Universidade Estadual de Campinas
Instituto de Computação

Marcus Felipe Botacin

Hardware-Assisted Malware Analysis

Análise de *Malware* com Suporte de *Hardware*

CAMPINAS
2017

Marcus Felipe Botacin

Hardware-Assisted Malware Analysis

Análise de *Malware* com Suporte de *Hardware*

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, na área de Sistemas de Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science, in the Computer Systems area..

Supervisor/Orientador: Prof. Dr. Paulo Lício de Geus
Co-supervisor/Coorientador: Prof. Dr. André Ricardo Abed Grégio

Este exemplar corresponde à versão final da Dissertação defendida por Marcus Felipe Botacin e orientada pelo Prof. Dr. Paulo Lício de Geus.

CAMPINAS
2017

Agência(s) de fomento e nº(s) de processo(s): CAPES
ORCID: http://orcid.org/0000-0001-6870-1178

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Márcia Pillon D'Aloia - CRB 8/5180

Botacin, Marcus Felipe, 1991-
B657h Hardware-assisted malware analysis / Marcus Felipe Botacin. – Campinas, SP : [s.n.], 2017.

Orientador: Paulo Lício de Geus.
Coorientador: André Ricardo Abed Grégio.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Malware (Software). 2. Tecnologia da informação - Sistemas de segurança. I. Geus, Paulo Lício de,1956-. II. Grégio, André Ricardo Abed. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações para Biblioteca Digital

Título em outro idioma: Análise de malware com suporte de hardware
Palavras-chave em inglês:
Malware (Computer software)
Information technology - Security measures
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Paulo Lício de Geus [Orientador]
Carlos Alberto Maziero
Sandro Rigo
Data de defesa: 28-07-2017
Programa de Pós-Graduação: Ciência da Computação

Universidade Estadual de Campinas
Instituto de Computação

Marcus Felipe Botacin

Hardware-Assisted Malware Analysis

Análise de *Malware* com Suporte de *Hardware*

Banca Examinadora:

- Prof. Dr. Paulo Lício de Geus
  IC/UNICAMP
- Prof. Dr. Carlos Alberto Maziero
  DInf/UFPR
- Prof. Dr. Sandro Rigo
  IC/UNICAMP

A ata da defesa com as respectivas assinaturas dos membros da banca encontra-se no processo de vida acadêmica do aluno.

Campinas, 28 de Julho de 2017

Acknowledgements

To my family, for supporting me.
To my friends, for walking along me.
To Paulo and André, for advising me since I was an undergraduate student.

Resumo

O mundo atual é impulsionado pelo uso de sistemas computacionais, estando estes presentes em todos aspectos da vida cotidiana. Portanto, o correto funcionamento destes é essencial para se assegurar a manutenção das possibilidades trazidas pelos desenvolvimentos tecnológicos. Contudo, garantir o correto funcionamento destes não é uma tarefa fácil, dado que indivíduos mal-intencionados tentam constantemente subvertê-los visando benefíciar a si próprios ou a terceiros. Os tipos mais comuns de subversão são os ataques por códigos maliciosos (*malware*), capazes de dar a um atacante controle total sobre uma máquina. O combate à ameaça trazida por *malware* baseia-se na análise dos artefatos coletados de forma a permitir resposta aos incidentes ocorridos e o desenvolvimento de contramedidas futuras. No entanto, atacantes têm se especializado em burlar sistemas de análise e assim manter suas operações ativas. Para este propósito, faz-se uso de uma série de técnicas denominadas de "anti-análise", capazes de impedir a inspeção direta dos códigos maliciosos. Dentre essas técnicas, destaca-se a evasão do processo de análise, na qual são empregadas exemplares capazes de detectar a presença de um sistema de análise para então esconder seu comportamento malicioso. Exemplares evasivos têm sido cada vez mais utilizados em ataques e seu impacto sobre a segurança de sistemas é considerável, dado que análises antes feitas de forma automática passaram a exigir a supervisão de analistas humanos em busca de sinais de evasão, aumentando assim o custo de se manter um sistema protegido. As formas mais comuns de detecção de um ambiente de análise se dão através da detecção de: (i) código injetado, usado pelo analista para inspecionar a aplicação; (ii) máquinas virtuais, usadas em ambientes de análise por questões de escala; (iii) efeitos colaterais de execução, geralmente causados por emuladores, também usados por analistas. Para lidar com *malware* evasivo, analistas tem se valido de técnicas ditas transparentes, isto é, que não requerem injeção de código nem causam efeitos colaterais de execução. Um modo de se obter transparência em um processo de análise é contar com suporte do *hardware*. Desta forma, este trabalho versa sobre a aplicação do suporte de *hardware* para fins de análise de ameaças evasivas. No decorrer deste texto, apresenta-se uma avaliação das tecnologias existentes de suporte de *hardware*, dentre as quais máquinas virtuais de *hardware*, suporte de `BIOS` e monitores de *performance*. A avaliação crítica de tais tecnologias oferece uma base de comparação entre diferentes casos de uso. Além disso, são enumeradas lacunas de desenvolvimento existentes atualmente. Mais que isso, uma destas lacunas é preenchida neste trabalho pela proposição da expansão do uso dos monitores de *performance* para fins de monitoração de *malware*. Mais especificamente, é proposto o uso do monitor `BTS` para fins de construção de um *tracer* e um *debugger*. O *framework* proposto e desenvolvido neste trabalho é capaz, ainda, de lidar com ataques do tipo `ROP`, um dos mais utilizados atualmente para exploração de vulnerabilidades. A avaliação da solução demonstra que não há a introdução de efeitos colaterais, o que permite análises de forma transparente. Beneficiando-se desta característica, demonstramos a análise de aplicações protegidas e a identificação de técnicas de evasão.

Abstract

Today's world is driven by the usage of computer systems, which are present in all aspects of everyday life. Therefore, the correct working of these systems is essential to ensure the maintenance of the possibilities brought about by technological developments. However, ensuring the correct working of such systems is not an easy task, as many people attempt to subvert systems working for their own benefit. The most common kind of subversion against computer systems are malware attacks, which can make an attacker to gain complete machine control. The fight against this kind of threat is based on analysis procedures of the collected malicious artifacts, allowing the incident response and the development of future countermeasures. However, attackers have specialized in circumventing analysis systems and thus keeping their operations active. For this purpose, they employ a series of techniques called anti-analysis, able to prevent the inspection of their malicious codes. Among these techniques, I highlight the analysis procedure evasion, that is, the usage of samples able to detect the presence of an analysis solution and then hide their malicious behavior. Evasive examples have become popular, and their impact on systems security is considerable, since automatic analysis now requires human supervision in order to find evasion signs, which significantly raises the cost of maintaining a protected system. The most common ways for detecting an analysis environment are: i) Injected code detection, since injection is used by analysts to inspect applications on their way; ii) Virtual machine detection, since they are used in analysis environments due to scalability issues; iii) Execution side effects detection, usually caused by emulators, also used by analysts. To handle evasive malware, analysts have relied on the so-called transparent techniques, that is, those which do not require code injection nor cause execution side effects. A way to achieve transparency in an analysis process is to rely on hardware support. In this way, this work covers the application of the hardware support for the evasive threats analysis purpose. In the course of this text, I present an assessment of existing hardware support technologies, including hardware virtual machines, BIOS support, performance monitors and PCI cards. My critical evaluation of such technologies provides basis for comparing different usage cases. In addition, I pinpoint development gaps that currently exists. More than that, I fill one of these gaps by proposing to expand the usage of performance monitors for malware monitoring purposes. More specifically, I propose the usage of the BTS monitor for the purpose of developing a tracer and a debugger. The proposed framework is also able of dealing with ROP attacks, one of the most common used technique for remote vulnerability exploitation. The framework evaluation shows no side-effect is introduced, thus allowing transparent analysis. Making use of this capability, I demonstrate how protected applications can be inspected and how evasion techniques can be identified.

List of Figures

List of Tables

Abbreviation List

AMT ..... *Active Management Technology*
APIC ..... *Advanced Programmable Interrupt Controller*
APT ..... *Advanced Persistent Threats*
ASLR ..... *Address Space Layout Randomization*
AV ..... *Anti-virus*
BIOS ..... *Basic Input Output System*
BPU ..... *Branch Prediction Unity*
BTS ..... *Branch Trace Store*
DBG ..... *Debugger*
CG ..... *Call Graph*
CFG ..... *Control Flow Graph*
CFI ..... *Control Flow Integrity*
DBI ..... *Dynamic Binary Instrumentation*
DBT ..... *Dynamic Binary Translation*
DEP ..... *Data Execution Prevention*
DLL ..... *Dynamic Linked Library*
DMA ..... *Direct Memory Access*
EPT ..... *Extended Page Table*
GDB ..... *GNU Debugger*
GPU ..... *Graphical Processing Unity*
HAL ..... *Hardware Abstraction Layer*
HAW ..... *Hardware-Assisted White-listing*
HPC ..... *Hardware Performance Counter*
HVM ..... *Hardware Virtual Machine*
I/O ..... *Input/Output*
IDS ..... *Intrusion Detection System*
IDT ..... *Interrupt Descriptor Table*
IOT ..... *Internet Of Things*
IPMI ..... *Intelligent Platform Management Interface*
IPS ..... *Intrusion Prevention System*
IRQ ..... *Interrupt Request*
ISR ..... *Interrupt Service Routine*
JOP ..... *Jump Oriented Programming*
KPP ..... *Kernel Patch Protection*
LBR ..... *Last Branch Record*
LOP ..... *Loop Oriented Programming*
LVM ..... *Logical Volume Manager*
LVT ..... *Local Vector Table*
MALWARE ..... *Malicious Software*

ME ..... *Management Engine*
MITM ..... *Man-In-The-Middle*
MMU ..... *Memory Management Unity*
MSR ..... *Model Specific Register*
MTF ..... *Monitor Trap Flag*
NDIS ..... *Network Driver Interface Specification*
NMI ..... *Non-Maskable Interrupt*
NPT ..... *Nested Page Table Translation*
NX ..... *No-Execute*
OS ..... *Operating System*
PAE ..... *Physical Address Extension*
PCI ..... *Peripheral Component Interconnect*
PE ..... *Portable Executable*
PEBS ..... *Precise Event Based Sampling*
PMI ..... *Performance Monitoring Interrupt*
PXE ..... *Preboot eXecution Environment*
ROP ..... *Return Oriented Programming*
SGX ..... *Software Guard Extensions*
SMI ..... *System Management Interrupt*
SMM ..... *System Management Mode*
SMRAM ..... *System Management RAM*
SOC ..... *System On a Chip*
STM ..... *Software Transactional Memory*
SVD ..... *Singular Value Decomposition*
SVM ..... *Secure Virtual Machine*
TCB ..... *Trusted Code Base*
TLB ..... *Translation Look-Aside Buffer*
TPM ..... *Trusted Platform Module*
TSC ..... *Time Stamp Counter*
TSX ..... *Transactional Synchronization Extensions*
USB ..... *Universal Serial Bus*
USB-HC ..... *Universal Serial Bus-Host Controller*
VM ..... *Virtual Machine*
VMCB ..... *Virtual Machine Control Block*
VMCS ..... *Virtual Machine Control Structure*
VMI ..... *Virtual Machine Introspection*
VMM ..... *Virtual Machine Monitor*
XD ..... *Execute Disable*

Contents

Chapter 1

Introduction

Motivation

In today's world, computer systems mediate most of our interactions, with either physical or digital media. In this scope, the proper working of such systems is always worrisome. One of the main threats against computer systems is malicious code (also referred to as malware—malicious software), since it directly affects systems' capabilities.

Malware can compromise systems—by luring users or exploiting vulnerabilities—and thus cause a wide range of damage, from data leaks to financial losses. As the technology is widespread, malware has potential to reach a significant fraction of the world population.

As an example of damage extension caused by malware, FEBRABAN—the Brazilian Bank Federation—estimated losses of around US$ 500 million in 2015 [62, 81]. RSA researchers, in turn, estimated losses of US$ 3.8 billion on Brazilian bank payment bills (a.k.a. *boletos*) [104]; according to the FBI, the Cryptowall ransomware has yielded US$ 18 million from its victims [188].

The usual way of handling malware is through analysis procedures, in which a given sample is run in a controlled environment in order to provide to the analyst information about its behavior [179]. By relying on collected data, security professionals are able to understand attacks, develop patches, vaccines and other countermeasures.

The effectiveness of incident responses depends on the capabilities of sample analysis , since a malware will keep infecting users if no countermeasure is taken. However, the analysis of an unknown piece of code may fail due to a lot of reasons: corrupted files, missing components and anti-analysis techniques.

Anti-analysis techniques are pieces of code intended to detect if a process is running under an analysis environment. If so, the execution is terminated and the analysis is unable to proceed. Some samples also rely on split personalities, in which, on detection of an analysis environment, the malicious behavior is averted and only mild or completely benign behavior is observed. This way, as no countermeasure could possibly be developed, the piece of code keeps threatening users.

Given this scenario, concerns about the samples that do employ some kind of anti-analysis trick arise naturally. A previous work of mine [28] analyzed more than 25 thousand malware samples operating in the Brazilian cyberspace between 2012 and 2015 and allowed observation of the first signs of evasive malware usage, which indicates armored

malware analysis may become a problem for our country in the near future.

The graph from Figure 1.1 shows how distinct anti-analysis technique usage evolved over time. In the 2012 to 2014 period, all techniques showed growth, which may indicate a tendency for future years.

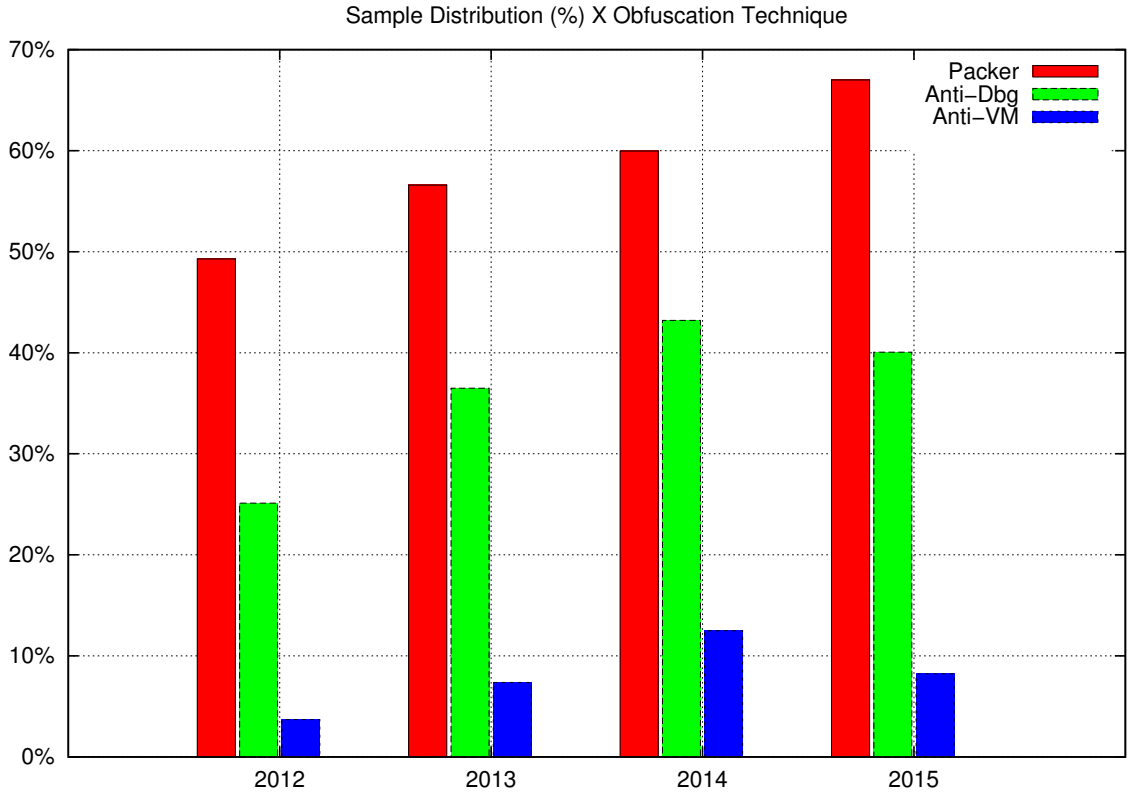

Figure 1.1: Anti-analysis technique usage evolution by sample (A sample may use more than one at a time). Source: [28]

A particular set of techniques of interest are those related to virtual-machine evasion. By looking at Table 1.1, we are able to identify its use in Brazilian samples. Although not prevalent yet, the knowledge of such techniques by some criminals is clear, so increased use is expected in the future.

Technique	**# Samples**
VMCheck.dll	2,729 (10.48%)
VMware *trick*	850 (3.26%)
VirtualBox Detection	306 (1.17%)
Bochs & QEmu CPUID *trick*	340 (1.31%)
VirtualPC *trick*	17 (0.07%)

Table 1.1: Identified anti-VM techniques and number of samples showing them. Source: [28]

In addition to the Brazilian scenario, many other anti-analysis techniques can be found

spread worldwide [30, 15], which was the main motivation to consider evasive malware is the primary concern of this work.

Recent studies about malware evasion have pointed that the key factor for analysis detection is that analysis solutions are not transparent—the analysis environment behaves in different ways in comparison to an ordinary user environment. If we could make those environments look more similar, samples would not have enough information to decide whether they are in an analysis environment or not.

In a more technical way, transparency is achieved by keeping samples' integrity and not introducing side effects. These two points are strong constraints for analysis solutions, since no code injection, hooking or emulation is allowed. As such, hardware-assisted approaches have appeared as an alternative way of fulfilling the transparency requirement, which was chosen to be this work's research line.

To start with, I present a comprehensive literature review on hardware-assisted techniques in the security context. This review provides a broad comparison of techniques, highlighting strong and weak points, and also pinpointing development gaps—security solutions which would benefit from hardware support. One significant gap I have identified is the use of performance counters for monitoring tasks. This way, I made a significant effort on developing their use on broader scenarios—for debugging and tracing—than they were originally designed to work with. Both contributions are presented as a paper collection, whose full content is presented in the next chapter.

Objectives

My main goal in this dissertation is to evaluate the use of transparent solutions to handle anti-analysis-armored malicious code. The literature review, presented further ahead, has revealed a gap in the use of performance counters for such purpose, thus I dedicated a great effort on studying this mechanism. This way, in general terms, my goals are presented as questions to be answered during the research development:

- **Could I develop a performance-counter-based malware analyzer?** This topic aims to answer whether the branch-collected data along with introspection procedures are enough to analyze malware and what limitations they present.
- **Could I isolate processes' actions?** This topic aims to answer whether the branch monitor mechanism can isolate malware actions from its system-wide collected data.
- **Is the conceived solution's overhead acceptable?** This topic aims to answer whether the added introspection procedures make the analysis still feasible when compared to the state-of-the-art tools.
- **Could the solution run in real-time?** This topic aims to answer whether the added introspection procedures allows for real-time monitoring or is limited to offline analysis.

- **Is the final solution transparent?** This topic aims to answer whether the data collection required for introspection interferes with analysis' transparency or not.
- **Is the solution easy to implement?** This topic aims to answer what the development cost is when compared to state-of-the-art solutions.
- **Is the solution portable?** This topic aims to answer whether the adopted introspection procedure can be ported to other systems or not.

My specific goals are the following:

- **Is Call Graph (CG) reconstruction possible?** This topic aims to answer which level of granularity I am able to achieve when handling function call data.
- **Is Control Flow Graph (CFG) reconstruction possible?** This topic aims to answer which level of granularity I am able to achieve when handling executed instruction data.
- **Could I develop a Debugger?** This topic aims to answer whether my solution can be used for real-time inspection or not, regarding inspected-state consistency.
- **Does the solution handle Return-Oriented Programming (ROP) attacks?** This topic aims to provide a comparison with existing branch-based monitoring tools.

