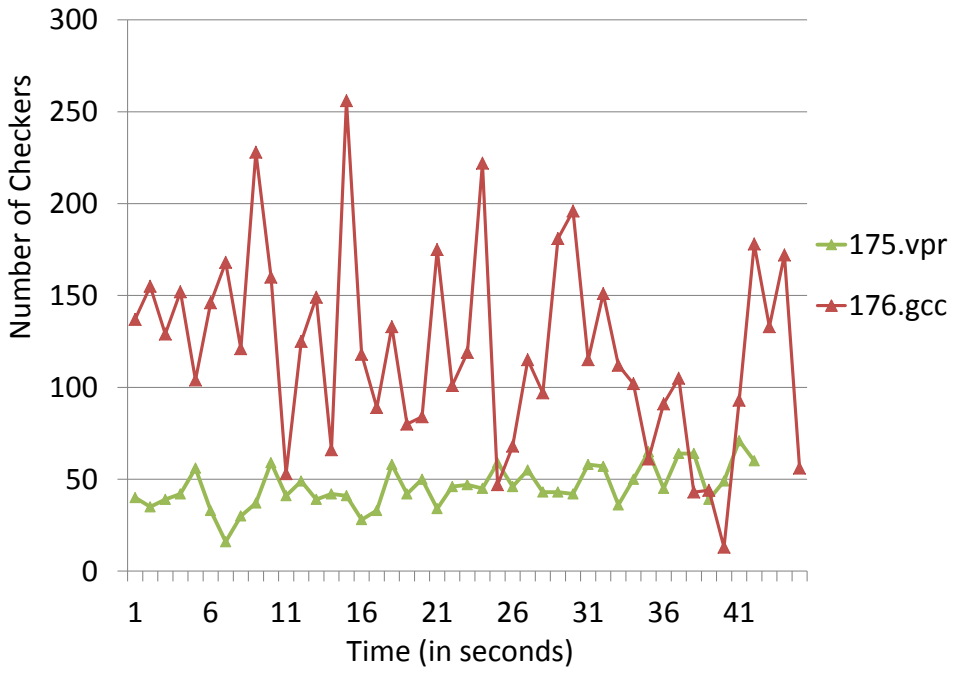


Figure 5.4: Number of checkers (guards and knots) executing per second for `175.vpr` and `176.gcc`. The protected range consists of both the application code and the translated instructions.

user and N is the number of guards. Various experiments were run, with these parameters (K and S) set to different values. From preliminary investigations, we found that a value of 7 for K, and 4096 bytes as the value for S yielded a good trade-off between performance and protections.

For knots, we also chose similar values for connectivity and protection-domain size. Our results are presented below.

5.3.1 Run-time Protection

We first analyze the run-time protection afforded by these techniques. The frequency of checker invocation is one useful metric of run-time protection. Figure 5.4 shows the frequency of combined checker execution per second for `175.vpr` and `176.gcc`. The figure illustrates that both guards and knots execute frequently for both benchmarks. The checkers are triggered by predicates.

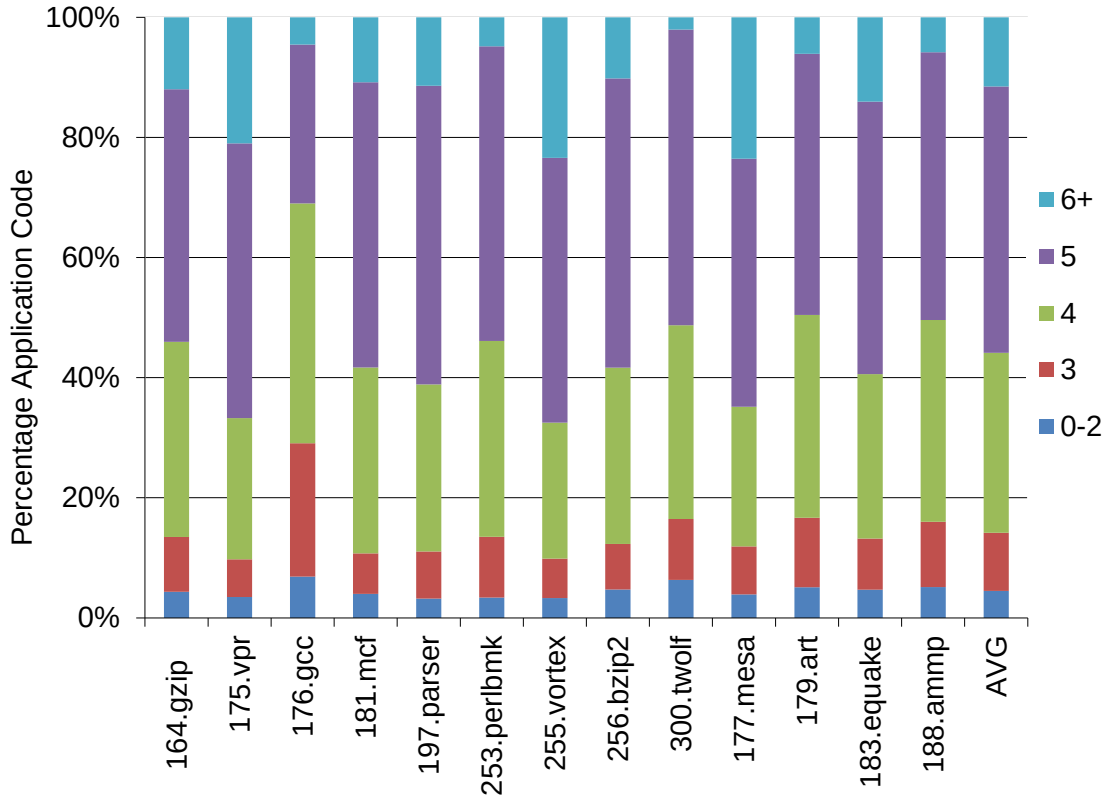


Figure 5.5: Guard connectivity

A high rate of execution of the guards is necessary, but not sufficient condition for robust tamper-detection systems. The range of protection also needs to be evaluated. In an effective system, each byte of the application should be protected by multiple checkers, so that even if an adversary manages to disable some of them, there are other checkers offering protection. Therefore, another important metric involves measuring the connectivity of guards. Connectivity is defined as the number of unique checkers covering a particular region of memory. Figure 5.5 shows that on average about 80% of the application text is covered by three to four checkers (this graph covers the guest application code, as well as the code residing in the software cache). This value indicates that on average, the adversary will have to disable 3-4 checkers to modify a single byte of application code. These checkers, in turn, are protected by 3-4 checkers each. Such a distributed protection scheme makes it difficult for the attacker to target any single point of vulnerability.

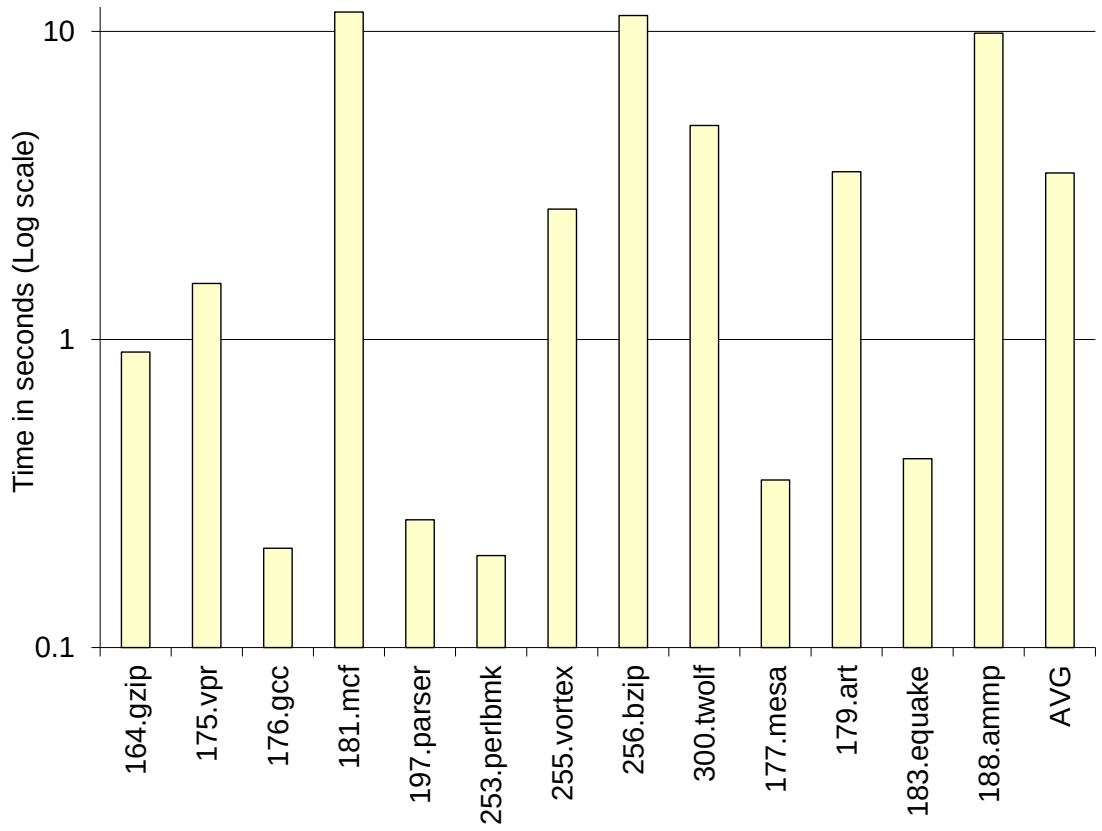


Figure 5.6: Average time delay between checks for each byte of the program. This metric gives an indication as to how long a modification can exist undetected.

The strength of tamper detection can also be evaluated by measuring the average time delay between successive checks on a program byte. This metric indicates how long modifications can exist in the system before detection. Figure 5.6 the measurement results for the C benchmarks in the SPEC 2000 benchmark suite. On average, checks are performed every 3.5 seconds for each byte in the programs, indicating that any modification in the code will be detected within 3.5 seconds on average. We believe that this value is sufficient protection for many classes of applications, such as word processors, web browsers and media players. For example, using these techniques to protect media players will prevent the adversary from viewing digital media for any significant length of time.

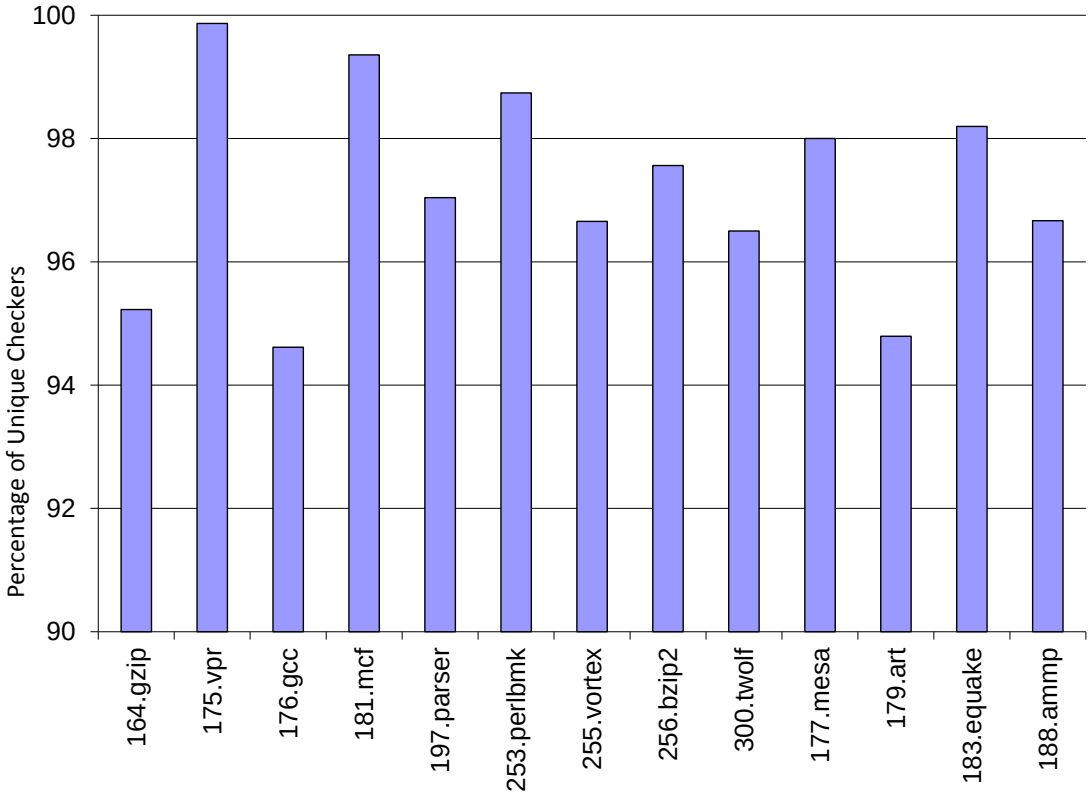


Figure 5.7: Percentage of unique checkers for each benchmark. This statistic indicates that a simple regular expression will fail to locate all of them.

5.3.2 Measuring Diversity of Checker Instances

As described in Section 5.2, we proposed instruction polymorphism to defeat attacks that use regular expressions. Figure 5.7 shows the percentage of unique checkers created for each benchmark. For our proof of concept, we constructed an instruction database for use by both Diablo and Strata. This database supports the creation of tens of thousands of unique checker instances. For all of the benchmarks, our design achieved upwards of 90% unique instances for every benchmark. Since the creation of checkers is driven by randomization, there is a small probability that the structure of two or more checkers are identical. The database can be inserted with more template instructions to make this probability even lower, and achieve greater occurrence of checker uniqueness. This graph demonstrates that there is a significant amount of diversity in the structure of the checkers. Thus, if the adversary is able to correctly identify one checker using a regular expression string, there is a low

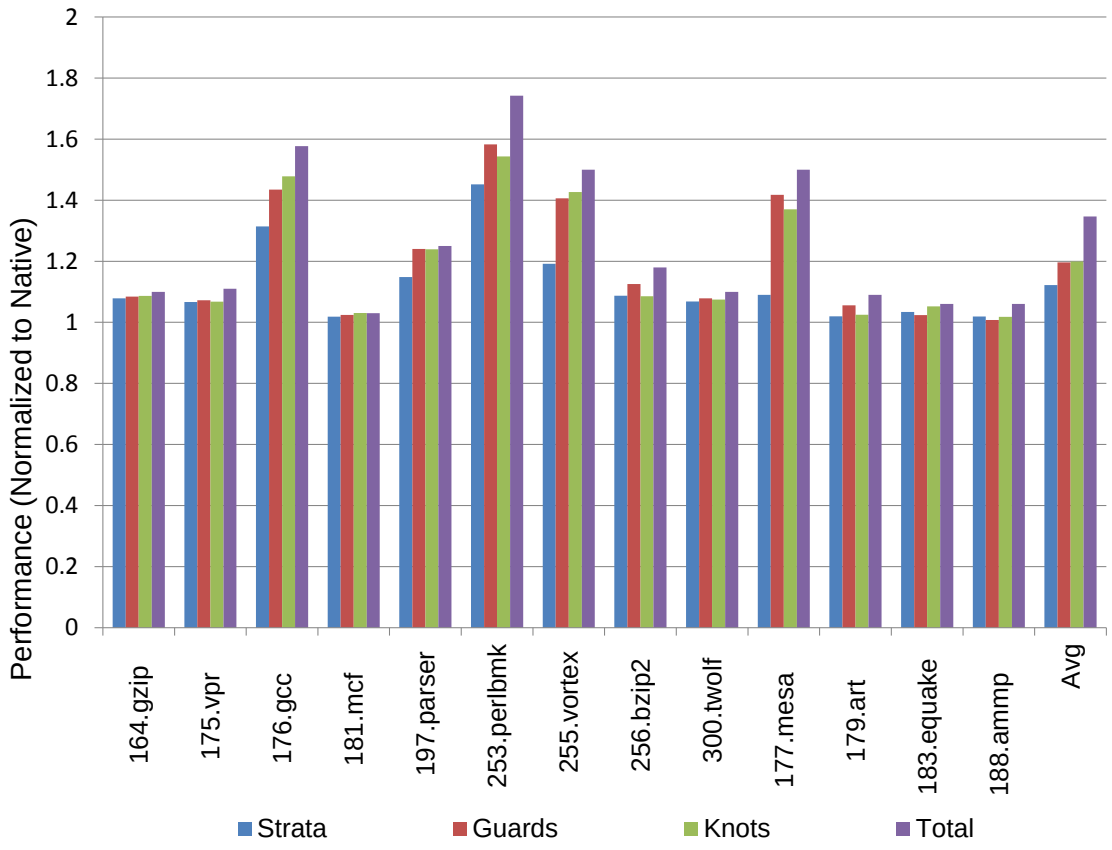


Figure 5.8: Performance overhead for the protection features normalized to native application run.

probability that another checker will be identified by that search string. Another point to note is that the checker structure and location changes with each invocation of the application, providing an ever-changing attack surface for the adversary. Based on the outcomes derived by previous work on software diversity, [11, 85, 90], we conclude that instantiation polymorphism is effective at defeating automated attacks based on regular expressions.

5.3.3 Performance

Figure 5.8 displays the performance overhead of Strata and the protection features normalized to native (*i.e.*, the baseline is the application running natively on the platform). Strata itself adds around 17% overhead to the run time [43]. Previous work has investigated techniques to reduce this overhead [53]. Research in this area is still ongoing.

The checkers add an additional overhead of around 15%. Both guards and knots add an overhead of around 7% each, which is logical considering that the execution rate of both is similar. The predicates that trigger checker execution are based on the exponential back-off method [94]. Initially the checker executes immediately when the predicate is triggered. Every time the checker is executed, the predicate is reconfigured such that it will have to be triggered twice more than the previous predicate value before the checker will execute, subject to a threshold value. This scheme multiplicatively reduces the rate of checker execution (and consequently, performance overhead) to gradually find an acceptable rate.

5.4 Discussion

This section discusses some of the protective properties of the checkers, as well as their robustness against established attacks.

5.4.1 Protection of Generated Code

The primary goal of the knots is protecting the translated code from tamper. To our knowledge, our scheme is the first attempt at tamper-proofing the software code cache. Static techniques can only protect the guest application code and the PVM. Previously, the adversary could make modifications to the code cache without fear of detection. In the presence of knots, such modifications will be detected. As we demonstrated, knots and guards complement each other, and should be used in conjunction to provide robust protection.

5.4.2 Software Diversity

Several researchers have advocated the use of diversity to make software applications less vulnerable to automated attacks [95, 96, 97]. The argument is that if every instance of a protection technique is different, it is more difficult to reuse an attack that was successful against one instance. In keeping with this viewpoint, we have created a system for checkers (guards and knots), that is based on diversity. First, we use polymorphic code to create the actual instances. In our prototype, more

than 90% of the instances possess unique structure. Secondly, the components of the checkers are not located together but distributed in memory. Each time the application is invoked, the structure and location of the checkers are altered, providing a constantly-changing execution environment for the application. Thus, the attack target for the adversary is in a state of flux across application invocations. In Chapter 6, we will introduce techniques to change the attack surface of the application at run time.

5.4.3 Circular Protection

All the techniques mutually reinforce each other to provide a strong tamper-resistant run-time environment. For example, guards check the encrypted binary as well as the PVM code. For an attack to be successful, the adversary will have to find and update all the guards (since each guard is protected by a network of multiple guards). Similarly, the code cache is protected by knots. Also, the guards in the application are protected via encryption. This code is only decrypted on demand, so the guards are not in plaintext simultaneously. Taken together, every component of the software package is being protected against tamper.

5.4.4 Effectiveness Against OS and VM attacks

Modified OSes have already been used to mount successful attacks against software checksumming systems. Such systems work on the assumption that the underlying hardware uses a von Neumann architecture (data reads and instruction fetches go to the same memory structure). Wurster, *et al.* demonstrated a skillful software-only attack on guards: separate data and instruction memory [20]. 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. In our system, the application code is encrypted on disk, and decrypted on an on-demand basis. To create a tampered copy, the adversary will have to obtain the decrypted instructions from the code cache. In the next chapter, we will introduce techniques to periodically delete code from the software cache. Under this obfuscation scheme,

the adversary will only be able to create a copy from the cache, if he is able to gather all the cache snapshots. Also, PVMs like Strata use self-modifying code, which has been shown to defeat split-memory attacks [98]. Any attempt to tamper with the underlying memory system would render the VM and consequently, the application, unusable.

5.5 Summary

Checksumming is a well-established techniques to thwart tamper attacks on software. Composing applications with PVMs adds a powerful, new dimension to checksumming. In this chapter, we discussed some of the novel features of using checksumming in a PVM-protected application. PVMs add dynamism to the guards, translating them to a new location before execution. This relocation makes it hard for the adversary to identify their position. Also, knots created by the PVM safeguard the previously-unprotected software code cache. Taken together, these techniques provide a comprehensive protection strategy against unauthorized tamper.

Chapter 6

Temporal Polymorphism

The previous chapter described the use of instantiation polymorphism to change the shape of guards and knots across application runs. Such a technique can provide protection against iterative attacks, where the adversary might try to run the application multiple times to gain useful information. To improve the tamper resistance of the program during execution, this chapter introduces the technique of temporal polymorphism. Temporal polymorphism presents the attacker an attack surface that changes at run time.

Temporal polymorphism is achieved by periodically flushing the code cache of the PVM, and continuing translation of the application [34]. As we described in Section 2.3, the PVM caches the instructions for optimization purposes. Compacting all the critical instructions (*i.e.*, the application's instructions) in a particular location (the code cache) creates a potential vulnerability in the system as the adversary can focus their analysis on this region. To provide a fluctuating attack surface, the instructions in the cache are deleted periodically, and the PVM continues to translate and cache the application's instructions.

This periodic flushing provides the foundations for the PVM to create a moving attack surface for the adversary. For example, after flushing, the PVM continues to translate and cache the application as before, but the location of the software cache can be changed. Therefore, each flush results in the the relocation of the attack target (the translated instructions). Also, the application's semantics

can be expressed using different opcodes after every flush, creating a diverse attack surface (*e.g.*, `add eax, 1` can be expressed as `sub eax, -1`, `mul eax, 2` can be expressed as `shl eax, 1`, *etc.*). Employing this feature enables the same original application block to appear as different blocks in the software cache. These three components (cache flushing, code polymorphism, and location randomization) together constitute temporal polymorphism. The major contributions of this chapter are:

- Design of a continuously shifting application attack surface. Such a scheme provides increased ability to protect against run-time attacks, *e.g.*, critical sections of the application are not executed *in-situ*, but from randomized locations in memory as the program executes. This dynamism is missing in current techniques.

- Formulation of decryption schemes that have a finer level of granularity than current techniques [19]. By combining cache flushing with on-demand decryption, the amount of code that is available in plaintext form can be reduced significantly.

- Development of stealthy schemes to trigger flushing. Using an external signal to trigger cache flushing creates a potential point of attack, because the adversary has full control over the external environment 1.3. Our scheme uses internal application properties to trigger flushing.

- Evaluation of the prototype implementing the ideas discussed in the chapter. We demonstrate the effectiveness of temporal polymorphism against published attack methodologies. Our analysis shows that such attacks fail to uncover useful information in the presence of temporal polymorphism. This obfuscation can be implemented with tolerable performance overheads.

This chapter is organized as follows: in Section 6.1, we describe temporal polymorphism in greater detail. Section 6.2, we present the results of the evaluation of temporal polymorphism. In Section 6.4, we discuss the effect of temporal polymorphism on previously-published attacks, and demonstrate that this scheme is effective at thwarting such attacks. We summarize our findings in Section 6.5.

6.1 Incorporating Temporal Polymorphism

As described in the previous chapters, the protected package consists of the encrypted application and the PVM. At run time, the PVM performs on-demand decryption of the application code, and caches the translated code in a software-managed memory buffer to amortize performance overhead [51]. Over a period of time the application code will materialize in the cache. Flushing the code cache at periodic intervals prevents the adversary from obtaining a sizable portion of the plaintext code. Flushing splits the run-time information into multiple pieces, forcing the adversary to splice them back together to fully analyze the application.

Flushing also creates an opportunity for the PVM to retranslate the code using a different set of opcodes. Such a scheme creates a constantly changing execution profile of the application. It regularly shifts the attack surface of the application as the protection mechanisms are relocated after each flush. This entropy is furthered increased by adding a random number of dead-code instructions to each basic block in the software cache. Randomized blocks combined with flushing ensure that code blocks regularly execute from different addresses in the cache. Therefore, periodic flushing, retranslation and dead-code instructions act together to impart temporal polymorphism.

In the next section, we discuss the factors affecting the trigger mechanism for the cache flush.

6.1.1 Triggering Cache Flush

Balancing the periodicity of code cache flushing plays an important role in determining the effectiveness of temporal polymorphism. At one extreme, a flush can occur after each instruction is executed. At any point in time, the adversary will only be able to disassemble one instruction, leaking the least amount of information. Unfortunately, such a scheme will lead to an inordinate amount of time being spent in translating instructions and not enough on actual application execution.

One of the major goals of this work has been to find stealthy schemes which flush the code at a fairly uniform rate, such that a high rate of polymorphism is achieved, yet the performance does not degrade significantly. The first scheme considered involved flushing based on a periodic signal generated by the operating system. On receipt of this signal, the system saves the current context

and deletes the stored instructions. Execution continues at the next PC scheduled for execution. However, the threat model includes the case where the application is run on under a malicious OS. The adversary could modify the OS kernel such that the signal is not delivered to the application at all, thus disabling the flushing completely.

Consequently, another technique was required for triggering the cache flushing. Ideally, this scheme should depend on an inherent property of the application itself, making it difficult to alter the system via external manipulation. The flushing should be triggered stealthily, without alerting the adversary as to when the code cache is cleared of the stored instructions. One such scheme involves counting a particular type of instruction at run time and triggering the flush when a threshold has been crossed. The instruction should be numerous enough and well spread out throughout the application code such that the flushing behavior is approximately periodic. As an example, we chose indirect jumps (including function returns) as our candidate instruction. The application is run in training mode and the average number of indirect jumps executed per second is calculated. Flushing occurs based on some function of this average.

6.1.2 Randomizing Code Locations in the Cache

Flushing enables code locations to change during execution. This code shifting hampers the adversary from launching iterative attacks on the application. As such, a small number of random instructions (between 2-8) are appended at the end of each basic block in the software cache. This randomization ensures that there is a high probability that relative distance between any two basic blocks is different across software cache flushes. As a consequence, code (including protective code, such as guards and knots) will execute from different locations during, and across runs.

We have built a prototype implementing these ideas. The next section presents an analysis of this prototype.

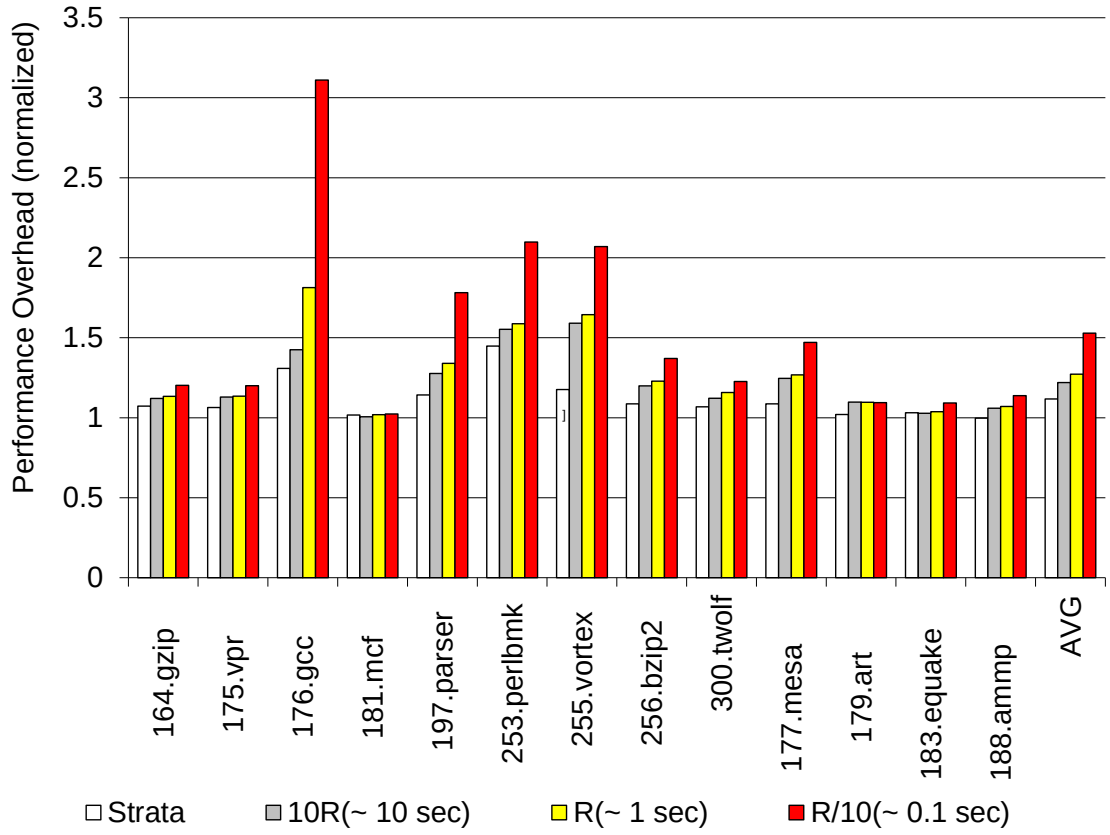


Figure 6.1: Performance overhead due on flushing based on indirect branch count

6.2 Evaluation of Temporal Polymorphism

In this section, we analyze the effects of indirect-branch-based flushing on performance and program security. Initially, the application is profiled, using SPEC's training input, to obtain the number of indirect branches executed per second (designated as **R** in the following discussion). Figure 6.1 displays the plot showing the performance overhead relative to unprotected execution. The performance overhead for flushing after every 10**R**, **R**, and **R**/10 branches was 25%, 30% and 55% respectively, compared to native execution. The overhead for flushing every **R**/10 branches is quite high but the other two options show promising results. We found the rate of flushing becomes somewhat inconsistent, if the threshold is set too low. In particular, flushing performed after every **R/10** branches differs when compared with flushing performed every 0.1 sec. The reason for this behavior is that indirect branches are not temporally uniform but are clumped together in time.

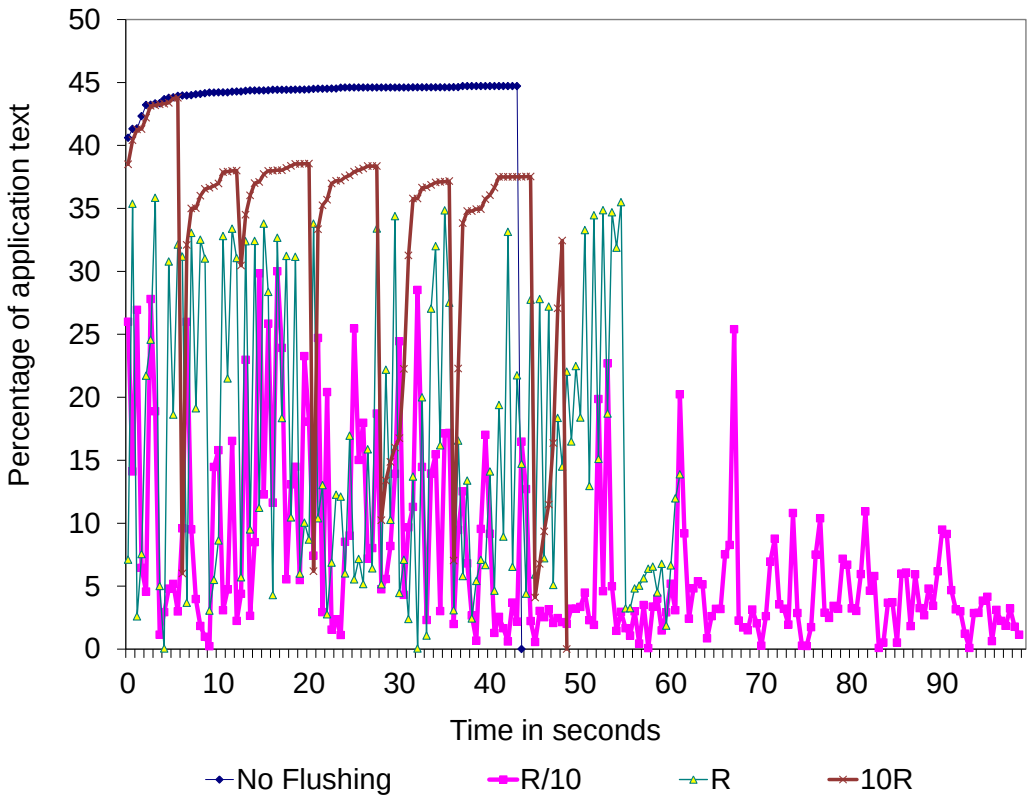


Figure 6.2: Rate of plaintext application code due to flushing on indirect branch count for `176.gcc`

In addition to temporal polymorphism, periodic flushing significantly reduces information leakage, when compared to current decryption schemes [19, 12]. This reduction is evaluated by measuring the rate at which application text appears in the code cache. Figure 6.2, which plots the rate at which the application text appears in the code cache for `176.gcc`. The plot shows that even without flushing, only some of the program text is present in the cache at a time. Flushing every 10 **R** branches per second shows some benefit, as much of the code is used in startup or tear down, after which it can be flushed out of the cache. Flushing every **R** branches is much more effective at keeping a significant portion of the application out of the cache. Flushing **R**/10 branches per second does the best job, as no more than 30% of the code resides in the cache at any point in time, while only adding an overhead of around 20%. This value is much better than bulk decryption [19], which has 100% application code decryption at startup. Decryption on a per-function basis typically has high

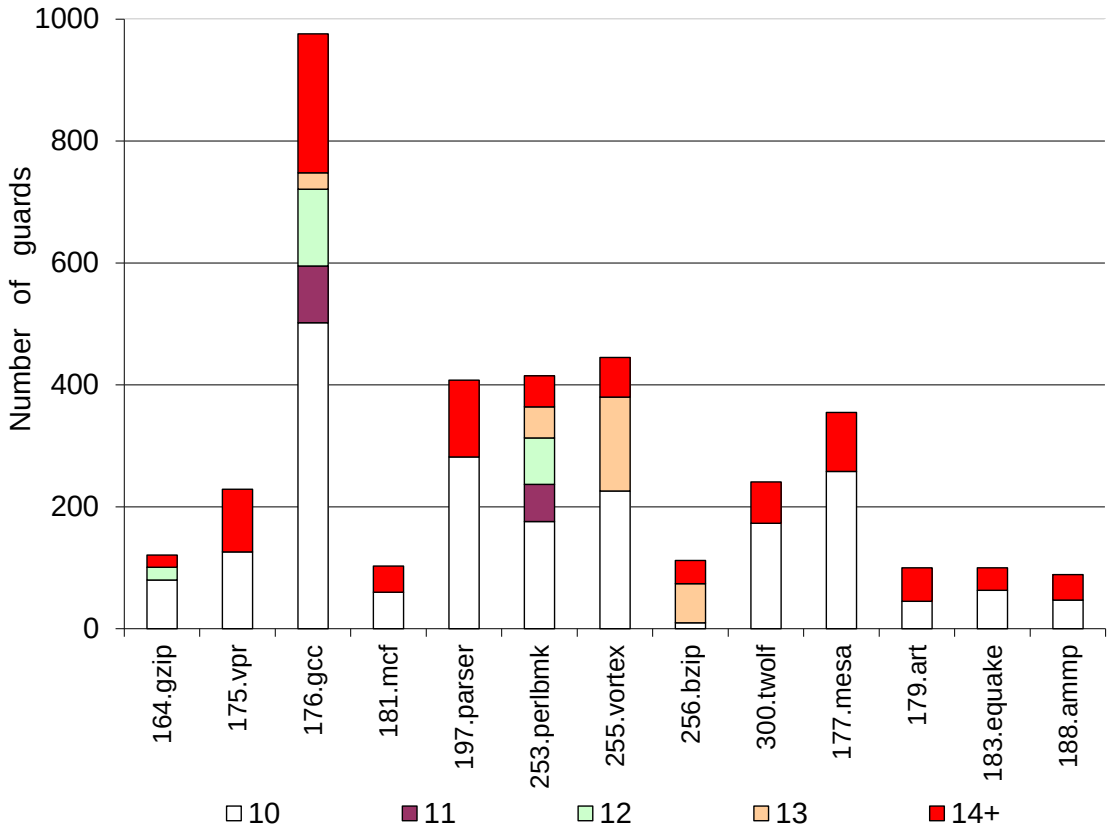


Figure 6.3: Distribution of code cache addresses for guards across 10 independent runs. The cache is flushed every second.

overhead (up to 8X), unless functions are encrypted selectively based on profiled information [12].

Flushing based on indirect branch count relies on training input to ascertain the average number of branch instructions executing per second. This average differs frequently when the application is run using reference inputs. There are other program characteristics that can be used to simulate a periodic signal, but the challenge is to find one which is consistently uniform across all application inputs. Developing an algorithm which provides consistent flushing which maintains the program's security without degrading performance is an area of ongoing research.

To illustrate the effectiveness of adding random instructions during translation to randomize code addresses, we analyzed the locations of protective code, in this case, application guards. Figure 6.3 shows the distribution of cache addresses of guards across 10 runs of the same executable. The software cache was flushed once every second so that even within a single run, guards execute from

multiple locations. Each guard had at least 10 distinct locations within the cache.

6.3 Robustness of the Flush-trigger Mechanism

As we described earlier, any external triggering mechanism for the software cache flush can be easily neutralized. The adversary has complete control over the external environment (including OS and the hardware), and can provide false information to the application to disable the trigger mechanism.

In light of this weakness, it is essential to trigger software cache flushing based on properties of the application itself. In our prototype, flushing depends on the dynamic count of indirect branches. The threshold, at which the software cache is flushed, is modeled to achieve a periodic effect *i.e.*, the threshold is chosen in such a way that the software cache is flushed at approximately regular intervals. Any other dynamic program property is equally applicable (*e.g.*, number of calls, number of direct jumps, *etc.*). Obscuring the cause of the flush is the first line of defense against tamper attacks.

An adversary attempting to disable flushing may try to overwrite the instructions that actually do the flushing (*e.g.*, replace them with `no-op`s). The presence of guards makes this attack less likely. As we have demonstrated in Chapter 5, each byte of the application (including guard code) is checked multiple times. Therefore, the network of guards provide robust resistance to tampering with the flushing mechanism.

Instead of modifying code, the adversary might attempt to change the threshold value, which triggers the software cache flush. This threshold is likely to be located in the global data section for the protected package. It is conceivable that an adversary is successful in locating the threshold value, and changing it to a large negative number. This change will effectively disable flushing. To thwart such attacks, copies of the threshold value can be made and distributed throughout the program data section. The sequence of code that updates the counter and checks if the threshold value has been reached, is generated dynamically and placed in the software cache. Multiple copies of this sequence can be generated. In this manner, the point of vulnerability (the threshold value, and the code updating the counter) is distributed, and hard for the adversary to locate and disable. Finally,

the flushing mechanism itself is located in the PVM, safeguarded by software guards. Any attempt to modify the flushing code would be detected by the guards.

Thus, taking these protection schemes together, we believe that it would be hard for the adversary to disable flushing. In the next section, we illustrate the effects of temporal polymorphism on established reverse-engineering attacks.

6.4 A Use Case

To further highlight and demonstrate the effectiveness of temporal polymorphism, we describe a published use case describing reverse-engineering methodologies. We then illustrate the effectiveness of temporal polymorphism against such attacks.

6.4.1 Analyzing the Control Flow Graph

The first step of any attack involves obtaining a basic understanding of the application. The control flow graph (CFG) is an essential data structure for program comprehension and analysis. It is a directed graph where the vertices represent basic blocks, and edges represent potential transfer of control flow from one block to another. A CFG can be dynamic or static. The static CFG is obtained by locating the start address of the application in the binary. Then, all the basic blocks are identified. Finally, all potential paths between the blocks are located. The dynamic CFG, on the other hand, is usually obtained from the trace of actual instructions executed. Depending on application input, the trace of one application invocation might be different from another. Consequently, the dynamic CFG can also be different from one invocation to another. In some cases, the static CFG can contain the superset of all possible dynamic CFGs in the application. In other cases, the dynamic CFG is partitioned into static control flow and data flow (*e.g.*, applications that use self-modifying code).

Obtaining the CFG from the binary in the presence of static protections can be computationally very expensive [61, 9]. Consequently, adversaries have increasingly focused on run-time techniques to obtain the CFG. Although CFGs obtained dynamically can be incomplete, they still provide

the adversary with useful information about the application. PVMs provide protection by making dynamic CFG construction and analysis highly resource- and time-intensive tasks.

To demonstrate this point, we studied dynamic reverse engineering schemes that have been shown to be successful in attacking software [99, 100] 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 adversary 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 [99]. Similarly, Udupa *et al.* used edge profiling to identify and remove unnecessary edges from the static CFG [100].

6.4.2 Thwarting Dynamic Analysis

To show the effectiveness of temporal polymorphism against the attack methodologies described in Section 6.4.1, we explored the applicability of such profiling techniques on two different run-time scenarios.

- The application executing without any protections (No protection).
- The application executing in the presence of a protective PVM (Protected). This PVM applies temporal and instantiation polymorphism to the application code.

To facilitate collection of application code blocks that have been translated, the application was run under an instrumentation framework in both 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 adversary.

We began by comparing instruction trace generation and block analysis in both cases. On comparison, we observed that packaging a protective PVM with the application makes analysis of

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

Table 6.1: 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.

the dynamic trace and CFG generation much harder. First, the periodic flushing and retranslation of application code increased the number of individual basic blocks substantially. 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, temporal polymorphism makes the code blocks appear different [101]. To summarize, temporal polymorphism increases the number of dynamic blocks by an order of magnitude. To obtain the dynamic CFG, the adversary would have to perform analysis on a larger instruction trace. Furthermore, to obtain a CFG closer to the actual application CFG, the adversary would have to perform analysis to reduce the number of blocks (*e.g.*, by identifying different blocks that were translated from the same original application block).

Temporal polymorphism alters a number of dynamic characteristics of the application, such as block execution frequency, and in and out degrees of the CFG nodes. Figure 6.4 shows the execution frequency of the dynamic blocks in both the 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 adversary 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

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

Table 6.2: 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.

function [99]. 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 adversary could initiate analysis. The adversary will have to increase the search space to locate the hot paths for the application.

The PVM also provides misleading information to the attacker. In the two 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 6.1 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 similar reordering when rankings were based on the in degree of the code blocks. Table 6.2 shows the list of the ten most frequently executed blocks when the application is run under the protection of

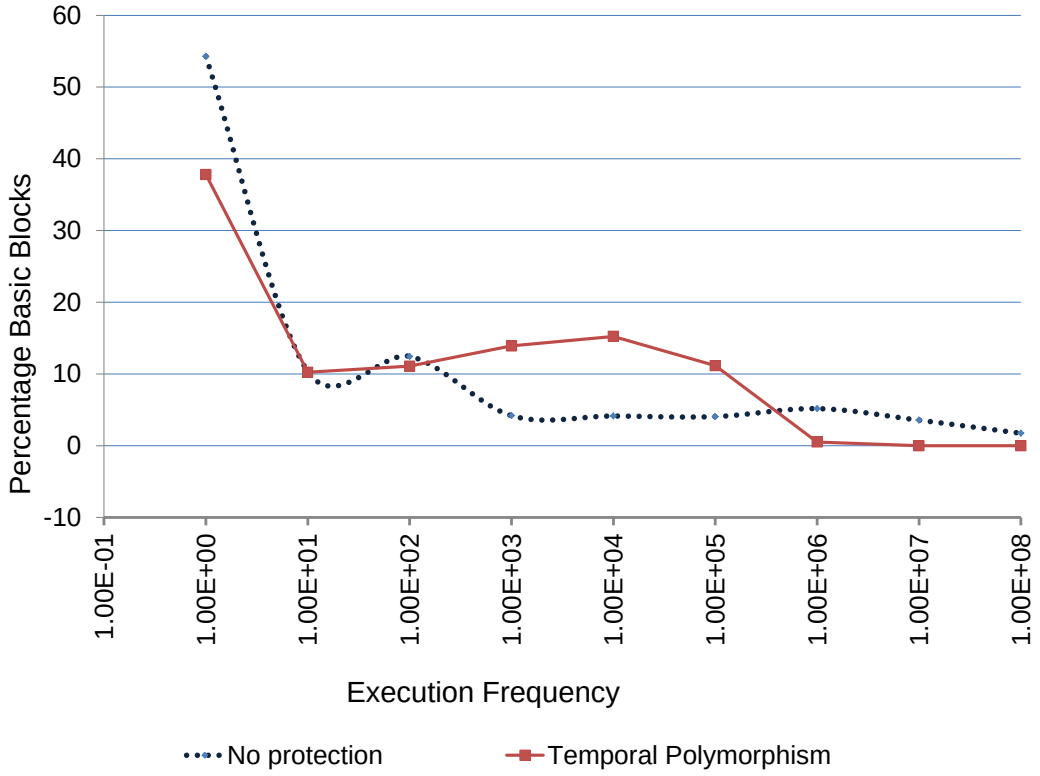


Figure 6.4: Execution frequencies for the application blocks under the two run-time scenarios (No protection, and Protected). The periodic flushing and retranslation of the application's code blocks by the protective VM drastically changes the execution frequency characteristics. Under the control of 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 adversary to expand their search space.

the PVM, along with their corresponding ranks in the unprotected run.

We observe that the blocks with a high rank in the unprotected run, are displaced in rankings by other blocks when the application is run under temporal polymorphism. Most instrumentation tools identify blocks based on their address. Since, under temporal polymorphism, application blocks are constantly flushed and rewritten at a different location, the instrumentation tool is not able to identify different blocks as originating from the same application block. The use of code polymorphism adds an extra layer of obfuscation. A retranslated block is likely to be composed of different opcodes, than its previous incarnation. Thus, this study demonstrates that frequency analysis is not useful to the attacker in the presence of temporal polymorphism. 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.

As should be obvious, 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 exists research which aims to reverse engineer PVM-protected applications, by identifying code belonging to the VM in the execution trace [102, 57]. 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.

6.5 Summary

In this chapter, we presented temporal polymorphism, a novel methodology for obfuscating application code. By periodically flushing the cached code, and retranslating it using different opcodes at different locations, this technique presents a changing attack surface to the adversary. Reverse-engineering methodologies that depend on iterative runs or profiled information, failed to obtain useful data in

the presence of temporal polymorphism. To impart trigger this scheme, we presented a technique based on the application itself (*i.e.*, the rate of indirect branch execution). Our evaluation shows that temporal polymorphism is a potent obfuscation scheme, reducing the leakage of plaintext information, as well as constantly relocating code. The analysis of the use case proved that known reverse-engineering attacks will fail on applications protected by this mechanism.

Chapter 7

Point-ISA: Binding the Application to the PVM

Chapters 5 and 6 illustrated the effectiveness of PVMs in improving the tamper detection and code obfuscation of applications. In particular, instantiation and temporal polymorphism can make it hard for the adversary to successfully reverse engineer the application. Consequently, any attack on the application has a higher chance of success if the protective PVM can be disabled.

This chapter investigates an attack methodology that targets applications protected using process-level virtualization. The basic premise behind this attack strategy is that the PVM is not tightly bound to the application, and can be replaced by the adversary. Using a replacement attack, the attacker can effectively remove any dynamic protection technique provided by process-level virtualization and proceed to analyze the application. After presenting a thorough investigation of this attack methodology, we present a novel approach to counter replacement attacks. The basic premise behind this solution is to create a unique relationship between the application and the protective PVM instance, based on the semantics of certain instructions. This relationship facilitates the detection of any attempt to execute the application without mediation by the protective PVM.

Some of the major contributions of this chapter are listed below.

- Design of a novel attack methodology, called replacement attacks, 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 [103]. 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.

- Demonstration that existing protection schemes, such as software checksumming guards fail to adequately protect virtualized applications from replacement attacks. Thus, a novel solution is required to thwart this class of attacks.

- Use of a comprehensive, two-part case study that describes in detail, the replacement attack methodology. The first part of this study describes the creation of a protected, virtualized application, and then how an attacker can 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.

- Design of a novel solution that thwarts replacement attacks. This solution, termed *Point-ISA* seeks to bind the application to its associated protective PVM instance. This novel relationship is achieved by inserting instructions into the application, whose semantics have been modified for this particular context (application and protective PVM instance). Any other interpreter

attempting to execute the application will interpret these instructions according to their original semantics and can be detected.

- Design of a prototype that implements our ideas. Using this prototype, we have performed extensive analysis on various issues related selecting instruction selection and attack response. We have also studied responses techniques in the case when a replacement attack has been detected. Our findings demonstrate that Point-ISA is effective against replacements attacks.

The organization of this chapter is as follows: In Section 7.1, we define the replacement attack methodology and give a formal description using our model. Section 7.2 describes two implementations of this attack. In Section 7.3, we analyze the effects of this attack on applications protected using the PVM-based techniques described in Chapters 5 and 6. Section 7.4 presents some techniques that an adversary can exploit to gain information before launching such attacks. The second part of this chapter deals with a solution strategy against replacement attacks. Section 7.5 describes this strategy, called Point-ISA, and defines it in terms of the formal model. In Section 7.6, we address some of the design decisions related to Point-ISA. Section 7.7 explains the creation of a prototype of this solution scheme, and evaluates its effectiveness. Section 7.8 provides a detail security discussion of this scheme. Finally, our results are summarized in Section 7.9.

7.1 Replacement Attack

The first part of this chapter explores the loose connection between the application and the protective PVM instance. The only requirement on the part of the PVM involves the ability to interpret the semantics of the application's instruction sequence. This weak binding stems from the need to make applications platform-independent, which was one of the major goals behind virtualization. The application is compiled once, and should be runnable on many platforms. This adaptability is facilitated by the virtual machine, which is implemented according to the specifics of the native platform. In this paradigm (exemplified by Java), it is essential that the application and the PVM be unbound.

We demonstrate that such adaptability leads to a serious weakness when PVMs are utilized in program protection. An able adversary can replace the protective PVM instance with a benign VM, and proceed to analyze the application unhindered. We define such attacks as *replacement attacks*, targeting software application packages protected by process-level virtualization. Once the PVM and its associated protection mechanisms are replaced, an adversary can apply standard reverse engineering techniques to study the application.

In the next section, we proceed to describe this attack methodology. This description is facilitated by extending the equational model introduced in Chapter 3.

7.1.1 Modeling the Attack

The main motive behind designing the model was to facilitate the description of various defenses and attacks involving PVM-protected applications. In this section, we extend the equations introduced in Section 3.1, to represent the replacement attack methodology.

We consider a generic software application that has been compiled to run on the Intel x86 platform. Equation 7.1 describes such an application.

$$P_{APP} =< I_{APP}^{x86}, IN_{APP}, OUT_{APP}, ASSETS_{APP} > \tag{7.1}$$

Recall that I_{APP}^{x86} refers to its instruction sequence in the x86 ISA. IN_{APP} and OUT_{APP} refer to its input and output set, respectively. Finally, $ASSETS_{APP}$ refers to its assets. IN_{APP} contains a subset of inputs, denoted by L_P that confirms that validity of application execution (*i.e.*, all the established rules for application execution have been met).

The software defender applies several transformations that are designed to protect the application. These transformations are represented by the operator, TR, which can be applied to an application. Equation 7.2 describes the protected application.

$$P_{TR(APP)} =< I_{TR(APP)}^{x86}, IN_{APP}, OUT_{APP}, ASSETS_{APP} > \tag{7.2}$$

$I^{x86}_{TR(APP)}$ refers to the transformed instruction sequence. The rest of the variables have the same meaning as in Equation 7.1. In our research, an example of a transformation could be to encrypt the instruction sequence, which is then decrypted at run time by Strata.

Next, we model the protective PVM (in this case, Strata), which runs the protected application. We assume that the hardware platform is an instance of the Intel x86 architecture. Equation 7.3 describes the Strata PVM.

$$P_{strata} =< I^{x86}_{strata}, IN_{strata}, OUT_{strata}, ASSETS_{strata} > \tag{7.3}$$

I^{x86}_{strata} represents the instruction sequence of Strata in the x86 ISA. The input set for Strata is a subset of $\rho_{APP} \times IN_{APP} \times C_{strata}$, where ρ_{APP} refers to all applications. In this example, we assume that the applications have been compiled for the Intel x86 architecture and subsequently protected. IN_{APP} represents the input set to the applications. Finally, C_{strata} refers to the configurations for Strata, to enable executing the application. Similarly, the output set for Strata, OUT_{strata}, consists of $OUT_{APP} \times O_{strata}$, where OUT_{APP} refers to the outputs generated by the application, and O_{strata} represents the outputs generated by Strata. $ASSETS_{strata}$ refers to the assets of the PVM.

We now focus on the run time. This application is interpreted by Strata, which applies dynamic protections techniques, similar to those described in Chapters 5 and 6. Referring to Equation 3.11, the interpretation by Strata can be modeled as:

$$\phi_{strata}(P_{TR(APP)}, in_{APP}, m^{in}_{strata}) \longrightarrow < out_{APP}, m^{out}_{strata} > \tag{7.4}$$

In Equation 7.4, ϕ_{strata} refers to the interpretation operation under Strata. $P_{TR(APP)}$ refers to the protected application, the operation $TR()$ indicating that the application can also be protected by techniques independent of Strata (*e.g.*, static protections). The input variable, in_{APP} consists of the application inputs. m^{in}_{strata} represents the initial memory state. At the conclusion of interpretation, out_{APP} is generated, and the final memory state is denoted by m^{out}_{strata}.

The Strata PVM is, itself a software application that is interpreted by the x86 platform. As we

described in Section 3.1, this framework can model nested interpretation. Equation 7.5 represents the interpretation of Strata on the Intel x86 platform.

$$\phi_{x86}(P_{strata}, in_{strata}, m_{x86}^{in}) \longrightarrow < out_{strata}, m_{x86}^{out} > \tag{7.5}$$

The x86 interpreter takes as input the Strata software application, P_{strata}, and an input, in_{strata}, while generating output out_{strata}. The memory state is transformed from m_{x86}^{in} to m_{x86}^{out}.

Expanding the variables in Equation 7.5 to include the representation of the protected application, $P_{TR(APP)}$, we obtain Equation 7.6, which illustrates the full nested interpretation.

$$\phi_{x86}(P_{strata}, < P_{TR(APP)}, in_{APP}, c_{strata} >, m_{x86}^{in}) \longrightarrow << out_{APP}, o_{strata} >, m_{x86}^{out} > \tag{7.6}$$

The input to the interpreter is composed of a tuple, comprising the protected software application $TR(P_{APP})$, the input to the original application, in_{APP}, and c_{strata}, the configuration settings for Strata to run the application. For example, if the application is encrypted, c_{strata} can represent the decryption key. The memory is in its initial state, which can be broken down into the memory state for Strata, $(m_{x86}^{in})^{strata}$, and the state for the application, $(m_{x86}^{in})^{APP}$. On successful completion, the output, out_{APP} is generated, and memory reaches its final state, denoted by $m_{x86}^{out})^{strata}$ and $(m_{x86}^{out})^{APP}$.

Next, we describe the replacement attack using our model. To remove PVM-enabled obfuscations, and to execute and analyze the application unhindered, the adversary replaces the protective PVM with a benign instance. We refer to such a PVM instance as the *attack PVM*. This PVM can also interpret any application that has been compiled to run on Strata, but enables the adversary to perform any analysis. The attack PVM can be represented by:

$$P_{attack} =< I_{attack}^{x86}, IN_{strata}, OUT_{strata}, ASSETS_{attack} > \tag{7.7}$$

Comparing Equations 7.3 and 7.7, we observe that the attack VM has the same input set, and

generates the same output set as Strata. I^{x86}_{attack} denotes the instruction sequence for the attack PVM on x86 machines. Finally, $ASSETS_{attack}$ represents the assets of the attack PVM.

We now proceed to model the attack, by modifying Equation 7.6.

$$\phi_{x86}(P_{attack}, < P_{TR(APP)}, in_{APP}, c_{attack} >, m^{in}_{x86}) \longrightarrow << out_{APP}, o_{attack} >, m^{out}_{x86} > \tag{7.8}$$

Equation 7.8 illustrates the replacement attack. The x86 interpreter operates on the attack PVM, P_{attack}. The inputs consist of a 3-tuple comprised of the protected application, $P_{TR(APP)}$, the input to the application, in_{APP}, and a configuration setting for the PVM, c_{attack}. During interpretation, the memory state is transformed from m^{in}_{x86} to m^{out}_{x86}. On conclusion, the interpreter outputs a tuple consisting of out_{APP}, which is the output of the application, and o_{attack}, which is the output specific to the attack PVM. The adversary can configure the attack PVM to generate additional information that could facilitate analysis. An example of o_{attack} could be the run-time trace of the application, which enables dynamic control flow analysis.

Thus, the replacement attack enables the adversary to remove any dynamic protections, and study the application unobfuscated. With the formal model in place, we proceed to describe the details of this attack against a PVM-protected application.

7.1.2 Description of the Attack

The replacement attack 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 $TR(P)$.

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

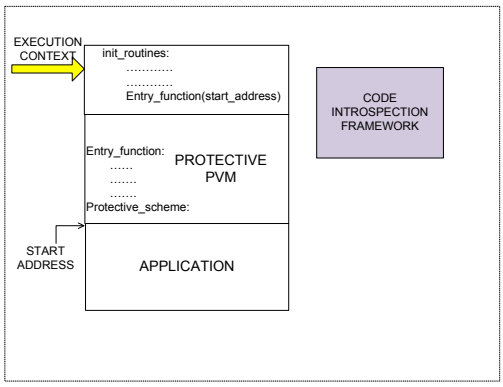
The attack occurs in two stages. In the first stage, the attack PVM has to be extended to decode the protected application, which involves understanding the guest ISA. If the ISA is encrypted, the decryption keys and algorithms must also be obtained and used to further extend the attack PVM by including the decryption algorithm and keys. Details on deciphering the guest application's ISA are given in Section 7.4.2.

Figure 7.1 illustrates the second stage of the attack on $TR(P)$. In Figure 7.1(a), the attacker invokes $TR(P)$, under a *code introspection framework* CIF observing instructions as they execute. Well known examples of CIFs include Pin [82] and QEMU [104]. The attacker modifies the CIF to locate the call to the entry function of the protective PVM.

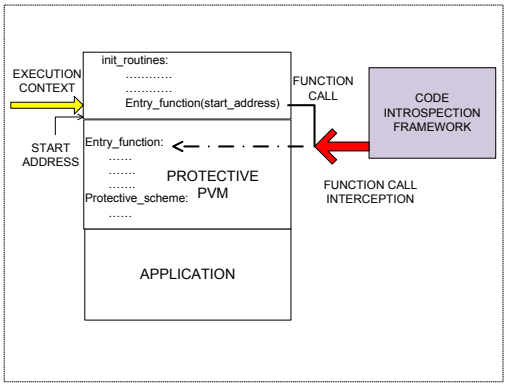
The initialization routine then proceeds to prepare the PVM's internal structures. As the entry function of the protective PVM is invoked, the CIF intercepts this call and extracts the start address, depicted in Figure 7.1(b). Details on identifying the PVM's entry function are given in Section 7.4.1

The CIF then proceeds to load and initialize the attack PVM, shown in Figure 7.1(c). The CIF then invokes this attack PVM with P's start address which has been extracted from the initial call.