Contributions

The contributions of this dissertation are those from the included papers. In order to make my claims clearer, I summarized them below.

The contributions from the paper "Who watches the watchmen" are the following:

- I present current state-of-the-art malicious code attacks and current tools' limitations.
- I claim the need for transparent systems to overcome current tools' limitations.
- I introduce the working mechanism of hardware-assisted, transparent approaches to a range of security tools.
- I present security tools based on this mechanism.
- I evaluate this mechanism according to several criteria, such as protection, transparency, overhead and development efforts.
- I highlight existing development gaps in the evaluated techniques.

The contributions from the paper "Enhancing Branch Monitoring for Security Purposes" are the following:

- **Current Threats and Solutions scenario review:** I present a review of the threat landscape scenario and current countermeasure and analysis tools, discussing their advantages and weaknesses. Specifically, I review transparent analysis solutions as well as branch-based ones.
- **Branch monitoring framework:** I propose a complete, modular framework based on hardware monitoring features, allowing for further applications that overcome current and future state-of-the-art limitations and weak points.
- **Transparent malware analysis:** I leverage my framework to build a transparent malware analysis tool with lower development efforts than the current state-of-the-art ones. As far as I know, no other malware tracer is based on this kind of monitoring.
- **Transparent debugger:** I demonstrate how to implement granular debugging based on my framework without using single-step flags. Again, I have no knowledge of other debugging solutions based on branch monitors.
- **ROP attack detector:** I present an improved implementation of current ROP detection heuristics, based on my framework, which does not require code injection and is a limitation of other approaches.
- **Hardware Improvements:** I suggest possible hardware enhancements for branch monitors based on the challenges I faced when developing my solution.

Publications

While the research was being pursued, some work have been accepted or submitted for publication, although not all of them were included in this dissertation. However, as each work contributed to the dissertation one way or another, I think it is reasonable to give them a mention. Below I discuss each one's implication for this dissertation.

- The paper "The Other Guys: Automated Analysis Of Marginalized Malware" was published in the "Springer Journal of Computer Virology and Hacking Techniques" (JCVHT) [26]. This paper is the result of a previous research and the presented sandbox was used as a ground-truth for detecting evasive samples. It also presents the sandbox mentioned in the TOPS article.
- The paper "One Thousand and One Nights: Brazilian Malware Stories" was also submitted to the JCVHT [28]. This paper is the result of a previous research and its threat review is used as underlying support for the Brazilian scenario landscape presented in the dissertation's introduction.
- The paper "Who watches the watchmen: A review of techniques, tools and methods to counterfeit anti-analysis techniques on modern platforms" was submitted to "ACM Computing Surveys" (CSUR) [29] and is embedded in this work as part of my contributions. This paper may be understood as my literature review and critical evaluation of the current hardware-assisted security scenario.

- The paper "Enhancing Branch Monitoring: From Control Flow Integrity to Malware Analysis and Debugging" was submitted to "ACM Transactions On Privacy and Security" (TOPS) [27] and is embedded in this work as part of my contributions. This paper may be understood as my final solution proposal for the stated problem. Preliminary results from this work were published in the "XVI Brazilian Symposium on Information and Systems Security" (SBSEG) in the form of the following articles: "VoiDbg: Projeto e Implementação de um Debugger Transparente para Inspeção de Aplicações Protegidas" [25]; "Detecção de ataques por ROP em tempo real assistida por hardware" [24]; "Análise Transparente de Malware com Suporte por Hardware" [23].

Background

Along the referred papers, I have assumed some concepts are known to the reader. In order to allow any reader to follow this text, I define them here.

Semantic Gap

The semantic gap problem is the information representation difference on distinct levels. As an example, consider the differences between programming languages and assembly code. As can be noticed, the first is richer than the second, given that it provides, for instance, independent, context-related variable names whereas the second works over context-independent, shared registers. The semantic gap problem can also be seen in other contexts, such as computer forensics. When handling memory dumps, data is seen as a byte-array whereas at the OS level bytes are seen as parts of complex context structures, such as processes, descriptors and handlers. It is important to notice that such representation difference is intrinsic to the distinct technologies involved and as such is present along this work, since we are interested in hardware solutions to monitor OS-level structures.

Introspection

Considering the stated semantic gap problem, we notice that fully bridging such gap, thus having the same representation, may be theoretically impossible, since it is technology-inherent. However, some information may be retrieved. As an example, low-level instruction addresses can be mapped to libraries' base address when these are known. Similarly, memory-dumped bytes can be interpreted as structs when these are known. This information recovering process is often named introspection, since one is outside looking to an inside structure, for a given representation level. Introspection procedures are discussed along this work, since all hardware-assisted approaches have to bridge this gap to some extent.

Outline

This dissertation is based on a two-paper collection, presented in Chapter 2 as they were submitted for publication. Chapter 3 summarizes my results and presents a discussion of their implications. Finally, I draw my conclusions and present future work in Chapter 4.

Chapter 2

Papers

Who watches the watchmen: A review of techniques, tools and methods to counterfeit anti-analysis techniques on modern platforms

Publication: This paper was submited for publication to the ACM Computing Surveys (CSUR)

Marcus Botacin[1], Paulo de Geus[2], André Grégio[3],
(1) University of Campinas
Email: marcus@lasca.ic.unicamp.br
(2) University of Campinas
Email: paulo@lasca.ic.unicamp.br
(3) Federal University of Paraná
Email: gregio@inf.ufpr.br

Abstract

Malicious software are still threatening users on a daily basis and their evolution goes from social-engineering-based bankers to advanced persistent threats (APTs). Recent research and discoveries have presented us to a wide range of anti-analysis and evasion techniques, in-memory attacks, such as Returned Oriented Programming (ROP), and systems subversion, including BIOS and hypervisors. This work presents a survey on techniques able to detect, mitigate and analyze these kinds of attacks, which require transparent and fine-grained environments as analysis resources. We cover current tools' limitations, such as not being fully-transparent, and introduce systems and techniques to overcome and/or mitigate these constraints. The work presents approaches based on hypervisor introspection, System Managment Mode (SMM) instrumentation as well as some hardware-based ones. We also present some threats based on the same techniques. Our main goal is to give to the reader a broader and more comprehensive understanding

of recently-surfaced tools and techniques.

Introduction

Malicious software (also known as malware) attacks are one of the main computer system security threats and their importance and spreading continuoe to grow. Reported data[9] shows malware dissemination exceeds rates of 60 thousand new samples per day, with malware for mobile devices having had a particular growth[189]. In all, losses amounted to around 200 million dollars in 2016's first quarter[63]. In order to handle security incidents generated by malware actions, researchers have proposed a lot of tools to mitigate (Intrusion Detection Systems), prevent (Intrusion Prevention Systems, packet filters), remedy (antivirus, vaccines) and analyze (sandboxes) such malware. But more than growing in number, malware also have been growing in complexity, being able to bypass filters by applying polymorphism, evade sandboxes by detecting virtualized environments and even to subvert whole system operation by taking hypervisor control on virtualized systems.

As threats have been evolving, security has been taking a reactive posture and engaging the opposing sides into an arms race. Some new tools were recently proposed in order to handle new security issues related to modern malware, but a clear scenario/panorama is yet to be formed, which motivates us to work on this survey.

Most new approaches are based on techniques to stealthly and transparently acquire data from analyzed systems, notably System Managment Mode (SMM) and Hardware-Assisted Introspection, which are explored in detail in further sections. Having this kind of system introspection allows the analyst to get a fine-grained view of samples, even from those employing common evasion techniques, given that the systems are hardware-supported and run on the most priviledged ring, as is desired for reliable experiments[159]. Moreover, these systems natively handle kernel data, which allows for rootkit analysis and detection, a recurrent limitation from analysis systems running on the guest OS.

This survey is not limited to sandbox analysis, rather also covering forensic procedures that can be performed with these new techniques and allow live-procedure without malware sample-awareness. Besides, we review whole system monitoring tools and present tools for system and hypervisor integrity, which are on the frontend of new malware attacks. In addition, we extend our overview to general attack classes when it is allowed by the investigated solution. In summary, we aim to provide a deep understanding of hardware-assisted isolated execution environments, as stated by Zhang [222].

As these techniques are still in their early uses, new threats to them are also still evolving, leading to a dangerous scenario of stealth and Operating System-independent threats, which we deem important to provide a current overview. Finally, we will present applications built with these underlying techniques and related work that might lead research in that area.

This work is organized as follows: Section 2.4 presents current monitoring tool approaches and their limitations, with special emphasis on their drawbacks and how malware are exploiting them. Section 2.5 describes the hardware-assisted hypervisor, its inner workings, current threats and how tools can be deployed using current processor support

for virtualization. Section 2.6 introduces the use of this priviledged processor mode to get a complete system view for both malicious and legitimate use cases. Section 2.7 presents the concepts of privileged rings and isolated execution environments. Section 2.8 presents some hardware-design tools to achieve system independent monitoring. Section 2.9 introduces complementary approaches to perform system introspection and behavior analysis. Section 2.10 show tables that aim to summarize the presented approaches. Finally, in Section 2.20 we present our conclusions and some directions for the future.

Initial Approaches and their limitations

Considering the limitations from static analysis[129], dynamic monitoring is the most effective technique for system monitoring and analysis, since the extracted behavioral information may support a diversity of incident-response measures[84]. However, the quality of these measures is directly affected by the quality of the dynamic analysis data extraction process. If a malware could evade the analysis process or subvert some monitoring mechanism, the incident response effort would be ineffective. Handling evasive and subversive malware is one of biggest challenges system monitors face nowadays. The particular challenges each monitor type faces are discussed below.

The survey by Egele et al. [59] points to analysis that can be done by instrumenting the monitored binary or system. Binary monitoring usually employs some kind of code injection, such as Microsoft detours or DLL injection, in order to instrument the binary. The injection may be detected by a malware that checks its memory integrity, thereby leading to an analysis evasion. System instrumentation may be added in or outside the monitored environment. When monitoring from inside the environment, the approaches often involve kernel changes by SSDT/IDT hooking, installing kernel callbacks or filters or wrapping handlers. Its main drawback is its uneffectiveness against rootkits, which run on the same privileged level and are able to perform the same kind of flow redirection in order to subvert the system, and in the process detecting the analysis monitor.

The system can be externally instrumented by performing a Dynamic Binary Instrumentation (DBI) or a virtual machine instrospection (VMI) procedure. DBI may be accomplished in either kernel or user-land, allowing for a fine-grained analysis at the instruction level. DBI's use in security was made popular by DynamoRIO[58] and PIN[79], with lots of tools built on top of it. However, execution in a DBI environment may be detected by executing an instruction whose translation turns into an abnormal behavior.

In order to overcome such limitations, some proposed work have the ability to translate some known failure-prone instructions into surely successful ones, notably VAMPIRE[196] and SPIKE[198], which allowed for stealth-fine-grained tools such as Cobra[197]. However, this approach has the disadvantage that all known fail-prone instructions have to be translated, which is not only impractical[87] but also prone to evasion by a malware that employs a new and unknown "evasion trick".

VMI, in turn, provides a native whole system view and fine-grained analysis capabilities through instruction-level inspection, being suitable for malware analysis, security policy enforcement and integrity checks. VMI-based techniques have also the advantage

of being able to monitor systems that prevent or restrict in-system monitoring, such as 64 bit Windows Kernel Patch Protection (KPP)[109], Many tools were build with the underlying VMI, like Anubis[16, 83] on top of QEMU[17], for malware analysis and its Danubis[134] version for drivers, exploring whole-system view capabilities. Bitblaze[182] and Virtice[152] are built on top of TEMU, a tiny version of QEMU, and provide a complete framework for malware analysis and system inspection. VMI has become popular to the point of having automatic instrumentation tools, such as Libvmi[100].

However, external monitoring approaches, such as VMI, have a major drawback, which is the gap between high-level (what is running at the O.S. level, e.g. a file opening) and low-level information (the machine instruction sequence being processed), also called semantic gap[127]. Despite some proposed automated techniques[64, 167, 164, 56], handling the semantic gap is still considered hard and requires specific knowledge and big development efforts. The semantic gap grows as one gets deeper into abstraction levels. Figure 2.1 shows abstraction levels for techniques presented in the following sections.

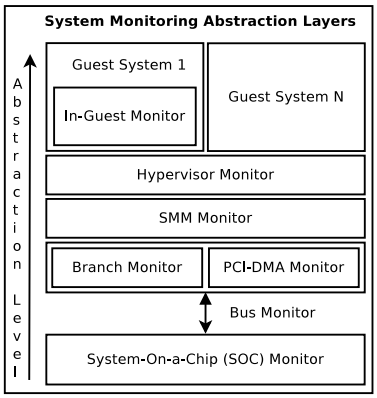

Figure 2.1: Abstraction levels for distinct monitoring techniques.

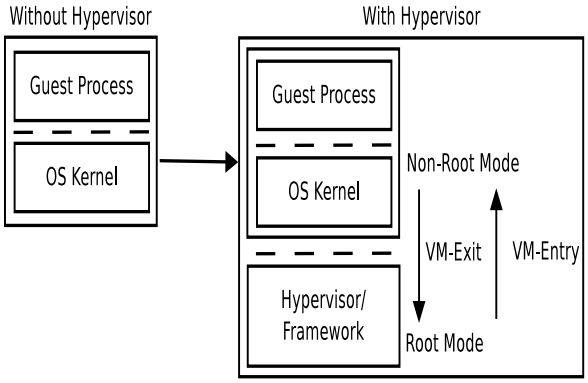

Figure 2.2: VM operating layers.

While possibly being able to bridge the semantic gap, VMI is not perfect since malware can detect virtualized environments in order to evade an analysis procedure[37]. Given that most VMI systems run on top of an emulator, such as QEMU, malware can detect the virtualized environment by testing instruction behavior, which often differs from the ones presented on a real CPU. Martignoni et al. [107] describe a method, called Red Pill, of fuzzy-testing a CPU emulator and being able to generate tests for emulator identification, soon extended by Shi et al. [177]. In addition, Paleari et al [140] developed a way of automatically generating red pills.

Although research efforts have been done to overcome these challenges, proposed solutions are very expensive, since they either require execution on multiple environments, such as BareCloud[89] and Splitmal[13] or require a physical machine, such as Barebox[88]. In order to overcome all these limitations, virtualization has to be made more transparent, something that can be achieved by hardware-assisted virtualization or, to go deeper into instrospection, by using SMM or similar.

Hardware Assisted Hypervisor-Based Approaches

This section focus on techniques based on hardware-assisted hypervisor (HVM) monitoring. Initially, we introduce the concepts of HVM operation, focused on the x86-64 architecture and covering implementations over both Intel VT-x[78] and AMD-v/SVM[5] platforms. We follow by introducing tools and applications developed over these platforms.

HVM background

Apart from 32 and 64-bit addressing modes, x86 and x86-64 CPUs have 3 operating modes: *Protected Mode*, which is the processor's native mode; *Real-Address Mode*, which extends the previous mode; and *System Management Mode*, covered in details in Section 2.6.

Virtualization instructions augment the CPU instruction set by adding two new operating modes, *root* and *non-root* modes, in a way so that guests and the host (VMM) are associated with the non-root and the root modes, respectively. The transitions from non-root to root mode are known as *VM-EXITS*, which work like an exception or a trap, but the set of spanning actions can be dynamically configured. After handling these events, the execution is resumed through *VM-RESUME* or *VM-ENTRY*. Figure 2.2 shows an overview of these new modes and events.

The extended instruction set offers many features to help a hypervisor implementation and improve system performance. Given that the new memory management unit can be used to track memory uses and guest registers are directly accessible from the hypervisor, we can implement a variety of security-oriented techniques that enforce policies. However, the main advantage offered is the capability of running code directly on the processor, with no need for instruction translation. This is particularly desirable for malware analysis and often a drawback for DBI-based systems, for instance, subject to instruction translation side-effects (Section 2.4). Since this analysis system is not susceptible to CPU emulation bugs, a malware running on such a system is not able to identify whether it is running inside a virtual environment or not, achieving the transparency requirement.

An important change when using virtualization instructions is the memory controller. There are different monitoring mechanisms according to the virtualization platform used, Intel or AMD, since the AMD's implementation has the memory controller on-die whereas Intel's has it externally. However, the main change is due to the double address translation mechanism. In a traditional hypervisor, guest virtual addresses are translated into guest physical ones, which are the host's physical ones. Intel and AMD have deployed techniques called Extended Page Table (EPT) and nested page table (NPT), respectively, that add an additional translation layer. On these systems, guest virtual addresses are translated into guest physical ones, but contrary to the previous implementations they are further translated before getting to the host's physical address lines. This process can be seen in Figure 2.3.

Knowledge of this mechanism is important since this second translation level could be instrumented so that memory access is monitored through translation-faults. However, such memory monitoring is not enough to cover the system as a whole, given that these

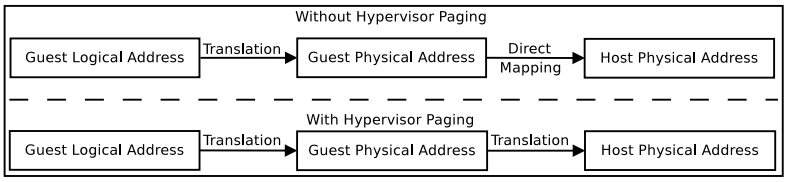

Figure 2.3: VM memory operation.

are CPU memory accesses; Direct Memory Access (DMA) may also happen and those are external to the CPU. DMA monitoring is enabled by another mechanism called IOMMU that intercepts Input/Output (I/O) actions. By following both approaches a more robust monitoring may take place.

A Hardware-assisted Virtual Machine (HVM) is configured through special control structures named Virtual Machine Control Structure (VMCS) and Virtual Machine Control Blocks (VMCB) on Intel and AMD systems, respectively. These blocks include the initial system state, memory allocation and vm-exits configuration. An advantage of using this technique is that the system does not need to be booted up in a virtualizated environment, but can be conveniently moved to one at runtime, which is called *late launch*. Late launch broadens the options for analysts, such as live-forensics, since it does not require reboot or shutdown. In the late launch case, the initialization blocks are set to the current system state, but this does require a driver to set specific registers at a privileged execution level.

When aiming to implement a security framework based on HVM, the usual approach is to instrument the hypervisor layer in root mode to collect information from non-root mode and then send them to an external client running on the host userland. In this case, the abstraction is similar to the one used in Operating Systems (OS): the VM monitor running in kernelspace and the analyzer in userspace. Even though the client is stealth for a malware, the hypervisor itself is not, given that the malware could map its memory to the hypervisor one and so find the monitor. A solution for that is to put the instrumented framework in a page without malware access, which is done through the new presented memory mechanism and its fault handler. Even more than the hypervisor analyzer, the internal driver could also be detected. To counter that, a tool may employ rootkit techniques to hide itself from the system or to use the same method to map the HVM-loader driver to a protected physical page.

The challenges faced when developing such kind of tool vary according to its monitoring goal: i) to register monitored values, as presented, is straightforward, since monitoring memory access can be done through translation faults and IOMMU; ii) system call tracing requires bridging the semantic gap by employing taint tracking and event static stack analysis; iii) finally, breakpoints and step-by-step execution require a more sophisticated approach.

Breakpoints are usually classified as software and hardware breakpoints. Software breakpoints are not transparent since they modify instruction bytes. Hardware breakpoints are limited in number and are shared between host and guest. Another way of implementing it is then required. Step-by-step execution has the same sort of limitation. The way tools overcome these limitations is to monitor the system by raising exceptions,

such as hooking the memory management unit to set a given page as read-only, thus causing a page fault at each execution, or to set performance counter registers to their maximum value in order to raise an overflow exception at each running step.

The next sections present practical tools and solutions developed using HVM, starting with an introduction to some threats based on this approach.