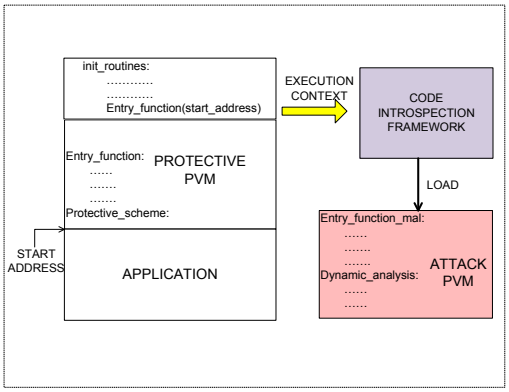
Thus, P now runs under the mediation of the attack PVM (shown in Figure 7.1(d)). The protective PVM is circumvented and fails to provide dynamic protection to P. The attack PVM can be used to perform tasks that helps the attacker understand P (*e.g.*, dump information, identify function locations, trace instructions, *etc.*).



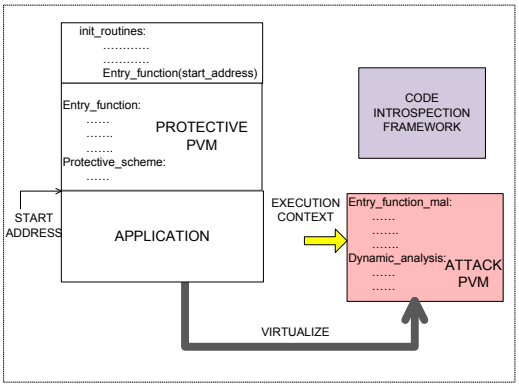
(a) The virtualized application, $TR(P)$, is run under an introspection framework.



(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 7.1: Steps illustrating the attack methodology on virtualized applications.

In Section 7.2, 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.

7.2 Use Cases

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, $TR(P)$. The target application was chosen from the integer benchmarks of the SPEC CPU2000 suite [105]. The

benchmarks were selected as examples of typical applications and they are commonly used to measure run-time 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. All our tests were carried out on the Intel x86 32-bit platform running Linux OS. All the components were initially compiled using *gcc*.

7.2.1 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 [52] that we extended to perform AES decryption). The second implementation uses an architectural simulator, PTLsim [83], 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.

Attack Using a Dynamic Binary Translator

This prototype uses Intel's run-time binary instrumentation framework, Pin [82], to replace Strata with another binary translator, HDTrans. 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

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

Figure 7.2: Listing of x86 assembly code snippet preceding the entry point into Strata.

binary code [106, 107, 108]. 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 [106]. HDTrans was subsequently modified to use AES decryption on the application code blocks prior to translation. Section 7.4.2 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:

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, 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 write the execution trace 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.

Attack Using an Architectural Simulator

The second prototype for the attack uses the PTLSim architectural simulator [83]. 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 7.2.1, and PTLSim is extended to decrypt instructions after fetching them from memory. At load time, PTLSim initializes its internal data structures and reads in $TR(P)$'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 7.2.1) 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.

7.3 Implications of the Attack

In section 6.4.2, we presented a use case to illustrate the obfuscation properties of the PVM. Continuing with the same example, we show that the replacement attack effectively reduces the

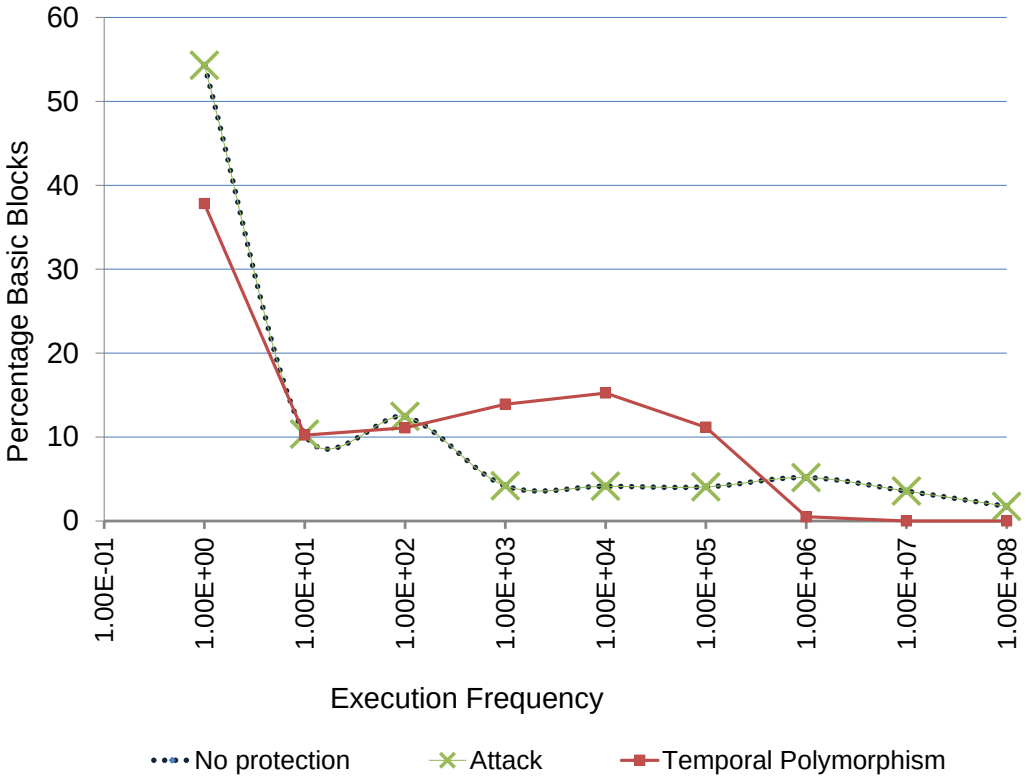


Figure 7.3: Execution frequencies for the application blocks under the three run-time scenarios (No protection, Protected, and Attack). The periodic flushing and retranslation of the application's code blocks by the protective VM drastically changes the execution frequency characteristics. Under the 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.

run-time execution environment of the protected application to that of an unprotected instance. To study the effects of the attacks, we applied the profiling techniques of Section 6.4 to an additional run-time scenario:

- 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).

The dynamic blocks in this third scenario were collected as before, *i.e*, by instrumenting the applications and identifying blocks based on their memory address.

Figure 7.3 is the extension of Figure 6.4 displaying the execution frequency of all three cases. An adversary attempting to analyze the protected application directly, will encounter a much larger

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 7.1: Original application addresses of the top ten 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.

space for exploration. As we discussed in Section 6.4, temporal and instantiation polymorphism lead to a large increase in the dynamic instruction and consequently, basic block counts. Figure 7.3 shows that replacing the protective PVM yields much more successful results. The attack reveals the original execution frequency of the application, which had previously been obfuscated by the PVM. For example, in the case study, 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. Once the protective PVM is replaced, the number of dynamic blocks and edges exposed return to their original values. Launching a successful replacement attack reduces the search space for the adversary by a considerable amount.

Replacing the PVM enables the adversary to obtain correct information about the application run time. Column 1 of Table 7.1 shows the top-ten most frequently executing blocks when the application is run without any protections. Column 2 displays the ranking of these blocks when the application is run under the control of the PVM. Finally, Column 3 reveals the ranking after the successful completion of the replacement attack. The PVM is able to reorder the blocks based on execution frequency. We observed similar reordering when rankings were based on the in-degree of the code blocks. Such a reordering can seriously affect analysis performed by the adversary. Replacing the protective PVM enables the original block rankings to be determined. Table 7.2 shows the list of the

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 7.2: 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.

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. This study shows that replacing the PVM enables the use of frequency analysis to uncover critical information about the application.

The replacement attack methodology is pivotal to the success of reverse engineering PVM-protected applications. Once the protective PVM has bee replaced, the application can be analyzed and its true characteristics studied without any obstruction.

7.4 Attack Discussion

Section 7.2 described two implementations of an attack that seeks to remove the protective PVM from a virtualized application, $TR(P)$. 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 $TR(P)$ failed to prevent the replacement. To successfully orchestrate the attack, some prior information about $TR(P)$ is required. In this section, we discuss some heuristics the attacker can use to determine this information.

7.4.1 Determining PVM Entry Function

To launch a successful attack, the location of the entry function(EP) 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 $TR(P)$ for distinctive instruction sequences that indicate the location of the entry function.

For example, 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 7.2.1 displayed the instruction sequence used by Strata to save state, which contains these two instructions. Dynamo-RIO, HDTrans and Pin also use these instructions to store register and flag values prior to initiating translation. 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 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.

7.4.2 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 [36, 11] or encryption [34]. 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 [106]. 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*) [108]. They successfully applied this technique to identify and extract cryptographic routines from a set of malware files. Similarly, Leder et al. examined data in-flow and out-flow in memory buffers to isolate cryptographic functions [107]. 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 [64]. 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 [19]. Techniques, such as white-box cryptography, were proposed to improve key management in encrypted systems [21]. However, researchers have since developed solutions to extract the key from such systems [67]. 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 [19, 41]. Once the the ISA is known, the attacker can successfully replace the protective PVM.

7.5 Point-ISA: Using Homographic Instructions to Semantically Bind the Application and the PVM

The replacement attack succeeds due to the fact that the application can be virtualized by any PVM that can interpret the ISA of P. This section proposes a solution that thwarts replacement attacks, by semantically binding P with a specific instance of the protective PVM(*i.e.*, P can only be interpreted by a unique PVM instance.

This protection scheme is facilitated by the implementation of a property adopted from the field of classical linguistics, called *homography* [109]. Homography refers to the property of words with identical representation, but different semantics. For example, *bark* has two meanings in the English language; the noise made by a dog, and the outer covering of a tree. The appropriate meaning conveyed by this word depends on the containing sentence.

We apply this property in the context of software virtualization to ensure that the protective PVM is not replaced. Traditionally, each instruction of an ISA possesses a unique semantics. In our proposed solution, certain instructions are selected from the ISA of the application, P. We refer to this set as I^{HG}, and each individual member of this set by i^{HG}. These instructions are inserted into the application at appropriate locations . Consequently, the PVM is configured to handle these instructions uniquely as follows: At run time, when the protective PVM encounters any of these select instructions, it interprets them in a custom manner,*different* from the semantics specified by the platform documentation, *i.e.*, the protective PVM instance will interpret homographic instructions using customized semantics, whereas all other interpreter instances will interpret it in the conventional manner (according to the platform documentation). When run under the protective PVM, the transformed application should generate outputs which are identical to the unprotected version, for any input combination.

The set of homographic instructions is not unique, but varies for each application. Implemented correctly, a replacement attack and subsequent interpretation on a generic interpreter will cause the application to behave in an undefined manner, and either lead to failure, or trigger appropriate response mechanisms. Thus, the replacement attack will fail in its objective of successfully executing the application without the protective PVM.

This solution methodology is termed as *Point-ISA*[1].

There are several components of this solution that need to be investigated, before a successful prototype can be implemented. Some of the primary areas of research include identifying the set I^{HG}, and designing seamless responses to a replacement attack. Care must also be taken to ensure that the transformed application running on the PVM instance generates outputs that are identical to those of the original application. The next few sections discuss these issues in detail.

Before exploring the various issues, we formalize this protection technique using our model.

[1] *Point-ISA* has been derived from the term, *point functions* in cryptography. Point functions return `true` for only one input, and false otherwise. In our case, i^{HG} corresponds to its correct value(*i.e.*, f^{-1} in only one context(*i.e.*, when it's interpreted by the associated PVM instance. In all other contexts, i^{HG} is interpreted according to the architectural ISA.

7.5.1 Formalizing Point-ISA

In Section 7.1, we used our model to describe replacement attacks. We now extend this model to characterize the Point-ISA scheme. We recall Equation 7.5, which describes the execution of the protected application on Strata. Strata itself is running on an Intel x86 hardware machine.

$$\phi_{x86}(P_{strata}, < P_{TR(APP)}, in_{APP}, c_{strata} >, m_{x86}^{in}) \longrightarrow << out_{APP}, o_{strata} >, m_{x86}^{out} > \tag{7.9}$$

In Equation 7.9, the input to the interpreter is composed of a tuple, comprising the protected software application $P_{TR(APP)}$, the input to the original application, in_{APP}, and c_{strata}, the configuration settings for Strata to run the application. For example, if the application is encrypted, c_{strata} can represent the decryption key. The memory is in its initial state, which can be broken down into the memory state for Strata, $(m_{x86}^{in})^{strata}$, and the state for the application, $(m_{x86}^{in})^{APP}$. On successful completion, the output, out_{APP} is generated, and memory reaches its final state, denoted by $m_{x86}^{out})^{strata}$, and $(m_{x86}^{out})^{APP}$.

The replacement attack results in the substitution of the protective PVM with an attack PVM, which can then be used to analyze the application without any hindrance. Recalling Equation 7.8,

$$\phi_{x86}(P_{attack}, < P_{TR(APP)}, in_{APP}, c_{attack} >, m_{x86}^{in}) \longrightarrow << out_{APP}, o_{attack} >, m_{x86}^{out} > \tag{7.10}$$

The x86 interpreter operates on the attack PVM, P_{attack}. The inputs consist of a 3-tuple comprised of the protected application, $P_{TR(APP)}$, the input to the application, in_{APP}, and a configuration setting for the PVM, c_{attack}. During interpretation, the memory state is transformed from m_{x86}^{in} to m_{x86}^{out}. On conclusion, the interpreter outputs a tuple consisting of out_{APP}, which is the output of the application, and o_{attack}, which is the output specific to the attack PVM. The adversary can configure the attack PVM to generate additional information that could facilitate analysis.

We now introduce the Ψ operator, which represents the Point-ISA protection scheme. This operator is applied to the software application, and results in a transformed application, $\Psi(TR(P_{APP}))$. Now,

the application can only be successfully executed under the mediation of its associated protective PVM, $strata_{APP}$. This protection scheme is orthogonal to other protection schemes, allowing it to be used in conjunction with other techniques.

Equation 7.11 describes the interpretation of a Point-ISA protected application on the native platform.

$$\phi_{x86}(P_{strata^{APP}}, < \Psi(TR(P_{APP})), in_{APP}, c_{strata^{APP}} >, m_{x86}^{in}) \longrightarrow << out_{APP}, o_{strata} >, m_{x86}^{out} > \tag{7.11}$$

The application has a unique PVM associated with it, denoted by $strata^{APP}$. The interpreter function, ϕ_{x86}, operates on the PVM application, $P_{strata^{APP}}$. The inputs consist of the protected application, $\Psi(TR(P_{APP}))$, the application input in_{APP}, and the configuration setting for the PVM $c_{strata^{APP}}$. The outputs consists of the output from the application out_{APP}, and the output from the PVM itself, o_{strata}. The memory state changes from m_{x86}^{in} to m_{x86}^{out}.

The interpretation of this Point-ISA protected application on the associated protective PVM is identical to the interpretation of the original application (represented by Equation 7.9). However, if this application is subjected to a replacement attack, the interpretation fails. This scenario is represented by the following equation.

$$\phi_{x86}(P_{attack}, < \Psi(TR(P_{APP})), in_{APP}, c_{attack} >, m_{x86}^{in}) \longrightarrow << (out_{APP})^{error}, o_{attack} >, m_{x86}^{out} > \tag{7.12}$$

In Equation 7.12, the x86 interpreter (ϕ_{x86}) operates on the attack PVM, P_{attack}. The inputs to the attack PVM consist of the protected application, $\Psi(TR(P_{APP}))$, the application input in_{APP}, and the configuration for the PVM c_{attack}. In this case, the application fails to execute correctly, and an error message is generated ($(out_{APP})^{error}$), along with any output from the attack PVM, o_{attack}. The memory state changes from m_{x86}^{in} to m_{x86}^{out}.

Modeling Homographic Instructions

The previous set of equations described the solution methodology at a conceptual level. We now proceed to expand our model, to describe Point-ISA at a finer level of granularity. To begin, we need to represent the interpretation of a sequence of instructions on the host machine via our model. Described in a simple form, interpreting an instruction (or a sequence of instructions) implies transforming memory of the computing machine from one state to another. To express this action in our model, we introduce a new operation, Φ_H. We describe the interpretation of a sequence of instructions in Equation 7.13.

$$\Phi_H(i_1^H, i_2^H, i_3^H .., m_H^{in}) \longrightarrow < m_H^{out} > \tag{7.13}$$

The operator, Φ_H denotes the interpretation of an instruction sequence on the host H. The sequence is denoted by $i_1^H, i_2^H, i_3^H ...$ The initial memory state is denoted by m_H^{in}, and the final state by m_H^{out}.

We can now utilize Equation 7.13 to represent homographic instructions. As we described, the semantics of the homographic instruction depend on the instance of the interpreter. We consider an instruction belonging to the x86 ISA, and its interpretation by a standard x86 interpreter, and the Strata protective PVM running on an x86 platform. Equation 7.14 describes these two scenarios.

$$\Phi_{x86}(i_{x86}^{HG}, m_{x86}^1) \longrightarrow < m_{x86}^2 > \tag{7.14}$$

$$\Phi_{strata}(i_{x86}^{HG}, m_{x86}^1) \longrightarrow < m_{x86}^3 > \tag{7.15}$$

When the instruction, i_{x86}^{HG}, is interpreted by a generic x86 interpreter, the memory state is transformed from m_{x86}^1 to m_{x86}^2. When that same instruction is interpreted by the *strata* instance, the memory state is transformed from m_{x86}^1 to m_{x86}^3. The premise behind Point-ISA is that this homographic behavior of i_{x86}^{HG} (which leads to memory states m_{x86}^2 and m_{x86}^3) can be differentiated programmatically enabling detection of the replacement attack.

We are now ready to design the Point-ISA solution. For this purpose, we utilize two functions f and f^{-1}, which are defined as follows.

$$f(m) \longrightarrow n \tag{7.16}$$

$$f^{-1}(n) \longrightarrow m \tag{7.17}$$

As Equation 7.16 represents, the function f takes an input m and transforms it to n, while the function f^{-1} transforms n back to m,(*i.e.*, the pair are the inverse of each other.

Consequently, these functions are inserted into the application. The function, f is implemented via a sequence of instructions. The function f^{-1} is implemented via a homographic instruction, i^{HG}_{x86}. The PVM is configured such that, when it encounters i^{HG}_{x86} at run time, it implements the semantics of f^{-1}. Equations 7.18 and 7.19 illustrate these scenarios.

$$\Phi_{strata}((f = \{i^1_{x86}, i^2_{x86}...i^k_{x86}\}), m^1_{x86}) \longrightarrow < m^2_{x86} > \tag{7.18}$$

$$\Phi_{strata}(i^{HG}_{x86}, m^2_{x86})) \longrightarrow < m^3_{x86} > \tag{7.19}$$

During software creation, the instruction sequence implementing f(*i.e.*,$i^1_{x86}, i^2_{x86}...i^k_{x86}$), is inserted at a suitable location in the application's CFG, followed by the insertion of i^{HG}_{x86} at a second location. The two locations are chosen such that both of them are guaranteed to be reached during program execution. In our prototype these components are aligned sequentially in program order. Dominator analysis can be used to make the arrangement less predictable. The protective PVM instance is configured such that i^{HG}_{x86} is interpreted according to the semantics of f^{-1}.

Under normal circumstances, this arrangement guarantees that the protected application produces outputs identical to those of the original application, for the identical set of inputs (we define this scenario as *output equality*). If a replacement attack occurs, the execution of a homographic instruction will leaves memory in an undefined state. We now proceed to describe appropriate response mechanisms to an attack.

Response to Replacement Attacks

During application execution, after instructions representing Equations 7.18 and 7.19 have been interpreted, m^1 and m^3 should be equal. This scenario is necessary for Point-ISA to maintain output equality. If the protective PVM instance is replaced, the attack interpreter will interpret i_{x86}^{HG} according to the ISA specifications, and not the semantics of f^{-1}. Consequently, output equality will no longer be achieved. We can respond to this event in two ways.

The first response mechanism, termed the *Value Modification* mechanism, involves letting the application continue execution. Since the memory values of the application are no longer in a predictable state (due to the execution of the actual semantics of i_{x86}^{HG}, as opposed to f^{-1}), there is a probability that the program will go down an incorrect path and fail. Such a failure is likely to be obscure due to the fact that the path taken is not predictable. The downside to this technique is that the modification may be ineffectual, resulting in normal program execution on the replacement PVM.

As such, the second response technique, termed the *Auditor* mechanism, involves the use of a guarding mechanism which checks to ensure that output equality has been maintained. If the checks detect that output equality has been broken, appropriate measures are taken. The downside to this scheme is that a knowledgeable adversary might be able to detect these checks and disable them. Therefore, to obtain a robust defense mechanism, a probabilistic mix of both schemes should suffice. Equation 7.20 describes functionality of the auditor function, which checks the memory, and triggers an attack response if they are not equal.

$$f_{audit} \longrightarrow \begin{cases} \text{if } m^3 \equiv m^1 \text{ continue execution} \\ \\ \text{if } m^3 \neq m^1 \text{ respond to attack} \end{cases} \tag{7.20}$$

m^1 denotes the state of memory prior to execution of the Point-ISA solution. m^3 denotes the state of memory after Point-ISA has successfully executed. If the two states are equal, it indicates that the execution has proceeded as expected. If the states are not equal, it signifies an attack, and an appropriate response is generated.

For the rest of this discussion, the components represented by Equations 7.18, 7.19, and the

optional part, 7.20 are referred to as the *components* of the Point-ISA methodology. In the next section, we describe the implementation of these components in greater detail.

7.6 Designing Point-ISA

There are several factors that can determine the effectiveness of Point-ISA. We address these issues in this section.

7.6.1 Identification of Homographic Instructions

The protective PVM instance interprets the redefined semantics of selected instructions. To enable such custom interpretation, the PVM must be able to distinguish the homographic instructions, I_H, from regular instructions that form part of the application. A straightforward manner to obtain such classification is to analyze the original, unprotected application, and choose an instruction that is not a constituent.

$$i_H \notin I_P \tag{7.21}$$

Consequently, the protective PVM instance is modified such that, on encountering i_H, it implements the semantics of f^-1. Each application to be protected via a PVM will possess its own I_H.

An adversary may be able to perform frequency analysis on the application, and decipher which instructions form part of I_H. To thwart such analyses, instead of a unique instruction, a unique combination of opcode and operands can be chosen. The opcode may be present in the original application, but the combination of the opcode and its operands is not. In this case, the protective PVM instance will have to decode the instruction completely (opcode and operands) before deciding whether to trigger the semantics of f^-1.

The next design issue involves devising techniques to insert the various Point-ISA components into the application such that they are executed in the correct order.

7.6.2 Insertion of Point-ISA Components

To recall, our semantic-binding scheme involves the insertion of a function, f, and a homographic instruction, i_H, into the control flow of the application, such that the output of the transformed application matches the original version, for all inputs (output equality). To achieve this property, i_H is interpreted by the protective PVM instance as f^{-1}.

To implement Point-ISA, the control flow of the application needs to be adjusted such that, when its components are scheduled for execution in correct order. For the purposes of our proof-of-concept, we inserted the Point-ISA components in sequential order at appropriate locations in the application. We describe our prototype in greater detail in Section 7.7.1

Maintaining output equality of the components is essential for correct functioning of Point-ISA. Reducing performance overhead is also critical, because a large increase is undesirable and makes the protection scheme impractical. To reduce excessive overhead, the point of insertion must not be on a frequently-executing path. This issue is similar to the issue of placement of checkers, discussed in Chapter 5. In the case of checkers, we needed to establish a balanced trade-off between checker execution and overhead, to maintain constant protection. As such, we devised predicated triggers. In the case of Point-ISA, we feel that the rate of execution of its components is not as relevant, since any one of them can trigger a response in the event of an attack. As such, we proceeded with utilizing profile information to guide the insertion process (*i.e.*, the application was run in profiled mode to obtain frequently-executing paths. Then, components are inserted based on a probability which is inversely proportional to path's frequency of execution).

7.6.3 Response to a Replacement Attack

If an adversary attempts to replace the protective PVM instance and execute the application on a different interpreter (software or hardware), the semantics of the i_H will revert to its original, and this event can be detected. We have devised two mechanisms to respond to such an attack.

Auditor Mechanism

In the first case, we employ the use of an *auditor* to detect the attack and take appropriate action. Recalling Section 7.5.1, the function, f updates a memory location's from , m^1, to m^2, whereas the protective PVM interprets i_H such that the memory location's value reverts back to m^1. The auditor is placed further along the same program path, and checks that the value of the memory location is indeed m^1. In case there is a mismatch, the auditor will trigger a response (*e.g.*, stopping application execution, or taking control along an incorrect path). Assuming that all the components (f, i_H, and the auditor) run correctly and in order, this technique guarantees that replacing the protective PVM instance will cause the application to fail.

Value Modification Mechanism

The use of an auditor leads to the creation of another point of attack. If the adversary is able to disable the auditor (*e.g.*, by replacing it with `no-op` instructions), a replacement attack will not be detected. To obfuscate Point-ISA, we modify the function, f to update a value belonging to the application (*e.g.*, modify a register) . The protective PVM instance is also updated such that the interpretation of i_H reverts the value of the variable.

The basic premise behind this response scheme is that, if the application is run under an attack PVM, f will be invoked as usual, but the interpretation of i_H will not revert the program variable. Further along this program path, the update caused by f could lead to program failure, although the final result is non-deterministic.

We have described two schemes that, although individually vulnerable to specific attacks, can mutually reinforce each other, to provide a robust protection for the protective PVM. Next, we evaluate the Point-ISA methodology, and describe the results in detail.

7.7 Evaluating Point-ISA

This section discusses the creation of a prototype implementing Point-ISA. Some of the design decision included techniques to select candidate instructions, identifying locations in the application

control flow graph where the homographic instruction could be inserted, and appropriate response mechanisms to tamper.

7.7.1 Designing the Prototype

The Point-ISA implementation depends on the complementary functions f and f^{-1}. Any number of such functions can be designed. For the purposes of our implementation, we defined f and f^{-1} in two ways, corresponding to the Value Modification and the Auditor mechanisms. Equation 7.22 describes the semantics of the functions in the case of the Value Modification Mechanism:

$$f, f^{-1}: \quad \sim \texttt{reg} \longrightarrow \texttt{reg} \tag{7.22}$$

In this scheme, f modifies an application value residing in a hardware register. `reg` can be any standard hardware register (`eax, ebx`, *etc.* on the Intel x86 platform).

Equation 7.23 describes the semantics of the functions in the case of the Auditor mechanism:

$$[MEM_1] = INIT$$

$$f, f^{-1}: \quad \sim [MEM_1] \longrightarrow [MEM_1] \tag{7.23}$$

$$f_{audit} : [MEM_1] == INIT = \begin{cases} \text{do nothing} & \text{if } \texttt{true} \\ \text{respond to tamper} & \text{if } \texttt{false} \end{cases}$$

In this scheme, the memory location, $\texttt{[}MEM_1\texttt{]}$ refers to the contents of a global memory variable that is created during the software protection process (*i.e.*, this location is not a part of the original application). This variable is initialized to a randomly selected value. f and f^{-1} consist of `negating` this variable. Finally, the audit function, f_{audit} checks whether the value of the variable has been reverted. For the sake of simplicity, we have provided only one version of f and f^{-}. A robust implementation of Point-ISA should possess several different versions of these functions.

In both these equations, `negation`($\sim$) is the primary logical operator. This operator was chosen because:

- On the 32-bit Intel x86 platform, the `negation` operator does not change any processor flags. As such, the flag register does not need to be preserved, which is beneficial for reducing overhead.

- The `negation` operator possesses the property of *involution* [110](*i.e.*, the function is its own inverse, making the implementation of f and f^{-1} identical.

The prototype is created using the link-time optimizer, *Diablo*. The inputs to Diablo consist of the original, unprotected application P, a set of homographic instructions (selection criteria for these instructions is described in Section 7.7.2) I_H, and an instance of the protective PVM (in this case, Strata). The logic of the protective PVM has been modified such that, during execution, when it encounters a homographic instruction, it implements the semantics of f^{-1}.

During software creation, we use Diablo to generate a flow graph of P. The nodes in this flow graph consist of basic blocks data structures (*i.e.*, sequence of instructions with a single entry point and a single exit point). The edges in this graph correspond to control flow branches (direct jumps, conditional jumps, calls, *etc.*). An edge typically connects a *predecessor* block (*i.e.*, the source block of the control transfer) to a *successor* block (*i.e.*, the destination block of the transfer) [111].

Once the flow graph information is obtained, Point-ISA is ready to be implemented. First, we consider the Value Modification mechanism, specifically Equation 7.22. Based on a selection criteria (described in Section 7.7.2), a target basic block X_P, belonging to P is identified. This block is subjected to liveness analysis in isolation [111], and registers are identified that contain live values exiting this block. An instruction sequence `neg`ating one of these registers, (*i.e.*, f) is inserted into a basic block data structure, BB_f. Next, an instruction is randomly selected from I_H, and inserted into another basic block structure, BB_{I_H}. The protective PVM is modified such that it interprets i_H as f^{-1}. The CFG is then modified such that X_P becomes the new direct predecessor of BB_f, and BB_f becomes the new direct predecessor of BB_{I_H}. All the previous successors of X_P becomes the new successors of BB_{I_H}. This modification ensures that any program path that contains X_P is guaranteed to execute BB_f and BB_{I_H}, enabling the Value Modification scheme.

Thus, if N_{succ} is the set of successors of X_P, the relationship is denoted by:

$$X_P \xrightarrow{pred} \{N_{succ}\}$$

We modify the CFG such that:

$$X_P \xrightarrow{pred} BB_f \xrightarrow{pred} BB_{I_H} \xrightarrow{pred} \{N_{succ}\} \tag{7.24}$$

The Auditor mechanism can also be implemented in a similar way, however the function $audit_F$, in Equation 7.23, needs to be handled as well. The instructions corresponding to $audit_F$ are stored in a new basic block, BB_{audit}. BB_{audit} is inserted into the CFG such that it becomes the new immediate successor of BB_{I_H}, and the new predecessor of all the original successors of X_P. Thus, the modified CFG would look as follows:

$$X_P \xrightarrow{pred} BB_f \xrightarrow{pred} BB_{I_H} \xrightarrow{pred} BB_{audit} \xrightarrow{pred} \{N_{succ}\} \tag{7.25}$$

where $\{N_{succ}\}$ is the set of successors to X_P in the original application, P.