HVM threats

Given the transparency and system-wide view of HVM systems, as presented in previous sections, exploiting a system using this technique is a straightforward thinking. It is hard to say who was the first person to propose such application for HVM, since most releases happened in underground hacker forums. However, undoubtedly, the first famous one was the BluePill rootkit[160].

Bluepill is implemented on Windows Vista using AMD technology and is able to perform late launch; a network backdoor, with no need for NDIS modification, is shown as use case. Other examples appeared, like the HVM Rootkit[133], which is AMD based and targets Windows XP machines. This tool takes a multi-core approach, setting up each core for an HVM. Its driver loading routine employs the physical page mapping for stealthiness.

In fact, in-guest, ordinary kernel rootkits are stealth enough against casual analysis, but to remain stealth before a specialized forensic procedure requires HVM-based ones. Despite none of these tools being a complete, definitive solution, similar ones have been improved by both analysts and criminals.

Malware Analysis

The previous HVM rootkit strategies can also be used for well-intended security purposes, such as malware analysis and debugging. It is not easy to identify this approach's emergence, as most of the poorly documented rootkits mentioned in the latter section have already proposed such instrumentation as a countermeasure. Presentations at hacker conferences have also shown incipient insights on HVM use for malware analysis.

A deeper understanding of HVM for malware analysis uses was provided by Dinaburg et al. [54], establishing formal foundations to attain transparency, from requirements—"higher privilege, no non-privileged side effects, identical basic instruction execution semantics, transparent exception handling and identical measurement of time"—to ways of fulfilling these requirements. The developed tool, Ether, is implemented as a Xen patch and runs Windows XP guests. The Analyzer architecture is client-server based and no internal code is required. Fine grained analysis is enabled by setting a trap flag for each instruction, causing considerable overhead. Memory access is traced by fault trapping on shadowed page tables. The same trap mechanism is employed to handle system calls, both on the new, fast syscall mechanisms and on the deprecated sysenter mode. System data is retrieved by classical VM instrospection techniques. In order to remain stealth, the Ether hypervisor changes PUSHF instruction behavior, since this instruction could detect the trap flags for single-step execution. It also changes the Timestamp Counter

(TSC) in order to avoid being detected through timing attacks. The study cases provided (a syscall tracer and an unpack tool, using shadow memory write tracking) show the tool's effectiveness against evasion tools available then. Despite its results, Ether is not a perfectly transparent tool, since some ways of fingerprinting it are known[145]. However, most of them are overcome by applying patches or by new VM extensions of modern processors. Moreover, Ether was a sound step towards being ahead of evasion tricks of its time.

CXPInspector[210] leverages Intel VT-x support on KVM to perform malware analysis on 64-bit Windows 7. Analysis challenges faced on 64-bit Windows Kernel include handling Address Space Layout Randomization (ASLR) and overcoming Kernel Patch Protections hook limitations, for which VMI is an alternative. In order to reconstruct high level semantics, it bridges the gap by using debug symbols and parsing PE executables. Branch addresses are reconstructed by using hardware branch facilities (discussed in Section 2.9). It also changes the TSC to prevent timing attacks. In addition to malware analysis capabilities, CXP is also able to perform application/system profiling by measuring the execution time spent on each memory page.

The CXPInspector also presents a more fine-grained concept for memory handling, named Currently eXecutable Pages (CXP), which allows for multiple scopes and granularities of the analysis. The three CXP granularities are: one memory region, a set of memory regions or one single memory page. By capturing transitions and flows among such CXPs, the system can trace events. In pratice, it is a way of implementing memory traps based on EPT or NPT facilities. CXP provides study cases of the Purple Haze 64-bit rootkit analysis and a profile of the Apache web server and its modules.

As an evolution of the ideia of analyzing malware on HVM, authors started to care about developing an *ad-hoc* malware analyzer VMM. The main reason behind it was reducing the Trusted Code Base (TCB), which is the code that should be trusted *a priori*. General-purpose VMM implements much more features than the one required for malware analysis, such as virtual devices and plenty of drivers. As is known, larger numbers of lines of code tend to generate more bugs, which increases the opportunity for malware evasion.

Given the above, Nguyen et al. [136] presented MAVMM, a malware analysis lightweight VM hypervisor. The TCB is reduced to 4K lines of code on MAVMM in comparison to millions of lines on well-known VMMs like Xen and VMWare. MAVMM is implemented using AMD-v instructions and runs a Ubuntu Linux as guest system. The hypervisor is loaded at boot time, in contrast to the late launch approach of Ether and others. The memory is protect using nested page technology. The tool is able to extract different features from the system, such as instructions, syscalls and memory accesses, by leveraging single-step execution and handling VM-Exits. It has 2 modes of feature extraction: Full and Compact modes. The first is a single step extraction whereas the second reduces the number of VM-Exits to a set of pre-registered reasons, achieving a significant analysis speed up. Authors pointed different possibilities of tool usage, such as syscall tracing and sample unpacking. The study case provided is a Linux rootkit analysis. The tool is also transparent as the HVM is employed. However, we should consider how likely to be fingerprinted and detected the environment is, as the hypervisor neither provides virtual

devices nor supports multiple guests.

More than just building analysis tools themselves, some authors have employed such tools for analysis improvements. Quist et al. [151], for instance, proposed using a modified version of the Ether HVM to improve AV detection accuracy. The work adds to Ether features for "deobfuscation: section and header rebuilding as well as automated kernel virtual address descriptor import rebuilding". With these repair mechanisms, AV showed detection rate improvements as high as 45%.

Malware Debugging

Beyond automatic execution tracers, complete debuggers were also implemented using HVM. Fattori et al. [61] presents a complete HVM framework which allows for tools to be built on top of it. Besides late launch, the framework also allows for online unloading and error handling, such as exceptions, thus being fault-tolerant. The framework is implemented using Intel VT-x on Windows XP. Its architecture is also client-server based, where low-level server information is translated into high-level semantics and delivered through a well-defined API. It allows for tracing function entries, syscalls, process switching and I/O operations. As MAVMM does, the system can also limit the tracing scope through a restricted mode. Traces can be filtered on function arguments, specific I/O ports and other features. The tool applies classical introspection techniques, including using Windows-provided debug symbols, in order to retrieve process and function names. Memory protection is achieved through extended page tables. For event tracing, the framework uses two different approaches: Exceptions are handled through hardware VM-EXITS whereas other mechanisms are handled through page-fault trap technique. This technique allows for transparent software breakpoints and watchpoints, without the need of hiding the trap flag, as in Ether. It also allows for syscall tracing, sice the syscall dispatch table is also trapped. The same happens for memory mapped I/O.

By making use of the framework presented above, the same authors introduced HyperDBG, a kernel debugger built on top of that framework. It boasts the same widespread functions of kernel debuggers, such as breakpoints, register inspection and tracebacks, supports guest write access and includes a hypervisor-based graphical user interface and hotkey support. Due to its self-contained implementation, HyperDBG is able to debug any kernel component, including components used on its GUI, such as the keyboard.

SPIDER [52] is a KVM implementation of hypervisor-assisted invisible breakpoints applied to Windows XP and Ubuntu systems. It allows for a complete system monitoring, including data breakpoints and process creation. It presents a client-server architecture, with its client implemented on top of the qemu-kvm one. It also prevents timing attacks by changing the TSC. The approach tries to combine the flexibility provided by software breakpoints but handling its instruction changes and control-flow-deviations side effects using the hypervisor MMU layer for spliting the data and code views, which causes a lower overhead than trapping each instruction, like Ether does. Code splitting is performed by setting the same virtual pages to two different physical pages, having each one mutually-exclusive attributes Reading or Executing. The data splitting decision is supported by the TLB separation on iTLB and dTLB in the x86 architecture, which reduces the number

of EPT violations. SPIDER monitors the system using the monitor trap flag (MTF), which causes a VM-Exit on each instruction. The SPIDER breakpoint handler checks if the current instruction is an ordinary (guest-set) or an invisible breakpoint, acting as a pass-through or a breakpoint hider, respectively. In the latter case, it will "clear the breakpoint and restore the first byte of the instruction that had been replaced". Whenever a write occurs, it happens on a data page, which means that self-modifying code is forcibly not allowed. This could lead to incorrect execution or even to malware evasion. In order to handle this situation, SPIDER synchronizes data and code pages after any change on them. The study cases presented show how SPIDER improves the tamper-resistance of the BEEP [98] *attack provenance system* and how SPIDER could be used to inspect instant messaging programs by inspecting them before the messages were encrypted.

A Combined Approach

Considering that, despite being transparent, HVM is an expensive approach, since it is harder to implement than in-guest monitors and has a greater performance penalty, and considering that emulation and DBT, despite being easy to instrument, are not transparent, a solution that connects both is desirable. Aiming to bridge this gap, V2e [215] presents a combined approach of capturing data on an HVM and precisely replaying it on an emulator. It is implemented on an HVM-KVM and replayed on TEMU, from both Linux and Windows XP as guests.

The most challenging task for such kind of implementation is to achieve a balance between captured data and replay feasibility (considering speed, precision and costs). For such goal, authors have introduced a formal definition of how a replay should look like. For the capture function, the HVM system implementation is based on using EPT/TDP/NPT for partitioning the memory space on mutually exclusive recorder and recorded pages. Besides, both TSC and DMA accesses are recorded. TSC is also changed in order to prevent timing attacks. The architecture is a client-server one for logging in userspace. Semantic gap bridging techniques are used for data parsing. For the replayer function, a conventional emulator has a series of problems in comparison to a real CPU. It uses block translation—a paradigm absent on real CPUs—, lazy flag calculation, translated code reuse and TSC redirection for host values. In order to overcome such limitations, V2e changes 3 instruction category behaviors: general-purpose (branches, data transfer and integer math), floating-point operations (FPU) and others. For the general class, lazy flag calculations are disabled; for the FPU one, the flags are passed to the host CPU by directly launching an assembly call; for others, they are turned into NOPs. Morevoer, as the same page table mechanism used for capturing is required for replaying, a software emulator one was developed, called physical page container. The study cases provided are the adore-ng and 12 other real-world malware samples.

A similar approach is implemented by Kang et al. [87] replaying Ether instructions on TEMU. This approach allows not only for executing evasive malware but also determines the points where the behavior differs between the reference and the emulated platforms. By analyzing divergence causes, the tool performs a dynamic state modification (DSM) that attributes "new values to specified execution state components, such as registers,

memory and so on, which represent a transient alteration to values during the samples' execution". Using this technique, however, does not remove the anti-emulation check. The tool aims to ensure the resulting modifications are robust—still effective when the same sample is executed with distinct inputs.

HVM for Security Policy Enforcement

Empowered by HVM system-view capabilities, other solutions were designed to enforce security policies, of which an important one is to enforce I/O policies, since some attacks such as information leakage rely on bad I/O policies.

To this end, Shinagawa et al. [178] proposed BitVisor, a single para-passthough[1] VM that can enforce I/O security policies. It consists of a specific-purpose hypervisor which implements only essential I/O drivers, reducing the TCB. Bitvisor works on a single VM guest as authors claims desktop users run only one system at once. Due to this claim, it has no need for VM-isolation, which also helps to reduce the TCB and the overhead in general. The para-passthrough approach requires intercepting only essential communications, such as those required for protecting the hypervisor and to enforce the policy itself. Hypervisor memory is protected through a BIOS hook that unmap critical regions. The tool is implemented using Intel VT-x on WIndows XP, Vista and Linux. As the system is pass-through, there is no active component on the guest system. In order to correctly enforce I/O policies, Bitvisor has to handle 3 different types of I/O routines: programmed I/O (PIO), memory mapped I/O (MMIO) and DMA. PIO are the IN and OUT instructions, handled by the VT-x port bitmap. which allows for intercepting specified ports. It has also to intercept PCI PIO in order to handle port remapping. MMIO device registers are mapped on memory regions, in a way shadow pages are suitable for such interception. DMA interception is handled by a new technique called shadow DMA. Modern systems use what is called DMA descriptor, a memory region in which DMA controls are mapped. However, monitoring such region is not effective since DMA memory accesses themselves occur in parallel. In order to overcome this situation, Bitvisor states a shadow DMA descriptor page, mapping the DMA descriptor in hypervisor memory and, after copying those blocks to the DMA controller buffer, performing a Man-In-The-Middle approach against the DMA controller, which allows access to all DMA communication. By using these monitoring processes, the authors present a case study of an ATA Host Controller, which enforces automatic storage encryption.

HVM for Attack Prevention and System Integrity

Besides malware analysis, HVM may also be employed to apply security policies, detect known attacks, avoid sensitive information leakage and assure system data integrity. This section presents an overview of HVM applied to achieve such goals.

In fact, techniques using VMM are already known for a long time. Kernel Guard[155], for example, is a framework for handling dynamic kernel rootkit through memory access

[1] A minimal interposition mechanism responsible for I/O filtering

policies. Lares[143] extends Xen dom0 with a new secure VM TCB providing secure services for an unprotected guest. Osck[76] defeats kernel rootkits by checking control-flow integrity. NICKLE[156] leverages mixed-page techniques to ensure trusted-code execution. Finally, Overshadow[36] introduces the concept of multi-shadowing physical pages to protect applications from untrusted kernels.

These approaches, despite being theoretically correct, are not fully transparent, which may be a limitation, either by the need of implementing all memory and I/O management by software or by the possibility of a malware evasion through virtualized-system detection. In addition, software implementations present greater overhead as compared to hardware ones. This way, as soon as the HVM extension was launched, VMM approaches started making use of it.

Secvisor [173] protects the kernel by assuring that only approved code may run, which may protect the kernel against injections and even 0-day attacks. Its threat model assumes the attacker has control of all but the CPU, MMU and IOMMU. It is implemented on the Linux kernel by leveraging the AMD-v instructions, and since it is a specific-purpose tool, it has a small TCB. Kernel code modifications are needed in order to handle specific actions, such as module loading, which requires approval and is performed through a hypervisor call (hypercall). The need for a patch may be considered a limitation for some but allowing users to create their own policies may be a justifiable trade-off. Secvisor virtualizes and intercepts MMU and IOMMU, protecting against DMA and memory writes by using the AMD Device Exclusion Vector (DEV) feature from AMD processors. Hypervisor memory is protected by assuring that its physical memory pages are never mapped into the Protection Page Table. Approved code execution is enforced by memory virtualization. User memory is marked as executable in user mode but not in kernel mode, in a way Secvisor needs intercepting all transitions in order to adjust flags. Secvisor also ensures that switching to kernel mode will occur only by setting the `IP` "to an address within the approved code". Likewise, kernel exits should target only user-mode code, avoiding kernel-flow redirections.

HVM for Forensic Procedures

HVM also has advantages in comparison to ordinary VMM for forensic procedures. HyperSleuth[106] employs the late launch and unload capabilities to implement a complete forensic framework, for both online and offline inspection. On top of it, authors implemented a memory dumper, a syscall tracer and a lie detector[2]. Hypersleuth is implemented on Win XP and requires a driver for loading its VMM. Since the system is aimed to be loaded on compromised systems, it has to check whether the procedure was correctly done, given that a malware running on such system could subvert the procedure. This is done through a sequence of challenges and responses. The framework also cannot trust on OS network software, so it implements its own network driver in order to send captured data through the network.

Syscall tracing is performed as in Ether's way by trapping memory pages, since VT-x hardware does not support tracing fast syscall entering (SYSENTER), whereas the

[2]A system which checks integrity from multiple perspectives in order to assure data consistency

memory dumper is implemented in a lazy way. When using it, the memory is dumped in a dump-on-write policy, similarly to copy-on-write in the Unix fork, by setting non-copied pages to read-only. This ensures the contents of the stored dump is current as of the time it was requested.

The dump occurs each time a page fault is triggered and also when halting and performing I/O, with the client being responsible for reassembling the data. Another advantage of such technique is that the system is not frozen, which avoids network connections to timeout. Those timeouts could be used to identifying the monitoring taking place. As for the offline analysis, the dump is checked by using the Volatility tool.

The lie detector is implemented through a userland program that asks for system properties, such as processes running, and sends this information to the trusted host. The intention is to trigger malicious behavior, such as process hiding. At the time the system resource call is performed, this program makes a hypercall to the hypervisor that performs the same action. Given that the hypervisor does not trust the OS and implements its own discovery mechanisms, it can detect even a lie behavior. The results are compared on the trusted host. Since the system is changing at runtime, this verification is performed repeatedly at variable intervals, in order to avoid any time measurement attack.

VMI limits

Despite all the benefits HVM can provide, some limitations are technology-inherent. In this section, we discuss these limitations and how they affect security measures development.

Firstly, we should be aware that the transparency claims made by different authors are not totally supported by processor vendors. According to Pearce et al. [144], VM transparency would be achieved by accomplishing the three Popek law requirements: efficiency, resource control and equivalence. Some authors, however, talk about the unfeasibility of such implementations, such as Garfinkel et al. [66]. In fact, vendors do not make such claims and often do not consider transparency a major requirement, even though a desirable one. This way, an analysis system will be as transparent as the vendor can provide.

Secondly, HVM-support improvements may modify the effectiveness and the way different security techniques are applied. Lengyel et al. [99] presents common pitfalls when designing VMI systems, delving into some of these modifications. The first one refers to TLB splitting. The TLB is usually splitt into data (dTLB) and instruction (iTLB) sections. Newer CPUs have a third section called sTLB, which caches the evicted/flushed entries from the previous ones, resulting in a considerable performance boost. TLB uses on security are well known, either for defense or attack purposes. Grsecurity[70], for instance, employed TLB in order to implement page execution attributes before NX was launched, whereas the Shadow Walker rootkit[183] achieves its stealthiness through a TLB poison. In this kind of attack, the rootkit remains stealth by taking advantage of the fact that "a single virtual address can point to distinct pages, according to which TLB is being used". By leveraging this technique, it could, for example, evade an AV software, since the latter would be unable to scan the malware pages.

Lengyel et al. also points that an effective defense mechanism against such attacks is

to periodically flush the TLB, since this reduces the analysis-time opportunity window. Windows 7 and later versions implement this approach. In addition, sTLB also makes it difficult to tell the existence of a stealth rootkit, as the former "can only store one version of the evicted TLB entries". More importantly, the VMI may be employed to skip the sTLB. By marking pages as execute-only—a feature from the EPT mechanism—data from iTLB and dTLB will differ and CPU address translation will go through the primed page-tables again, restoring stealth techniques eventually used. Moreover, newer CPUs rely on tagged TLB, a mechanism that labels data in order to manage it, resulting in another performance boost. This way, the TLB is not flushed anymore, thus opening a new opportunity for stealth rootkits. Another evasion way is related to the EPT mechanism. When a VM-Exit occurs, its violation reason (Read, Write, Execute) is specified. However, only the start address is specified, thus "an attacker who is able to break the assumption that the violation happened at exactly the pointer location may evade an analysis". As far as we know, Intel is working on such limitation.

When building a VMI-based system we should also care about monitor triggering, since a passive monitor (snapshot-based) may be evaded by a timed execution. Wang et al. [201] present applications using this kind of evasion, which includes stealth file transfer and a backdoor.

Given the above, although HVM-based systems have raised the bar against evasive samples, general sandbox solutions may still be detected by advanced threats, like the examples shown by Brengel et al. [31]. A complete discussion of HVM evasion is provided by Pek et al. [146]. Although some evasion tricks have already been mitigated since then, either by introducing new hardware or by leveraging evasion-aware programming guides, that work provides a way of understanding how transparent machine vendors are implementing these capabilities.

SMM-based techniques

This section presents techniques and tools based on the System Management Mode (SMM). Initially, we present an SMM background, which includes its advantages and drawbacks. Secondly, we present tools and how they overcome SMM limitations to achieve whole system monitoring.