Both these schemes are applied multiple times to blocks selected based on profiled information to obtain a distributed protection mechanism.

7.7.2 Selection Criteria for Homographic Instructions

Selecting the set I_H is crucial to the success of Point-ISA. These instructions should belong to the set of commonly used instructions of most applications. Such unobtrusiveness provides a natural cover from techniques such as frequency analysis. For our prototype, we created the set I_H, based on unique instruction opcodes. We statically analyzed a collection of applications binaries (SPEC CPU2000 and `binutils` package), and stored their instruction opcodes in a database. Let this set be denoted by I_D Subsequently, while processing a benchmark b, its unique opcodes are collated. This set is denoted by I_b. The set of homographic instructions can be obtained by calculating the set

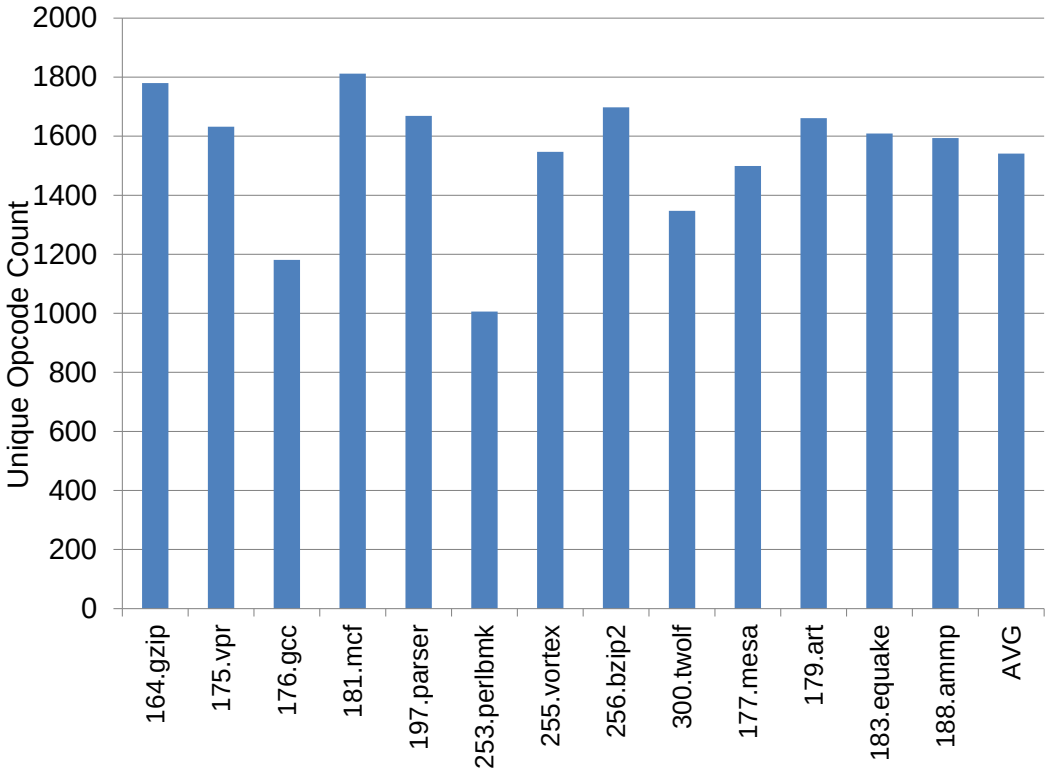


Figure 7.4: Number of opcodes in the database that can form part of I_H, for each benchmark.

difference between I_D and I_b. This equation can be represented as follows:

$$I_H = I_D \setminus I_b := \{x \in I_D,\ x \notin I_b\}$$

where $\setminus$ denotes the set difference operation. Any opcode in the database that does not occur in b, can be used to form I_H. Once I_H has been identified for a particular application, an instance of the protective PVM is created that handles each member instruction, i_H accordingly.

Figure 7.4 displays the number of opcodes that can be used to form the set I_H, for each benchmark of the SPEC CPU2000 suite. As the figure illustrates, each benchmark has several options. Because all these opcodes occur in standard applications, their presence in the protected application should be well camouflaged. The options for I_H can be further increased by considering instruction operands in addition to the opcodes. In this case, the number of options increased by several orders of magnitude.

During the software creation process, Diablo selects one instruction at a time using a random scheme. This instruction is then inserted in basic block B, and the process continues as described in Section 7.7.1.

7.7.3 Performance

As with any other protection scheme, low overhead is an important factor in the design of Point-ISA. High overhead may limit the adoption of this technique.

The selection of block X_P is directly related to the overhead associated with Point-ISA. Since all the inserted blocks are predecessors of this particular node, a high execution frequency for this node will translate into a high execution frequency for the newly inserted nodes. A high frequency adds to the overhead. Therefore, we need a judicious arrangement to select X_P. In Chapter 5, one of the schemes designed to insert checkers into the application involved generating probabilities of basic blocks that was inversely proportional to their execution frequency. The downside of such a scheme is that, in certain applications, the standard deviation of the checker frequency can be inordinately high. This disadvantage is not quite so bothersome in Point-ISA. As long as the Point-ISA components execute, protection will be achieved. The relative spread of their execution times is not relevant.

Therefore, we utilize the inverse-frequency scheme to generate probabilities for each basic block of P. During software creation, the selection of X_P depends on its assigned probability. Figure 7.5 shows the overall performance overhead for the SPEC CPU2000 benchmarks. On average, this technique adds an overhead of 5% over Strata. Since the placement method is probabilistic, there is a non-zero chance that the Point-ISA components will be placed in a hot path (*e.g.*, `177.mesa`). In such a case, it is judicious to rerun the software creation process to get a more favorable binary.

Figure 7.6 shows the average time delay between two successive execution of the Point-ISA components. The error bars correspond to the standard deviation. The high standard deviation in benchmarks such as `181.mcf` and `255.vortex` indicate that the components execute in intermittently in groups, rather than periodically. As we mentioned previously, the rate of execution of these components is not as relevant to the protection offered.

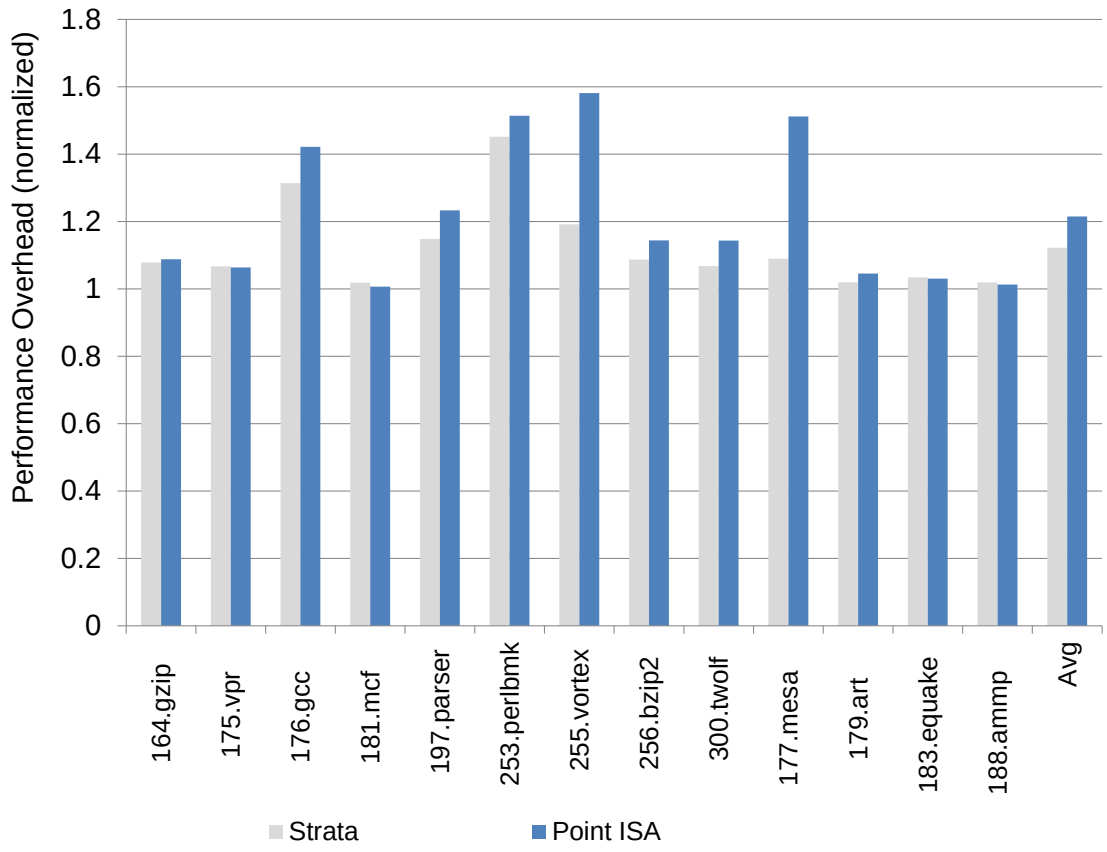


Figure 7.5: Performance overhead for Point-ISA normalized to native. Overhead for Strata is provided for reference.

7.8 Security Discussion

This section discusses the impact of Point-ISA on program protection. Specifically, we consider the robustness of this scheme against particular attack methodologies. We also examine the presence of any weaknesses that an adversary might exploit and offer alternate schemes.

7.8.1 Effectiveness against Replacement Attacks

Point-ISA was designed specifically to thwart replacement attacks. Therefore, its effectiveness against such attacks is paramount. We applied the process described in Section 7.7.1 to several benchmarks, and subjected them to replacement attacks. All the protected benchmarks were able to repel the attacks. Most of the benchmarks exited with a message generated by the auditor. However, some

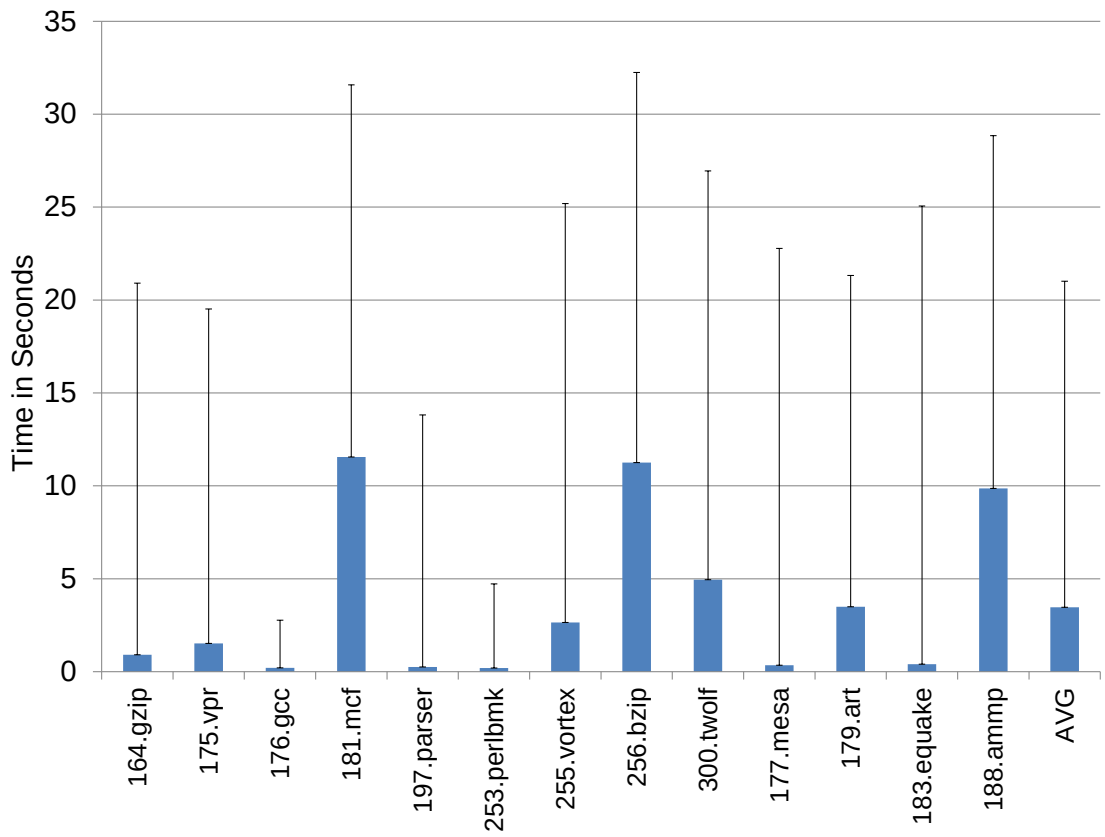


Figure 7.6: The average delay between the invocations of two Point-ISA components. The error bars correspond to the standard deviation.

benchmarks generated the `SIGSEGV` faults, which indicate that the Value Modification mechanism had been triggered and led to the corruption of the application.

7.8.2 Effectiveness against Reverse Engineering

In Section 7.7.1, we presented a straightforward mechanism to implement Point-ISA, which consisted of inserting the Point-ISA components in a predictable order into the CFG of P. Such a naive implementation can be disabled by a knowledgeable adversary.

Data-flow analyses can be used to make Point-ISA more robust against reverse-engineering attacks. `use-def` is commonly-used analysis technique which identifies instructions that define and use the variables in an application. The sequence of instructions that mark the *definition* and *use* of a variable are termed *chains*. In the Value Modification Scheme, the use of `def-use` chains can be used

to identify multiple locations to insert the Point-ISA components. Blocks BB_f and BB_{I_H} can be inserted randomly at any site within the confines of the `use-def` chain associated with the live value being modified. This additional entropy makes it harder for the adversary to identify and disable these Point-ISA components. The amount of entropy will vary depending on the variable chosen. Applying this scheme multiple times should give a good spread.

To increase the robustness of the Auditor mechanism, we utilize dominator analysis [111]. In any directed graph (*e.g.*, the CFG of an application), a node d is called the dominator of node e if all paths from the graph entry point (i.e., program entry point) to e pass through d) [111]. Similarly, the node e post-dominates d if the paths from d to the graph exit point pass through e. Referring to Equation 7.25, the components BB_f, BB_{I_H}, and BB_{audit} can be placed on the post-dominator path of X_P to application exits. Care must be taken to ensure that the components all execute the same number of times. This restriction can be enforced trivially by avoiding their placement in any loops.

$$1_P.. \xrightarrow{sdom} BB_f.. \xrightarrow{sdom} BB_{I_H} \xrightarrow{sdom} 2_P.. \xrightarrow{sdom} BB_{audit}.. \xrightarrow{sdom} X_P \tag{7.26}$$

The blocks 1_P, 2_P.. refer to basic block blocks of the original application. Equation 7.26 implies that the components are inserted at random locations in the dominance tree of X_P. Thus, this scheme makes Point-ISA more unpredictable and consequently, harder to reverse engineer.

7.8.3 Tampering with the Protective PVM

A knowledgeable adversary can attempt to uncover the special semantics of the homographic instructions by investigating the instruction handling logic of the protective PVM instance, and comparing with the ISA manual. Instruction handling is one of the most complicated components of any PVM implementation. Applying traditional program protection techniques to the PVM can make reverse engineering even harder. Thus, while it is theoretically possible to analyze the PVM and discover the homographic instructions, we do not see a trivial way to achieve this task. The adversary will have to manually investigate the handling of each possible instruction. By expanding

the domain of homographic instructions to include instruction operands as well, this task can be made still harder.

7.8.4 Synopsis of Overall Protection

Point-ISA is designed to be used in conjunction with static protections and a protective PVM. The static protections safeguard the application and the PVM binary from analysis and tamper. At run time, the associated PVM instance continuously applies tamper-resistance techniques, making dynamic analysis harder. Point-ISA ensures that this protective PVM instance is not replaced by a benign PVM instance by an adversary attempting to obviate security measures. Thus, taken together, such a protective configuration lays the foundation for a robust tamper resistance execution environment.

7.8.5 Co-designing Applications and protective PVMs

The use of homographic instructions can be extended, to codesign applications with protective PVMs. During software creation, large sequences of code (*e.g.*, a function block) of the application can be replaced by a homographic instruction. At run time, when the PVM encounter this instruction, it will interpret it according to the semantics of the replaced block. In this scenario, further analysis is required to ensure that, in the case of a replacement attack, the interpretation of the homographic instruction actually leads to a successful detection of the attack. This topic is a source of further research.

7.9 Summary

In this chapter, we explored the relationship between the application and the protective PVM instance. We discovered that the PVM is not strongly bound to the application. Such a weakness can be easily exploited by a knowledgeable adversary. We prototyped a PVM-replacement attack methodology based on this weakness, which completely obviated all dynamic protections based on PVMs. To combat this attack, we adapted the property of homography from classical linguistics, and applied

to to the field of software virtualization. Using homographic instructions, we designed a solution methodology, Point-ISA, that can thwart replacement attacks. Our prototype provided encouraging results on the success of this solution. PVMs have already been shown to provide strong dynamic protections. Combined with Point-ISA, the effectiveness and robustness of PVMs increases beyond the state-of-the-art.

Chapter 8

Establishing a Data Dependence between the Application and the PVM

At this point, we recap software protections from the view of the application's state. *Prior* to application execution, static techniques protect the binary from reverse engineering. This area of protection has been thoroughly studied, and several robust mechanisms have been proposed [9, 5, 6, 81]. Chapter 4 proposes new techniques that protect virtualized applications from analysis. Thus, these mechanisms perform effectively at thwarting static attacks.

During execution, protection is provided by the associated PVM. Chapters 5 and 6 describe effective schemes that obfuscate the run time of the application, repelling reverse engineering attacks. Chapter 7 illustrated a weakness in current PVM-based protection technology, and proposed a solution that establishes a semantic relationship between the protective PVM instance and the application. Incorporating these techniques into the application provides a robust tamper-resistant environment at run time.

The question remains whether it is possible for an adversary to extract critical information *after*

execution has concluded. For example, the adversary might be able to obtain a protected copy of the software application and proceed to run it to completion under the control of an analysis tool (*e.g.*, an instrumentation framework, or a modified OS). The question we are addressing is, whether the adversary can extract the application's assets from any post-execution analysis. Examining the susceptibility of PVM-based protections against such exploitation is the focus of this chapter.

As previous research uncovers, it is indeed possible to inspect software, and extract critical information using such post-execution analyses (i.e, by examining the application *trace*, which refers to the sequence of executed instructions) [57]. Such attacks consist of isolating unique application attributes (*e.g.*, system calls) from the trace, and performing a data flow analysis . In traditional PVM systems, the data access patterns of the application, and those of the PVM are distinct. As such, examining the flow of data could reveal all the relevant information about the original application.

This chapter deals with the investigation of such dataflow attacks on PVM-protected applications. An implementation of this attack methodology was first realized by Coogan, Lu and Debray [57]. The authors termed this attack methodology as Value-based Dependence Analysis (VDA). Their scheme consists of identifying instructions that perform arithmetic and logical operations directly on application data, neglecting other operations (e.g, address calculation, conditional jumps, *etc.*)[1] Applying this methodology to a virtualized application's trace reveals those instructions that implement the core functionality of the application. VDA can be classified as a search-space reduction technique.

In this chapter, we thoroughly investigate this attack methodology. We begin by formally defining the attack methodology using our model. We demonstrate that Coogan, *et al.*'s implementation can be easily defeated by binary translators, and propose a more robust attack that targets all types of PVMs. Next, we describe a novel scheme that thwarts this attack. This solution mechanism, called DataMeld, involves establishing a relationship between the data values of the PVM and the protected application. Our evaluation reveals that our solution significantly reduces the amount of relevant

[1]Inherently, VDA is similar to a well-known security technique, *taint checking*. In taint checking, program variables that are susceptible to malicious data (*e.g.*, program inputs) are identified, and the flow of their data values is tracked throughout application execution. The major difference between VDA and taint checking is that, tainting is a forward flow analysis (the data values are tracked in program order), whereas VDA is a backward flow analysis (the data values are tracked from application attribute to program entry).

information that can be extracted by the adversary, post execution.

The major contributions of this chapter can be summarized as follows:

- Description of a class of data-analysis attacks on PVM-protected applications, called Value-based Dependence Analysis (VDA). Initially, it was thought that PVMs were effective at obfuscating the execution trace of the application. However Coogan, *et al.* demonstrated that it is relatively straightforward to extract the relevant information from the trace [57]. We represent this attack in our model to facilitate a better understanding.

- Design of a run-time solution that thwarts Coogan's attack implementation, called dynamic address-mode transformation. We demonstrate that the original VDA scheme can be thwarted by changing the addressing modes of instructions. We employ this scheme via a software dynamic translator. We use a case study to demonstrate that this solution can defeat Coogan, *et al.*'s implementation.

- Design of a more robust implementation of the VDA methodology. The new scheme, called modified VDA (MVDA), has more overhead (in terms of redundant information extracted) than Coogan, *et al.*'s implementation, but is effective against addressing-mode changes. We extend our case study to demonstrate that MVDA is able to extract critical information from the PVM-protected trace.

- Conception of a novel scheme that consists of blending the data of the PVM and the protected application, called DataMeld. This scheme operates by modifying the data values of the application using the values of the PVM in a semantically-neutral manner. The basic premise behind this solution is to obfuscate the dynamic data flow of the application by using the PVM.

- Introduction of a metric for calculating the memory coverage of variables, called instruction coverage for variables (ICV). ICV values facilitate the selection of PVM variables for binding with the application.

- Finally, the design of a robust prototype for evaluating these ideas. We demonstrate that DataMeld increases the obfuscation in the trace of the PVM-protected application. Thus dataflow attacks like VDA have a reduced chance of success.

The rest of the chapter is organized as follows: in Section 8.1, we explore the concept of data-based attacks on the PVM-protected trace. We discuss the attack devised by Coogan, *et al.* in this section. In Section 8.2, we demonstrate that modifying the addressing modes of instructions that perform direct register transfers can thwart VDA attacks. We employ a use case to illustrate our assertion. In Section 8.3, we describe a modified version of the VDA attack, that can withstand addressing mode transformations. Section 8.4 describes a robust solution to the VDA methodology that obfuscates the dataflow of the application by blending it with the data flow of the PVM. Section 8.5 presents a the concept of instruction coverage for variables, to facilitate selection, and describes the design of a tool to calculate this metric. Section 8.6 we review past work that has investigated extracting data structure information from binaries. We utilize this work for implementing DataMeld, which is described in Section 8.7. In Section 8.8, we present an evaluation of this obfuscation, by calculating the amount of information leakage. Finally, we summarize the chapter in Section 8.9.

8.1 Value-based Dependence Analysis (VDA) Attacks

Software applications modify data values during processing. Any non-trivial application will apply a myriad of operations on many different data values during the course of its execution. Because most modern processors have a finite set of hardware registers, these values are temporarily stored in memory locations while other values are being computed. Dataflow analysis investigates the transfer of data between memory locations and hardware registers on the native machine, and by extension, the instructions that access such data values.

The introduction of a virtualization layer in an application changes the dynamic execution environment. Within the context of data flow, the PVM and the application are two separate software entities, each with their own set of data values, which do not intersect. This dichotomy forms the premise behind the VDA attack methodology by Coogan, Lu, and Debray [57]. The

critical aspect of their analysis is the identification of an attribute that belongs exclusively to the application. The authors decided on utilizing system calls as the attribute, since most PVMs use a limited set of system calls (usually related to file I/O, such as `open, read`, *etc.*). Because the system call arguments and return values are standardized for every platform, it is easy to identify them. Once these values are recognized, VDA operates by tracking their flow through the instruction trace. In this manner, all the critical instructions (*i.e.*, those belonging to the application) can be identified. The instructions belonging to the PVM in the dynamic trace will be ignored. Results have indicated that this scheme can aid reverse engineering by reducing the search space by orders of magnitude [57].

The VDA attack methodology can be elegantly represented in our formal model. Recalling Equation 7.6, which describes execution of a an application on a software interpreter, which in turn, is running on an Intel x86 hardware platform.

$$\phi_{x86}(P_{strata}, < P_{TR(APP)}, in_{APP}, c_{strata} >, m1_{x86}) \longrightarrow << out_{APP}, o_{strata} >, m2_{x86} > \tag{8.1}$$

In Equation 8.1, the hardware interpreter operates on the Strata application, P_{strata}. The inputs to Strata comprise of a tuple, comprising the protected software application $P_{TR(APP)}$, the input to the original application, in_{APP}. The configuration settings for Strata are also given as input, c_{strata}. On completion, output out_{APP} is generated, along with any outputs from Strata, o_{strata}. The memory of the host machine is transformed from $m1_{x86}$ to $m2_{x86}$.

Next, we analyze the memory used by the interpreter. As we have described, these can be broken down into two separate components, one belongs to the application, and the other to Strata. Consequently, in Equation 8.1, the memory state at the beginning of interpretation, $m1_{x86}$ can be split into $m1_{x86}^{strata}$, and $m1_{x86}^{APP}$, represented in Equation 8.2.

$$m1_{x86} = m1_{x86}^{APP} \cup m1_{x86}^{strata} \tag{8.2}$$

Similarly, the output memory can also be split in to $m2_{x86}^{strata}$, and $m2_{x86}^{APP}$, represented by Equation 8.3.

$$m2_{x86} = m2_{x86}^{APP} \cup m2_{x86}^{strata} \tag{8.3}$$

Rewriting Equation 8.1 with these changes, we have:

$$\phi_{x86}(P_{strata}, < P_{TR(APP)}, in_{APP}, c_{strata} >, m1_{x86}^{APP} \cup m1_{x86}^{strata}) \longrightarrow << out_{APP}, o_{strata} >, m2_{x86}^{APP} \cup m2_{x86}^{strata} > \tag{8.4}$$

Equation 8.4 illustrates the interpretation of a PVM-protected application, with the distinct memory components. The inputs to Strata comprise of a tuple as before, comprising of $P_{TR(APP)}$, the input to the original application, in_{APP}, and the configuration settings for Strata, c_{strata}. On completion, output out_{APP} is generated, along with any outputs from Strata, o_{strata}. In this case, the memory components have been split. The input memory can be described as the union of the memory state of the application and the PVM. The same case applies to the memory state at the end of interpretation.

The basic premise behind this attack methodology is that the memory components are mutually exclusive. Thus, for the input components, we represent this situation in Equation 8.5

$$m1_{x86}^{APP} \cap m1_{x86}^{strata} = \emptyset \tag{8.5}$$

The same scenario holds for the output memory components, represented in Equation 8.6.

$$m2_{x86}^{APP} \cap m2_{x86}^{strata} = \emptyset \tag{8.6}$$

To summarize, the sets of data values accessed by the application and the protective PVM are distinct.

Given Equations 8.5 and 8.6, if an adversary is able to identify a member of the application's data value set, and iteratively analyze the flow of the value, it could reveal other relevant values that belong to the application, and the instructions that access them (Relevance implies that the value is a partial product that is pertinent to the eventual computation). Once the application's instruction sequence is obtained in this manner, the adversary can apply any number of known reverse-engineering techniques to obtain its representation at a higher level of abstraction and analyze its functionality.

Now that we have established the main concepts behind the VDA methodology, we describe the implementation by Coogan, *et al.*.

8.1.1 Coogan, *et al.*'s implementation of VDA

The domain of Coogan, *et al.*'s investigations involved malware protected by software interpreters, such as VMProtect [36] and CodeVirtualizer [37]. This point is pertinent, because malware is minuscule in size, compared to generic applications this research is attempting to protect. When their methodology is applied to generic software, the amount of protection offered is reduced. But the general principle still holds.

Their implementation consists of two steps. The first step involves data analysis of the trace to recognize instructions that belong to the original application. The second step involves performing control flow analyses on the instructions identified to generate program structures (*e.g.*, call-return pair, jumps, *etc.*) such that the representation is at a higher level of abstraction. The control-flow analyses are not specific to reverse engineering virtual machines, but can be applied to attack any generic obfuscation technique. As such, we focus on the first part of the attack,(*i.e.* the data-flow analysis).

To initiate the attack, a run-time trace of the interpreted application is generated and recorded. The next step involves identifying system calls that belongs to the original application. System calls belonging to the interpreter can be easily identified by generating the execution trace of two different PVM-protected applications and matching the common system call invocations.

Once the system calls are identified, the data path of their arguments are investigated. As we have established, the goal of this analysis is to identify all data values and, by extension, instructions that directly or indirectly access these values. The direction of this analysis is reverse to execution order,(*i.e.*, from the system call back to the next system call or the first instruction executed). The analysis terminates when another system call is encountered or the first instruction of the trace is reached.

A notable feature of their VDA implementation is that it essentially focuses on the flow of *values* and neglects non-essential computations, such as address calculation. As such, the extracted information deals almost entirely with the algorithms that have been implemented in the application. By ignoring such non-essential constructs, this feature simplifies the adversary's work load.