SMM Background

The System Management Mode (SMM) is one of CPU operating modes and is intended to be a mechanism for implementing system control features, such as power management. SMM operating abstraction is similar to the one presented for HVM; the system under monitoring executes in ordinary CPU modes (guest-analogous) and the SMM monitor (hypervisor-analogous) in SMM mode. An SMM enter is triggered by a System Management Interrupt (SMI) and exit by executing the `RSM` instruction.

The RSM exit instruction can only be executed in SMM mode, which protects the code, whereas the SMI may be triggered in a variety of ways, such as by PCI devices, writing directly on CPU pins, using a periodic timer or even ACPI/APIC interrupts.

Some events need to be rerouted in order to trigger an SMI, which can be done through the chipset or the Interrupt Descriptor Table (IDT).

When an SMI is triggered, the whole execution context is saved in the System Management RAM (SMRAM) and the corresponding event handler is executed. The SMRAM is addressable only in SMM mode, whose address range is rerouted to VGA when running in other CPU modes, thus protecting its content. An inherent limitation is that, as every protected data and code is stored in SMRAM, any development is limited to a very limiting size of a few KB.

When in SMM mode, the addressing mode is a direct mapping to physical pages, without any translation mechanisms, a significant difference as compared to the HVM case. In order to bridge the semantic gap of virtual pages, use of the CR3 registers may be made by SMM code developer/system analysts. Another drawback is the fact that memory addressed this way is restricted to 4GB, even in Physical Address Extension (PAE)-enabled systems: the transition to other CPU modes, which fully address system memory, requires exiting the SMI mode. To overcome this limitation, one may use some kind of subversion or insertion of a callback instruction directly in the program code.

The SMM code is initialized by the Basic Input/Output System (BIOS), thus the original BIOS should be replaced by the instrumented one. For this task, Coreboot[44] and SeaBios[172] are often used.

Building an analyzer in SMM mode is a natural follow-up, since it is well protected from the "guest" system, which remains completely unaware of the former's presence and therefore composes a strong threat model. Besides, such analyzer would also run instructions directly on the processor, with no address translation but still having memory access. We can think of such a system as a more fine-grained bare-metal analyzer.

The architecture of this kind of system also follows a client-server model, with the server being the SMM code and the client some code on an external machine, often network connected. As it runs in a low-abstraction level, with no O.S. support, all comunication needs to be implemented by the analyst, including network drivers and protocols. Additionally, as data will be transmitted through an insecure media, cryptography and error detection/correction also need to be implemented in the client-server communication. Analyzers' initialization may be done using SMI triggering facilities, such as listening to an external serial port or using the Intelligent Platform Management Interface (IPMI).

Security system features may be implemented by making use of the SMM architecture, according to its intended features (data watchpoints, I/O monitoring, step-by-step execution and so on). The way all of them work is very similar, triggering an SMI for such actions. Breakpoints, for example, may be implemented by triggering an overflow event on performance counters.

When building such systems, it is important to reduce the attack surface for properly handling malware that try to guess whether they are running on an SMM-monitored environment. Some malware counteractions may include BIOS overwriting, which may be avoided by employing hardware-assisted Trusted Boot. It is also important to realize that kernel malware have access to debug and performance counter registers, which may be harmful to an analysis process based on such values. A corresponding countermeasure would be to periodically trigger an SMI to check their values.

Another problem is BIOS fingerprinting. As BIOS is rewritten, original hash values and strings are changed. This way, a malware could compare these values to known BIOS-vendor ones in order to detect the monitoring environment. An effective way to overcome such problem in to perform an online BIOS flashing to the original one after the modified one was loaded in memory. However, this is an open problem for systems that do not allow BIOS online flashing.

As for the future, SMM mode will certainly undergo changes, the most significant one being the SMM virtualization (STM), which would deny most SMI requests. A possible solution to this might be rewriting larger parts of BIOS code in order to handle such events earlier, with a significant increase in complexity.

The next sections present tools developed using the SMM mode. We start by presenting some threats that make use of SMM for its inherent stealthiness and follow with some security-related tools.

SMM Threats

Rootkit development has exhibited an adaptative evolution, counterparting advances in system security. In the same way HVM was employed for malicious purposes, SMM has already been targeted. Its attractiveness comes from the same virtues exhibited by HVM: transparency, system-wide instrumentation capabilities and OS independency. An SMM rootkit also has advantages in concealing its memory footprint, given that SMRAM is hardware-protected, and in surviving reboots and re-installations, since it is BIOS-stored.

The first work referring to an SMM rootkit is probably the one presented by Duflot et al. [57], showing a privilege-escalation attack against x86 OpenBSD. In this attack, the authors bypass secure-level protections by installing their own SMM handlers, allowing unrestricted access to physical memory. Following that, the practical Phrack magazine highlighted some work intended to handle SMM for other purposes, as in [32] and [207].

Academically, Embleton et al. [60] presents the construction of an SMM keylogger by redireting the keyboard Interrupt Request (IRQ) on the chipset to SMM using the Advanced Programmable Interrupt Controller (APIC). The pressed keys are logged and transmitted through the network interface. For such implementation, a kernel driver was used to set SMM mode. An advantage of this technique is that no IDT hooking is employed, since one can have an out-of-band access through the chipset APIC redirection. Redirection is performed directly on the chipset redirection table. The network card operation has to be manually implemented, working in a client-server way on the PCI bus and encapsulating data on UDP packets, which are then transmitted when buffers are full.

Another SMM keylogger is presented by Schiffman and Kaplan [166]. In this implementation, the authors hijack USB without tunnelling data to SMM, which allows for interception to occur before the kernel is aware of the event. This is achieved by having the USB Host Controller reroute the interrupts to a USB-PS/2 emulation SMM handler. It constitutes a much stealthier way of hijacking the keyboard events, since keystrokes could be successfully intercepted, replaced and injected, on tests performed in a Linux environment. The authors also discuss how to extend the approach to perform Man-In-

The-Middle (MITM) attacks and to hide USB devices.

SMM for Debugging

Given the system view and transparency offered by the SMM mode, it is naturally suitable for debugging purposes. Zhang et al. [218] presented MALT, a complete debugging framework based on SMM which implements basic debugging facilities, such as breakpoints, register access and memory examination. It is implemented on a client-server architecture, offering possibilities of communication by using a GDB-like protocol or through its user-friendly interface, allowing for integration with several popular debugging clients, such as IDAPro and GDB.

MALT's threat model is designed so as to handle armored malware and rootkits. As the SMM TCB resides in BIOS, it does not handle firmware rootkits. Its implementation is based on replacing the original BIOS for coreboot and SeaBios as payload. SMI triggering is done by rerouting actions, and the semantic gap is bridged based on CR3 register addresses[3].

MALT is able to provide four different levels of step-by-step debugging: instruction-level, branch-level, far control transfer level and near return transfer level. Step-by-step is implemented by overflowing performance counter registers in order to trigger SMIs. Breakpoints are also implemented using this technique, comparing the EIP of the current instruction with the stored breakpoint address.

MALT protection is performed by intercepting register reading actions in order to check for kernel access to specific system registers. In addition, online flashing is performed, replacing the BIOS with the original image before the debugging process starts, so fingerprinting is avoided. For configurable timers, their values are recorded after switching into SMM. The study case provided is about debugging of a kernel on a crash, from both Windows and Linux, including a complete backtrace.

SMM for Forensic Purposes

The whole-system view of the SMM mode is suitable for system forensic procedures, such as complete memory dumps. Usage of these capabilities was the premise of many works, like the ones presented below.

One of them is SMMDumper[154], able to acquire volatile memory contents on running systems and therefore helping with digital forensic analysis and incident response. SMMDumper employs the SMM BIOS hardware support in order to be resilient to malware attacks. It also overcomes the SMM-imposed 4GB barrier to take whole-memory snapshots. SMMDumper is based on two modules: i) a collector; and ii) a trigger. The former is a boot-time loaded SMM-resident module responsible for capturing data and transmitting it to the trusted host, whereas the latter is responsible for activating the collector using SMIs. There is also an external client, the trusted host, which receives the collected data. Data transmission is supported by a system-plugged cryptographic device. It is used for signing data in order to asssure data integrity. In order to communicate with

[3]As the CR3 stores memory page information, it allows for process identification and isolation

the machine's NIC, SMMDumper implements its own network driver. Data is transmitted through UDP packets, and since UDP is prone to data loss, lost packets are asked back by the trusted host to SMMDumper, which recreates and retransmits them.

SMMDumper activation is done by pressing a pre-defined key sequence. In order to implement such functionality, the authors have implemented an SMM-based keylogger. The SMM ISR is responsible for extracting the scancode from the keyboard controller buffer by reading from the I/O port. SMI triggering is generated by modifying the I/O APIC Redirection Table. In order to overcome the 4GB limit, code is injected in mapped pages and EIP is modified in the `State Save Map` (SSM) so that it points to the code that was injected. Once an `RSM` instruction is raised, `EIP` is restored from the SSM and thus the system execution resumes from the custom code. SMMDumper's evaluation shows it is practical for 6GB dumps.

A slightly different solution is presented by Wang et al. [203], which leverages PCI DMA access to complement the SMM mode in order to perform whole-system memory analysis. In fact, using PCI DMA for memory acquisition is not new. However, it is hard to obtain the semantics of a memory dump without knowing the values of CPU registers at the time that the memory snapshot was retrieved, such as the Interrupt Descriptor Table Register (IDTR) or the base address of the current page table (CR3). To address these challenges, a new approach where SMM register data is supplied is presented.

Analysis may be performed either online or offline, with the CPU state transmitted from SMM to an external client through a GDB-compatible protocol. Consistency is guaranteed because during SMM the OS enters and remains in suspended state. SMI is triggered through IPMI and the system is supported by coreboot and seaboot, with a GDB stub written directly on it. The BIOS also holds the NIC driver, which is a better solution than an in-system driver in a subversion-prone scenario. Another solution for data transfer would be to build a specific-purpose NIC, but then again that would be cost-prohibitive.

Memory capture is performed by computing the difference between two consecutive memory snapshots, thus reducing the amount of data to be recorded. This strategy requires all memory pages to be marked as read-only and having the exception handler responsible for recording the memory pages to be modified and sending this information to its caller. A limitation of this approach is that PCI card access to physical RAM memory may be blocked on both Intel and AMD solutions.

SMM for attack detection and prevention

SMM transparency and its hardware protection make it a suitable environment to malware attack analysis and identification. For this purpose, Zhang et al. [221] presented SPECTRE.

SPECTRE is a specific-purpose SMM tool that allows for memory inspection. Its specialization brings with it a small TCB, self-protected by the SMM mode. Its threat model covers most kinds of attack, with the exception of hardware trojans, thus allowing for host and guest memories' safe inspection. It is implemented using coreboot and assumes the SMRAM will be locked after its loading.

Its architecture is client-server based and "a heartbeat message is sent securely to the monitor machine through a gigabit NIC. When a suspicious behavior is detected, an alert is transmitted as part of the heartbeat message". SPECTRE uses system timers to periodically generate SMI. This approach may lead to evidence evasion for a malware that could act on such time intervals. As other SMM-based tools, SPECTRE also has to reconstruct virtual pages, since SMM only sees physical ones.

As a modular tool, SPECTRE has rules written for heap spray, buffer overflow and rootkit (kernel integrity) detection. Shellcode's NOPs are used as identifiers for memory overwriting. This detection is performed through a regex matcher implemented inside the tool. The tool was evaluated against Windows and Linux attacks.

SMM for I/O Integrity

SMM can also be used to ensure I/O integrity, which means known ports will not be mapped to other ones in order to prevent potential malicious actions. Such integrity is important since after compromising an I/O controller, attackers can change memory via DMA or by compromising I/O devices.

Trusted Platform Modules (TPMs) are able to protect firmware and IOMMU integrity at boot time, but not at runtime. The Input/Output Memory Management Unit (IOMMU) tries to protect memory from DMA attacks. However, the root entry table's base address and other configuration registers may also be under the attacker's control on specific scenarios. Besides, the National Vulnerabilities Database (NVD)[138] shows that many firmware vulnerabilities were discovered since 2010, therefore enlarging the attack surface.

As such, authors have proposed I/O Check[220], a solution which employs SMM to check I/O configurations and firmware integrity, enumerating all I/O devices in order to achieve its goals. I/O Check assumes the system is supported by a TPM hardware for its boot and also assures BIOS image integrity. It also assumes that SMRAM is locked in the BIOS after loading. Its verification has as base the premise that the "DMA Remapping ACPI table should never change after booting" and that "the base address of the configuration tables for the DMA remapping unit should be static". Attack detections are notified through audible beeps. It also assures NIC integrity by storing its original hash value in SMRAM and by periodically reading the NIC's memory firmware code, computing the current image's hash value and comparing it with the saved value.

Who protects the hypervisor?

In Section 2.5.7, we presented systems whose goal was to protect the guest system running inside them by running a trusted hypervisor. However, attacks to hypervisors are well known and widely deployed today. Rootkowska and Wojtczuk [163], for example, present an attack to the Xen Hypervisor by redirecting memory reads/writes from the internal guest to the host. Sharkey [176], in turn, presents attacks able to trap special instructions under secure hypervisors. This way, protecting hypervisors from attacks and corruption is an important feature in security systems. Given the nature of the SMM mode, it is

well suited for such purpose, to the way of being employed by a variety of tools, some of which presented below.

One of these examples is HyperSentry[11], a hypervisor integrity checker for cloud environments. Its architecture consists of an agent inside the hypervisor and a client in SMM. Its output is composed by the memory data plus the hash calculation and, despite being able to access memory, Hypersentry has to overcome another challenge, which is bridging the hypervisor semantic gap from within the SMM mode.

In order to assure integrity at boot time, Hypersentry employs trusted boot capabilites of modern hardware to lock SMRAM after its loading. Since the hypervisor can re-route and mask SMI, it requires an out-of-band channel. So, SMIs are triggered by using IPMI specific hardware from IBM servers. When an SMI is triggered, the system has no power over whether the hypevisor is running on VMX Root or Non-Root mode. However, in order to handle VMCS data, the CPU must be in root mode. To overcome this challenge, HyperSentry proposes a new fallback technique, which "guarantees that the CPU falls back to VMX root operation". It is based on redirecting the SMI APIC from the performance counter overflow after a code injection, returning control to the monitor.

The system has some limitations: protected registers from the Intel TXT platform were not used; cache was not used in order to prevent cache poisoning attacks; when in SMM mode, interrupts are disabled, which may lead to a crash if it lasts too long; in a multiprocessor scenario, when monitoring an event on a specific core, other cores are frozen in order to ensure consistency.

The case study provided is a Xen Hypervisor monitoring. The authors have verified its code integrity using SHA-1, its control flow pointers in the IDT and whether its physical memory guest isolation was functional.

A similar approach for hypervisor protection is employed by HyperCheck[202], which uses a PCI DMA card to collect memory data as well as SMM collected register data to handle virtual address translation. The system was implemented using two prototypes: the first is an NIC emulation on QEMU and the other is a real PCI NIC. The system also has an analyzer to which data is transferred through the network card. This transfer is protected using a random hash in order to avoid replay attacks, with the key being locked in SMRAM. In order to prevent attacks where a fake device asks for the key, TPM hardware can be used. In addition, a random-interval scan is performed to avoid timing attacks. The study case provided was DMA attacks against the Xen Hypervisor, having both Linux and Windows XP as guests. However, some kinds of attack are not detected or prevented by using this technique, for instance using dynamic function pointers or returned-oriented attacks.

SMM security issues

SMM has been criticized for allowing system subversion in many ways, such as attacking the TXT subsystem, which would allow complete system subversion. SMM virtualization was proposed as a way of sandboxing SMI requests. Some exploits, however, presented sandbox escapes and therefore still compromising the system, as pointed by Rutkowska [162]. Intel's answer was to made available its CHIPSEC tool[42], intended

to assure correct configuration of hardware parameters in order to make the system more secure[41].

The battle of the rings

Modern processors isolate distinct classes of applications by hardware-imposed privilege limits. The privilege levels in the x86 architecture are known as rings, where userland applications run on ring 3 whereas the kernel runs on ring 0. The other rings were aimed to be used by dynamic libraries but are not used in practice. Recently, as new monitoring mechanisms have been proposed, new ring names were christened in order to highlight the higher privileges they impose. This way, HVM became known as `ring -1`, since it is more privileged than the kernel's and userland's, and SMM became known as `ring -2`, since it can monitor even hypervisors. Currently, a more privileged system mode has been used to monitor even SMM, known as `ring -3`. This latter is called Management Engine (ME) and is presented in the following section.

Management Engine: the lord of the rings

The Management Engine (ME) is another management mode present in Intel's chipsets, originally aimed to support Intel Active Management Technology (AMT). However, Intel recently started using it for executing system sensitive applications.

ME can be seen as an embedded processor, having its own timer, RAM and ROM memories, and DMA. As this mode cannot address system memory, DMA is used to transfer system data to ME mode. As in the SMM mode case, memory is accessed as physical addresses. As ME can externally monitor and control the processor, it can interfere even in SMM and BIOS codes, besides kernel and userland rings. These capabilites resulted in backdoor suspicions[200].

Similarly to the other monitor modes, ME may be suitable for many security related tasks, from the malicious ones like data stealing to the transparent analysis of protected software. A talk on BlackHAT[186] presented the use of the ME mode to implement a system rootkit. In practice, ME implementation flaws[77] may allow an attacker to take control of the victim's machine at a very deep level.

ME usage for security purposes, however, is still a limited research field, having few published articles or available tools when compared to other solutions. Academically, ME was investigated by some authors, for instance Ververis [199], but a wide range of research still awaits further progress. The results may be similar to other tools', providing the community with transparent malware analyzers, tracers, debuggers, forensic tools and so on. Finally, environments like ME are not an exclusivity of Intel's. A similar solution is present in AMD processors, called Secure Processor[6].

Isolated rings and SGX

As more privileged rings came to be used for inspection, privacy-concerned applications had to be moved to a place where they could not be monitored. This way, isolated

rings/modes were developed, ones that are system-independent and cannot be monitored by other system facilities.

The most notable isolation solution nowadays is Intel SGX. Isolated rings like SGX, however, are not exclusive from Intel processors. ARM processors have a similar ring named Trusted Zone[8], used for instance on recently launched Android Nougat[45]. This paper, however, focus on Intel's solution research since they are more developed and widespread.

Intel Software Guard Extensions (SGX) is a set of instructions that allows software to run in an isolated mode called enclave. This mode is O.S. independent[4], and its encrypted memory pages are destroyed after usage. The SGX underlying crypto systems also allow software verification and attestation, which aims to offer tamper-proof capabilities. SGX integrity itself is assured by hardware TPM and TXT systems.

As SGX is based on a new instruction set, programs should be rewritten to include such instructions, which allow for enclave initiation, attestation, execution launch and destruction. Some research has already been conducted in order to clarify SGX crypto API[10] and SDK usage[103]. Besides a new programming model, SGX also requires special hardware—SGX capabilites are available on modern Intel processors, like the Skylake microarchitecture. In order to provide a research environment, Jaim et al. presented the OpenSGX[85], a QEMU extension to cover SGX instructions.

An example of a benign application taking advantage of SGX capabilities is the protected chat[75] presented by Intel, in which images are processed inside the enclave and therefore protected from external capture. However, malicious actors can also make use of these capabilities to their benefit, as presented in other solutions. Van Prooijen's work[190] illustrates how an attacker could remotely attest its payload is not tampered. It also points at the hardness of reverse engineering SGX code. This scenario is bound to bring about new research in coming years since it is currently an open question.

The idea of malware attestation have first appeared in Davenport and Ford's work [47]. In this article, authors also highlight SGX capabilities for the anti-cheat gaming industry.

Another class of attacks which applies even in the context of SGX is the side-channel information retrieval. Schwarz et al.[171] presented a malicious sample able to retrieve RSA keys from co-located enclaves by monitoring cache access patterns.

Hardware-based techniques

Although hardware design is normally out of reach for most security research, many defense concepts aforementioned count on hardware support for their implementations, while others are even mostly hardware-focused. In this article we cover the main ones.

The first widely-recognized attempt to implement a hardware based security monitor was `Copilot`[147], which aims to assure kernel integrity. The solution is very similar to the one presented by Hypercheck, using a PCI card to collect memory data snapshots and analyze them. Since it is a dedicated hardware, it is protected against tampering. Its architecture follows the well-known client-server model, where the monitor is responsible

[4]Many management operations are performed by CPU microcode and not by software

for analyzing the received data and identifying threats.

As presented previously, this approach has the disadvantage of not getting CPU register values, which imposes limitations on context comprehension and introspection. However, it was a first step towards overcoming virtual address translation on approaches using external hardware, which was achieved by deriving page information from the Linux's `System.map` file. The developed prototype helps the authors make clear the main benefit of their approach, which is the minimal processing overhead; measurements indicated only 1%.

Besides these advantages, Copilot-like approaches have a significant drawback related to its snapshot characteristic, rendering it susceptible to timing attacks. Also known as transient attacks, timing attacks allow for an attacker to remain stealth by executing malicious activities during the time-interval between 2 snapshots.

In order to overcome the previous disadvantage, Moon et al. [125] proposed Vigilare, a System-On-a-Chip (SOC) implementation that snoops the memory bus in order to perform real-time analysis for kernel integrity evaluation. Aiming to give a better understanding of the issues related to transient attacks, the authors implemented two Vigilare versions, each one implementing a distinct capture strategy: a snapshot-based one (SnapMon) and a snoop-based one (SnoopMon). They were implemented using a LEON3 processor running Snapgear Linux.

Snapmon is a straightforward implementation of Copilot's approach, using DMA for acquiring memory. Its memory verification is performed through hash comparisons to identify any modifications.

In turn, Snoopmon had to overcome some implementation challenges. One was to handle virtual address translation, which was done by following Copilot's System.map approach. However, the biggest challenge was to handle lots of data at once. If Vigilare could not analyze all the bus traffic that Snooper provided, the results would be compromised. This way, the tool was designed to have a selective bus traffic filter. which recognizes only meaningful information while truncating unnecessary data. This approach also allowed Snooper to filter data on traffic bursts.

Vigilare also proposes two ways of protecting its memory content, be it data or instructions. Firstly, it uses a separate hardware memory, with no guest access; secondly, "it implements a memory region controller which specifically drops all memory operation requests from the host system". The latter may reduce hardware costs in comparison to the former. Experimental results have shown that the snapshot approach, even through the use of a randomized snapshot interval, is susceptible to transient attacks. Its detection rate highly depends on luck, whereas SnoopMon is able to detect all attempts. Despite Vigilare's effectiveness against static kernel code modification, it is not capable of handling dynamic kernel modifications, such as process list changes.

In order to overcome this limitation, Lee et al. [97] proposed Ki-mon, "an event-triggered verification scheme for mutable kernel objects". Much like Vigilare, Ki-mon is a SOC-based system, running Snapgear Linux on a Leon3 processor and also able to snoop bus traffic. Its memory acquision is performed through a structure called Value Table Management Unity (VTMU) which, besides snooping the bus, is also able to filter its capture and perform DMA access.

Ki-mon introduces a new form of bus traffic monitoring in order to verify changes in objects values. It also presents callback verification routines that can be instrumented to handle specified events. This is named hardware-assisted whitelisting (HAW) and its registers can be configured to be active in different ways, including a pass-through operation. Contrary to Vigilare that only alerts occurrences of memory modifications—which is sufficient for static code modification detection, but not for dynamic ones—though, Ki-mon "provides the ability to extract data values in write traffic for invariance assurance". It is also able to generate events reporting address and value pairs for memory modifications, using whitelist-based filtering while doing so. The tool also allows an event-triggered callback verification and provides an API intented to enable monitoring rule development. Some rules were developed and tested on real rootkits to check the solution's effectiveness.

Finally, there are other approaches that inspect memory traffic by using other hardware features, such as Processor Trace capabilities. Such approaches, however, follow the same previously-presented working principles. Kargos[126], for instance, is a high-frequency snapshot-based solution.

A brief discussion on hardware-based approaches

An intrinsic limitation of bus monitoring approaches is the handling of memory cache access. If reading a write-through cache is straightforward, reading a write-back cached data on the bus may lead to incorrect values. In addition, instrumenting the cache bus may range from difficult to impossible in many architectures. This way, such implementations should be aware that "it is possible to devise a transient attack that may reside in write-back cache before the updated cache contents is flushed to the memory bus" (Moon et al.).

An additional discussion is needed regarding how portable such approaches are. If, on one hand, protecting SOC systems is essential in an Internet of Things (IoT) scenario, on the other hand the well-deployed x86 architecture suffers from a lack of understanding of external bus monitors. Vigilare's authors have pointed the problem of a high transmission rate and burst transmissions but a deeper investigation is required. Besides, although kernel integrity is an essencial issue to be addressed, more bus monitoring applications should be explored. A natural scenario seems to be extending such data invariants from kernel to hypervisor integrity monitoring, such as on HyperSentry and HyperCheck solutions.

Finally, external hardware approaches have the significant disadvantage of being passive tools, without the possibility of naturally acting to block threats. This way, researchers have yet to propose ways to alert users when a violation is identified.

Other Approaches

Besides using special processor operating modes and developing special purpose hardware, system monitoring for security purposes may be performed by using additional hardware features, such as performance counters. Additionally, new trends have also introduced new threats, such as GPU ones. This section gives an overview of these tools and threats.

Performance Counters

The first category of approaches presented here is based on Hardware Performance Counters (HPCs), a specific-purpose CPU feature that provides detailed information about hardware and software events, such as cache misses, instructions retired, branches and others. They can be used for hardware verification, debugging purposes, CPU scheduling, integrity checking, performance monitoring and so on. Such feature is available on both AMD and Intel platforms, being known as Last Branch Record (LBR) and/or Branch Trace Store (BTS) on the last.

By making use of branch monitoring capabilites of HPCs, Willems et al. [209] developed `BranchTrace`, a branch monitor able to detect accidental or intended incorrect behavior of dynamic analysis in an emulated environment. The approach is motivated due to newly developed `delusion-attacks` that are able to detect CPU emulators using the different instruction execution side effects between an emulated and a real machine.

The authors remark that the deviating behavior may be fixed in the emulators. However, this would require special handling for a variety of instructions that can be used in conjunction with `rep`. This not only takes considerable effort to implement, but would reduce the performance of string copy operations, for example. As such, they chose to develop the BranchTrace monitor to collect data that allows for identifying these cases without taking into account the effects of SMC, caching effects or other kinds of delusion and/or detection attacks. BranchTrace was developed using Branch Trace capabilites on Intel processors.

Interception happens on each taken branch, including conditional and unconditional jumps, calls, interrupts and exceptions. Supplied data is the addresses of the source and target instructions of the branches. However, the authors claim it is still possible to reconstruct context from such information. They also suggest extending the information by using Windows debug symbols and disassembling the nearest instructions. For context reconstruction from such data, the authors devised a technique called binning, in which the crash reports are automatically grouped into different classes, each one consisting of crashes that resulted from the same root cause.

For the practical evaluation of this approach, they extended a tool called CWXDetector that is capable of detecting exploitation attempts and extracting shellcode used during exploitations. The tool only becomes active after the first shellcode is executed, resulting in no information gain about the path that led to such exploitation. The way authors found to gain more information about the exploitation's path was to utilize BranchTrace combined with this tool. The extended tool was used to examine a set of 4,869 malicious PDF documents. The clustering results were normalized and a similarity distance was applied. The results were compared against the PDF analysis framework Wepawet.

The authors also suggested using this technique for handling Return-Oriented Programming (ROP) attacks by applying an heuristics approach: if a RET without a corresponding CALL is detected, it is labeled as ROP-RET. Regarding ROP attacks, many tools try to address this problem by leveraging branch monitors. As examples, approaches like `CFIMon`[213], `KBouncer`[142] and `ROPecker`[39] make use of the branch record mechanism to apply Control Flow Integrity (CFI) policies, in a similar way as the aforemen-

tioned ROP-RET. Other approaches, such as the one of Pierce et al. [148], address the ROP problem by using the branch misprediction monitor, since the RET targets are not well distributed in memory and thus causes prediction errors.

Another kind of approach that employs HPC for attack detection is presented by Kompali and Sarat [91], in which a Vtune extension was developed to monitor the Branch Prediction Unit (BPU). Initially, a baseline is defined by running benign applications on the system as a training set. Afterwards, a modified version of the `Win32/Renos` malware was evaluated. Results show that "branch prediction miss rates are below threshold for a clean system". However, in infected systems, "BPU produces a high rate of prediction misses". The same approach was extended to runtime memory allocation and usage.

An additional extended approach is presented with HPCHunter[12], which uses HPC data to build a support vector machine (SVM)-based event feature selection for real time malicious program detection.

Many authors addressed this problem of online detection, such as Yuan et al. [217] and Demme et al. [51], and even extending it to the kernel [204]. The biggest advantage of the performance monitoring approach is its lower overhead when compared to other solutions[102]. The biggest challenge of these systems, however, is feature selection[185].

Performance monitors are not restricted to complex processors, as many embedded systems also present performance monitoring capabilities. `Confirm`[205] is an approach to validate firmware on embedded systems.

Graphics Processing Units

In addition to HPCs, Graphics Processing Units (GPUs) have also been employed for security purposes, both for attack and defense. We found tools and techniques leveraging GPUs for packer detection[72], IDS implementation[4, 194], security log processing[18], cryptography[192], AV parallelism[193] and even polymorphism[195].

The most significant GPU-implemented threat is a keylogger[95], in which DMA is remapped to be acessible from the GPU in a way no hook is required. Once the GPU is aware of the keyboard buffer location, the GPU can retrieve data directly from the system's memory pages. Since GPU usage has grown tremendously in the last few years, we consider this kind of threat as a very relevant aspect to be considered in security systems. POSTER work[184] suggested an approach to monitoring DMA access in order to detect DMA malware. The approach consists in trying to identify DMA side-effects, such as on the timestamp counter (TSC) and HPC ones.

Although the work is preliminary, it was a first step towards taking advantage of this possibility. We notice that an approach to detect DMA malware could also explore the HVM system resources, such as IOMMU monitoring, presented in the previous sections. An extension of the GPU DMA monitoring approach was given in the work by Koromilas et al. [93], which leveraged these capabilities to perform kernel integrity monitoring on a periodic snapshot basis.

Transactional Memories

Transactional memory is a concurrency control hardware mechanism that allows operations to be executed atomically, through using a concept similar to database transactions. Transactional memory support is present on modern systems' platforms; this paper will cover Intel's TSX solution.

TSX is a set of instructions which add transactional memory support on x86. TSX is able to monitor a small region of memory in order to verify whether a transaction can be committed or not. TSX's usage requires code rewriting since new instructions (`XACQUIRE, XRELEASE, XBEGIN, XEND, XABORT` and `XTEST`) should be used.

Due to the monitoring capabilities of TSX, it can be used for security purposes by monitoring specific system regions. An example of monitoring application supported by transactional memory might be to monitor thread memory in an efficient way, such as proposed by Muttik et al [132]. The efficiency is due to the fact that transactional memory is assynchronously activated, thus without polling needs. In addition, it is more granular (64 bytes) than trapping the page-fault mechanism, which is 4K byte granular, in general.

Another TSX-support usage is for CFI enforcement[131]. The main advantage of this approach is that the malicious transaction is not only detected but also not committed, keeping the system on a secure state. TSX research is still taking place, but it is easy to imagine use cases, such as detecting system components touched by malware samples. Another kind of application which will probably benefit from TSX is policy enforcement, as can be seen in Birgisson et al's work[19], which proposes a memory introspection mechanism.

Summary

In this section, we present an overview of the presented technologies, tools, and solutions, aiming to ease comparisons. As the tabulated items are themselves related in two distinct ways—by the employed technology and by the solution goal—we present both, so a given solution may appear more than once.

Table 2.1 presents the HVM-based tools and solutions.

Table 2.2 presents the SMM-based tools and solutions.

Table 2.3 presents the privileged rings-based tools and solutions.

Table 2.4 presents the hardware-based tools and solutions.

Table 2.5 presents the performance counters-based tools and solutions.

Table 2.6 presents protection mechanisms employed by the presented solutions.

Table 2.7 presents a comparison of solutions according to their purposes.

Table 2.8 presents a comparison of solutions according to their overhead.

Table 2.9 presents a comparison of solutions according to their required development effort.

Table 2.1: Summary of HVM-based tools and solutions.

Tool	**Purpose**	**Target OS**	**Technology**	**Hypervisor**	**Resources**
BluePill	offensive	Windows Vista	AMD SVM	–	network backdoor
HVM Rootkit	offensive	Windows XP	AMD SVM	–	–
Ether	malware analysis	Windows XP	–	Xen	syscall tracer, unpacker
CXPInspector	malware analysis	Windows 7 x64	Intel VT-x	KVM	memory tracking, profiling
MAVMM	malware analysis	Ubuntu Linux	AMD SVM	own hypervisor	syscall tracer, unpacker
HyperDBG	debugging	Windows XP	Intel VT-x	–	kernel debugger, graphical user interface
SPIDER	debugging	Windows XP Ubuntu Linux	–	KVM	trap flag Hider, unlimited breakpoints
V2E	execution replay	Linux Windows XP	HVM DBT	TEMU	transparent collection, execution replay, emulated instruction changes, emulated page table translator
Kang et al. 2009	execution replay	Windows XP	HVM DBT	Ether TEMU	transparent collection, execution replay, dynamic state modifications
BitVisor	policy enforcement	Windows XP Windows Vista Linux	Intel VT-x	–	para-passthrough, I/O policy, DMA MITM
SecVisor	attack prevention	–	AMD DEV	–	code execution policy, code authorization, kernel-userland flow integrity
HyperSleuth	forensics	Windows XP	Intel VT-x	–	syscall tracer, dump on write, lie detector
Quist et al. 2001	unpacking	Windows XP	Intel Vt-x	Ether	binary section and header rebuilding, VAD import rebuilding, AV submission

Table 2.2: Summary of SMM-based tools and solutions

Tool	**Purpose**	**Target system**	**Trigger**	**Resources**
Duflot et al. 2007	offensive	OpenBSD	—	unrestricted physical memory access
Embleton et al. 2008	offensive	–	keyboard IRQ on APIC chipset	keylogger
Schiffman and Kaplan 2014	offensive	Linux	USB-PS2 emulation handling rerouted from USBHC	keylogger, UDP transmission
MALT	debugging	Windows, Linux	performance counter overflow	register access, memory inspection, step-by-step, BIOS flashing, GDB integration
SMMDumper	forensics	–	APIC-redirected known key-pressed sequence interrupt	memory dump, UDP packets, code callbacks
Wang et al. 20011	forensics	–	IPMI	memory dump, consecutive snapshots, PCI DMA, SMM-based semantic gap bridging, BIOS NIC driver
SPECTRE	attack detection	Windows, Linux	periodic timer	memory pattern matching, BIOS NIC heartbeat
I/O Check	I/O integrity	–	–	APIC remapping check, NIC firmware hash check
HyperSentry	hypervisor integrity	Intel VT-x, Xen Hypervisor	IPMI, performance counter overflow	hash-based integrity check, SMM-based semantic gap bridging, root-non-root transition bridging
HyperCheck	hypervisor integrity	QEMU, real NIC, Xen hypervisor, Linux, Windows	–	SMM-based semantic gap bridging, hash-based integrity check

Table 2.3: Summary of privileged rings-based tools and solutions.

Solution Class	**Ring**	**Underlying Technology**	**Capabilities**	**Limitations**
HVM	-1	extended processor instruction set	attack: hypervisor attacks, defense: kernel/userland monitoring	hypervisor rewriting, semantic gap, considerable overhead
SMM	-2	system BIOS	attack: boot attacks, defense: kernel, userland and hypervisor monitoring	BIOS rewriting, semantic gap, locked BIOS
AMT	-3	Chipset	attack: whole system view, defense: kernel, userland, hypervisor and BIOS monitoring	chipset dependent, semantic gap
SGX	–	Processor Enclave	attack: malware integrity check, defense: running software cannot be monitored	API-dependent code
hardware	–	physical external hardware	attack: –, defense: tamper-proof monitoring	hardware development efforts, bus monitoring rate, semantic gap
performance counter	–	processor feature	attack: side-channel detection, defense: low-overhead tracing and profiling	no process isolation, kernel mechanism configuration
GPU	–	PCI Card	attack: DMA snooping, defense: DMA monitoring	semantic gap, DMA blocking
TSX	–	concurrency control hardware	attack: –, defense: commit-based control flow	limited block size

Table 2.4: Summary of hardware-based tools and solutions.

Tool	**Technology**	**Data Collection**	**Filtering**	**Vulnerabilities**
Copilot	PCI Card	snapshot	No	timing, dynamic state modification
SnapMon	SOC	snapshot	no	timing, dynamic state modification
SnoopMon	SOC	snooping	yes	–
Ki-Mon	SOC	snooping	yes (Hardware-Assisted Whitelisting)	–
Kargos	–	snapshot	–	–

Table 2.5: Summary of performance counters-based tools and solutions.

Tool	**Class**	**Type**	**Vendor**	**Purpose**	**Intel.**	**Intro.**	**Tools**	**Underlying Solution**	**Injection**	**Limits**
Branch Trace	branch monitor	—	—	delusion attacks, ROP	root cause analysis, CALL-RET CFI	Windows debug symbols	shellcode extractor, ROP detector	CWXDetector	—	no CG or CFG generated
CFIMon	branch monitor									
KBouncer	branch monitor	LBR	Intel	ROP detector	CALL-RET CFI	–	process blocking	–	yes	LBR gadget-length, code injection
ROPecker	branch monitor	LBR	Intel	ROP detector	CALL-RET CFI	–	—	—	—	LBR gadget-length, static code database
Pierce et al. 2016	events monitor	—	—	ROP detector	baseline					
Kompali and Sarat 2014	events monitor	–	Intel	abnormal behavior	baseline	N/A	CPU usage, mem usage, cache usage, branch prediction	Vtune	–	no process information, Vtune limits
HPCHunter	events monitor	–	–	abnormal behavior	baseline + SVD/SVM					
Yuan et al. 2011										
Demme et al. 2013										
Wang and Guo 2016										
Malone et al. 2011										
Tang et al. 2014										
Confirm										

Table 2.6: Protection mechanisms used by overviewed solutions.

Solution Class	**Mechanism**	**Protection**
HVM	trap flag	exception handler
HVM	loader driver	rootkit hider technique
HVM	hypervisor code	page fault handler trap
HVM	timing	TSC change
SMM	BIOS change	online BIOS flashing
SMM	timing	TSC change
performance counter	MSR disabling	periodic kernel checking
DMA	DMA blocking	device exclusion vector

Conclusions

In this work, we presented a complete overview of monitoring tools on modern systems. We took a deep look into HVM and its application as well as SMM ones, which includes threat and defense mechanisms. We also presented some hardware-based related tools and their tamper-proof advantages and early-developed tool limitations.

We do believe that this summarizing work will help researchers improve knowledge in the area, since there are still have unsolved issues. Among them, despite well-known ones such as transparency claims which are not supported by the vendors, we have new ones. An emerging one is nested virtualization support. Questions such as: "How do we support a VM inside another one?" and "How do we bridge such coupled semantic gap?" are still unanswered.

We are also aware of a trend on moving to VM is happening: Qubes[161] is a modified Linux OS that isolates each application on a VM; Windows 10 has implemented the concept of Virtual Secure Machines (VSMs), "loading a microkernel with its own drivers, called Secure Kernel Mode (SKM) environment"[82]; the Edge browser also moved to a micro-VM environment[53]; Samsung, in turn, implemented a hypervisor-based approach for kernel protection[165].