Listing 8.1: Value-based Dependence Analysis of PVM-protected applications, as defined by Coogan *et al.* [57]

```
VDA(T, C)
inputs:T  Trace of PVM-protected Application, P
       :{C}  Set of system calls belonging to P
S = {}
i = last instruction of T
while (i ∈ T ≠ empty)
do
         if (i ∈ C)
         then
                  S = {}
                  S = S ∪ use(i)
                  mark(i)
         endif
         if (def(i) ∈ S )
         then
                  S = S − def(i)
                  S = S ∪ use(i)
                  mark (i)
         endif
         i = prev(i)
done

use(i):
inputs: i  a read operand
         if (i reads register r)
         then
                  return r
         else if (i references memory address a )
                  return a
         endif
         return NULL

def(i):
inputs: i  a write operand
         if (i writes register r)
         then
                  return r
         else if (i writes memory address a )
                  return a
```

```
                endif
                return NULL
```

Listing 8.1 illustrates the VDA algorithm together with the modified definitions for `use` and `def`. The analysis begins at a system call invocation known to be part of the application. For each such system call, the Application Binary Interface (ABI) information is used to identify its arguments. At the onset, a set S is initialized with the locations containing these arguments. Then the trace is scanned in reverse, and each instruction is processed as follows: if I defines a location $l \in S$, I is marked relevant, l is removed from S, and the set of locations used by I is added to S. This scan continues until S is empty, the beginning of the trace is reached, or another system call belonging to the application is reached. If the terminal condition is another relevant system call, this process is repeated. The definition of `use` and `def` has been modified from the classical definition, to focus on the values (registers and addresses), rather than the locations of these values. This modification reduces the amount of information extracted, because address computations are ignored. Therefore, only the instructions dealing with core computation are identified.

We apply Listing 8.1 to an example code snippet obtained from the trace of a PVM-protected application, displayed in Listing 8.2 in static single assignment (SSA) form[2]. We then study the effects of VDA on this snippet. The system call, *print*, has been identified as part of the application. The application of VDA to Listing 8.2 works as follows:

Listing 8.2: Example assembly code

```
/*I0*/   add esp1, 8              ;pop off two arguments of the stack
/*I1*/   popf                     ;restore flags
/*I2*/   popa                     ;restore registers
/*I3*/   lea esp2, [esp1 + 32]    ;pop off 32 bytes off the the stack
/*I4*/   jmp [esp2-24]            ;jump to address
/*I5*/   mov ecx1, 0x34           ;mov 0x34 to ecx
/*I6*/   mov eax1, [ecx1 + 0x1000] ;move the value of memory location a to eax
/*I7*/   mov ebx1, 0x124          ;move 0x124 into ebx
/*I8*/   add eax2, ebx1           ;add ebx to eax and store in eax
/*I9*/   mov ebx2, eax2           ;store value in ebx, as argument for syscall
/*I10*/  mov eax3, 4              ;load syscall number into eax
/*I11*/  int 0x80                 ;invoke system call ABI with argument
```

[2]SSA is a representation technique in which each assignment denotes a new variable [111]. New variables are typically denoted by a subscript on the name of their container. Thus, eax_1, and eax_2 refers to different variables that reside in `eax`

- I_{11}: invokes the system call. ABI information reveals that system call, *print* takes one argument, which is placed in register `ebx`. So, this instruction is tagged, and the set S is initialized with register `ebx`, *i.e.*, $S = \{ebx\}$

- I_{10}: moves the system call to the register `eax`. This instruction is tagged. S remains unchanged, *i.e.*, $S = \{ebx\}$.

- I_9: moves a value from register `eax` to `ebx`. Since `ebx` is a member of S, we add `eax` to S, and remove `ebx`, *i.e.*, $S = \{eax\}$. This instruction is tagged.

- I_8: adds eax and ebx, and stores it in eax. This instruction is tagged. Thus, $S = \{eax, ebx\}$

- I_7: stores an immediate value in `ebx`. Since `ebx` is in S, we remove it from the set, $S = \{eax\}$, and we tag this instruction.

- I_6: stores a value from memory location, $(ecx_1 + 0x1000)$ to `eax`. Thus $S = \{(ecx_1 + 0x1000)\}$. Note that `ecx` is *not* included in S because it deals with address calculation.

- I_5: stores an immediate value in `ecx`. Since `ecx` is not in S, this instruction is ignored. $S = \{(ecx_1 + 0x1000)\}$.

- I_4: makes an indirect jump off the stack. This instruction is ignored.

- I_3, I_2, I_1, I_0 perform processing which does not involve memory location $(ecx_1 + 0x1000)$. Hence, these instructions are ignored.

Thus, VDA tags the following instructions as belonging to the application:

$$\{I_6, I_7, I_8, I_9, I_{10}, I_{11}\} \tag{8.7}$$

Applied iteratively on the instruction trace, VDA is able to identify all the application instructions that are used for core computation. Thus, one of the major protections offered by the PVM is successfully removed.

In the next section, we provide a solution to Coogan, *et al.*'s implementation of VDA. This solution is based on software binary translators, and exploits the fact that their implementation ignores instructions that perform address calculation.

8.2 Effectiveness of VDA on Binary Translation Systems

The VDA implementation designed by Coogan *et al.*, addresses software interpreters, although the basic theory is applicable to any PVM technology. An optimization devised by the authors involved ignoring any instructions that deal with address calculation, and only focusing on memory and registers contents, containing partial computation products. We describe a scheme that thwarts Coogan, *et al.*'s design by transforming the addressing modes of application instructions at run time.

8.2.1 Dynamic Transformation of Addressing Modes (DTAM)

DTAM operates by transforming addressing modes. It has been designed specifically for software dynamic translators (SDT) as these translators are able to generate and modify code at run time. The main premise behind this solution involves changing direct data transfer to indirect transfers. Making this transformation on the application is relatively straightforward using SDTs. We validate this claim using Strata.

Our DTAM implementation is targeted towards PVM-protected application running on 32-bit Linux platform. We assume that the size of an `int` variable is 4 bytes. At application start up, the PVM allocates a chunk of contiguous memory, M_{base} of size $n * 4$, located at address $base$. Then, this memory is initialized, such that the offset $4 * i$ contains the value i.

$$[base + index * 4] = index \tag{8.8}$$

During execution, whenever the PVM encounters application instructions containing direct register transfers, it translates such transfers into based indexed indirect transfers [112]. Consider the simple transfer listed in Equation 8.3.

Listing 8.3: Direct register transfer

```
        mov eax, ebx
```

Strata translates this instruction into the sequence listed in Listing 8.4.

Listing 8.4: Indirect data transfer, created by Strata at run time.

```
        push ecx
        mov ecx, base
        mov eax, [ecx + ebx*4]
        pop ecx
```

The register that is used to store the base address of the memory block,in this case, `ecx`, can be any register that is not being used as an operand in the direct transfer. Because Coogan, *et al.*'s implementation ignores instructions dealing with address calculation, it will fail to track the flow of data through the base and index registers (`ecx` and `ebx` in Listing 8.4, affecting the effectiveness of VDA.

We apply dynamic address-mode transformation to the code snippet presented in Listing 8.2. Listing 8.5 illustrates the modified snippet, as generated by Strata.

Listing 8.5: Example assembly code using based, indexed indirect transfers

```
init_array:                                 ;function to initialize the memory array
                  mov ebx, 0xdeadbeef       ;load start address
L1:                                         ;start of loop
                  mov ecx, 0                ;set counter to zero
                  mov [ebx], ecx            ;set counter value at location
                                            ;(start address + 4 * counter)
                  add ebx, 4                ;increment the address
                  inc ecx                   ;increment counter
                  xor ecx, n                ;check for loop termination
                  jnz L1                    ;iterate if loop not broken
                  ...
                  ...
        /*I0*/    add esp1, 8               ;pop off two arguments of the stack
        /*I1*/    popf                      ;restore flags
        /*I2*/    popa                      ;restore registers
        /*I3*/    lea esp2, [esp1 + 32]     ;pop off 32 bytes off the the stack
        /*I4*/    jmp [esp2-24]             ;jump to address
        /*I5*/    mov ecx1, 0x34            ;mov 0x34 to ecx
        /*I6*/    mov eax1, [ecx1 + 0x1000] ;move the value of memory location b to eax
        /*I7*/    mov ebx1, 0x124           ;move 0x124 into ebx
        /*I8*/    add eax2, ebx1            ;add ebx to eax and store in eax
        /*I9*/    push edx1                 ;save register
        /*I10*/   mov edx2, 0xdeadbeef      ;set location of array in memory
        /*I11*/   mov eax3, [edx2 + eax2]   ;load location (0xdeadbeef + eax) into eax
        /*I12*/   pop edx3                  ;restore register
        /*I13*/   mov ebx2, eax3            ;store value in ebx, which acts as argument for syscall
        /*I14*/   mov eax4, 4               ;load syscall number into eax
        /*I15*/   int 0x80                  ;invoke system call ABI with argument
```

Strata adds a function, `init_array`, to initialize the block of memory. Applying the VDA scheme to this code snippet yields:

- I_{15}: invokes the system call. ABI information reveals that system call, *print* takes one argument, which is placed in register `ebx`. So, this instruction is tagged, and the set S is initialized with `ebx` ,(*i.e.*, $S = \{ebx\}$
- I_{14}: moves the system call to register `eax`. This instruction is tagged.
- I_{13}: moves a value to register `ebx`. As `ebx` is in set S, we remove it, and add `eax`, *i.e.*, $S = \{eax\}$. This instruction is tagged, and $S = \{eax\}$.
- I_{12}: restores a value to register `edx`, which is not in S. This instruction is ignored.
- I_{11}: moves a value from memory to register `eax`. As `eax` is in S, we remove it from the set, and add the memory location. This instruction is tagged, and $S = \{(eax_2 + edx_2)\}$.
- I_{10}: loads the start address of the memory buffer into `edx`. This instruction is ignored.
- I_9: saves `edx` on the stack. This instruction is ignored.
- I_8: adds the value of `eax` and `ebx`. As `eax` is no longer in S, this instruction is ignored.
- I_7: loads a value in *ebx*. This instruction is ignored.
- I_6: loads a value from memory into eax. This instruction is ignored.
- I_4 through I_0: are all ignored by Coogan, *et al.*'s implementation VDA, as none of them write to any of the contents of the set, S.

This analysis will eventually tag the instructions of the function `init_array`, completely ignoring the instructions that actually implement the core functionality. Thus, in the original code snippet of Listing 8.2, Coogan, *et al.*'s analysis only identifies the following instructions as belonging to the application.

$$\{I_{11}, I_{13}, I_{14}, I_{15}\} \tag{8.9}$$

The instructions performing core computations (I_5 through I_8) are ignored.

As our investigation illustrates, this scheme significantly reduces the effectiveness of VDA attacks. In particular, arithmetic and logical operations are ignored. As such, the adversary will be unable to identify the majority of instructions that form the original application.

Caveats

Although this solution is effective at thwarting VDA attacks, there are a couple of points that should be recognized.

A point of concern involves the size of the M_{BASE}(*i.e.*, n. Theoretically, it should be equal the maximum possible value that can be stored in a variable of type `int`. However, such a value would lead to a prohibitively high memory overhead. To reduce overhead, n should be set to a reasonable value. At run time, the protective PVM will transform the addressing mode only if the value being transferred using register direct mode is less than n.

We also acknowledge that this technique has a high performance overhead, because it replaces a simple register-to-register transfer with instructions that access memory. If the instruction to be replaced exists on a frequently-executed path, the overhead implications could be severe. As we have already seen in Chapter 5, profiling the application provides a simple technique to identify basic blocks (and instructions) that are on *hot* paths. When applying this protection scheme, the application should be profiled, and only those direct transfers targeted for replacement, that do not lie on such paths.

In the next section, we modify Coogan, *et al.*'s original design of VDA to be effective in the presence of DTAM.

8.3 Modified Value-based Dependence Analysis (MVDA): An Example of White-hat Attack

The area of program protections is attritional by nature. The adversary designs techniques to steal assets from applications. The software defender develops mechanisms to protect applications from such attacks. Then, the adversary constructs schemes to disable the protections, and so forth. Aware of this cyclical nature of protections, most security researchers often attempt to break their own protection techniques, as a measure of robustness. This technique, called 'white hat' or ethical hacking, offers the security provider a view of the protections from the perspective of the adversary. Such schemes often lead to discovery of weaknesses and loopholes, that may not be clear initially (such as the attack on Skype [19]).

In this section, we switch our viewpoint to that of an adversary. In the last section, we demonstrated that Coogan, *et al.*'s implementation of the VDA attack is relatively straightforward to overcome, specially using a PVM based on binary translation that can perform code manipulations at run time. Rather than wait for an adversary, it would be prudent to attempt to design the next step in the evolution of the VDA methodology. The results of such an analysis could be used to make protections more robust.

The DTAM scheme operates by modifying address transformations. We opted to attack this aspect of the solution. Thus, we modify VDA to include the operands used in address calculation in its analysis as well. That is, the analysis of the dataflow should include the address generation operands as well.

Applying this modification to our definition of VDA in Listing 8.1 yields the modified definition displayed in Listing 8.6.

Listing 8.6: Modified definitions of def and use, to thwart trivial solution.

```
use(i):
   inputs: i  a read operand
           if (i reads register r)
           then
                   return r
           else if (i references memory address a )
                   return a and any registers used in calculation
           endif
```

```
                return NULL

        def(i):
        inputs: i  a write operand
                if (i writes register r)
                then
                        return r
                else if (i writes memory address a )
                        return a
                endif
                return NULL
```

In this listing, the function `use`, is modified to return any registers that were used to generate the address. For example, on a Intel x86 platform, applying `use` to the instruction in Listing 8.7 returns `ebp, esi`, and the address pointed to by (`ebp + esi`), as illustrated by Listing 8.7.

Listing 8.7: Example of address calculation in an instruction

```
        mov  eax, [ebp + esi]
```

We now apply this modified VDA algorithm to the code snippet presented in Listing 8.5. This analysis yields:

- I_{15}: invokes the system call. ABI information reveals that system call, *print* takes one argument, which is placed in register `ebx`. So, at the onset, the set S is initialized with `ebx`, i.e., $S = \{ebx,\ eax\}$
- I_{14}: moves the system call number to register `eax`. This instruction is tagged, and $S = \{ebx\}$.
- I_{13}: moves a value to register `ebx`. Since `ebx` is in set S, we remove it, and add `eax`, *i.e.*, $S = \{eax\}$
- I_{12}: restores a value into `edx`. Since this register is not in S, we ignore this instruction.
- I_{11}: stores a value into `eax`, which is present in S. This instruction exemplifies our modification. Now, we replace `eax` in S with `edx`, `eax`, and the memory address `(eax+edx)`, *i.e.*, $S = \{eax, edx, (eax + edx)\}$. This instruction is tagged.
- I_{10}: moves an immediate value into `edx`, which is present in S. Hence, `edx` is removed, and this instruction is tagged, *i.e.*, $S = \{eax,\ eax{+}edx)\}$.

- I_9: stores the value in `edx`. This instruction is ignored.
- I_8: stores a value in `eax`, which is in S. Hence, `eax` is removed, and the operands, `ebx`, and `eax` are inserted. This instruction is tagged, and $S = \{$*eax, ebx, (eax+edx)*$\}$.
- I_7: moves an immediate into `ebx`. This instruction is tagged, and $S = \{$*eax, (eax+edx)*$\}$.
- I_6: moves a value from memory to `eax`, which is in S. Hence, this instruction is tagged, and the memory location, b is added. The register used in address calculation is also added to S, resulting in $S = \{$*(eax+edx), (ecx + 0x1000), ecx*$\}$.
- I_5: moves an immediate value to `ecx`. This instruction is tagged. $S = \{$*(eax+edx), (ecx + 0x1000)*$\}$.
- I_4 through I_0: are ignored, since they do not involve writing to locations a and b.

This analysis results in the following instructions being tagged:

$$\{I_5, I_6, I_7, I_8, I_9, , I_{10}, I_{11}, I_{12}, I_{13}\} \tag{8.10}$$

Comparing Equation 8.7, Equation 8.9, and Equation 8.10, we observe that MVDA does output more instructions than Coogan *et al.*'s implementation, but less than the actual trace size. All the core instructions are present in the output of MVDA.

This running example illustrated the effects of the VDA methodology over a small code snippet. We now expand the scope of the analysis, to include the whole application trace, and evaluate the effects of VDA. The next section describes our results.

8.3.1 Evaluation

To judge the effectiveness of both VDA and MVDA techniques, we decided to compare the dynamic instruction count (DIC) obtained by applying these techniques to run-time traces generated under different scenarios. We performed our experiments on a subset of the SPEC 2000 benchmarks under different scenarios. The count in each scenario is normalized with respect to the DIC in the original, unprotected application. The first scenario under consideration involves calculating the

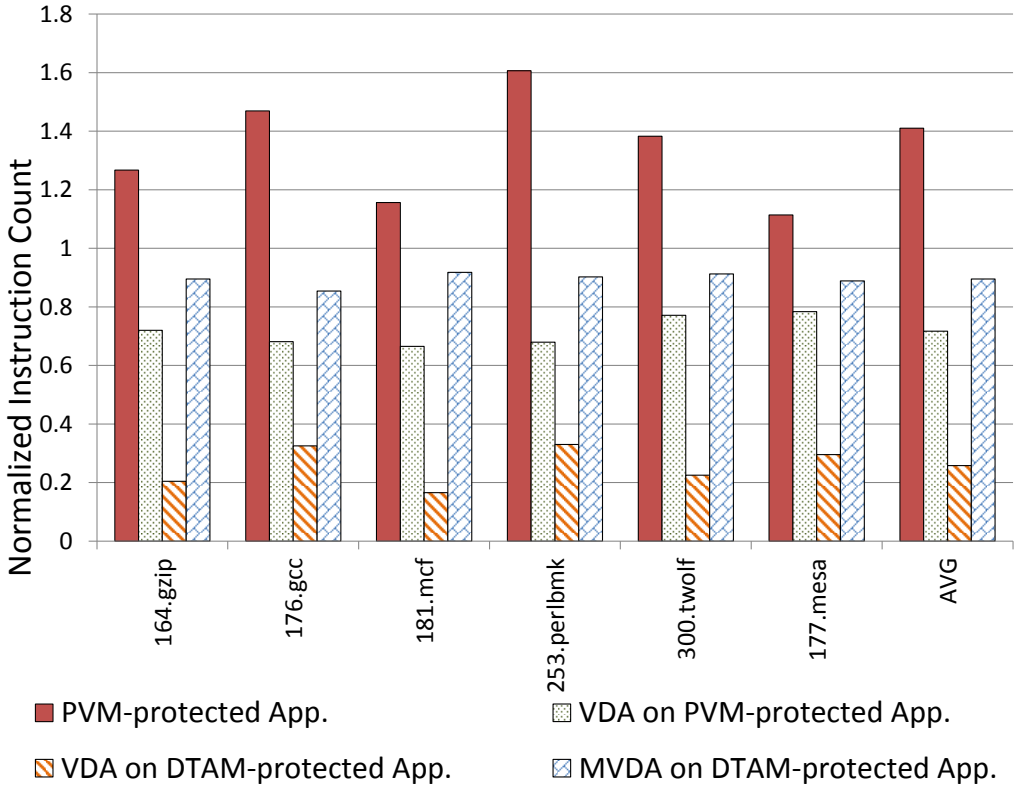


Figure 8.1: Dynamic instruction counts, normalized to the original application. The four scenarios include running the PVM-protected application, the count of the instructions obtained when VDA is applied to the PVM-protected application, the instruction count when VDA is applied to an application protected with dynamic address-mode transformation, and finally, MVDA applied to an application protected with dynamic address-mode translation.

DIC of a PVM-protected application. Next, we display the DIC obtained from applying VDA to the PVM-protected application. The third scenario consists of the DIC obtained from applying VDA to a DTAM-protected application. The last scenario involves obtaining the DIC from applying MVDA on an application protected via DTAM. Figure 8.1 illustrates the results.

From the perspective of the adversary, the optimal condition is represented by the scenario where VDA is applied to a trace of a standard PVM-protected application. As we have previously described, VDA only focuses on instructions that perform the core computation for the application. It ignores all superfluous instructions (such as address computations, and those instructions that belong to the PVM). Any technique that outputs information greater than this case has likely extracted some of

the superfluous instructions. Any technique that outputs information less than this case has likely ignored some of the instructions dealing with core computations. Therefore, both these cases are worse.

As the figure illustrates, on average, the PVM introduces 40% more instructions to the benchmark. Since applications have DIC in the order of billions, this overhead represents a significant increase on the part of the adversary. Analysis of second scenario reveals that VDA reduces DIC by more than half, compared to the first scenario. This result indicates that more than half of the dynamic instructions of the PVM-protected application consist of the PVM or address-calculation instructions. The remaining instructions form the core of the application (about 70% on average), and are of the most value to the adversary.

The third scenario reveals that DTAM removes most of the relevant instructions. On average, applying VDA on this protection scheme uncovers only 25% of the original instructions. As we demonstrated via an example in Section 8.2, most of the instructions obtained in this scenario consist only of the data transfers. The arithmetic and logical instructions that form the bulk of any complex application, are completely missed by VDA in the presence of DTAM protections.

Finally, MVDA is able to render DTAM protections ineffective, uncovering close to 90% of the original application. This figure is slightly higher than the result obtained from Coogan, *et al.*'s analysis (Scenario 2), which we assume to be optimum, but is still better than Scenario 3.

Although this change to VDA increases the amount of information to be processed, it makes VDA more effective at disabling all PVM technologies. To effectively defeat this modified version of VDA, a more robust solution is required. In the next section, we present such a solution that involves the protective PVM modifying data that belongs to the application.

8.4 DataMeld: Blending Data between the Application and the PVM

Now, we revert back to our original viewpoint, that of the software defender. The goal now is to design a solution to the robust version of VDA methodology introduced in Section 8.3. In this section, we present a revolutionary approach to virtualization that thwarts VDA attacks, by removing the separation between the application and the PVM data sets. The basic idea involves inserting the PVM data into the data flow of the application, without affecting functionality. In the presence of this obfuscation scheme, data-based analyses, such as VDA, will be unable to discern the individual components (the PVM and the original application). VDA will track instructions belonging to the PVM as well. Consequently, the effectiveness of VDA will be reduced, nullifying it as a search-space reduction technique.

We call this scheme *DataMeld*, because it obfuscates the application by blending its data flow with that of the PVM. To achieve optimum protection, all the data values of the protective PVM should affect the the application's data values. However, extracting this information from the PVM can be an onerous task. Thus, part of the research involves the creation of a tool that analyzes memory usage patterns of the PVM and outputs those variables that provide the maximum coverage in terms of the PVM code. The software defender can then proceed to blend the dataflow of the two components. We explain this process in greater detail in the next section.

DataMeld is achieved by applying arithmetic and logical operations on application values using PVM data as operands, and vice versa. However, these operations can not be applied arbitrarily. Care must be taken to ensure that this manipulation does not affect the outputs of the application. Thus, these data manipulations should have no effect over the course of the application lifetime. We achieve this effect by applying a series of operations on the data values of the application, and then, applying the inverse of these operations, using PVM data values as operands. This action occurs at different points in the run time of the PVM-protected application.

One such convenient point is when the PVM obtains control and translates the application. When

control is transferred to the PVM, the application data values can be extracted and modified. After the PVM completes translation, the values are reverted and control is transferred to the translated code.

The data values of the application can be accessed easily by the PVM. Prior to transferring control, the application's context is stored on the software stack. As both the application and the PVM share this stack, the PVM can scan the stack, and read the values. These values are subsequently modified using different PVM variables and stored back in their original locations on the stack. After the translation of the current block is completed, the PVM will revert the modifications and store the values in their original location on the stack. In this manner, dataflow of the application is transformed to include data from the PVM as well.

As we mentioned previously, these modifications should have no net semantic effect, so that the transformed application generates the same outputs as the original version, for all inputs. A knowledgeable adversary can attempt to locate these operations based on this premise. To make this scheme more robust, the modifying operations can utilize techniques, such as pointer aliasing [23, 8], and branch functions [81]. These techniques make it harder for the adversary to analyze the trace.

In essence, the effect of this scheme is to bind the data flow of the application with that of the PVM. To maximize this binding (*i.e.*, maximize obfuscation), the chosen PVM variables must be affected by as many PVM instructions as possible. We define the number of instructions that affect the value of a variable as *instruction coverage* for *variables* (ICV). This metric can serve as an indicator of the strength of the protection against VDA. Variables with a higher ICV value will generate more obfuscation comparable to variables with lower values. One of the contributions of our research is a tool that calculates the ICV values for the variables in a software application.

Figure 8.2 illustrates the workflow for protecting an application via DataMeld. The steps of the workflow are as follows:

1. The application is packaged with Strata, and subjected to the ICV tool, described in Section 8.5. This tool generates the ICV values for the memory references used by Strata. The top-ten memory references are chosen, according to decreasing ICV values.

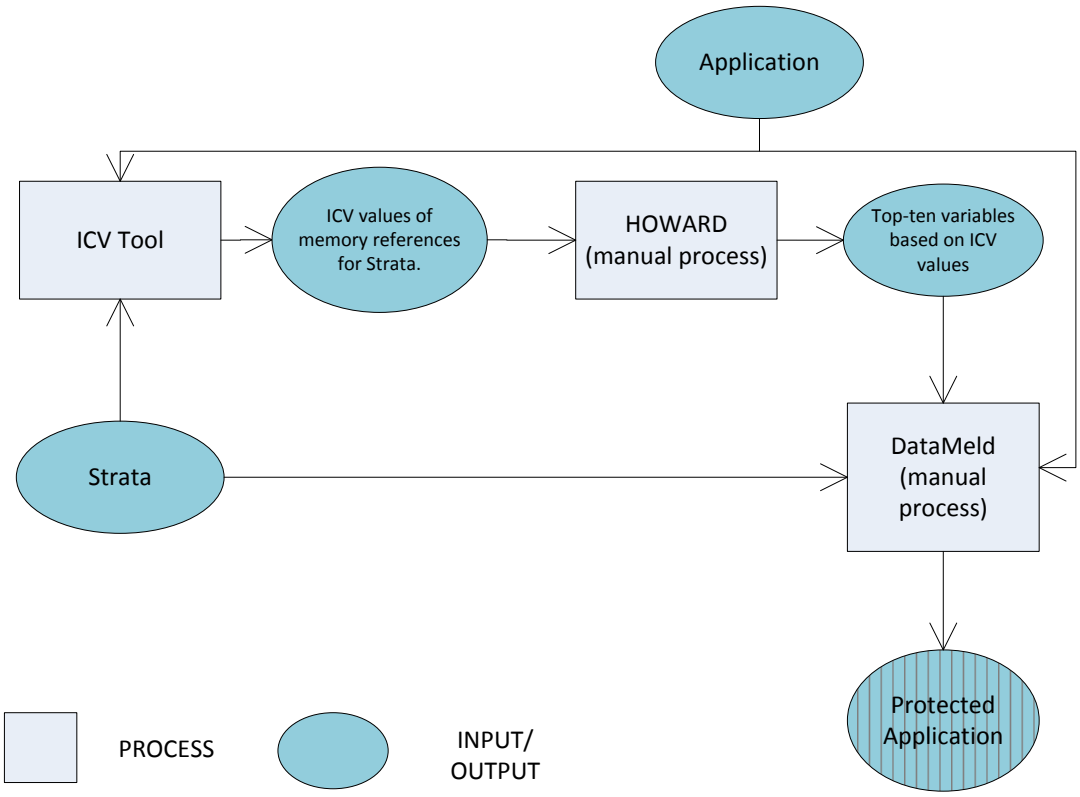


Figure 8.2: The workflow for the application protection process via DataMeld. First, the memory references with most instruction coverage are calculated. Subsequently, the variables in source code that correspond to these references are identified. Finally, DataMeld is implemented using Strata

2. Previously-derived heuristics (HOWARD) [113] are utilized to identify the variables that correspond to the memory references generated in Step 1. The heuristics facilitate identifying data structures from the source code. This step is performed manually. This part of the workflow is described in greater detail in Section 8.6.

3. The source code of Strata is manually modified such that at run time, Strata can extract and modify application values from the stack, using the variables identified in Step 2. Subsequently, the PVM-protected application package is created, as in previous cases.

We now proceed to explain the ICV calculation in detail.

8.5 A Tool to Calculate ICV

ICV calculation is based on dataflow analysis (DFA), which tracks the flow of data throughout execution, and has been used in software security techniques, such as taint analysis of software programs [114, 115, 116]. ICV basically computes the number of instructions that have affected the value of a particular variable. The goal is to obtain the ICV value for all the data variables in the PVM. The PVM variables with higher ICVs can then be used as candidates in the implementation of DataMeld.

Our tool has been designed to work on PVM-protected applications. We are only interested in the dataflow of the PVM, hence the analysis phase ignores data flow of the original application. Our tool calculates dynamic ICV for the PVM (*i.e.*, the number of dynamic instructions affecting the data variable).

The tool analyzes the run-time of the application. It operates on a per-function basis. On startup, each data memory location in the application is assigned with a ICV counter, initialized to zero. Whenever a memory write instruction is encountered, the ICV of the destination operand is updated based on the ICV counter of its source operands, plus the current instruction. The rules for calculating the updated value are described in Section 8.5.1. On most systems, general-purpose registers are used to perform arithmetic and transfer operations, as such, each general-purpose register is also assigned with a ICV counter. After the application exits, each memory reference is collated and ordered based on the counter value.

In the next section, we describe the rules that guide ICV calculation.

8.5.1 Rules Regulating ICV Calculation

This section describes the rules for calculating the ICV of application memory locations. These rules have been extended from the DFA rules proposed by Kemerlis et al. [117]. In the following discussion the term `dst` refers to the destination operand, whereas `src*` refers to the source operands.

The rules are based on the type of opcode of the instruction.

- ALU: These operations typically consist of 1 or 2 source operands writing to a destination operand. Examples of such instruction include `sub, add`, and `mul`. For such instructions, the ICV for the destination operand can be calculated as follows.

$$ICV_{dst} = ICV_{src1} + ICV_{src2} + 1$$

  All literal values have an ICV of zero.

- XFER: These operations consist of data transfers from the source to the destination. Both the source and the destination can be a register or a memory location. The rule for such instructions is as follows:

$$ICV_{dst} = ICV_{src} + 1$$

  As before, literal values have an ICV of zero.

- CLR: Examples of such instructions include `cpuid, setxx` *etc.* Zeroing out operations are also included in this category (*e.g.*, `xor eax, eax`. The ICV for such locations is:

$$ICV_{dst} = 1$$

- SPECIAL: This class contains instructions that cannot be handled appropriately by the above primitives. Examples of such instructions include `xchg, cmpxchg, lea` *etc.*

The tool ignores all other instructions, such as `FPU, MMX`.