If this trend consolidates as a `de facto` standard, we will have another turn, since just detecting VM environments will not be enough for malware authors: they will have to detect the monitoring process itself, which is much harder.

The hereby presented approaches solve most of the timing problems when transparently analyzing a system. However, external approaches still remain effective, such as NTP time measurements over encrypted connections.

Acknowledgments

This work was supported by the Brazilian National Counsel of Technological and Scientific Development (CNPq, Universal 14/2014, process 444487/2014-0) and the Coordination

Table 2.7: Comparison of solutions according to their purposes.

Purpose	**Technology**	**Advantages**	**Disadvantages**
offensive	HVM	late launch	–
offensive	SMM	–	BIOS locking
offensive	SGX	remote attestation	–
malware analysis	HVM	late launch	hypervisor rewriting, overhead, semantic gap
malware analysis	SMM	–	BIOS rewriting/locking, semantic gap
malware analysis	performance counter	low overhead	limited data capture, limited process information
malware analysis	GPU	low overhead	limited to DMA data, DMA blocking, semantic gap
debug	HVM	easy register access	
debug	SMM		limited to SMI
attack detection	HVM	–	vulnerable to hypervisor attacks
attack detection	SMM	inspect hypervisors	have to implement network support
attack detection	GPU	low overhead	DMA-limited
attack detection	PCI-card	low overhead	DMA-limited
attack detection	hardware	tamper-proof	no active component
policy enforcement	HVM	IOMMU	–
integrity check	HVM	kernel monitoring	hypervisor attacks
integrity check	SMM	hypervisor check	coupled semantic gap
integrity check	DMA	low overhead	timing attacks
side effects	event counter	low overhead	limited process isolation
ROP	branch monitor	runtime code monitor	increased overhead
ROP	event monitor	side effects detection	limited process information

Table 2.8: Comparison of solutions according to their overhead.

Technique	**Overhead**	**Reason/Limitation**
HVM	high	single-step trap at hypervisor level
SMM	high	single-step trap at BIOS level
performance counter	medium-low	branch-step trap at kernel level
GPU DMA	near-zero	GPU blocked for other calculations
dedicated PCI DMA	near-zero	specific purpose
external hardware	zero	no interruption/interference

Table 2.9: Comparison of solutions according to development effort.

Technique	**Development effort**	**Reason/Limitation**
HVM	high	hypervisor rewrite
SMM	high	BIOS rewrite
external hardware	high	hardware project
dedicated PCI DMA	medium	device driver
performance counter	medium	ordinary kernel driver
GPU DMA	low	GPU program

for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).

Enhancing Branch Monitoring for Security Purposes: From Control Flow Integrity to Malware Analysis and Debugging

Publication: This paper was submited for publication to the ACM Transaction On Privacy and Security (TOPS)

Marcus Botacin[1], Paulo de Geus[2], André Grégio[3],
(1) University of Campinas
Email: marcus@lasca.ic.unicamp.br
(2) University of Campinas
Email: paulo@lasca.ic.unicamp.br
(3) Federal University of Paraná
Email: gregio@inf.ufpr.br

Abstract

Malware and code-reuse attacks are the most significant threats to current systems operation. Solutions developed to countermeasure them have their weaknesses exploited by attackers through sandbox evasion and anti-debug crafting. To address such weaknesses, we propose a framework that relies on modern processors' branch monitor feature to allow us to transparently analyze malware, thus reducing evasion effects. Transparency is also key for debuggers, since modern software (malicious or benign) is anti-analysis armored. We achieve transparent code execution control by using the branch monitor hardware's inherent interrupt capabilities, keeping the code under execution intact. Previous work on branch monitoring have already addressed the ROP attack problem, but require code injection and/or are limited in their capture window size. Therefore, we also propose an ROP detector without these limitations.

Introduction

Malicious Software (malware) is a persistent threat to current systems; dynamic analysis is one of the main techniques used for malware profiling, identification of features and development of countermeasures. Malware authors continuously seek ways of preventing their code from running inside analysis environments (sandboxes) to thwart detection, while operating system improvements complicate its instrumentation for malware monitoring. Hence, hardware-assisted analysis approaches have been developed to overcome these issues.

In addition to sandboxes, debuggers are also important tools for software development and analysis, as they support code inspection and, consequently, its validation. In systems security, debuggers can be used to assist malware analysis and reverse engineering, allowing researchers to investigate several execution paths. However, both legitimate software (for intellectual property protection) and malware (for detection avoidance) are often equipped with anti-debug techniques. Therefore, we need to accomplish transparency to overcome these techniques and so perform more dependable malware analysis.

Along with malware infection, code injection used to be one of the main attack vectors to subvert systems functioning. The adoption of non-executable pages supported by hardware (No eXecute - NX/eXecute Disable - XD) and data execution prevention (DEP) eliminated this problem in practice. However, attackers found another way of leveraging control flow deviation by chaining blocks of code (gadgets) through `ret` instructions. This is known as Return-Oriented Programming (ROP) and is currently the main injection vector. Recently, techniques based on Control Flow Integrity (CFI) and code length arose to counter such attacks with reasonable effectiveness[5].

Based on the aforementioned issues, we introduce a hardware-assisted solution to address sandbox evasion and anti-debug equipped malware in a transparent way, and to detect ROP attacks in real time while overcoming limitations of existing state-of-the-art, hardware-assisted approaches. In summary, we make the following contributions:

[5]Despite attacks as Control Flow Bending and Jujutsu

1. **Current Threats and Solutions scenario review**: we present a review on the threat landscape scenario and current countermeasure and analysis tools, discussing their weaknesses. Specifically, we review transparent analysis solutions as well as branch-based ones.

2. **Branch monitoring framework**: we propose a complete, modular framework based on hardware monitoring features, allowing for further applications that overcome current and future state-of-the-art limitations and weak points.

3. **Transparent malware analysis**: we leverage our framework to build a transparent malware analysis tool with lower development efforts than the current state-of-the-art ones. As far as we know, no other malware tracer is based on such kind of monitoring.

4. **Transparent debugger**: we demonstrate how to implement granular debugging based on our framework without using single-step flags. Again, we have no knowledge of other debugging solutions based on branch monitors.

5. **ROP attack detector**: we present an improved implementation of current ROP detection heuristics, based on our framework, which does not require code injection, a limitation on other approaches.

6. **Hardware Improvements**: We suggest possible hardware enhancements for branch monitors based on the challenges we faced when developing our solution.

The remainder of this paper is organized as follows: in Section 2.15, we define the threats to be addressed, review the current threat and countermeasure scenario, and introduce the hardware feature used in our solution; in Section 2.16, we review the state-of-the-art solutions and pinpoint their weaknesses; in Section 2.17, we state the basis for our framework; in Section 2.18, we present solutions developed upon our framework; in Section 2.19, we discuss our contributions, proposal limitations and future developments; finally, in Section 2.20, we present our conclusions.

Background and threat model

In this section, we present background information about the covered topics, and introduce our threat model, which will guide our project decisions and the development itself.

Malware analysis and evasion

Malware analysis makes use of a set of techniques used to inspect malicious artifacts in order to build a knowledge about it, which allows the development of detection and countermeasure mechanisms. The techniques are usually classified as static or dynamic [179]. Each one presents limitations exploitable by malware. Static analysis may be susceptible to both theoretical (e.g., opaque constants [128]) and practical (e.g., packing, encryption, and obfuscation [65]) limitations. Dynamic analysis is often employed as an additional

analysis layer in order to overcome such limitations [59], relying on the sample's execution in a controlled environment (sandbox). The execution usually happens on an emulator, due to instrumentation issues, or in a virtualized environment, due to scalability issues. Code execution in a non-native environment may be used by malware to detect an analysis environment and thus to hide its malicious intent.

Environment detection is performed mainly through fingerprinting or side-channel effects on instruction execution [105], since emulator implementations do not exactly look like physical processors. Currently, there are tools to automatically detect such side effects [177].

Current solutions for evasive malware

Many authors tried to address evasive malware issues, either by detecting their split personalities under an emulator [13] and counter-measuring its side effects [198], or by running the sample on a bare metal environment [89]. Trying to mask execution side effects is an ineffective solution since it can only assure the execution of samples which employ known detection tricks. Regarding this scenario, a non-evadable malware analysis tool should be able to run code in a native processor, which is called transparent execution.

bare metal systems often present another issue related to detection: their intrusiveness over the traced sample. Systems which rely on techniques such as DLL injection can be detected by hashing sample's own memory. For this reason, higher-privileged monitoring tools are required [159]. However, as the operating system (OS) evolve, high-privileged instrumentation becomes harder due to new security mechanism – Windows Kernel Patch Protection (KPP) [109], for example, denies kernel hooking, a frequent approach for API interception. This way, a non-intrusive instrumentation is a requirement for bare metal malware analysis on modern OS.

Developing a transparent and non-intrusive malware analysis system is one of the goals of this work.

Transparency

Along this paper, our most frequent claim regards transparency, which we understand as the sample behaving under analysis the same way as it behaves out of an analysis environment. In other words, the sample must not get to be aware of an analysis procedure. In order to give more formal foundation to this claim, we adopt the Dinaburg et al. definitions for Ether sandbox:

1. **Higher privilege.** The analyzer has to have more privileges than the analyzed instance.

2. **No non-privileged side-effects.** Any instruction which induces side effects should be handled by an exception able to hide its effect.

3. **Identical Basic Instruction Semantics.** Each executed instruction has the same effect and lead to the same next instruction.

4. **Transparent Exception Handling.** Given an exception on a given i^{th} instruction, the exception handler must return to the $i + 1^{th}$ instruction;

5. **Identical Measurement of Time.** The measurement of time has to be identical within and without the analyzer. As this requirement is hard to fulfuill—exception handling is not constant time, for instance—, small differences are tolerable.

It is clear a bare metal-based system naturally fulfills most of these requirements. In this work, we discuss how to fulfill the other ones when developing an inspection mechanism.

Debuggers: requirements and implementations

Debuggers can be used to assist malware analysis and reverse engineering, allowing researchers to investigate several execution paths. In general, a debugger should provide:

- **Small Step Execution**: A debugger should allow for region of interest inspection in a granular way—from single step to function call trace.

- **Breakpoint Information**: A debugger should assure that the breakpoint region is known. In other words, it should provide predictability to the execution. The combination of the two aforementioned requirements matches the context requirement [158]. Due to the latter, probing approaches are not suitable for debuggers.

- **Context Inspection**: Given a breakpoint, the debugger should be able to retrieve information about the current execution context, such as memory and register values and/or function called.

In addition, there is another characteristic of a good debugger, though not always implemented in practice:

- **Transparency**: It is most welcome that the software under analysis be unaware of a debugger, so it can execute without any restriction. Currently, a lot of software employ anti-debug techniques in order either to prevent intellectual property stealing, in the legitimate case, or to avoid detection, in the malicious case.

Current debugger implementations

Current debuggers are built on top of distinct techniques, such as OS support, emulation or injection, and hardware support. However, most of them are not transparent, showing the same limitations discussed previously in Section 2.15.1.

Most OSs provide interfaces that allow process control. Some Unix-like systems, for instance, provide the *ptrace* API, which is the base for tools like the `strace` tracer and the `GDB` debugger. This solution is not transparent since the presence of the tool can be discovered with the tool itself (`if (ptrace(PTRACE_TRACEME, 0, NULL, 0) == -1)`, then it is detected). Windows also provides its own debug facilities [113]. However, they are also not transparent, being detected by the `IsDebuggerPresent` API [118].

Emulation and injection-based systems suffer from the same problems stated in Section 2.15.2. Other debug solutions rely on hardware features, such as the `step-by-step` execution defined by setting a `trap flag` in a `debugctl` MSR register, which can be detected by samples through reading that register, therefore being also not transparent. Providing a transparent debug solution is another of the goals of this work.

ROP attacks

Code injection used to be one of the main attack vectors to subvert systems functioning. The adoption of non-executable pages supported by hardware (NX/XD) and data execution prevention (DEP) eliminated this problem in practice. However, attackers found another way of leveraging control flow deviation by chaining blocks of code (gadgets) through `ret` instructions. This is known as Return-Oriented Programming (ROP) and is currently the main injection vector. This kind of attack is not prevented by existing mechanisms since the code chains are not composed of any externally-injected material but perfectly legitimate code in system memory. In a general way, the chains are composed of a few instructions, which when combined and properly chosen may satisfy the attacker's desire to execute specific, arbitrary computation [67].

Figure 2.4 illustrates a ROP attack which adds and subtracts two values in memory. In a real scenario, those could be addresses of a function or a malicious payload. Notice that the hexadecimal code on the right side is part of the process's image, either pieces of its own binary or of any library it is linked to. Gadgets can make use of existing instructions by aligning the return jump to the byte boundary of a complete instruction, but are often built based on instruction unalignment/misalignment. By returning to a memory address out of phase in relation to the original opcodes, the intermediate bytes are understood by the CPU as completely different opcodes than the original ones.

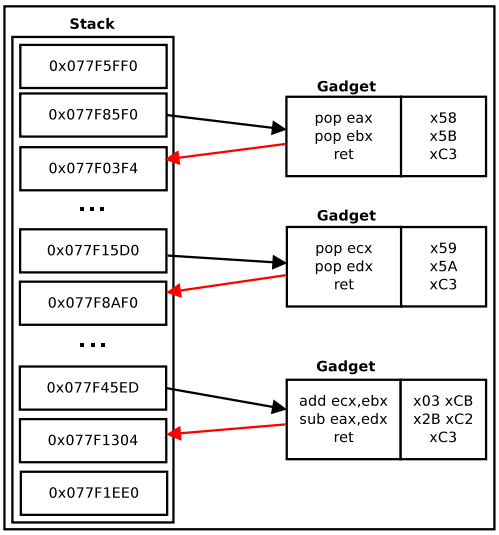

Figure 2.4: Example of a ROP attack - Computing with memory values. Left side shows a program stack filled with malicious payload (gagdets addresses); right side illustrates how the possible gadgets look like.

Recently, techniques based on Control Flow Integrity (CFI) and code length arose

to counter such attacks with reasonable effectiveness. However, many issues are still posed by them, such as recompilation or code-injection. One of our goals is to propose a hardware-assisted system that overcomes limitations of state-of-the-art, hardware-assisted approaches to detect ROP attacks in real-time.

Performance monitoring

Modern processors have many sensors on their platforms which allow for monitoring performance event indicators. These sensors may also be used for other purposes, such as security ones, as proposed in this work. Each vendor presents their own set of monitors—Intel [78], AMD [5], and ARM [7]. Due to availability issues, this work is based on the Intel platform. It is composed of two sub systems: PEBS (Precise Event Based Sampling) and LBR/BTS (Last Branch Record/Branch Trace Store). The first is responsible for collecting information about general system events, such as instructions retired, cache hit/misses, branches predicted and so on. The second is responsible for monitoring control flow deviations. It can store both source and target addresses of deviation instructions. Despite being called branch monitors, they can monitor any control flow deviation instruction, including `CALL`, `RET` and exceptions, beside ordinary branch ones (`JMP`,`JNE`). Figure 2.5 illustrates a branch stack.

FROM	TO
0x0FFFF418	0x0FFF8F36
0x0FFF1510	0x0FFFCF2E
0x0FFF8014	0x0FFF0523
0x0FFF81b3	0x0FFFE057

Figure 2.5: Example of a Branch Stack.

Both schemes have two storage options: register and memory-based. The first option allows the system to store data in a limited number of specific purpose MSR registers, whereas the second one provides an unlimited storage in OS memory pages. The branch monitoring mechanism is named LBR when operating in the first mode and BTS in the second. Collecting data in MSR registers requires polling, which may cause data loss in the LBR case. The memory-based approach for PEBS and BTS, in turn, have the advantage of having the ability to generate an interrupt when a given threshold is reached; this way no data is lost by the capturing mechanism.

Both systems are activated by setting special flags in MSR controls and impose theoretically zero overhead, since they are processor features. For MSR access a kernel driver is required. In addition, as a processor feature, a physical machine is required, since virtual machines do not emulate such special MSRs. The mechanisms operate in a system-wide manner. Therefore, no process information is available for filtering. The LBR mechanism, however, is able to filter deviation types (branch, call or ret). Both LBR and BTS can filter at the capture level—kernel or userland.

Threat model

The assumptions presented here will guide the solution's development and evaluation. We have the following threat model for malware analysis and debug scenario.

- **Evasive malware**: we target samples with virtual machine detection as the anti-analysis mechanism.
- **Userland threats**: we assume that samples will execute in userland and no kernel activity is performed. This assures the integrity of our kernel driver, as stated by Rossow et al. Monitoring userland threats is a frequent assumption on malware sandboxes. It can be seen in solutions like Cuckoo Sandbox [71] and CWSandbox [208].
- **Transparent analysis**: we assume that our solution will be running on a physical machine with performance monitoring support.
- **System API usage**: we assume that sample-OS interactions are performed through default system APIs. This allows us to introspect system addresses in order to reconstruct sample flows.
- **Modern OS**: we assume that the execution environment is a modern OS, where kernel instrumentation is denied by modern protection mechanisms.

Below, we show the threat model for the ROP scenario.

- **Return-based**: we assume that attacks are based on return gadgets, ignoring other forms of code reuse attacks, such as Jump- or Loop-oriented ones.
- **Unobtrusive monitoring**: we aim to monitor software without any code injection or emulation.
- **No prior information**: our solution is aimed to monitor any binary in the system without additional information about it.

Related work

Transparent malware analysis Currently, two main kinds of techniques are employed in transparent malware analysis: HVM (Hardware Virtual Machines) and SMM (System Management Mode) instrumentation. HVM are systems which rely on virtualization instructions available in modern processors—Intel VT-x and AMD SVM—to build a transparent environment, since these technologies allow running code on the physical processor. In addition, they offer instrumentation facilities, such as double page translation mechanisms. The transitions from root to non-root mode also provide a way to monitor system events. Systems like the malware tracers Ether [55] and MAVMM [136] make use of this technique to build their transparent systems. SMM mode, in turn, is a specific processor mode to manage the system at a very low level. It consists of an executable portion of

code resident in the system BIOS, triggered by special interrupts called System Management Interrupt (SMI). This mode is well isolated from other execution modes by address redirection. It allows transparent execution since it is bare metal based. Systems like MALT [219] and SPECTRE [221] make use of SMM instrumentation to transparently monitor systems. One main disadvantage of HVM and SMM approaches is their development complexity: HVM requires writing an instrumented hypervisor. In some cases, such as in MAVMM, a hypervisor has to be built from scratch, since a minimal Trusted Code Base (TCB) is required. SMM, in turn, requires rewriting BIOS code—a task allowed only on unlocked systems. In addition, such systems cannot rely on any library, given their low-level placement. Another issue is the overhead imposed by the instrumentation routines. HVM exits may impose an overhead of the same magnitude as the system execution's, such as on Ether's case [55], therefore being impractical for some uses. Our solution intends to be a lightweight version in the same line as these approaches, allowing native code execution and low level inspection but with a significant reduction in overhead and development efforts.

Debugger Most of the current debugger developments are focused on complex software architectures, such as high level constructions [40], distributed systems [101, 168, 74], and GPU programming [175]. However, few efforts were made toward a more transparent debugger. The closest attempts to build a transparent debugger as proposed in our work were made by MALT [219] and HyperDBG [61], which employ SMM introspection and HVM, respectively. Our work intends to be a lightweight version in the same line as these approaches, requiring smaller costs of development and performance, although it implies on some restrictions, such as analysis inside the kernel and therefore requiring protection to avoid kernel subversion.