Next, we apply these rules to a small code snippet, and calculate the ICV values for the variables in that example.

Listing 8.8: Example code to illustrate Instruction Coverage for Variables. Note that the implementation actually works at the binary level.

```
1  int datum1 = 4; //      ICV_datum1 = 1
```

```
2   int datum2 = 5; //      ICV_datum2 = 1
3   int datum3 = datum1 * datum2; // ICV_datum3 = ICV_datum1 + ICV_datum2 + 1
```

lstsetlanguage=C,numbers=none,escapeinside=@@

Listing 8.8 displays the snippet. Initially, the ICV for the variables `datum1`, `datum2`, and `datum3` is zero. In Line 1, `datum1` is written, therefore, its ICV is set to 1. Similarly, in Line 2, the ICV for `datum2` is set to 1. Finally in Line 3, the ICV for `datum3` is set to the ICV of its operands plus the current instruction,(*i.e.*, 3.

We have created a tool that calculates ICVs based on these rules, using Intel's Pin instrumentation framework. This pintool analyzes PVM-protected applications, and calculates ICV values based on the above rules. For our prototype, we used Strata as the protective PVM. Since this tool only calculates ICV for variables in Strata, a technique is required to trigger the pintool when Strata obtains control (to continue analysis), and when it relinquishes control (to pause analysis). We achieve this trigger by instrumenting the `strata_build_main` (which indicates control has been transferred to Strata), and the `targ_exec` (which indicates that control has been transferred to the translated block) functions. Whenever a write occurs, the ICV of the destination operand is updated by applying one of the rules to the ICV values of the source operands. The tool maintains the ICV values in a map data structure, indexed by memory address. At the end of execution, this map contains the ICV values of all the memory addresses accessed by the PVM.

8.6 Mapping Memory References to Variables in Source Code

The pintool generates ICV values for memory references of Strata. We need a technique to ascertain the variables in source code that correspond to these references, as the DataMeld is implemented at the source code level. This information can be extracted from heuristics that recover data structures from binary code. Extensive research has already been performed in this area, such as the CodeSurfer project designed by Balakrishnan and Reps [118, 119, 120]. Others research of note include Laika [121], and Rewards [122]. In light of the established body of research, we proceeded with reusing some of these techniques for our goals.

For our purposes, we adopted the techniques for dynamic data structure excavation from the *Howard* system [123, 113]. Howard has been designed to extract data structures from generic binaries. This information is extracted by running the application, and analyzing the data usage pattern. Since we are only interested in applying these techniques to protective PVMs like Strata, we simplified them accordingly. For our initial study, we have restricted our analysis to local function variables.

During execution, whenever a `call` instruction is encountered, it signifies a new function is to be analyzed. A `ret` instruction signifies the end of analysis for the current function. The frame pointer (`ebp`), or the stack pointer (`esp`) is typically used in operations on local variables. Local variables and arguments are accessed via positive or negative offsets off of these registers.

While processing a function, if an operation that is relevant to ICV calculation occurs, and one of the operands is a memory location accessed via the stack or frame pointer, its offset is recorded, along with the ICV calculations. When the entire analysis terminates, we collate this per-function data and attempt to match the offsets with the variables from source code manually. Although this method is not precise, we were able to obtain sufficient information to facilitate implementation of DataMeld.

In the next section, we describe the creation of the DataMeld system.

8.7 Implementing DataMeld

Once the variables are recognized, Strata is modified to extract and the applications values on the stack. The main code modifications occur in the builder function of Strata, `strata_build_main`.

The prototype of this function is as follows:

Listing 8.9: Function prototype of strata_build_main

```
fcache_iaddr_t strata_build_main (app_iaddr_t to_PC, strata_fragment *from_frag)
```

This function is invoked when a new application block has to be translated. Prior to invoking this call, the application's context (*i.e.* all the register values, as well as status flags) are pushed on

to the program stack. Once this function is invoked, these values can be accessed via its function arguments. An example code snippet that performs this operation is as follows:

Listing 8.10: Strata code snippet to access the locations where application values are stored

```
eax_address = (int *)( ((char *)&from_frag) + 36);
ecx_address = (int *)( ((char *)&from_frag) + 32);
edx_address = (int *)( ((char *)&from_frag) + 28);
ebx_address = (int *)( ((char *)&from_frag) + 24);
```

In Listing 8.10, the variable `eax_address` contains the address of the stack location containing the `eax` values of the application prior to context switch. Similarly, `ebx_address` contains the address of the location storing `ebx`' values, and so on.

Once these locations are extracted, code can be added in `strata_build_main` to modify these values. An example follows:

Listing 8.11: Strata code modifying the applications values

```
*(eax_address) += frag->fPC;
```

Listing 8.11 illustrates the modification of the application's `eax` values by a variable of Strata, `frag->fPC`.

Prior to `strata_build_main` returning control to the newly translated block, the above mentioned change must be reverted. This modification is illustrated in Listing 8.12.

Listing 8.12: Strata code reverting the applications values

```
*(eax_address) -= frag->fPC;
```

In this manner, the application's values can be tainted by the PVM values. These examples perform simple calculations, but any complex calculations can be performed, as long as the values on the stack prior to `strata_build_main` returning control are identical to the values at the point of invocation of this function. Applying VDA to such a protected application will lead to a large portion of the PVM appearing in the results, thwarting analysis. In the next section, we present some of the results of our evaluation.

Rank	File name	Function name	Variable name
1	targ-build.c	targ_classify	insn
2	targ-build.c	targ_classify	opcode
3	targ-build.c	targ_normal	insn
4	targ-build.c	targ_normal	frag
5	targ-build.c	targ_normal	class
6	targ-build.c	targ_pcrel_branch	insn
7	build.c	strata_build_main	frag
8	targ-build.c	targ_create_trampoline	tramplist
9	build.c	strata_build_main	to_PC
10	build.c	strata_create_trampoline	patch

Table 8.1: Table displaying the top-10 variables in terms of ICV values. These variables are used for creating the DataMeld system.

8.8 Evaluation

We successfully prototyped the DataMeld system, using Strata, and performed analysis on its protection properties. We summarize our efforts in this section.

8.8.1 ICV Tool Output

As before, we utilized Strata as the prototype for the protective PVM. Initially, we ran the ICV tool on a simple PVM-protected application, and obtained 10 variables that had the highest ICV counts. We then manually modified the Strata source code, such that when Strata was in control, it would extract the application values from the stack and modify its contents, using these variables as operands.

Table 8.1 displays the ten variables with the highest values of ICV. For the prototype, we used simple arithmetic operations to modify the data values of the application, using these variables. The operations that are performed on the application variables are not static, but selected at random. Care must be taken to ensure that these operations are reverted when control returns to the translated code.

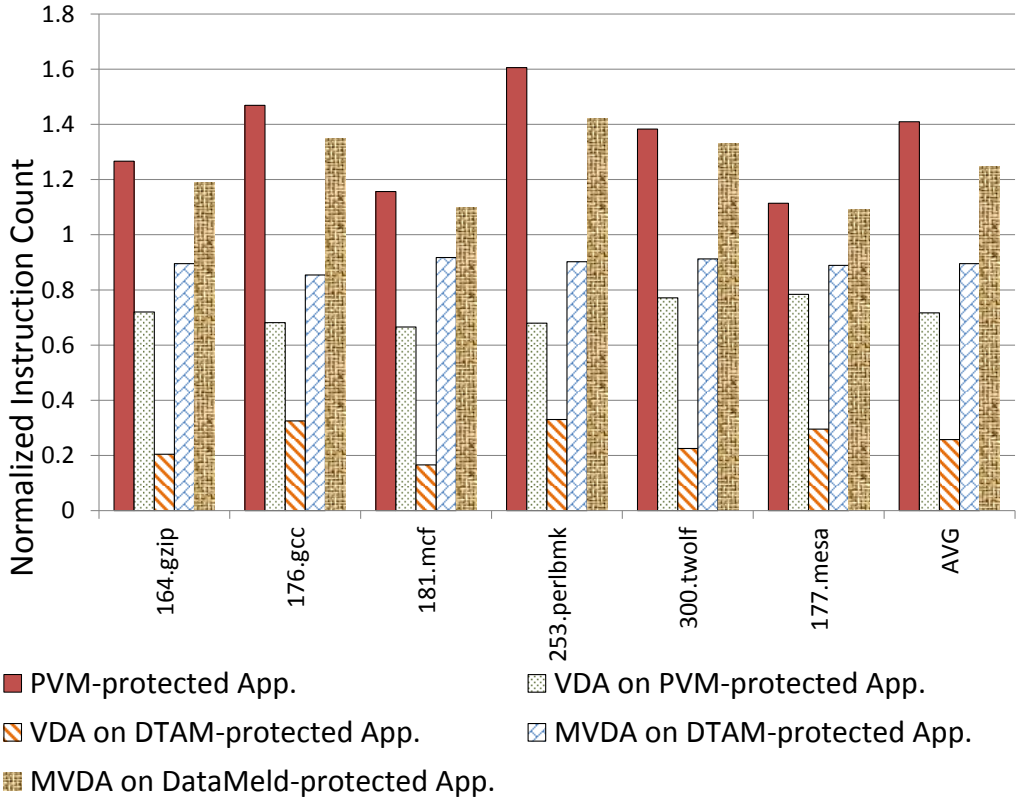


Figure 8.3: Dynamic instruction counts, normalized to the original application across all the five scenarios. To recap, these scenarios include running the PVM-protected application as is, the DIC obtained when VDA is applied to the PVM-protected application, the DIC when VDA is applied to an application protected with DTAM, the DIC obtained when MVDA is applied to an application protected with DTAM, and finally, the DIC when MVDA is applied to an application protected via DataMeld.

8.8.2 Measuring Obfuscation

To measure obfuscation caused by DataMeld, we added an additional scenario to the analysis performed in Section 8.3.1. This additional scenario comprises of calculating the DIC obtained by applying the MVDA analysis on a DataMeld-protected application. Figure 8.3 displays the DIC for a few benchmarks, together with the DICs obtained from the previous four scenarios described in Section 8.3.1.

As the figure illustrates, DataMeld increases the amount of superfluous instructions obtained by MVDA, thereby thwarting the attack. On average, the DIC for DataMeld-protected applications is 25% more than that of an unprotected application. Recalling Section 8.3.1, the optimum scenario

for the adversary comprises of applying VDA to a PVM-protected application, which resulted in a DIC that was 30% less than that of an unprotected application. As such, our results indicate that DataMeld does increase the amount of obfuscation in the trace. These experiments on DataMeld were performed by using the top ten variables in terms of ICV values. By increasing the number of values, DataMeld can provide more obfuscation. Each additional variable will likely result in VDA generating more instructions (*i.e.*, those instructions that interact with the additional variable).

These investigations reveal that DataMeld can improve the robustness of the protective PVM, against dataflow-based attacks like VDA. Employing this scheme removes another weakness that could potentially be exploited by an adversary, preventing them from removing protections and acquiring the application's assets in an unauthorized manner.

8.9 Summary

In this chapter, we presented our investigation into the flow of data between the application and the protective PVM. We began the discussion by describing the work of Coogan, *et al.*, in designing an attack on PVM-protected applications by obtaining the trace and performing dataflow analysis, called Value-based Dependence Analysis (VDA). We modeled this attack methodology within our framework. We then proceeded to extend the attack to binary translation systems. The original version of VDA could be defeated by software dynamic translators, but we demonstrated a modified version of VDA that is effective at disabling any PVM technology. Finally, we designed a robust solution, DataMeld, that offers protection by conflating the dataflow of the PVM with that of the application. Our results show that DataMeld is effective at thwarting dataflow-based attacks on PVM-protected applications.

Chapter 9

Composable Virtual Machines

Upto this point in the dissertation, this dissertation has exclusively focused on a single layer of virtualization to combat reverse engineering and tamper. The PVM makes it hard to statically analyze the application. At run time, the PVM applies various dynamic protection techniques to the application, such as temporal polymorphism, and knots. Point-ISA and DataMeld ensure that the adversary is not able to remove the protections. All these techniques thwart the adversary from targeting the attack surface (*i.e*, the application), and successfully acquiring the assets of the application.

One point of concern is that the PVM itself is not as well protected. The PVM is not as attractive a target as the application for the adversary, but there is still potential for an attack that could disable the protective properties of the PVM and leave the application vulnerable. The major weakness for the PVM is its stationary target surface. As opposed to the application, the PVM executes in a predictable manner. If an adversary were able to locate a vulnerability, they could launch an automated attack on the PVM.

A straightforward solution to this problem involves nesting virtualization layers to impart all the dynamic protections to the PVMs themselves. The concept of nesting virtualization layers was first proposed by Popek and Goldberg [44]. In essence, the adversary would have to break through a dynamically-protected PVM layer before even attempting to attack the application.

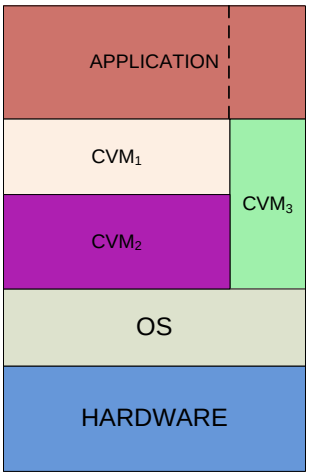


Figure 9.1: A conceptual overview of a CVM-protected application package. The application is partitioned (based on certain criteria) into two sections. The first section is protected by a set of two CVMs, while the second section is protected by one CVM instance.

In this chapter, we expand on the idea of nesting virtualization layers, by proposing the concept of *composable virtualization*. We define composable virtualization as applying a set of virtual machines (which could be null) to different partitions of an application, for program protection. Since the virtual machines themselves are software applications, they too can be subjected to composable virtualization. These virtualization layers are termed as Composable Virtual Machines (CVMs). The basic steps involved in this scheme comprise of partitioning the application, and assigning each partition to a set of CVMs. These partitions are then composed together, to create a executable package that represents the protected application.

Figure 9.1 illustrates the conceptual overview of an application protected by CVMs.

The application is partitioned into two sections. The first partition is protected by a set of two CVMs (CVM_1 and CVM_2). The set is arranged in such a way that CVM_2 protects CVM_1, which in turn protects the first partition. The second partition is protected by a single CVM (CVM_3).

The goal of this chapter is to introduce research on composable virtual machines. An extensive analysis and evaluation of multiple virtualization layers is beyond the scope of this dissertation. Rather, we will be analyzing some of the logistics involved in composing applications using CVMs, and their properties at run time. Our hope is that this study will facilitate further analysis and

research into composable virtualization, and pave the way for their widespread adoption.

The main contributions of this chapter are as follows:

- Introduction of a novel protection technique, called *composable virtualization*. In this scheme, the application is partitioned, and assigned to a set of virtual machines that protect it and themselves as run time.
- The use of a case study to investigate some of the protection properties of this technique. For example, we demonstrate that CVMs can better protect software caches, compared to the use of a single PVM instance. The goal of the case study is to provide an initial insight into composable virtualization, and foster future research in this area.
- The design of an optimization technique to alleviate some of the performance overheads associated with multiple layers of virtualization. This optimization is targeted towards software dynamic translation, and can be used to reduce overheads significantly.

The rest of the chapter is organized as : Section 9.1 discusses past work on program partitioning for security. It is our belief that such past work can be co-opted for use in the CVM methodology as well. Section 9.2 describes the experimental set up for creating CVM-protected packages. Section 9.3 discusses the use of a case study to investigate some of the properties of CVMs. In Section 9.4, we investigate the sources of overhead in CVMs, and proposes optimizations. In Section 9.5, we discuss some of the issues related to CVMs, and propose some specific areas for further research. Finally, we summarize this chapter in Section 9.6.

9.1 Partitioning Applications

The high-level design philosophy for creating a CVM-protected application consists of partitioning an application, and assigning each partition to a set of CVMs, (or scheduling the partition to run natively). At run time, each partition will run under mediation of its assigned set of CVMs. The CVMs will dynamically protect the currently-executing partition, as well as each other, creating a network of protection.

Partitioning an application for security purposes has been investigated in the past [124, 125, 126, 127]. Primarily, such techniques have facilitated a distributed computing paradigm, in which different parts of the application execute on various hosts, each with a different level of trust. Zdancewic, *et al.* devised language-based scheme for protecting confidential code during computation in a distributed network containing untrusted hosts [124]. Narayanan, *et al.* designed a compiler-guided technique for secure code partitioning among a set of hosts. Their scheme targeted hosts that execute secure embedded application in parallel [125]. Sondergaard, *et al.* also devised techniques to annotate programs with annotations with protection specifications, and to partition the annotated programs. In most of these works, the exact mechanism to partition the application has been left to the discretion of the software defender. These solutions typically enable the defender to mark sections of the code as critical.

Having investigated the plethora of research work, we have opted against devising a partitioning scheme of our own. We feel that the partitioning scheme is highly dependent on the needs of the protection configuration. For example, it might be tolerable, from the perspective of the software defender, to run certains sections of the application natively and without protections. The more relevant issue is to compose the protected package, once a partitioning scheme has been decided. In the next section, we describe techniques to compose a CVM-protected application.

9.2 Creating CVM Packages

The CVM infrastructure is based on software dynamic translation (Strata) [51], which we described in Section 2.3. To create applications protected via CVMs, we utilize the *Diablo* link-time toolchain [62]. The application is first compiled using the standard `gcc` compiler. The SDT library (*i.e.*, Strata) is also compiled in a similar manner.

Each CVM instance is created from the Strata library. To create the multiple instances, we employed the `objcopy` utility from the `binutils` suite of tools. `objcopy` copies the contents of one object file (including library and executable files) to another, while providing options to modify the output in several ways. One of the options involves modifying the names of all the global

symbols in the output file by adding a prefix. This option can be used to ensure that the different CVM instances do not share any global symbol names, thus removing any name conflicts. Utilizing `objcopy` provides a simple technique to create several CVM instances, without performing extensive source code changes. It is important to note that creating CVM instances with different protection properties, some amount of code modification may be required. In our subsequent discussion, we will assume that different CVM instances possess identical protection properties.

Strata has three functions that are of relevance. The `strata_init` function initializes the Strata library. The `strata_start` function prepares for application virtualization. Finally, the `strata_exit` function deallocates any resources that Strata might be using, and relinquishes control completely.

Once all the object files and CVM instances are created, they are provided as input to Diablo. As we described in Chapter 4, Diablo processes the inputs and creates the CFG for the application. At this point, Diablo can be programmed to insert calls to `start_init`, `strata_start` and `strata_exit` for each CVM instance at appropriate locations in the CFG corresponding to each partition. If a CVM instance is itself scheduled for partition, Diablo can be programmed to insert the calls to the CVM accordingly.

Consequently, Diablo generates machine code for this modified CFG and writes it to an executable file. Various static protection can also be applied to the executable at this stage. This file represents the CVM-protected application binary. At run time, when a call is made to `strata_start`, the corresponding CVM instance starts mediating the application. When a call is made to `strata_exit`, the corresponding CVM instance releases resources, and yields control permanently.

This technique provides a platform to the software defender to craft different configuration of CVMs. In the next section, we will focus on some of their protection properties. We will base our discussion of CVM protection on a case study, comprising of two CVM configurations.

9.3 A Use-case Study

The protection offered by CVMs depends, to a large extent, on the partitioning scheme. For example, the defender may choose to partition the application such that only one function runs under a CVM.

In such a case, CVM-enabled protections will only be active when that particular function is invoked.

To obtain insight into the protection offered by CVMs, we performed a case study consisting of two simple CVM configurations, called *series* and *nested*. Each configuration consists of an unprotected application, P_{APP}, and two CVM instances, Strata_1 , and Strata_2. In the series configuration, the two CVM instances are in series of each other *i.e.*, at any point in time, only one CVM instance is active. In the nested configuration, both CVM instances are active simultaneously.

9.3.1 Series Configuration

Figure 9.2 illustrates the creation and the run time of a package protected by the series configuration of CVMs. During software preparation, Diablo generates the CFG of the application. The set of blocks representing the `main` function of the application is partitioned equally. Diablo then inserts the entry and exit calls to Strata_1 and Strata_2 to the two partitions, as shown in the figure.

On program startup, Strata_1 assumes control, and start mediating the execution of its partition. It applies various dynamic schemes to protect the run time from attacks. When the corresponding `strata_exit` is invoked, Strata_1 deallocates its resources, and yields control. Consequently, Strata_2 assumes control, and mediates the execution of its partition. It too, applies various dynamic schemes to the run time. The application then runs to completion.

We analyzed this configuration, in terms of security and performance. The performance overhead is similar to the case of an application running under a single instance of Strata. There is some overhead involved for the startup and shutdown associated with every CVM instance, but for most of the benchmarks, it was negligible.

The protection offered by this configuration is incremental, compared to the single PVM scenario. The dynamic protection at any point is dependent on the CVM instance in control. Over the entire application run, the protection applied is the average of the two CVM instances.

As the protections offered by the series configuration is incremental over a single PVM instance, the rest of this chapter deals exclusively with the nested configuration of CVMs.

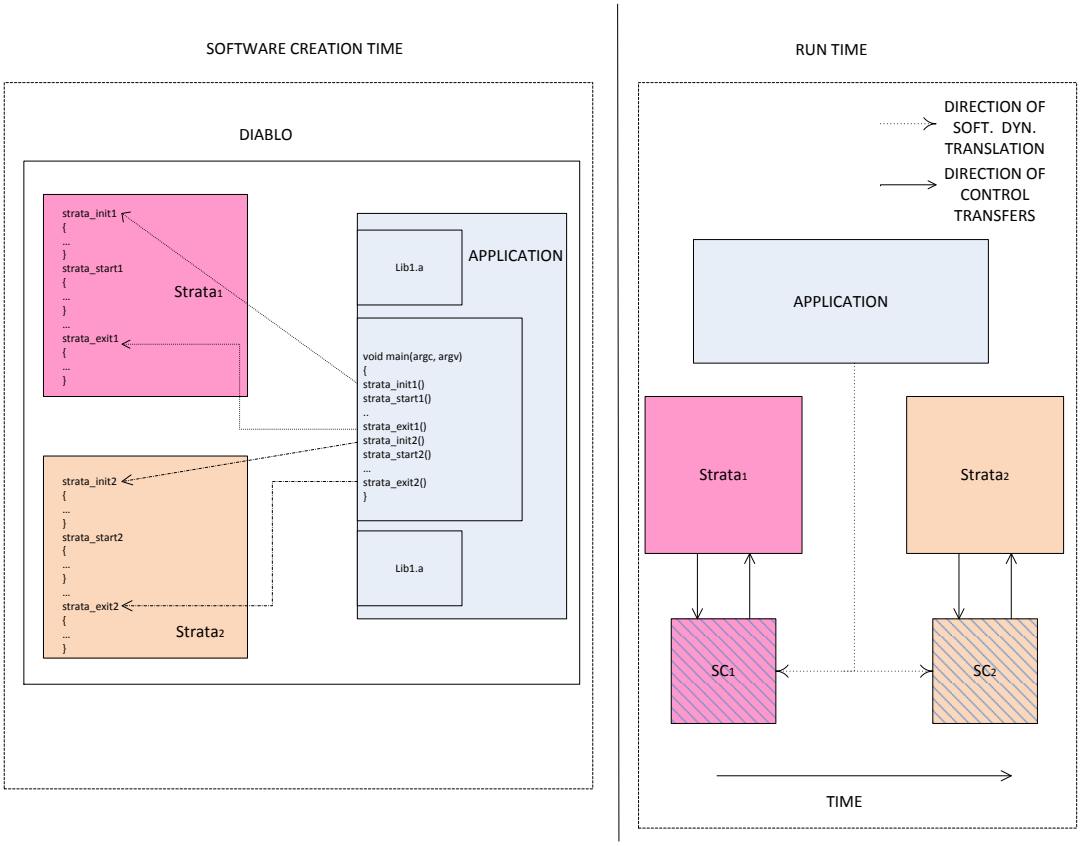


Figure 9.2: A high-level overview of the series configuration. During software creation, Diablo synthesizes the CFG of the application and inserts the entry and exit functions of the CVM instances, Strata_1 and Strata_2, by dividing the `main` function equally. At run time, first Strata_1 translates its partition, followed by Strata_2.

9.3.2 Nested Configuration

Figure 9.3 illustrates the creation, and the run time of the nested configuration. As before, during software preparation, Diablo generates the CFG of the application. In this case, all the blocks comprising `main` are encapsulated by the start and exit functions of Strata_2. Consequently, Strata_2 is encapsulated by Strata_1, ensuring that Strata_2 executes under the control of Strata_1.

At run time, Strata_1 assumes control. It proceeds to translate code from Strata_2 to its software cache, SC_1. Then, control is transferred to the translated code in SC_1, which in turn, starts translating the application's code to Strata_2's software cache (SC_2). As control is about to be transferred to SC_1, Strata_1 captures control, and starts translating code from SC_2, to its own cache SC_1. Thus, all the instructions that execute natively belong to Strata_1 or reside in its software cache, SC_1. This

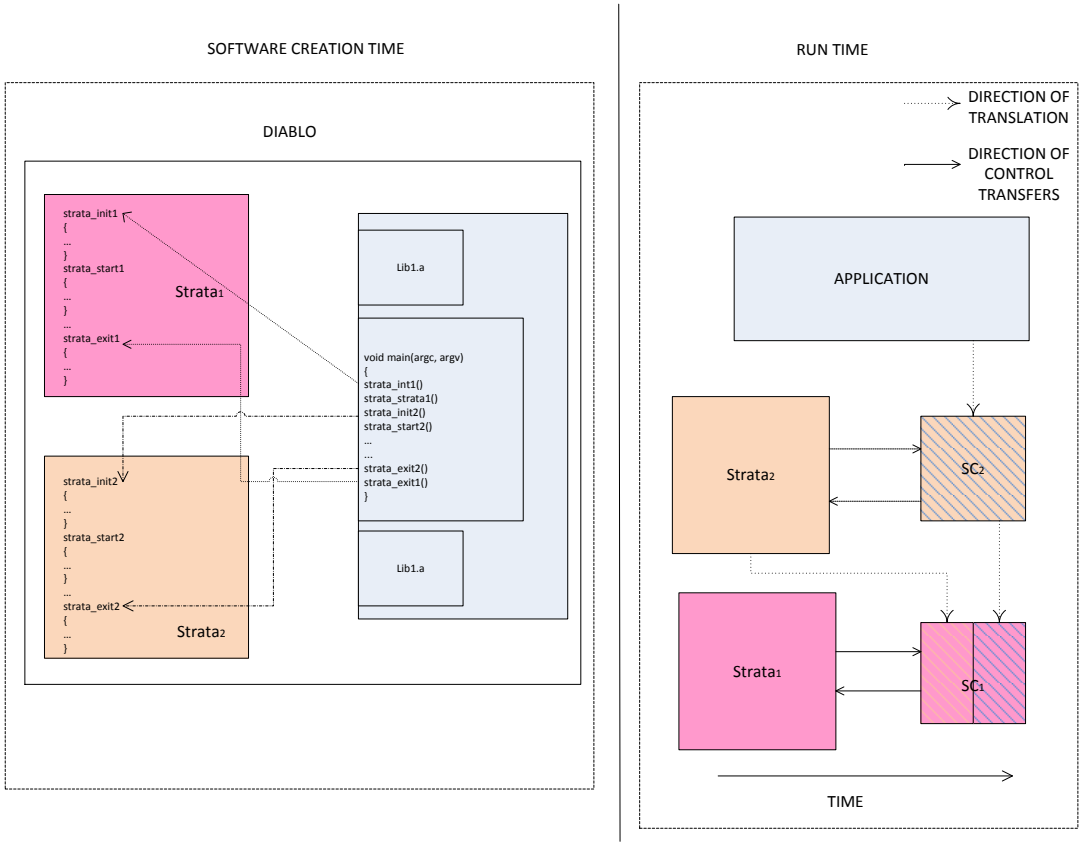


Figure 9.3: A high-level overview of the nested configuration. During software creation, Diablo synthesizes the CFG of the application and inserts the entry and exit functions of the CVM instances, Strata_1 and Strata_2. In this case, Strata_2 is encapsulated within Strata_1. At run time, Strata_2 translates the application to its software cache, SC_2. Strata_1 translates Strata_1 and SC_2 onto SC_1. Since the code resident in SC_2 does not execute directly, it can be encrypted.

feature is important to note as it indicates that the code residing in the software caches of inner[1] CVMs (such as Strata_2) do not execute directly. Therefore, they can be transformed in a manner that thwarts analysis (*e.g.*, encryption).

Next, we focus on some of the properties of the nested configuration that protect against reverse engineering.

[1]In this discussion, positions of the virtualization layer are described relative to the application. The application is located at level 0, the innermost CVM layer at level 1, and so on. Any CVM that is interpreted directly by the hardware is said to be at the outermost level.

Obfuscation of the Software Cache

One of the major goals of composable virtualization involves increasing the obfuscation of software cache containing translated code. To evaluate this feature, we analyzed the software caches of the CVMs.

As we mentioned, code resident in the software cache of any CVM that is not directly in contact with the native platform, will be translated by another CVM. As a means of thwarting analysis, the contents of such an inner-level software cache can be encrypted. In such a case, the decryption key must be possessed by any CVM that is translating this code. In the current example, Strata_2 responsible for mediating Strata_2, and must possess the corresponding decryption key. This feature can be extended to multiple layers of virtualization, as long as the decryption keys are shared appropriately. Thus, if a CVM at level n controls the execution of a CVM at level $n-1$, it must posses the decryption key for the encrypted code residing in the software cache of the CVM at level $n-1$.

Care must be taken to ensure the encryption algorithm is robust, and the key is not easy to extract. The logistics of encryption and key protection are similar to the case of encrypting the application, which was discussed in Section 4.1.2. that discussion applies in this scenario as well.