ROP attacks ROP detection approaches can be classified in compilation-time, instrumentation, binary-rewriting and run-time. Approaches based on compilation time such as control flow locking [21, 139] aim to avoid vulnerable constructions generation, thus reducing ROP exploitability. The main disadvantage of this approach is that it cannot be applied to existing binaries. Instrumentation approaches [49, 35, 68] are applicable to existing binaries without recompilation. These solutions aim to detect exploitation in real-time. However, they suffer from limitations of the instrumenting tools they are built on. Binary rewriting solutions [73, 141, 206] can also be applied to existing binaries and do not require instrumentation. They rewrite the binary on first execution, hardening it against exploitation. The main disadvantage of this approach is its limitation to handle dynamically generated code. A broader approach is to leverage hardware monitors in order to inspect application flows. This approach can be applied to existing binaries and do not rely on emulated environments. Hypercrop [86], for instance, leverages HVM to identify ROP attacks. However, its overhead is prohibitive. A lightweight approach to hardware assisted monitoring is to leverage performance counters. ROPecker [38] and KBouncer [142] make use of the LBR mechanism to identify and counter ROP attacks. ROPecker and KBouncer are the closest related to ours in the scope of ROP detection. However, they present some limitations, such as using the limited LBR instead of the BTS

storage and requiring DLL injection. Our work is intended to overcome such limitations.

Branch Monitoring Distinct solutions have been deployed using performance monitors in a general way. Beside the ones on ROP detection, they were also applied in other attack evaluations. Yuan et al. [217] relies on performance data provided by Linux Perf[6] in order to identify attacks. Kompalli [92] works in a similar way but its underlying tool is Intel Vtune[7]. Both solutions, however, are intended to detect system misbehavior in a general way, whereas our proposal is to trace specific process activity. In this sense, the work closest to ours is [209], which is able to rebuild some traces from a program crash. However, this solution is not aimed at Control Flow Graph (CFG) reconstruction or debugging. It is also limited to using the few LBR registers. Our work solves it and provides a broader solution to malware tracing.

The usage of LBR and BTS. Most solutions based on branch monitoring make use of the LBR mechanism instead of the BTS one. This way, many comparisons made in this work consider only LBR, such as on ROPecker's and KBouncer's study cases. CFIMon [212], for instance, makes use of BTS to enforce a CFI policy. Unlike this work, it uses a 2-phase heuristics, which requires binary prior-analysis. This way, it is not directly comparable to this work, which implements a 1-phase policy, such as in KBouncer.

The work in [3] is the only one we are aware which addresses the specific usage of BTS for security purposes. In such work, BTS is used to validate control flow path transitions. The work hereby presented, however, differs from it in many ways, since our goal is to implement security tools, such as malware tracers and debugging facilities, whereas the cited work is concerned only with implementing runtime policy enforcement. Its syscall trace system is more focused on abnormal behavior detection than on tracing a given binary itself, which is proposed by us. Given these differences, we are able to provide more flexible solutions, which allows us, for instance, to reduce overhead. Moreover, our solution presents the same advantages of the aforementioned approach, such as not requiring binary modifications.

The work of Soffa et al [180] makes use of both LBR and BTS in order to discuss the application of hardware monitors in the context of software engineering. Although this work had already suggested branch monitor use in order to track executed paths, it can be considered as a first step, since many aspects needed to be further discussed, such as: i) External Library Inspection; ii) Branch-capture granularity; iii) Process Isolation; iv) Implementation aspects. In this scope, the hereby presented work can be considered as a second-step, discussing these missing points and presenting real-world sample evaluation.

Proposed framework

The framework is general in nature and can be applied for collecting and evaluating control flow deviation data, in particular by the applications developed as part of this work, which implements extensions to the framework in order to apply security policies.

[6] `https://perf.wiki.kernel.org`

[7] `https://software.intel.com/en-us/intel-vtune-amplifier-xe`

We are mainly concerned with tracing program paths, so we adopted the branch monitor subsystem of the performance monitor hardware as our base mechanism. Given that it is able to collect the address of executed instructions, we can reconstruct the whole scenario of code execution over binaries, libraries and function calls through introspection.

To avoid losing instruction data, we opted for the BTS mechanism instead of the LBR one. The advantage relies on the ability to make use of interrupts on our system, which assures its state is coherent at inspection time. We defined a 1-instruction threshold; this implies the system will be interrupted at each control flow deviation instruction, therefore warranting the precise identification of which process is executing such instruction. This way we can easily and stealthly filter process actions, even though the BTS system itself is unable to provide such information. Conversely, solutions using LBR require intrusive hooks to provide similar capabilities, such as in [2].

Our system architecture is designed as a client-server one, in which the kernel driver is responsible for managing BTS data (server) and the user-land application for applying policies on the collected data (client). Data collection may be synchronous or asynchronous, in order to best fit a given policy. Data is collected in a system-wide fashion as it is provided by the BTS mechanism and filtered in the user-land client, so that distinct policy implementations are allowed, as shown on Figure 2.6.

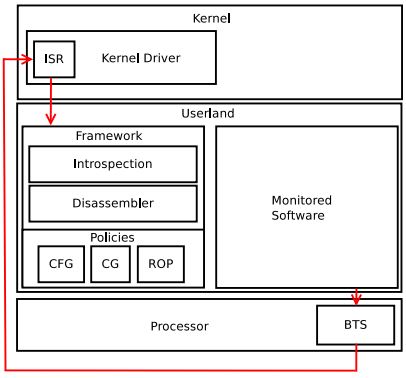

Figure 2.6: Proposed Architecture. The processor fetches branch instructions from the monitored code, which trigger the BTS threshold. The raised interrupt is handled by an ISR at a kernel driver. The captured data is sent to the userland framework where introspection and disassembling are performed and policies are applied.

The proposed architecture does not require any injection or interaction with the analyzed process, as data is collected by the processor and processed by distinct, independent pieces of software. It also provides transparent execution, since instructions are executed on the real processor. This framework is also easier to implement than other transparent approaches: it requires only a kernel driver and additional user-land software, with no need to write a hypervisor or rewrite the BIOS.

As for the solution's performance, it presents theoretically zero[8] overhead at capture time, since it is performed on hardware, plus some overhead on analysis and/or policy application. This overhead, however, is application-dependant. As our goal is to minimize this overhead, we projected the system so that we could make many tasks using offline

[8]The BTS monitor by itself, not considering data collection and analysis

processing, such as address introspection in case of a simple trace. However, some tasks require online processing, as is the case with attack detection; as policies are implemented as independent programs, we can run them on a different processor thread/core, which minimizes performance impact over the analyzed software.

In the next sections, we cover details about the framework implementation and the characteristics that make it flexible. It was implemented on 32- and 64-bit versions of Windows 7 and 8[9].

Driver: all about the basis

We need to access the `debugctl` MSR register, thus requiring the deployment of a driver. This driver is also required for allocating and supplying OS memory pages to the BTS mechanism, as well as providing the Interrupt Service Routine (ISR) to handle BTS interrupts and the I/O routines which send data from kernel to user-space. Each of these features are detailed next.

Handling Interrupts

Interrupt handling is the most important step of data collection, because this is where data preservation is effectively performed. As BTS data is stored in memory pages, we can collect it by simply using a pointer. Installing the ISR, however, is the hard part: the BTS interrupt mechanism looks into the Local Vector Table (LVT) to find out how to deliver the interrupt. The LVT defines if it is delivered through an SMI, NMI or fixed mode (the default option). In the latter case, an entry into the Interrupt Description Table (IDT) is performed. The ISR address is placed on the corresponding position of the IDT. On Windows systems, the defined IDT entry may already be allocated to another portion of the system. Changing the LVT vector offset could be an option but, in our tests, no IDT entry was available. Hooking IDT is not an option anymore on newer Windows versions since the Patch Guard mechanism prevents it.

Hooking the specific performance handler could be an option—it could be done by calling the `_HalpSetSystemInformation()` from `HalDispatchTable` in order to change the `HalpPerfInterruptHandler`—but this presents side effects. A non-hooking solution is to change the delivery mode on LVT to Non-Maskable Interrupt (NMI), a high priority interrupt originally aimed at exception checking. As an NMI interruption is immediately handled, it is a good choice for our monitoring goal. The NMI ISR is registered using the `KeRegisterNmiCallback` [108] function. When an NMI happens, process execution is suspended so that we can correctly retrieve its PID. It is performed by using the `PsGetCurrentProcessId` [119] function. This information, along BTS branch data, are stored on a queue, detailed below, to be collected by an I/O call. Finally, the ISR is also responsible for re-enabling the BTS interrupt mechanism, since it is disabled as soon as an interrupt happens.

[9]We chose to implement such solution on Windows since it is the desktop OS most targetted by malware samples

Handling Data

When an interrupt occurs, the BTS buffer is full, so we have to copy its data to some other place in order to free the buffer and re-enable the monitor. In our solution, BTS buffer entries are copied to a Windows kernel list, pushed in a FIFO basis, in order to keep proper order. In the event the buffer is not freed and the monitor is just re-enabled, entries would be overwritten, thus causing the same effect of using the LBR monitor.

Performing I/O

The client receives data from kernel through I/O routines, with each application presenting distinct requirements for data collection. Applications that are intended to provide real time monitoring may require an asynchronous I/O in order to get results as soon as possible, whereas tracing tools may get delayed data through synchronous I/O. Each mode is detailed below:

Synchronous I/O In this mode, ISR collected data is enqueued on a circular kernel list [123] in a FIFO way. Data is transferred to user-mode though IRP [117] generated from a ReadFile [120] call, since the driver handle is opened as a file. The client periodically asks for more data through polling the ReadFile call. As the queue follows the FIFO rule, the data corresponds to ordered events.

Polling frequency In this mode, BTS data will generate data at a rate higher than the consumption by the polling client. However, no data is lost since it is moved from the BTS entry to the kernel list. The driver client, however, should define a compatible polling frequency in order to not exhaust kernel memory. In our tests, a second-long interval was enough.

Asynchronous I/O In this mode, the collected data is not stored on a queue, but on a single structure, since it is expected to be consumed as soon as it is retrieved. Once an interrupt occurs, the driver fires a previously cached I/O request in order to alert user-mode code that the requested data is available. The client is then responsible for releasing the I/O routine and collecting the stored data. The client should first release the I/O since an interrupt is intended to be fast, being protected from locking by a timer watchdog. This collection mode is named Inverted Call. Notice that the client must have distinct threads for handling the kernel call and processing the data independently, thus not blocking the ISR and not causing data loss.

What happens after an interrupt

It is natural to think that interrupts will keep being raised during the ISR, which would overload the system. However, the BTS mechanism has some filtering features which help us deal with this. The main one is the execution level filter, which allows us to disable interrupts generated by the kernel, thus no branch generated by the ISR is captured.

Handling monitor branch data

As the BTS captures branches in a system-wide way, the client-generated branches could also be captured by the client itself. A direct way of preventing such behavior is to run the client on a core distinct from the one the monitor is running on. An alternative approach would be to perform PID tracking in the kernel.

Clients: where the magic happens

The user-land clients are the active analyzers of our system. They are responsible for retrieving the data collected from the driver and applying their policies. They can be built with complete independence from the drivers. To exemplify this claim, we have implemented clients both in C and Python. The clients are responsible for keeping system information in memory and to use them for the analysis process. The basic information processing they perform are introspection and context retrieval mechanisms, which aim to enrich the raw data collected. We detail both below.

Introspection

The information provided by the BTS mechanism (addresses) have very little meaning in the context of a program execution. These addresses must be translated into high level constructs so that analysts could gain more knowledge about the system state. As instruction addresses point to the main memory, we can identify which loaded modules are resident in each memory region. The loaded modules may be the main binary or dynamic library code images; in the latter case, we can dig into their structure to identify known function addresses/offsets, giving information about which functions were called and/or what is being executed. The same could be done for binary images if we had debug symbols, which is not usually the case.

Code images in memory can be enumerated by using the `EnumProcessModules` [114] function. Each of their base locations can be retrieved with the `GetModuleHandle` [115] function. However, code images change their placement at every system startup due to the Address Space Layout Randomization (ASLR) mechanism, as shown in Table 2.10. Therefore, as we run on a bare metal system[10], which requires rebooting for state restoring, this code image address enumeration procedure should be repeated at every boot, thus being intrinsically ASLR-aware. If the intended usage scenario is not a sandbox, which requires rebooting, one may just repeat this procedure before every process invocation in order to get per-process, ASLR-aware data.

Table 2.10: ASLR - Library placement after two consecutive reboots.

Library	NTDLL	KERNEL32	KERNELBASE
Address 1	0xBAF80000	0xB9610000	0xB8190000
Address 2	0x987B0000	0x98670000	0x958C0000

[10]By bare metal we mean a physical machine running an actual processor, with no emulation or virtualization.

Given a BTS-provided address, we can identify the corresponding library it refers to by looking to the closest base address previously retrieved from module enumeration. By looking to the difference between the base address of a given library and the address pointed to by the BTS, we can compute an offset, which is mapped to a library internal function, thus leading to a higher level semantic construct. Function offsets can be obtained by inspecting host libraries through the use of tools like `DLL Export Viewer` [137]. Table 2.11 shows function offsets for the `NTDLL` library, as an example. The whole introspection process is illustrated in Figure 2.7. It is important to notice that such offset extraction occurs automatically before analysis begins by considering a list of module names supplied by the analyst. Once the extraction is performed, an offset database is created. We are able to reuse such data since, unlike module addresses, function offsets do not change due to ASLR.

Table 2.11: Function Offsets from `ntdll.dll` library.

Function	***Offset***
NtCreateProcess	0x3691
NtCreateProcessEx	0x30B0
NtCreateProfile	0x36A1
NtCreateProfileEx	0x36B1
NtCreateResourceManager	0x36C1
NtCreateSemaphore	0x36D1
NtCreateSymbolicLinkObject	0x36E1
NtCreateThread	0x30C0
NtCreateThreadEx	0x36F1

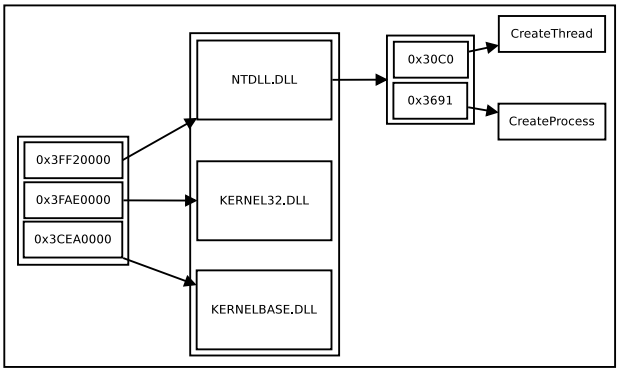

Figure 2.7: Instrospection mechanism: from raw addresses to functions.

Looking into memory

In addition to looking into what an address represents, sometimes it is useful to look to what the address content is—it can be executed instructions, as directly pointed to by the BTS mechanism, or function arguments, given by an introspection process. Given a memory address, the contents can be retrieved by using the `ReadProcessMemory` API [121]. It is important to notice we are allowed to perform such memory read since our framework

execute with administrative privileges. It is also important to notice that memory reads, unlike writes, do not violate the transparency requirement.

The ability to read memory allows us to read instruction bytes, which can be used to enrich the software analysis. However, the bytes need to be translated into a higher-level construct in order to be understood, i.e., instruction opcodes. Given an instruction address, we can easily get the opcode from the instruction represented by the first byte of the memory dump by using a simple table. The bytes representing the CALL and RET instructions are shown respectively in Table 2.12 and Table 2.13. Notice that the same opcode may have slightly different meanings according to the following bytes (addressing modes). However, our solution does not need to look to these immediate values to calculate addresses, since the branch monitor provides the already calculated target addresses. This way, we can only look to the first byte and thus speed up some kind of instruction interpretation, as is required for the ROP detector through the CALL-RET CFI policy.

Table 2.12: `CALL` Opcodes.

Opcode	Mnemonic	Opcode	Mnemonic
0xE8	CALL rel16	0x9A	CALL ptr16:16
0xE8	CALL rel32	0x9A	CALL ptr16:32
0xFF	CALL r/m16	0xFF	CALL m16:16
0xFF	CALL r/m32	0xFF	CALL m16:32

Table 2.13: `RET` Opcodes.

Opcode	Mnemonic	Opcode	Mnemonic
0xC3	RET	0xC2	RET imm16
0xCB	RET	0xCA	RET imm16

Despite disassembling only one byte, we may also face the scenario in which a block of code is provided. As x86 instructions are not fixed-size, we need to find out if the following bytes are immediate arguments or a new, following instruction. The disassembly of multiple instructions in our system is performed by two libraries: Pybfd [69], a Python interface for `libopcodes` on Linux, is used for offline processing whereas Capstone [33], a Windows disassembler, is used for real-time processing.

Besides knowing how to interpret instructions from a memory dump, we need to know how to retrieve addresses from branch information. The straightforward dump of the first byte indicated by some branch data is not able to identify all instructions executed. That would require additional data, which can be obtained by looking to two consecutive branches. The destination address of the first branch indicates a place where the execution will start; the source address of a consecutive branch indicates that the code execution left the block at that point. As no other branch may have occurred, all intermediate instructions were effectively executed. Therefore, reading memory starting on the first address up to the second leads to all executed instructions. Figure 2.8 illustrates it with data from Listing 2.1. The execution flow enters a block of code at the first branch target address (`0x48ff5ab8`) and leaves it on the source address of the next taken branch (`0x48ff5ac0`). As no other deviation occurred, all instructions stored in that range were effectively executed. The opcodes of such instructions are identified through the disassembly of 8-bytes (the exit address minus the entry address) starting from the entry address.

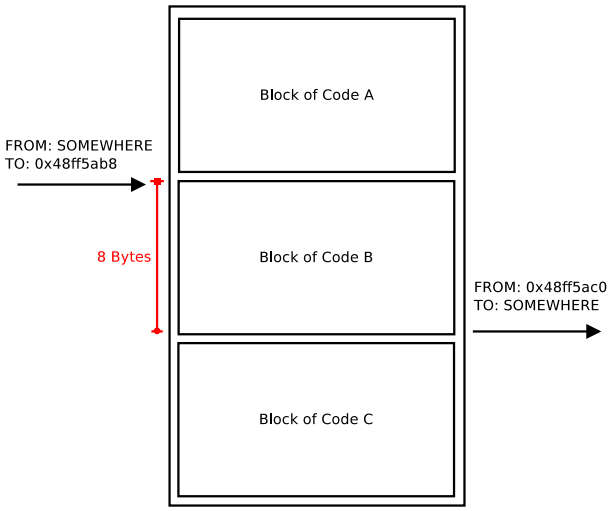

Figure 2.8: Block identification of two 8-bytes consecutive branches.

Listing 2.1: Block identification—from `0x48ff5ab8` to `0x48ff5ac0`

```
1 PID: 4876 FROM: 0x48ff5ab0 TO: 0
      x48ff5ab8
2 PID: 4876 FROM: 0x48ff5ac0 TO: 0
      x48ff5ad0
```

Validation

In order to validate our framework, we have implemented the same ideas on Intel PIN, a dynamic binary translator. In the emulated prototype, we considered only branch data and reconstructed instruction blocks by relying on two consecutive branches, as shown in Listing 2.2.

Listing 2.2: Instruction Instrumentation on PIN.

```
1 VOID Instruction(INS ins,
      VOID *v){
2     if(INS_IsBranchOrCall(
          ins))
3         Disasm(last,current)
```

We have run sample programs on both solutions and compared the results for each execution, considering the framework as correct since all of them matched. As an example, Listing 2.3 and Listing 2.4 present the execution of a given piece of code under PIN and our solution, respectively. One can verify that the execution of the same block (`0x90` offset)[11] resulted on the same number of disassembled instructions (`0xc96 - 0xc90 = 0x7`).