We now focus on the software cache of the CVM at the outermost level, *i.e.*, the CVM that executes on the native platform. In our example, this CVM is represented by Strata_1. Figure 9.4 illustrates its software cache. As the figure shows, there is interleaving between the code from these two components. There is no clear demarcation between the code of Strata_2, and the application.

As discussed in Section 4.2, interleaving code from the application and the VM enables greater entropy, and leads to greater obfuscation. It is more difficult for the adversary to distinguish the application code from the CVM code. We decided to reuse the experiments of Section 4.2 on the contents of this software cache, to obtain an insight on the entropy of the system. Each basic block that originated from the application was assigned a bit value of '0', and each basic block from Strata_2 was assigned a value of '1'. A string was then obtained based on the layout of the software cache, and compressed using the `LZ78` algorithm.

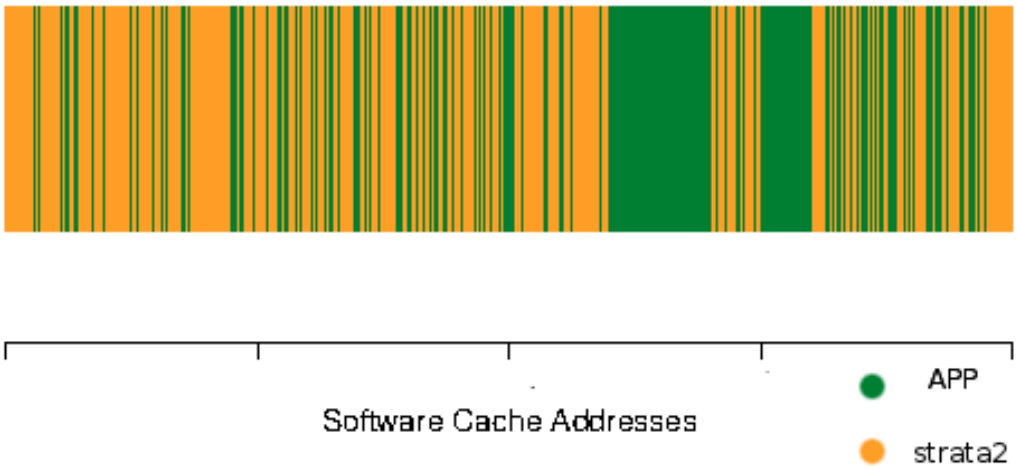


Figure 9.4: This figure illustrates the software cache of Strata_1. This software cache contains the code from Strata_2, and its software cache SC_2, which is basically translated application code. As can be seen from the figure, the code from the components are interleaved, to provide better obfuscation.

In the case of an application protected by a single PVM, the compression ratio was 149. In the case of the nested configuration, the compression ratio was observed to be 15.63. These results indicated indicate that the nested CVM configuration provides higher entropy to the software cache. Therefore, the nested configuration provides more protection from analysis to the software cache.

This result is encouraging, from the viewpoint of the software defender. With more complex CVM configurations, the software caches will possess more entropy, making it harder for the adversary to extract meaningful information.

Obfuscation of the Control Flow Graph

The previous experiment indicated that the low-level information possesses higher randomness in the presence of CVMs. In this section, we demonstrate that the high-level information (*i.e.*, the CFG) is also more obfuscated, further thwarting the adversary from successfully reverse-engineering the application. As discussed previously, the CFG is a useful tool for the adversary to comprehend the functionality of the application.

To validate our claim of increased CFG complexity, we reviewed past work and identified a metric, Cyclomatic Complexity, for measuring complexity of program graphs. It should be noted at the outset that metrics in security can be subjective, and provide only a limited scope for evaluating the absolute effectiveness of obfuscations.

Cyclomatic Complexity (CC) was designed by Thomas McCabe, and is used to indicate the complexity of graphs. It measures the number of linearly independent paths through an application's code [128]. In the context of reverse engineering, a higher value of CC implies that there are more program paths that need to be analyzed. Consequently, more effort is required from the adversary. CC has been used previously in the field of program obfuscation [129, 130]. McCabe, *et al.* defined the Cyclomatic Number (CN), for a undirected graph G, as:

$$CN(G) = e(G) - n(G) + 2 * p(G) \tag{9.1}$$

where $e(G)$ represents the number of edges of the graph, $n(G)$ denotes the number of nodes in the graph, and $p(G)$ denotes the number of exit nodes in the graph [128].

Our experiment consist of comparing the cyclomatic complexities of the CFGs obtained from the software caches in two different configurations; the application running under a single PVM, and the application running under the nested CVM configuration (*i.e.*, the application running under Strata_2, which runs under Strata_1). In the nested configuration, we only consider the software cache of Strata_1.

The CFG is built from the executable code located in the software cache. Whenever control exits the software cache and enters the SDT library, processing of the CFG stops. When control returns to the software cache, a new CFG component is started. So the dynamic CFG consist of several disconnected components representing the different translated blocks. In the case of the nested configuration, only the the software cache of Strata_1 (SC_1) is considered. This software cache contains code translated from the inner CVM (Strata_2), as well as code originating from the application.

We performed the analysis for CC on the benchmarks of the SPEC CPU 2000 suite. The values for $e(G)$, $n(G)$, and $p(G)$ were obtained by running the two configuration under the Pin instrumentation

Benchmark	CC for Single PVM	CC for Nested CVMs	Percent Increase from Original
176.gcc	1604	80109	4894.77
181.mcf	351	9828	2701.65
256.perlbmk	803	32903	3997.51
179.art	181	5130	2734.25

Table 9.1: Cyclomatic Complexity of the dynamic CFGs obtained from the software caches, when run under a single PVM, and a nested CVM configuration comprising of two CVMs. The edges and node (basic blocks) were calculated using the Pin instrumentation framework. The CFG corresponds to the executable code residing in the software software cache. In the case of the nested CVMs, the CFG corresponds to the code in the cache of Strata_2. The benchmarks were run under the test mode to expedite the study.

framework. Table 9.1 displays the cyclomatic complexity for some of the benchmarks, under the two scenarios.

As the table illustrates, the nested CVM configuration has more independent paths in its CFG, compared to the case where only a single PVM is used. On closer examination of Equation 9.1, we observed that the increase in CC in the nested configuration was mainly triggered by the increase in the number of exit nodes (denoted by $p(G)$ in Equation 9.1, which has twice the weigth of the other parameters). This higher value of $p(G)$ implies that there are more disconnected components in the software cache of the outer CVM (Strata_1). An adversary would have to collate all these extra components to successfully in an effort to reverse engineer the application.

This experiment indicates that CVMs have the potential of increasing the obfuscation of protected applications. Simple nesting of CVMs yields CFGs that are significantly more complex. With more elaborate partitioning and utilization of numerous CVMs, foundations can be laid for robust program protection. An adversary will have to expend significantly more effort than current state-of-the-art, to successfully obtain relevant information.

As previously mentioned, the increased complexity of the CFG in the nested configuration is reflected in the run-time performance. In the next section, we investigate this overhead of nested CVM, and suggest optimizations that can improve performance.

9.4 Performance Overhead of Nested CVMs

The previous section provided some insight into the protection properties of CVMs. However, to enable practicality, the overhead of CVMs must be tolerable. In this section, we discuss the performance implications of nested CVMs, and suggest techniques to alleviate the overhead.

9.4.1 Experimental Setup

The performance evaluation was performed on the SPEC CPU 2000 benchmarks. The CVM instances were created from the Strata, as described in Section 9.2. The experiments were carried out on a 32-bit AMD Athlon processor, running Ubuntu 12.04. The code was compiled using the `gcc-2.95` compiler, and the protected package was created using Diablo. The older version of `gcc` was used, as newer versions are not compatible with Diablo [62].

9.4.2 Self-modifying Code

The performance overhead was observed to be 35X over native execution, on average. This high overhead is due to the *self-modifying* aspect of software dynamic translation. As we have described previously, the SDT translates and caches instructions from the application one at a time till a control transfer instruction is encountered. If the target block (also called a *successor* block) of the transfer instruction has previously been translated and cached, the SDT will append a direct transfer instruction to that block. If the target is absent, the SDT will append a sequence of instructions that transfer control back to itself so that it can translate the code located at the target address. This sequence of instructions is known as a *trampoline*. Each trampoline is associated with a target address.

When the target application block actually appears in the software cache the corresponding trampoline is rewritten with a `jump` instruction, that transfers control directly to the translated target block, without invoking the SDT again. This instruction rewriting is known as *patching*, and is an example of self-modifying code [54].

Currently, SDTs handle self-modifying code by flushing the entire software cache, and continuing translation at the next application block scheduled to be executed. In our example of nested CVMs (which are based on software dynamic translation), Strata_1 obtains control at start up, and begins translating Strata_2's code to its software cache, SC_1. Consequently, control is transferred to the translated code corresponding to Strata_2. This code translates the application to the software cache, SC_2, and appends a trampoline to those blocks whose target successors have not been translated. Consequently, Strata_1 again obtains control and copies the instructions from SC_2 to SC_1.

When the translated code for Strata_2 patches a trampoline in its software cache (SC_2), it causes Strata_1 to flush SC_1 entirely. After the flush, Strata_1 must translate a significant portion of Strata_2 (such as the initialization parts), before any code corresponding to the application can be translated and executed. Therefore, this patching of trampolines in the software cache of Strata_2 leads to excessive overheads.

9.4.3 Super-patching of Trampolines

The generic mechanism to handle self-modifying code in SDTs (*i.e*, flush the entire software cache) is expensive and leads to high overheads in nested CVMs. Since the software cache flushes are exclusively caused by the patching of trampolines, we devised a novel mechanism, called *super-patching*, that alleviates some of the performance overheads associated with nested CVMs.

In Section 9.4.2, we described how the creation of a patch in the software cache of Strata_2, causes the entire software cache of Strata_1 to be flushed. Instead of flushing, a simple optimization involves propagating the patch in the software cache of Strata_2 into the software cache of Strata_1. When any trampoline is being patched to its target block (TB) in SC_2, Strata_2 sends that information about the patch and the TB to Strata_1. Strata_1 no longer flushes the cache but stores this information for later processing. When the TB is translated from SC_1 to SC_2, Strata_1 artificially patches the *translated* trampoline to the translated TB in its software cache. This mechanism is called super-patching, because the patching in SC_1 triggers this patch in SC_2. With this mechanism, the extraneous flush has been removed. After implementing this technique, we observed significant reduction in

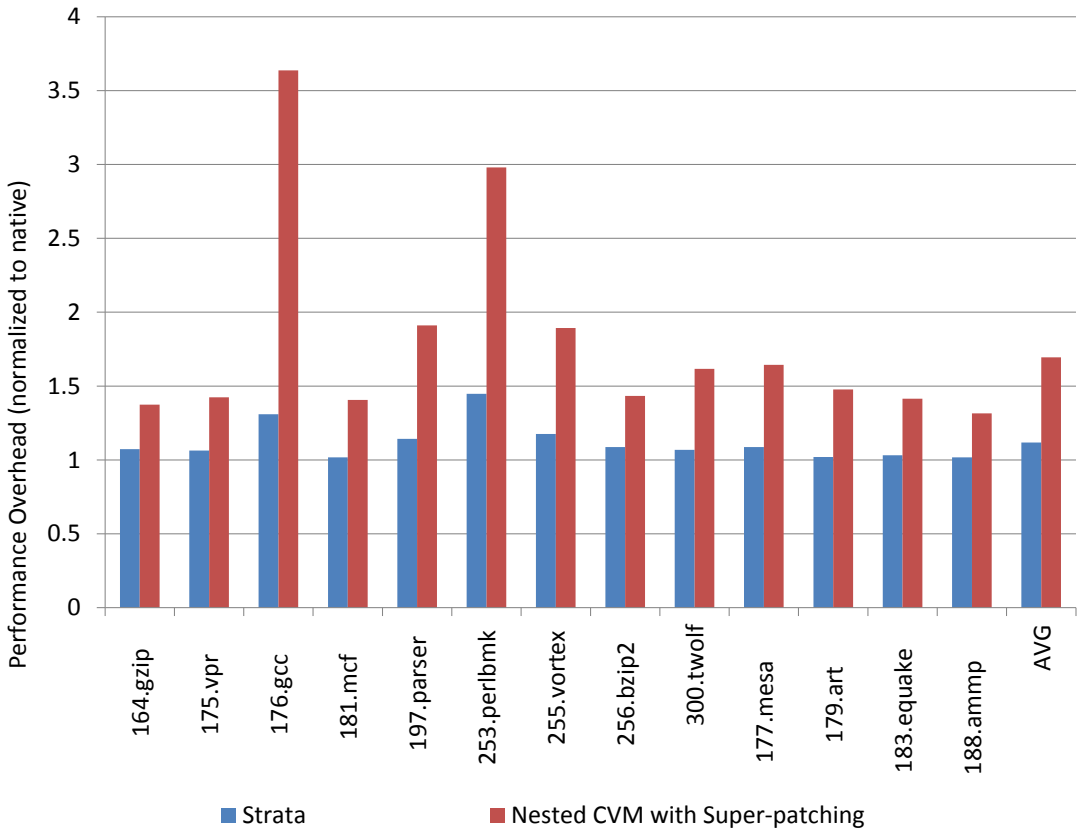


Figure 9.5: Performance overhead for super-patching in the nested CVM configuration. The performance overhead of Strata is also provided for comparison. On average, super-patching reduces the overhead of nested CVMs to 70% over native execution.

performance overhead.

Figure 9.5 compares the overhead for an application running under a single Strata instance, and the nested configuration. Due to the reduction in unwanted software cache flushes, the total overhead is now 70% over native. One of the reasons for the high overhead pertains to indirect branches. Applications with a high occurrence of indirect branches are known to cause performance issues in SDTs [54]. Since the SDT itself uses indirect branches during run time, adding the second layer causes the overhead to increase significantly. This amplification is specially visible in benchmarks such as `176.gcc` and `253.perlbmk`, which also have a high occurrence of indirect branches.

Super-patching can lead to a reduction of overhead in nested CVMs. However, there are trade-offs involved between this optimization and protection. In the next section, we discuss these trade-offs

Benchmark	CC for Single PVM	CC for Nested CVMs with Super-patching	Percent Increase from Original
176.gcc	1604	6078	278.92
181.mcf	351	989	181.76
256.perlbmk	803	2997	273.22
179.art	181	r5632	193.92

Table 9.2: Cyclomatic Complexity of the dynamic CFGs obtained from the software caches. In this case, the nested CVM configuration has super-patching enabled. The CC values indicate that super-patching significantly reduces complexity, which has a negative effect on the protection properties of CVMs.

and other issues in more detail.

9.5 Issues Regarding CVMs and Scope for Future Work

In this chapter, we have introduced the concept of composable virtualization to combat reverse engineering. Our preliminary investigations reveal that CVMs can thwart analysis of code. There are several issues that need to be resolved to make CVMs practical. Resolving these issues provides a rich source for future research.

9.5.1 Performance Trade-offs

In Section 9.4.3, we introduced the concept of super-patching to alleviate some of the performance overheads associated with nesting CVMs. At the high-level, super-patching can be viewed as introducing *short cuts* between paths in the CFG of the protected application. Super-patching creates a path from a translated block to its target, overwriting a trampoline which would have taken execution via the SDT library.

This creation of a short cut inherently reduces the complexity of the dynamic CFG, which is an issue from the protection aspect. Extensive use of super-patching will make the CFG easier to analyze, and therefore, easier to reverse engineer. To demonstrate this reduction in complexity, we repeated the Cyclomatic Complexity (CC)experiments of Section 9.3.2 on the nested CVM configuration, with super-patching enabled. Table 9.2 displays the results of the experiment.

As the findings demonstrate, the CC values have been significantly reduced. This experiment indicates that indiscriminate use of super-patching actually obviates most of the protections afforded by nested CVM configurations, facilitating the task of the adversary.

Therefore, we propose selective super-patching as a technique to improve performance overheads, as well as, to maintain the protective features of CVMs. Rather than applying super-patching to every translated trampoline blocks, some selection criteria should be used. One criteria that can be appropriate in this scenario involves super-patching those blocks that are executed frequently. The outer CVM (Strata_2) can maintain a count of translated trampolines, and super-patch them after their execution count reaches a threshold. Otherwise, it should flush its software cache, and continue translation.

Future work can investigate and compare different selection strategies for super-patching. An appropriate strategy should balance performance overhead and Cyclomatic Complexity.

9.5.2 Security Evaluation of CVM Methodology

In this chapter, we investigated code layout and dynamic CFGs to gain some insight into the protective qualities of CVMs. Our preliminary results indicate that CVM can provide robust protection of software. We believe that further demonstration of CVM protections will facilitate wider adoption of this methodology to safeguard applications.

To further highlight the protective properties of nested CVMs, consider the following example of analyzing the dynamic CFG of the protected application. The dynamic CFG is made up of nodes (basic blocks) that belong to the application and the various constituent CVMs. These nodes are interleaved with each other, as control keeps transferring between the CVMs and the translated application code. An adversary attempting to extract the critical information from such a CFG, will have to excise those parts that belong to the CVMs. We believe that the adversary's task can be reduced to the problem of *subgraph isomorphism*, which involves identifying a smaller subgraph in a larger graph. This problem is well-established in the field of theory of computation, and in the general case, is NP-complete to solve [131].

Formally analysing the accuracy of this reduction hypothesis is an area for future research.

9.5.3 Incorporating Protection Techniques into CVMs

Most of the experiments described in this chapter used standard VMs with no protections added. An important direction for future work involves incorporating the various protection schemes described in the previous chapters into CVMs. Techniques such as temporal polymorphism and code checkers will potentially improve application protection. Techniques such as Point-ISA and DataMeld will potentially ensure that CVMs cannot be replaced at run time. Incorporating these techniques, and ensuring that the results are practical will most likely be a challenging task.

9.6 Summary

This chapter introduces the concept of composable virtualization for program protection. The application is partitioned, and each partition is assigned to a set of protective CVMs (which could be null). The goal is to create a network of protective entities that safeguard the application and themselves from tamper and reverse engineering. We demonstrated that CVMs can successfully obfuscate the code present in the software cache. We also demonstrated that CVMs obfuscated the CFG of the protected application. These protection comes with a higher performance overhead. Consequently, we provided an optimization technique to alleviate such overheads. Further analysis is required on the trade-off between security and performance. Our hope is that this work will initiate further discussion on CVMs, and lead to innovative and robust techniques that make it hard for the adversary to extract critical information.

Chapter 10

Related Work

In Chapter 1 we stressed the need of tamper-resistance schemes in today's world. This need has naturally led to research directed at protecting programs from tamper and reverse engineering. In this chapter, we chronicle some of this past research. We describe several program protection schemes developed over the past few years, and also discuss notable attacks that defeated previously-robust defenses.

When discussing tamper-resistance tehcniques, it is useful to classify them according to their major characteristics. In Chapter 1, we described tamper resistance schemes as consisting of tamper-detection techniques and code obfuscations techniques. We can further divide these categories based on their nature (*i.e.*, hardware or software), their domain (*e.g.*, static or dynamic), use of an extra software layer (*i.e.*, a VM) to apply the protections, *etc.* Based on our analysis, we classify past work into the following categories:

- Self-aware tamper-detection schemes, where the system performs introspection from time to time to verify the code being executed [2, 3, 1].

- Obfuscation schemes, where the protection schemes thwart reverse engineering, making it hard to identify critical aspects of the application [81, 23]. Obfuscation schemes have been applied to various domains, so this class is subcategorized as follows:

  - Static obfuscation schemes, where the application is protected from static analysis [9, 7].
  - Dynamic obfuscation schemes, where the run time of the application is protected from analysis [132, 133].

- Remote tamper-resistance techniques, where the critical part of the application code is mediated by a remote server [5, 6, 134].

We proceed to describe past research in each of these categories.

10.0.1 Self-aware Integrity Verifiers

The first major technique of tamper resistance involves self-aware systems (*i.e.*, the program is augmented to compute a checksum over a region of code which was then compared to a pre-calculated value. Aucsmith introduced an implementation called Integrity Verification Kernels (IVK), which verifies the integrity of critical code segments [1]. Much of the work in tamper resistance is based one or more of Aucsmith's ideas. Another introspection scheme was introduced by Horne et al. [3] At run time, a large number of embedded code blocks, called testers, verify the integrity of code (using a linear hash function and an expected hash value); if the integrity check fails, an appropriate response is pursued. The use of a number of testers increases the attacker's difficulty of disabling testers. Chang and Atallah, proposed a scheme involving a set of guards which can be programmed to carry out arbitrary tasks: one example is checksumming code segments for integrity verification providing software tamper resistance [2]. Another suggested guard function is repairing code (*e.g.*, if a damaged code segment is detected, downloading and installing a fresh copy of the code segment).

Jacob, *et al.* proposed oblivious hashing, which involves compile-time code modifications resulting in the computation of a running trace of the execution history of a program. Here a trace is a cumulative hash of the values of a subset of expressions which occur within the normal program execution [4]. This technique does not prevent attackers from using dynamic techniques to identify and remove guards.

Tamper detection can be implemented by checking the the validity of computed results [135]. The application can also check for the validity of the execution environment, although the actual methods are system-specific [136]. Tan et al. investigated response techniques to software tampering [93].

Self-aware systems work on the assumption that the underlying platform is based on the Von Neumann architecture. The Von Neumann architecture uses the same memory to code and data. So reads, writes, and instruction fetches all access the same memory. Most software systems assume the underlying platform follows the Von Neumann architecture as well.

On the other hand, many modern architectures use a *Harvard* architecture, where code and data are stored in different memory spaces. Typically, it is the responsibility of the Operating System to handle consistency issues with regardd to the two memory spaces.

Wurster, *et al.* were able to craft an attack on self-checking systems using a modified OS [20]. In the attack, the authors created a copy of the application and made modifications to the code. At run time, the original, untampered application was placed in the data memory unit, whereas the tampered application was placed in the code memory unit. Data reads (for the integrity verification) accessed the data memory unit, and did not report any modifications. Instruction fetches, on the other hand, accessed the code memory unit, which fetched code from the tampered application. In this manner, the authors were able to invalidate the checking mechanism. This attack is referred to as the *split-memory attack*.

Giffin, *et al.* proposed a novel solution to the split-memory attack [98]. Their solution involved the insertion of self-modifying code to applications. Successfully executing self-modifying code indicates that the underlying system follows the Von Neumann architecture. Thus, the authors suggested running self-modifying code at application start up, to ensure that the underlying platform has not been tampered.

Self-checking systems are still popular. Many applications such as Skype, Adobe, Arxan *etc.* use software guards to protect against code tampering.

10.1 Obfuscation

To successfully reverse engineering an application, the adversary must first be able to analyze and comprehend its flow. Obfuscation techniques focus on defeating analysis of code. Obfuscation techniques can primarily be classified as static or dynamic. A subclass of obfuscation techniques use an addition software virtualization layer to protect programs. Because this dissertation deals with PVM-enabled protections, we will discuss these techniques as well.

10.1.1 Static Obfuscations

Static analysis is a common technique, and is used by most adversaries to extract information about the instructions implementing the application. Disassembly is usually the first step in the process of static analysis, and involves statically disassembling the binary executable code and restoring its corresponding assembly code. However, generating completely accurate assembly code is difficult, and researchers have developed several methods to improve the processs accuracy [137]. Linear sweep linearly scans over the code, disassembling instructions, assuming that every instruction is followed by another instruction. GNUs `gdb` implements this technique. Recursive traversal takes control flow into account. However, as some branches are input-dependent, usually not all target addresses can be statically derived and disassembled. Disassembling such instructions often requires the use of techniques that recover indirect jump tables [138].

Protecting applications from such static analysis has been investigated extensively and has produced encouraging results. Collberg, *et al.* presented a seminal work on software obfuscations [8]. Their work categorized different types of code transformations (*e.g.*, control flow obfuscation, layout obfuscation, data obfuscation, *etc.*). Collberg also proposed the concept of *opaque predicates*. Opaque predicates are expressions which are hard to predict statically [7].

Control flow obfuscation is one of the more commonly-used approaches against reverse engineering. Saumya Debray and his research group have proposed several novel techniques in this area. Popov, Debray, *et al.* proposed using operating system signals to obfuscate the static CFG of the application [81]. Linn, Debray, *et al.* designed the concept of *branch functions*, to obscure the

target of `call` instructions [23]. Any call to functions are rerouted through a branching function that performs complex calculations to retrieve the target address. These calculations were hard to identify statically. Chenxi Wang, *et al.* devised the concept of control flow flattening, in which the control flow graph of the application was transformed into a series of `switch-case` statements. The authors demonstrated that attempting to statically locate the target of the switch statements was NP-complete.

Encryption is also a commonly-used technique to thwart static assembly. Previous encryption techniques have suffered from coarser levels of decryption granularity. Biondi, *et al.* successfully reverse engineered Skype, by extracting the plaintext code from memory [19]. Cappaert, *et al.* presented a partial encryption approach, in which application code is partitioned into small segments and encrypted [12]. The encrypted code segments are decrypted at run time by users. Thus, the partial encryption ameliorates the faults of illuminating all of the binary code at once as only the essential segments of the code are decrypted at run time.

Software defenders often try to ensure that a successful attack against one instance of an application will not be effective against another instance. This property can be implemented via the use of *software diversity*. Cohen was among the first researchers to address the potential of code diversity as a defense mechanism against attacks [95]. Collberg, *et al.* looked at models from biology and history, classify them into primitives, and illustrate how to map them onto a digital world where software also requires defense mechanisms [139]. Code polymorphism is a common example of software diversity. Malware writers often use this technique to evade virus scanners [61]

These static techniques provide viable protection to the application from attacks. However, these techniques are susceptible to dynamic attacks. In the next section, we describe some techniques to thwart such dynamic attacks.

10.1.2 Dynamic Protection Techniques

Recently, adversaries have started using dynamic schemes to extract critical information from the application. Dynamic schemes typically involve running the application on a simulator, and providing

forged inputs to break protection techniques. Such attacks have been shown to be very effective. Barak, *et al.* were able to successfully prove that perfect obfuscation of general applications is impossible when they are run in an environment under the complete control of the adversary [22].

A number of researchers have attempted to decrease the rate of information leakage at run time. These techniques usually involve discovering the critical application instructions in stages. For example, on-demand decryption consists of decrypting parts of the application that are scheduled to be executed. Consequent to execution, the parts can be re-encrypted. This scheme was implemented in the Shiva system [140].

Kanzaki, *et al.* describe a technique to overwrite program instructions with dummy ones. The application is crafted in such a way that each dummy instruction is restored prior to executing it. Similarly, Madou, *et al.* propose a rewriting engine that updates function bodies at run time [133]. Additionally, the authors propose to cluster similar functions into a common template. Finally, Mavrogiannopoulos, *et al.* classify self-modification techniques based on the attackers toolset capabilities [141].

Software virtualization has also been previously used to create a trusted execution environment, both at the system level and at the process level. The Proteus system uses software diversity to prevent tampering [11]. The authors discuss diversifying multiple characteristics such as instruction semantics, construction encoding and operand encoding to ensure that successful attacks on one instance of the application do not succeed against any other instance. However the high overhead, (more than 50X), makes it impractical for use in most settings.

The Terra system implements a trusted virtual machine monitor which can be used to create closed-box platforms where the developer has complete control, which co-exist with standard open platforms [40]. However, it requires hardware support to validate the software stack. Chen et al. discuss Overshadow, a system that cryptographically isolates an application inside a Virtual Machine Monitor from the guest OS it is running on. This system offers another layer of tamper resistance, even in the case of total OS compromise [142]. However the VMM is open to compromise [143]. This research provides a uniform protection mechanism in which all components are protected.

10.2 Remote Tampering of Software

As we mentioned previously, Barak, *et al* results indicate that it is impossible to protect applications when they are run under the control of the adversary [22]. Many researchers advocate partitioning the application based on criticality and running the most critical parts on a secure server. The non-essential parts can run on the user's machine. When a crucial computation needs to be performed, a request is sent to the server. When the server concludes its computations, it transmits the results to the client. In this manner, the assets of the applications are not not directly accessibly by the adversary. Shesadri, *et al.* proposed the Pioneer system, in which a software-based primitive ensures verifiable code execution [5]. This system is based on a challenge-response protocol between an external trusted entity, called the dispatcher, and an untrusted computing platform, called the untrusted platform. The dispatcher communicates with the untrusted platform over a communication link, such as a network connection. After a successful invocation of Pioneer, the dispatcher obtains assurance that the application on the untrusted platform is unmodified.The authors then implemented this scheme for embedded devices as well [6].

Collberg, *et al.* proposed a similar scheme, in which the server side continuously obfuscates and replaces the application code blocks on the client side, to thwart analysis and modification [144].

Such systems have shown to be effective at thwarting analysis and modification. However, they need stringent Quality-of-Service guarantees from the underlying hardware and network connections, thereby constricting their use, specially for mobile devices [33]. Such guarantees reduces their applicability in various situations.

10.3 Hardware Approaches

A number of hardware-based protection techniques have also been proposed [145, 146, 26]. Since this research focuses on software-only solutions to reverse engineering, we will not be discussing these approaches.

Chapter 11

Summary

In this chapter, we summarize the contributions from this dissertation, and describe some avenues for future research.

11.1 Conclusions

Many software systems perform critical tasks and need to be protected from analysis and tamper. Of particular interest is the improvement of the dynamic security of the application. The thesis of this dissertation is that composing applications with process-level virtual machines can effectively hamper reverse engineering and tamper attacks on software. To support this thesis, we have proposed several new techniques, and evaluated them via metrics, as well as attack methodologies.

Chapter 1 discussed the motivation for improved program protections. Current techniques are not adequate against dynamic analysis. Novel schemes are required that provide a moving attack surface. PVMs can provide such a fluctuating surface. Techniques that are based on PVMs will be much more dynamic and harder to reverse engineer.

Chapter 2 introduced the concept of virtualization. Virtualization involves the insertion of an extra layer of software between the application and the underlying platform. The software layer can be at the process-level, or at the system level. The focus of this dissertation is exclusively on process-level virtualization. This chapter also discussed the concept of software dynamic translation,

which is an efficient technique for virtualizing software. One of the main attributes of a software dynamic translator (SDT) involves a software cache, where translated code is stored. The ideas presented in this work were prototyped using the Strata SDT library, developed at the University of Virginia [51].

One of the research contributions of this dissertation is the design of a model, in Chapter 3. The purpose of this model is to to describe software applications and interpretation. This model provides an insight into attack methodologies targeted at software, as well as techniques to thwart such attacks. This model was utilized in later chapters to illustrate attack methodologies, and their solutions.

Although the focus of this dissertation was to improve the dynamic protection of application, PVMs can protect on-disk binaries as well. Chapter 4 demonstrated that PVMs can be used to increase the entropy of the binary. The increase in entropy is achieved by interleaving code from the application and the PVM. The increased entropy provide higher levels of protection from static analysis.

Chapter 5 explored software guards in the context of process-level virtualization. The fluctuating nature of PVM mediation enables guards to execute from random locations in memory, making it difficult for the adversary to locate them. The use of instantiation polymorphism introduced diversity into the scheme, preventing the adversary from using a generic attack methodology on all instances of the protected software. The chapter also introduced knots, to protect code located in the software cache of the SDT. Previously, this code was left unprotected.

The ability of PVMs to continuously shift the attack surface was extensively investigated in Chapter 6. The software cache can be flushed periodically to thwart attackers. After each flush, the SDT continues translating the application, and caching them in a different memory location. Also, the instruction opcodes used to implement the application's semantics can be varied. All these techniques taken together, constitute temporal polymorphism, which creates a widely-shifting attack surface. The chapter described the evaluation of various properties of temporal polymorphism, and demonstrated its effectiveness against established dynamic analysis techniques.

Having demonstrated the effectiveness of PVMs at thwarting attacks, the focus of research turned to investigating weaknesses in the PVM itself. Chapter 7 illustrated that the PVM can be replaced dynamically, enabling the adversary to analyze the program unhindered. The primary weakness in current PVMs is the lack of binding between the application and the PVM. To combat this weakness, we proposed the concept of *homographic* instructions. The basic idea involves customizing the semantics of selected instructions according to the associated protective PVM instance. Interpreting these instructions under any other context will yield different semantics, and can be used to detect a replacement attack. The solution scheme is termed as Point-ISA. Point-ISA can be used to semantically bind the application to its protective PVM.

Coogan, *et al.* demonstrated that, by analyzing the execution trace of a PVM-protected application an adversary can identify critical instructions. The approach involves tracking instructions accessing certain data values. Chapter 8 extensively investigates this attack methodology termed Value-based Dependence Analysis (VDA). We extended this idea to include attacks on SDTs as well. We then proposed a novel defense against this attack scheme, termed as DataMeld. The approach involves interleaving the data flow of the application with that of the PVM. This interleaving results in obfuscation of dataflow analyses. Our results indicated that VDA attacks lose their effectiveness significantly in the presence of DataMeld.

Finally, we expand the scope of PVM-based protection by proposing the novel concept of composable virtualization. The basic idea involves partitioning the application, and assigning each partition to a set of protective PVM instance. The PVM instances can protect each other as well, creating a network of protection. Our preliminary results indicate that composable virtual machines can significantly increase the effort required by the adversary to extract vital information from protected applications.

This research has proposed several new ideas on program protection, and has also opened up new avenues for research. We discuss some of these avenues in the next section.

11.2 Scope for Future Work

The research presented in this dissertation significantly advances the state-of-the-art. This assertion has been amply demonstrated by evaluating some of the proposed protection techniques against published attack methodologies. There still exist topics that need to be investigated further. In this section, we describe some of these new avenues for research.

11.2.1 Expanding the Scope of Homographic Instructions

In Chapter 7, we introduced the concept of homographic instructions. Such instructions possess custom semantics when they are interpreted by the protective PVM instance. These custom semantics are different from their standard semantics, which is determined by the ISA. The scope of these instructions can be expanded beyond software virtualization. One such area is hardware-software codesign for software anchoring. Software anchoring is defined as binding an instance of a software application with a particular hardware system, *i.e.*, that instance of the software can only run on that particular hardware platform [147]. There are many scenarios where software anchoring is useful, such as loss of critical armament technology, where it is paramount that the adversary is unable to rehost the software on possibly counterfeit hardware.

Researchers have advocated using Physically Uncloneable Functions (PUFs) for software anchoring [147]. However, PUFs do not provide any protection from software reverse engineering. With the use of homographic instructions, the adversary is no longer aware of the complete semantics of the application (because the semantics of these homographic instructions are not known to the adversary). Consequently, the adversary will be unable to extract high level information from the software.

An area of future research could involve techniques to make this hardware design for homographic instructions easier.

11.2.2 Alternate Threat Models

The protection techniques in this work addressed a restrictive threat model (on the part of the software defender). Once the software is released and obtained by an adversary, they could perform various analyses to extract vital information. An innovative area of research involves evaluating these techniques under different threat models. For example, it would be interesting to study the impact of protective PVMs if the application ran partly on a secure server.

Changing the threat model could uncover weakness in the current techniques, as well as provide new avenues for protection.

11.2.3 Expanding the Scope of Protections

The protection techniques proposed in this work, are closely tied to software dynamic translation. Consequently, the practicality and acceptability of these techniques depends on SDTs. As SDTs are expanded to newer platforms and environments, their use in program protection will also increase.

The techniques described in this work were prototyped using Strata on 32-bit Intel machines, running the Linux kernel. Currently, work is being done to expand Strata to run on 64-bit machines, as well as the Microsoft Windows platform. Mobile systems also represent a new frontier for software dynamic translation. The instances of reverse engineering on mobile computing devices is increasing significantly, and it is our belief that these protection techniques will be notably effective on such platforms. Architectures often have idiosyncrasies that need to be considered (for example, on ARM platforms, the pc can be used as a general-purpose register). Issues such as power consumption, and memory footprint can often cause problems when techniques are transfered from general-purpose computing systems to mobile platforms [148]. Therefore, these issues need to be studied carefully for proper adoption of these techniques on mobile systems.

11.2.4 Appropriate Program Partitioning

An integral part of composable virtualization involves partitioning the application. In Section 9.1, we enumerated several established schemes to partition the application for security purposes. An

interesting avenue for research involves testing the suitability of these partitioning schemes with respect of CVMs. Research could also focus on designing new partition schemes, which may be more appropriate for composable virtualization.

11.2.5 Incorporating Protections into CVMs

Chapter 9 provided a preliminary insight into the potential of Composable Virtual Machines to provide protection. The next stage of research involves incorporating various techniques discussed in Chapters 4, 5, 6, 7, and 8 into CVMs, and evaluating their strength. There are numerous configurations that can be applied to protect the applications. For example, the software defender could vary the rate of software cache flushing, or rate of insertion of guards. We believe that further investigation of CVM protections to be one of the more interesting avenues.

11.3 Summary

The software defender and the adversary are engaged in a constant battle for control of critical software. The defenders continue to design new mechanisms to protect applications from reverse engineering and tamper. The adversary continually attempts to break these protections, and extract valuable assets from the application. In this dissertation, we have investigated the use of PVMs in the area of program protections. Our research indicates that a holistic design of a protected package, comprising of virtual machines and the application, can provide robust security. As part of our research, we have devised several PVM-based protection techniques that advance the state of the art. Our evaluation has demonstrated that these techniques can withstand current attacks methodologies, and will require significant effort on the part of the adversary to disable them. Based on past experience, it is likely that the adversary will keep devising new strategies in an attempt to disable these protections. Continuous innovation is required to keep the adversary at bay.

Glossary

Checker Sequence of instructions that verify the integrity of code.

Composable virtualization Multiple virtualization layers applied to different partitions of an application, to protect it from attacks.

Guard Sequence of instructions that verify the integrity of statically -generated code.

Knot Sequence of dynamically-generated instructions that verify the integrity of PVM-generated code.

Output equality A transformed version of an application is said to possess output equality, if it produces the same output as the original version, for any set of inputs.

Patch Overwriting of a trampoline in the software cache, to transfer control to the translated instruction.

Software cache Software-managed memory buffer where software dynamic translators cache the translated code, to amortize the overhead of translation. This region is also known as a code cache.

Trampoline A code sequence appended to the end of a translated block, that returns control back to the software dynamic translator. It is also responsible for passing as argument, the address containing the next application instruction scheduled for execution

ABI Application Binary Interface

CIF Code Introspection Framework, which is capable of monitoring and instrumenting the code being executed

CVM Composable Virtualization Machines.

DIC Dynamic Instruction Count. The dynamic instruction count of the application.

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

ICV Instruction Coverage for Variablesr. Number of instructions that affect the value of a variable.

PVM Process-level Virtual Machine. A software layer that virtualizes one single process.

SDT Software Dynamic Translator. A process-level virtual machine that implements binary translation with caching.

SSA	Static Single Assignment. An intermediate representation in which each variable is assigned only once
SVM	System-level virtual machine. A software layer that virtualizes an entire Operating System.
VDA	Value-based Dependence Analysis. An analysis technique which consists of identifying application data values from an execution trace, and performing flow analysis on these values in the reverse execution order.
VMM	Virtual Machine Monitor
$\boldsymbol{I^{HG}}$	The set of instructions that are classified as homographic for a particular benchmark
$\boldsymbol{P_{T(A)}}$	An application that has been tampered by an adversary.
$\boldsymbol{P_{TR(A)}}$	An application that is protected by tamper-resistance techniques, static and dynamic

Bibliography

[1] David Aucsmith. Tamper-resistant software: An implementation. In *Proceedings of the 1st International Workshop on Information Hiding*, pages 317–333, London,U.K, 1996. Springer-Verlag.

[2] Hoi Chang and Mikhail J. Atallah. Protecting software code by guards. In *DRM '01: Revised Papers from the ACM CCS-8 Workshop on Security and Privacy in Digital Rights Management*, pages 160–175, London, UK, UK, 2002. Springer-Verlag.

[3] B. Horne, L. R. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Digital Rights Management Workshop*, pages 141–159, London, U.K., 2001.

[4] Matthias Jacob, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *MM&Sec '07: Proceedings of the 9th Workshop on Multimedia & Security*, pages 129–140, New York, NY, USA, 2007. ACM Press.

[5] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP '05: Proceedings of the 20th ACM Symposium on Operating Systems Principles*, volume 39, pages 1–16, New York, NY, USA, December 2005. ACM Press.

[6] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. SWATT: Software-based attestation for embedded devices. In *SP '04: Proceedings of the 25th IEEE Symposium on Security and Privacy*, pages 272–282, Oakland, CA, May 2004.

[7] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 184–196, New York, NY, USA, 1998. ACM.

[8] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. *University of Auckland Technical Report*, page 170, 1997.

[9] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, University of Virginia, Charlottesville, VA, USA, 2000.

[10] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *DSN '01: Proceedings of the International Conference on Dependable Systems and Networks*, pages 193–202, Goteborg, Sweden, 2001. IEEE Computer Society.

[11] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *DRM '06: Proceedings of the ACM Workshop on Digital Rights Management*, pages 47–58, New York, NY, USA, 2006. ACM Press.

[12] Jan Cappaert, Bart Preneel, Bertrand Anckaert, Matias Madou, and Koen De Bosschere. Towards tamper resistant code encryption: practice and experience. In *ISPEC '08: Proceedings of the 4th International Conference on Information Security Practice and Experience*, pages 86–100, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[14] Oleh Yuschuk. Ollydbg: A window debugger. `http://www.ollydbg.de`, 2006.

[15] y0da. Lordpe: A pe file editor. `http://http://www.woodmann.com/collaborative/tools/index.php/LordPE`, 2009.

[16] Global Software Piracy Study for 2007, May 2008. Business Software Alliance.

[17] Raghunathan Srinivasan, Charles Colbourn, and Aviral Shrivastava. Protecting anti-virus software under viral attacks, 2007.

[18] The Beast Worm. http://lists.virus.org/dshield-0310/msg00337.html.

[19] Philippe Biondi and Desclaux Fabrice. Silver needle in the skype. In *Black Hat Europe*, Amsterdam, the Netherlands, 2006.

[20] Glenn Wurster, P. C. van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 127–138, Washington D.C., U.S.A, 2005. IEEE Computer Society.

[21] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-box cryptography and an AES implementation. In *SAC '02: Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography*, pages 250–270, London, UK, 2003. Springer-Verlag.

[22] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Crypto '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, pages 1–18, London, UK, 2001. Springer-Verlag.

[23] Cullen Linn and Saumya Debray. 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)*, pages 290–299, Washington D.C., U.S.A, 2003. ACM Press.

[24] Kelly Heffner and Christian S. Collberg. The obfuscation executive. In Kan Zhang and Yuliang Zheng, editors, *Information Security, Proceedings of the 7th International Conference on Information security*, volume 3225 of *Lecture Notes in Computer Science*, pages 428–440. Springer, 2004.

[25] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Language and Systems*, 16(5):1467–1471, 1994.

[26] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. AEGIS: Architecture for tamper evident and tamper resistant software. In *SC '03: Proceedings of the 17th Annual International Conference on Supercomputing*, pages 161–171. ACM Press, 2003.

[27] Robert M. Best. Preventing software piracy with crypto-microprocessors. In *The IEEE Spring COMPCON*, page 466, 1980.

[28] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 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*, volume 35, pages 168–177, New York, NY, USA, 2000. ACM Press.

[29] David Lie, John Mitchell, Chandramohan A. Thekkath, and Mark Horowitz. Specifying and verifying hardware for tamper-resistant software. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 166, Washington D.C., U.S.A, 2003. IEEE Computer Society.

[30] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 272–280, New York, NY, USA, 2003. ACM Press.

[31] Benjamin A. Kuperman, Carla E. Brodley, Hilmi Ozdoganoglu, T. N. Vijaykumar, and Ankit Jalote. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, 48(11):50–56, 2005.

[32] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *VM '04: Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[33] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *CCS '09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, pages 400–409, New York, NY, USA, 2009. ACM.

[34] Sudeep Ghosh, Jason D. Hiser, and Jack W. Davidson. A secure and robust approach to software tamper resistance. In *IH '10: Proceedings of the 12th International Conference on Information Hiding*, pages 33–47, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[36] VMProtect Software. VMProtect. `http://vmpsoft.com/`, 2008.

[37] Oreons Technology. Codevirtualizer. `http://oreans.com/codevirtualizer.php`, 2009.

[38] Oreans Technologies. Themida. `http://oreans.com/themida.php`, 2009.

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

[40] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 193–206, New York, NY, USA, 2003. ACM Press.

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

[42] Mathias Payer and Thomas R. Gross. Fine-grained user-space security through virtualization. In *VEE '11: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 157–168, New York, NY, USA, 2011. ACM Press.

[43] Wei Hu, Jason D. Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 2–12, New York, NY, USA, 2006. ACM Press.

[44] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17:412–421, July 1974.

[45] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 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*, pages 15–24, Washington, DC, USA, 2003. IEEE Computer Society.

[46] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, pages 265–275, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[47] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *USENIX '02: Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[48] Jim Smith and Ravi Nair. *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.

[49] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[50] Paul Klint. Interpretation techniques. *Software-Practice and Experience*, 11(9):963–973, 1981.

[51] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa. Retargetable and reconfigurable software dynamic translation. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, pages 36–47, Washington D.C., U.S.A, 2003. IEEE Computer Society.

[52] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, and Prashanth P. Bungale. HDTrans: An open source, low-level dynamic instrumentation system. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 175–185, New York, NY, USA, 2006. ACM.

[53] Jason D. Hiser, Daniel Williams, Wei Hu, Jack W. Davidson, Jason Mars, and Bruce R. Childers. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 61–73, Washington, DC, USA, 2007. IEEE Computer Society.

[54] Jason D. Hiser, Daniel Williams, Adrian Filipi, Jack W. Davidson, and Bruce R. Childers. Evaluating fragment construction policies for SDT systems. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 122–132, New York, NY, USA, 2006. ACM Press.

[55] Kim Hazelwood and James E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, pages 89–, Washington, DC, USA, 2004. IEEE Computer Society.

[56] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 209, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[57] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *CCS '11: Proceedings of the 18th ACM conference on Computer and Communications Security*, pages 275–284, New York, NY, USA, 2011. ACM.

[58] Cataldo Basile, Stefano Di Carlo, Thomas Herlea, Brecht Wyseur, and Jasvir Nagra. Towards a formal model for software tamper resistance. *http://www.cosic.esat.kuleuven.be/publications/article-1280.pdf*, 2009.

[59] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[60] Matthew A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[61] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[62] Bruno De Bus, Bjorn De Sutter, Ludo Van Put, Dominique Chanet, and Koen De Bosschere. 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*, pages 211–220, Washington D.C., U.S.A, 7 2004. ACM Press.

[63] Adi Shamir and Nicko van Someren. Playing "hide and seek" with stored keys. In *FC '99: Proceedings of the Third International Conference on Financial Cryptography*, pages 118–124, London, UK, UK, 1999. Springer-Verlag.

[64] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. In *USENIX '08: Proceedings of the 17th USENIX Security Symposium*, volume 52, pages 91–98, Berkeley, CA, USA, May 2009. USENIX Technical Association.

[65] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *CCS '12: Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 305–316, New York, NY, USA, 2012. ACM.

[66] Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box DES implementation for DRM applications. In *DRM '02: Proceedings of the Digital Rights Management Workshop*, pages 1–15, London, UK, 2002. Springer.

[67] Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *Selected Areas in Cryptography*, pages 227–240, Hiedelberg, 2004. Springer-Verlag.

[68] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In *SAC '07: Proceedings of the 14th International Conference on Selected Areas in Cryptography*, pages 264–277, Berlin, Heidelberg, 2007. Springer-Verlag.

[69] Hamilton E. Link and William D. Neumann. Clarifying obfuscation: Improving the security of white-box DES. *ITCC '05: Proceedings of the International Conference on Information Technology: Coding and Computing*, 1:679–684, 2005.

[70] Brecht Wyseur. *White-Box Cryptography*. PhD thesis, Katholieke Universiteit Leuven, 2009. Bart Preneel (promotor).

[71] Brecht Wyseur. White-box cryptography: Hiding keys in software. *MISC Magazine*, 5(5):65–72, 2012.

[72] R. A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver and Boyd, 2nd edition, 1943.

[73] Roberto Giacobazzi and Andrea Toppan. On entropy measures for code obfuscation. In *SSP '12: Proceedings of the 2012 ACM SIGPLAN Software Security and Protection Workshop*, pages 1–8, New York, NY, USA, 2012. ACM.

[74] C. E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, January 2001.

[75] G. Marsaglia. Random numbers fall mainly in the planes. *Proceedings of the National Academy of Sciences of the United States of America*, 60:25–28, 1968.

[76] S Pincus. Approximate entropy as a complexity measure. *Proceedings of the National Academy of Sciences of the USA*, 88:2297–2301, 1991.

[77] S. Pincus and R. E. Kalman. Not all (possibly) "Random" sequences are created equal. *Proceedings of the National Academy of Sciences of the USA*, 94:3513–3518, 1997.

[78] Georges Hansel, Dominique Perrin, and Imre Simon. Compression and entropy. In *STACS '92: Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science*, volume 577, London, UK, 1992. Springer Lecture Notes in Computer Science.

[79] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 2006.

[80] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *ECML PKDD '11: Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III*, pages 522–536, Berlin, Heidelberg, 2011. Springer-Verlag.

[81] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *SSYM '07: Proceedings of 16th USENIX Security Symposium- Volume 18*, pages 19:1–19:16, Berkeley, CA, USA, 2007. USENIX Association.

[82] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 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*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[83] M.T. Yourst. 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*, pages 23–34, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[84] Sudeep Ghosh, Jason Hiser, and Jack W. Davidson. Software protection for dynamically-generated code. In *PPREW '13: Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, pages 1:1–1:12, New York, NY, USA, 2013. ACM.

[85] Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Software piracy prevention through diversity. In *DRM '04: Proceedings of the 4th ACM Workshop on Digital Rights Management*, pages 63–71, New York, NY, USA, 2004. ACM.

[86] Daniel Williams, Wei Hu, Jack W. Davidson, Jason D. Hiser, John C. Knight, and Anh Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *IEEE Security & Privacy*, 7(1):26–33, 2009.

[87] Sandeep Bhatkar and R. Sekar. Data space randomization. In *DIMVA '08: Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22, Berlin, Heidelberg, 2008. Springer-Verlag.

[88] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *SSYM '05: Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, pages 17–17, Berkeley, CA, USA, 2005. USENIX Association.

[89] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a board range of memory error exploits. In *SSYM '03: Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, pages 8–8, Berkeley, CA, USA, 2003. USENIX Association.

[90] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *ACSAC '12: Proceedings of the 28th Annual Computer Security Applications Conference*, pages 319–328, New York, NY, USA, 2012. ACM.

[91] Serge Chaumette, Olivier Ly, and Renaud Tabary. Automated extraction of polymorphic virus signatures using abstract interpretation. In *NSS '11: 5th International Conference on Network and System Security, NSS 2011, Milan, Italy, September 6-8, 2011*, pages 41–48. IEEE, 2011.

[92] Philippe Beaucamps. Advanced metamorphic techniques in computer viruses. In *CESSE '07: Proceedings of the International Conference on Computer, Electrical, and Systems Science, and Engineering*, pages 10–23, Venice, Italy, 2007.

[93] Gang Tan, Yuqun Chen, and Mariusz H. Jakubowski. Delayed and controlled failures in tamper-resistant systems. In *Proceedings of 8th Information Hiding Workshop*, pages 216–231, Alexandria, VA, U.S.A, July 2006.

[94] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP, Vol 2: Design, Implementation, and Internals*. Prentice Hall, 2 edition, 1994.

[95] Frederick B. Cohen. Operating system protection through program evolution. *Journal on Computing Security*, 12(6):565–584, October 1993.

[96] Mark Stamp. Risks of monoculture. *Communications of the ACM*, 47(3):120–, March 2004.

[97] J.A. Whittaker. No clear answers on monoculture issues. *IEEE Security & Privacy*, 1(6):18–19, 2003.

[98] Jonathon T. Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 23–32, Washington D.C., U.S.A, 2005. IEEE Computer Society.

[99] Matias Madou, Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *DRM '05: Proceedings of the 5th ACM workshop on Digital Rights Management*, pages 75–82, New York, NY, USA, 2005. ACM Press.

[100] S.K. Udupa, S.K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In *WCRE '05: Proceedings of the International Working Conference on Reverse Engineering*, volume 0, pages 45–54, Los Alamitos, CA, USA, Nov. 2005. IEEE Computer Society.

[101] Jean-Marie Borello and Ludovic Mè. Code obfuscation techniques for metamorphic viruses. *Journal of Computer Virology*, 4:211–220, 2008. 10.1007/s11416-008-0084-2.

[102] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *SP '07: Proceedings of the 2009 30th IEEE Symposium on Security and Privacy*, pages 94–109, Washington, DC, USA, 2009. IEEE Computer Society.

[103] Sudeep Ghosh, Jason Hiser, and Jack W. Davidson. Replacement attacks against VM-protected applications. In *VEE '12: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, pages 203–214, New York, NY, USA, 2012. ACM.

[104] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[105] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, July 2000.

[106] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automatic identification of cryptographic primitives in binary programs. In *RAID '11: Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, pages 45–65, London, UK, 2011. Springer-Verlag.

[107] Franck Leder, Peter Martini, and Andrew Wichmann. Finding and extracting crypto routines from malware. In *Proceedings of the IEEE 28th International Performance Computing and Communications Conference (IPCCC)*, pages 394 –401, Washington, DC,USA, December 2009. IEEE.

[108] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *NDSS '10: Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2010.

[109] R. L. (Robert Lawrence) Trask. *A dictionary of phonetics and phonology / R.L. Trask*. London ; New York : Routledge, 1996. Includes bibliographical references.

[110] Alfred Whitehead and Bertrand Russell. *Principia Mathematica*. Number v. 2 in Principia Mathematica. University Press, 1912.

[111] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[112] Intel Corporation. *IA-64 application developer's architecture guide*. Intel Corporation, 1999.

[113] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS '11: Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, 2011. The Internet Society.

[114] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *ISSTA '07:Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 196–206, New York, NY, USA, 2007. ACM.

[115] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS '04: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, New York, NY, USA, 2004. ACM.

[116] Harry J. Saal and Israel Gat. A hardware architecture for controlling information flow. In *ISCA '78: Proceedings of the 5th Annual Symposium on Computer Architecture*, pages 73–77, New York, NY, USA, 1978. ACM.

[117] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: Practical dynamic data flow tracking for commodity systems. In *VEE '12: Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, pages 121–132, New York, NY, USA, 2012. ACM.

[118] Gogul Balakrishnan and Thomas Reps. DIVINE: Discovering variables in executables. In *VMCAI '07: Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 1–28, Berlin, Heidelberg, 2007. Springer-Verlag.

[119] Gogul Balakrishnan. *WYSINWYE: What you see is not what you execute*. PhD thesis, University of Wisconsin at Madison, Madison, WI, USA, 2007. AAI3278779.

[120] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In *PEPM '06: Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 100–111, New York, NY, USA, 2006. ACM.

[121] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *OSDI '08: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 255–266, Berkeley, CA, USA, 2008. USENIX Association.

[122] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS '10: Proceedings of the 17th Annual Network and Distributed System Security Symposium*. The Internet Society, 2010.

[123] Asia Slowinska, Traian Stancescu, and Herbert Bos. DDE: Dynamic data structure excavation. In *APSys '10: Proceedings of the First ACM Asia-Pacific Workshop on systems*, pages 13–18, New York, NY, USA, 2010. ACM.

[124] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: secure program partitioning. In *SOSP '01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 1–14, New York, NY, USA, 2001. ACM.

[125] S. H. K. Narayanan, M. Kandemir, and R. Brooks. Performance aware secure code partitioning. In *DATE '07: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1122–1127, San Jose, CA, USA, 2007. EDA Consortium.

[126] Dan Søndergaard, Christian W. Probst, Christian Damsgaard Jensen, and René Rydhof Hansen. Program partitioning using dynamic trust models. In *FAST '06: Proceedings of the 4th international conference on Formal aspects in security and trust*, pages 170–184, Berlin, Heidelberg, 2007. Springer-Verlag.

[127] Tao Zhang, Santosh Pande, Andre dos Santos, and Franz Josef Bruecklmayr. Leakage-proof program partitioning. In *CASES '02: Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 136–145, New York, NY, USA, 2002. ACM.

[128] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[129] Bertrand Anckaert, Matias Madou, Bjorn De Sutter, Bruno De Bus, Koen De Bosschere, and Bart Preneel. Program obfuscation: a quantitative approach. In *QoP '07: Proceedings of the 2007 ACM Workshop on Quality of Protection*, pages 15–20, New York, NY, USA, 2007. ACM.

[130] Mariano Ceccato, Massimiliano Di Penta, Jasvir Nagra, Paolo Falcarin, Filippo Ricca, Marco Torchiano, and Paolo Tonella. Towards experimental evaluation of code obfuscation techniques. In *QoP '08: Proceedings of the 4th ACM Workshop on Quality of Protection*, pages 39–46, New York, NY, USA, 2008. ACM.

[131] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the Third annual ACM Symposium on Theory of Computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[132] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, pages 170–176, Washington, DC, USA, 2003. IEEE Computer Society.

[133] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *The 6th International Workshop on Information Security Applications (WISA 2005)*, volume LNCS. Springer Verlag, August 2005.

[134] Mariano Ceccato, Mila Dalla Preda, Jasvir Nagra, Christian Collberg, and Paolo Tonella. Barrier slicing for remote software trusting. In *SCAM '07: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 27–36, Washington D.C., U.S.A, 2007. IEEE Computer Society.

[135] Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.

[136] T. Holz and F. Raynal. Detecting honeypots and other suspicious environments. In *Proceedings from the Sixth Annual IEEE Information Assurance Workshop*, pages 29–36, June 2005.

[137] Kaiping Liu, Hee Beng Kuan Tan, and Xu Chen. Binary code analysis. *IEEE Computer*, 46(8):60–68, 2013.

[138] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *IWPC '99: Proceedings of the 7th International Workshop on Program Comprehension*, pages 192–, Washington, DC, USA, 1999. IEEE Computer Society.

[139] Christian Collberg, Jasvir Nagra, and Fei-Yue Wang. Surreptitious software: Models from biology and history. *Communications in Computer and Information Science*, 1:1–21, 2007.

[140] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Survey*, 44(2):6:1–6:42, March 2008.

[141] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[142] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. 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*, pages 2–13, New York, NY, USA, 2008. ACM Press.

[143] Nguyen Anh Quynh. Hijacking (Xen) virtual machine for fun and profit. In *Bellua Security Conference*, Jakarta, Indonesia, 2007.

[144] Christian Collberg, Jasvir Nagra, and Will Snavely. bianlian: Remote tamper-resistance with continuous replacement. Technical report, Tuczon, AZ, USA, 2003.

[145] Peter Williams and Rick Boivie. Cpu support for secure executables. In *TRUST '11: Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, pages 172–187, Berlin, Heidelberg, 2011. Springer-Verlag.

[146] Luis F. G. Sarmenta, Marten van Dijk, Charles W. O'Donnell, Jonathan Rhodes, and Srinivas Devadas. Virtual monotonic counters and count-limited objects using a tpm without a trusted os. In *STC '06: Proceedings of the 1st ACM Workshop on Scalable Trusted Computing*, pages 27–42, New York, NY, USA, 2006. ACM Press.

[147] Mikhail J. Atallah, Eric D. Bryant, John T. Korb, and John R. Rice. Binding software to specific native hardware in a VM environment: the PUF challenge and opportunity. In *VMSec '08: Proceedings of the 1st ACM Workshop on Virtual Machine Security*, pages 45–48, New York, NY, USA, 2008. ACM.

[148] Ryan W. Moore, José A. Baiocchi, Bruce R. Childers, Jack W. Davidson, and Jason D. Hiser. Addressing the challenges of DBT for the ARM architecture. In *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 147–156, New York, NY, USA, 2009. ACM.