[11]Base addresses are changed on distinct executions

Listing 2.3: Sample code running under PIN.

```
1 From: 0000000077332F89 To: 0x7732ec90 Disasm of 1 instr: call
2 From: 000000007732EC97 To: 0x7732ecab Disasm of 1 instr: jnz
3 Disasm of 0x7 bytes from 000000007732EC90: 0x48 0x3b 0xd 0x39
      0x8e 0xe 0x0
```

Listing 2.4: Sample code running under Branch Monitor.

```
1 Binary Branch.Tester.exe at <0x1ca1> to Binary Branch.Tester.
     exe at <0x1c90>
2 Binary Branch.Tester.exe at <0x1c96> to Binary Branch.Tester.
     exe at <0x1c9a>
3 should disasm from 7ff6d6ec1c90 to 7ff6d6ec1c96
```

Applications

In this section, we present security applications built upon the presented framework. Each of them makes use of a distinct feature from it, exemplifying distinct ways branch monitors can be applied for security purposes. For the sake of readability, since capturing data at branch level produces huge amounts of data, we present CG and CFG reconstruction based on minimal examples. However, the evaluation tests presented on further sections, such as 2.18.5 and 2.18.6, are based on real samples. The complete logs for such samples can be found on the project page [12].

Malware Tracer

Tracing malware is an important step for malware analysis procedures. Traces can provide information about malware behavior and its interaction with the system, which can be used to group similar samples, develop anti-virus vaccines, patch vulnerabilities, and so on. Our malware tracer follows directly from the data obtained from the BTS mechanism, as instructions are supplied. In addition, we can enrich the data by adding extra information, as presented in Section 2.17. We have implemented two analysis features in our prototype: a call-graph viewer and a Control Flow Graph (CFG) rebuilder. The first allows for identifying a sample's behavior whereas the second can provide granular information about executed instructions, which allows heuristics development like one based on tainting.

Call Graph

The CG represents function calls and their relationships. To exemplify the CG visualization application, we will rely on a simple code whose host process was named `NewToy.exe`: `scanf("%s",val); printf("%s\n",val);`. `CALL` and `RET` instructions are directly captured by the BTS mechanism and function identification is performed by the previously mentioned introspection process. However, the BTS mechanism has no filter itself, incurring in the capture of the `CALL` and `RET` instructions inside libraries in a given process

[12] `https://sites.google.com/site/branchmonitoringproject/`

scope. Following code inside libraries might be useful in some situations, but not always. So, we provide the ability to skip these instructions using a filter in the client. This selection may be understood as debugging's `Step Into` and `Step Over` navigation.

Step Into Figure 2.9 shows an excerpt from the Step-Into CG from the example code, presenting the analysis of `printf` function internals. After the binary under analysis calls the `printf` entry point, we can find `calls` to internal functions responsible for the `printf` behavior—`locks`, for instance—which assures I/O ordering, since `printf` is a non-reentrant function.

Figure 2.9: Step-Into call graph, all intermediate calls represented.

Step Over Figure 2.10 shows the Step-Over CG from the same code in which only the called functions appear. The presented excerpt covers the `return` from the initial `scanf` function at the `0x3f` offset to our binary (`NewToy`) and then the `call` to the entry point of the `printf` function, which will print the read value. Internal aspects of both functions are ommited in this mode.

Figure 2.10: Step-Over call graph, only CALL/RET represented.

Control Flow Graph

CFG is the most granular inspection view possible of a code at the instruction level. By inspecting it, one can perform taint analysis [170] or identify malicious payloads [135, 216]. In order to rebuild the CFG of a given sample, we rely on the disassembly solution previously presented. We apply it repeatedly so that we could retrieve information about each block surrounded by two deviation instructions. Similar to CG's case, the captured data also contains information about library internals. Here, our sole interest is about binary information, so we used the same approach of the Step-Over filtering. For example, the code from Listing 2.5 results on the corresponding CFG of Figure 2.11.

Listing 2.5: Example code for CFG validation purposes.

```
1 a=0;
2 scanf("%d",&n);
3 for(i=0;i<n;i++)
4       if(i%2==0)
5             a++;
6       else
7             a--;
8       printf("%d\n",a);
```

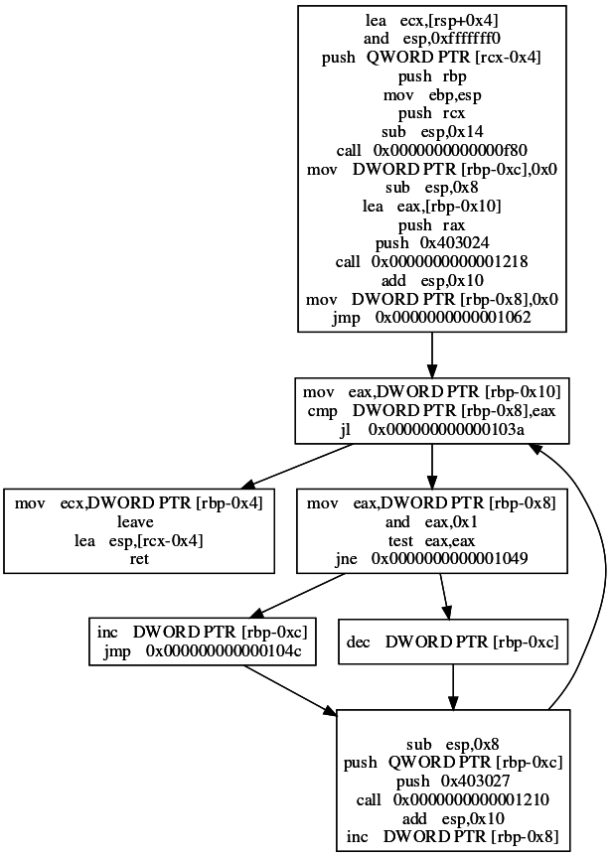

Figure 2.11: Reconstructed CFG from Listing 2.5 example code.

We can observe a match between the presented code and the generated CFG, where the first block is related to set up routines, such as pushing the stack frame. This block leads to a decision block related to the `for` statement. If the iteration reached its limit, the code proceeds to the left block, where the `main` function is finished by the execution of a `ret` instruction. Otherwise, the execution proceeds to another decision block, associated to the `if` statement—we should notice the `and eax,0x1` instruction, associated to the decision calculation. Even values result in the left block being taken (notice the `a++`

represented as an `inc`) and odd ones in the right one (notice the `a-` represented as a `dec`). The `call` in the last block is the `printf` invocation. The execution iterates through the `for` backward edge.

As the provided branch information is related to the effectively taken deviation, our solution has the advantage of capturing and disassembling real instructions; it does not suffer from alignment tricks (often used for anti-analysis), which is a common problem on static disassembly.

A brief comparison. Some may find similarities between the hereby presented approach and the one presented by Paleari in his Fuzztrace tool[13], detailed in a blog post[14]. In addition to having distinct purposes, the solutions present other differences. Paleari rebuilds the CFG based on perf-supplied edges. The presented study case is a heat map of executed blocks, a task which our solution is also able to perform. Since our framework is modular, a heat map policy would require writing a few lines of code.

Paleari's solution, however, is more limited since it does not track external function calls. Our solution, instead, is able to track and introspect into these functions. In addition, we are able to filter how deep we dig into these libraries by selecting the step-into and step-over modes. In the step-into mode, the CFG is rebuilt in the same way as in the `viewer` tool from Fuzztrace. In the step-over mode, however, a stack is used to filter out instruction blocks according to the monitored code (internal or library). All details about our CFG reconstruction algorithm are presented in the next sections.

Additionaly, Fuzztrace only provides instruction addresses as tracing data, which requires performing a static disassembly in order to match addresses and instructions. Conversely, our solution is able to perform online, dynamic code disassembly, providing the executed block as tracing data.

Self Modifying Code. Many malware samples perform in-memory code changes, also known as Self Modifying Code (SMC) [214, 50]. This is a technique often used for packing samples in order to avoid static detection. Our solution is able to handle packed samples since we can monitor their whole behavior, during and after unpacking (intended execution). In case one wants to monitor the code modification itself, the tracer needs to be run using asynchronous I/O since, in order to reduce overhead, we have implemented the presented version using synchronous I/O, which causes delayed code memory reads. As an advantage, the SMC detection can be performed concurrently with the CFG reconstruction, by the same algorithm, as shown in the next section/paragraph.

The CFG-SMC algorithm. In this section, we detail the CFG reconstruction, covering the step-over execution and external function calls. We also show how we can use the same algorithm (Algorithm 1) to perform SMC detection. The algorithm takes as input a list of instruction blocks and the flags which enable/disable the step-over and SMC detection modes. The algorithm iterates over the instruction block list (line 6), updating the current and previous blocks (line 28), adding edges between them when needed (line 26).

In the step-over mode (line 7), library `CALLs` (line 9) will be exit nodes from the graph whereas `RETs` (line 11) will be entering nodes. In this mode, the instructions in between

[13] `https://github.com/rpaleari/fuzztrace`

[14] `http://roberto.greyhats.it/2015/02/fast-coverage-analysis-for-binary.html`

are not considered, thus the `pass` instruction (line 14). As these blocks are passed, the previous node is kept in the `CALL` instruction and later edge-linked to the node coming after the `RET`. Notice that when the step-over mode is enabled, as the libraries are pushed into the stack, the library nodes are printed at a distinct CFG level. If necessary, one may instrument the pass step to generate the whole CFG for a given level (the library CFG, for instance). An example of such generation modes is provided below.

Figure 2.12 shows the step-into case. As the stack is limited to the first level (`binary`), the library internal code, highlighted in the internal bounded box, is inserted as ordinary binary code blocks. Figure 2.13 shows the step-over case. As the stack changes from `binary` to `library`, these are printed at distinct levels. The two bounded boxes present, respectively, the binary code (and its library call node) and the library code itself. An existing corner case is related to branches whose targets are instructions inside other blocks. In this case (line 19), the existing block has to be split (line 20) in order to keep the CFG's definition (set of non-branch instructions limited by a branch).

The same block traverse algorithm can be used as base for SMC detection. In this case, a shadow memory is used, thus requiring additional memory. When a block node is created (line 15), its memory content is hashed and stored in a shadow block (line 18). Notice that when a block is split (line 20), block hashes need to be updated (line 22). After the point where the current block is defined, we can check whether the current block hash matches its shadow (line 24). In case any difference is found, an SMC code is identified (line 25). The detection routine can be used to immediately return or update the block hash and keep detecting code changes. Notice that in the SMC case distinct CFG visualization modes should be used, since the dynamic block content makes plotting harder.

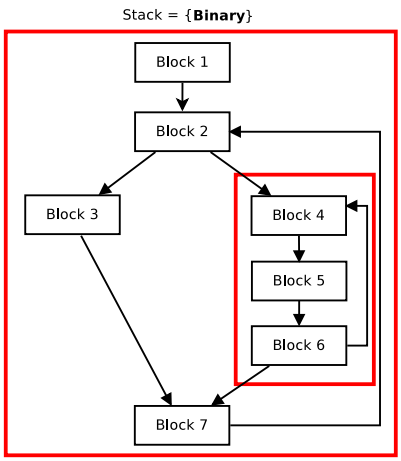

Figure 2.12: Step-into CFG.

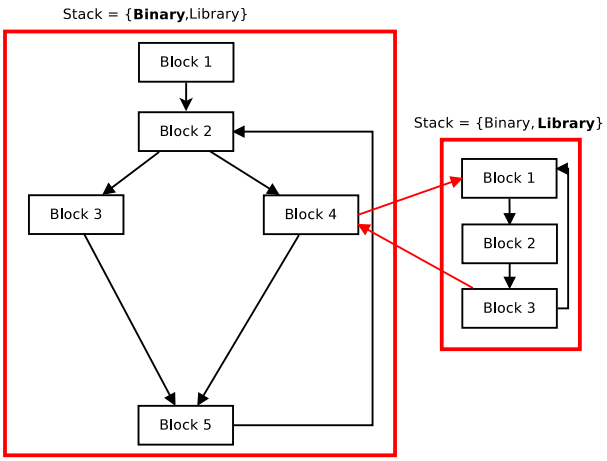

Figure 2.13: Step-over CFG.

Modular malware

Many malware use modular approaches to deploy the functions required for infection, such as dropping a file or downloading a payload from the Internet. This way, splitting their maliciousness through many processes actually presents a lower malicious profile.

ALGORITHM 1: CFG reconstruction and SMC detection

```
   Data: blocks, StepOver, SMC.Detection
   Result: CFG, SMC.Detected
 1 Stack = {Binary}
 2 CFG = ∅
 3 Shadow = ∅
 4 create_node(block[0],Stack,CFG)
 5 previous = block[0]
   /* Iterate over the blocks in an ordered way                          */
 6 for current in blocks[1:n] do
       /* Case filtering is enabled                                      */
 7     if StepOver then
           /* Case it's a library.  Otherwise, run                       */
 8         if Introspection(current) is LIBRARY then
               /* CALLs are pushed on the stack                          */
 9             if Instruction(current) is CALL then
10                 Stack.push(Library(current))
               /* RETs pop data from the stack                           */
11             elseif Instruction(current) is RET then
12                 Stack.pop()
               /* Ignore any other internal instruction at the for level */
13             else
14                 Pass
                   /* Instead of passing, one can generate a CFG for the
                      library                                             */
       /* Create non-existing nodes in the graph                         */
15     if not node_exists(current,CFG) then
16         create_node(current,Stack,CFG)
           /* Case SMC.Detection is enabled, compute the block hash the
              first time                                                  */
17         if SMC.Detection then
18             shadow[current]=Hash(Instruction(current))
       /* Case there's a branch to the middle of a previous block        */
19     elseif not match_first_address(current,CFG) then
           /* split the previous block                                   */
20         current, splitted = split_CFG(current,CFG)
           /* Update block hashes to include the splitted one            */
21         if SMC.Detection then
22             shadow[splitted]=Hash(Instruction(splitted))
23     if SMC.Detection then
           /* Check current block has the same content than before       */
24         if shadow[current] is not Hash(Instruction(current)) then
25             SMC.Detected()
       /* add edges                                                      */
26     if not edge_exists(previous,current,Stack,CFG) then
27         create_edge(previous,current,Stack,CFG)
28     previous=current
```

This effectively achieves a lower malware ranking among Anti-Virus tools and is currently a common behavior on modern malware samples. Although our introspection process is able to identify the call to APIs such as `CreateProcess` [110], we are not able to collect the created process PID and thus not able to filter its activities. In order to overcome this limitation, we installed a Process callback [111] which delivers new PIDs to our client to be monitored. This way, the created process is added to the monitored list plus the initial PID, which could be a suspended process, as usual on most sandboxes solutions, or even a running process whose PID is known.

Real malware tests

As our tracing tool is built upon our framework, it allows malware analysis in a transparent way. In order to validate such property, we ran some evasive samples in our environment so as to verify if they executed as expected in a real, victim machine. The samples choice was based on static analysis tools that identify Anti-VM techniques. We selected four samples[15] said to employ QEMU tricks according to PEframe (`https://github.com/guelfoweb/peframe`) and executed them in our proposed framework and in our QEMU-based internal sandbox solution (unpublished). All of them evaded the execution in the QEMU-based sandbox, not providing any useful behavior to be analyzed. However, they executed normally under our proposed malware tracer's monitoring.

Debugger

The malware tracer allows us to understand a great deal of a sample's behavior through its execution, but it is not able to suspend the execution at an arbitrary point in order to provide a deeper introspection view. This could be a useful approach for detecting bugs or complex constructions, especially those with stealth attack intentions. Extending our framework to provide such facility is a straightforward path.

Project

A debugger presents some differences on how to use the framework's features. We present an overview in this section.

Goals achievement In order to achieve the **small step execution** goal, we rely on the BTS mechanism. Although it does not allow step-by-step execution, it provides sequential block-by-block granularity which, with the help of a block disassembler, brings basically the same functionality. The **breakpoint information** goal is achieved by relying on introspection during interrupts. Finally, the **context inspection** goal is achieved by using system APIs. The data consistency is assured due to the raised interrupt which precedes API calls.

[15] Samples MD5: f03c0df1f046197019e12f3b41ad8fb2, 2b647bdf374a2d047561212c603f54ea, 7a4b29df077d16c1c186f57403a94356, 340573dd85cf72cdce68c9ddf7abcce6

Debugger working flow As we need to suspend the process execution to inspect it, the strategy here is different from the Tracer's. In addition, the process suspension must proceed as soon as an interrupt occurs; to accomplish this, we made use of the inverted I/O call. The debugger working flow is as follows: (i) at a given moment, the processor fetches a deviation instruction whose source and target addresses are stored by the BTS mechanism; (ii) an interrupt is then raised since we have defined a 1-threshold—at this point, the process under analysis is active, but interrupted; (iii) the ISR routine releases the cached I/O in order to alert the user-mode client, which receives the alert, suspends the process execution and finishes the I/O routine; (iv) when the ISR receives the I/O completion signal, the interrupt is released and the process is now in suspended state; (v) then, all introspection and context retrieval processes take place; (vi) when the process is resumed in the client, the whole debugging process is restarted.

Debugger resources One of the most important resources in a debugger is its inspection capabilities. Our solution presents the following ones:

- **Process management**: our solution is able to create a new suspended process to be inspected or to attach to an existing one.
- **State inspection**: our solution is able to identify function execution, loaded libraries and to read context registers.
- **Step execution**: our solution is able to perform branch step execution at the block, function and library levels.
- **Integration**: our solution can be integrated to other debugging tools, such as GDB.

Debugger client implementation

Although the client was built upon our framework, some features were implemented in the client itself. The details are given below.

Process management Processes are created using the `CreateProcess` API. Processes need to be at the suspend state in order to be inspected consistently. Therefore, new processes are created using the `CREATE_SUSPENDED` flag whereas existing ones should be suspended by calling a specific API. There are three known methods for suspending a process: (i) enumerating all threads for a given process and calling `SuspendThread` [124] to each one, which may lead to a deadlock due to thread desynchronization; (ii) calling `DebugActiveProcess` [112], which is detectable by `IsDebuggerPresent`; and iii) using the undocumented[16] API `NtSuspendProcess`, which was used in our solution.

Context values Context values are obtained by using system APIs. We rely on our framework to perform introspection and disassembly. In addition, register values are retrieved by using the `GetThreadContext` [116] API.

[16]Undocumented functions are often found on web forums despite not being present in any official media.

GDB integration Although we have our own interface to our debugging solution, we opted to integrate it with GDB in order to make use of its extensions and facilities. The integration is done using a `stub`, a small protocol that transfers data from our back-end to GDB. We based our implementation on [130] efforts, porting it to Windows. The use of our solution with GDB allows an analyst to inspect Windows systems from distinct platforms and/or over the Internet. The current GDB stub implementation allows for `step` and `info register` commands. We intend to make everything available to the community after publication.

Validation test

To evaluate our approach's transparency, we have implemented some *tricks*. Our goal is not to provide an exhaustive list of anti-analysis tricks but to demonstrate that practical aspects match theoretical ones we have been drawing along this text.

IsDebuggerPresent. It is the default way of checking a debugger's presence on Windows. This code (shown in Listing 2.6) detected the debugger when running under ordinary debuggers, but not on our system.

Listing 2.6: Simplest debugger detection code.

```
1 if(IsDebuggerPresent())
2     printf("debugged\n");
3 else
4     printf("NO DBG\n");
```

CheckRemoteDebuggerPresent. A way of checking debugger's presence on a remote host that is also able to detect whether a process is being debugged when attached to itself. The code (shown in Listing 2.7) did not detect the debugger on our system.

Listing 2.7: 2nd Simplest debugger detection code.

```
1 CheckRemoteDebuggerPresent(GetCurrentProcess()
    ,&result);
2 if(result)
3     printf("debugged\n");
4 else
5     printf("NO DBG\n");
```

OutputDebugString. This function is the default way of printing a message in the debugger. The resulting `eax` values changes according to whether the debugger is attached or not, thus allowing the debugger presence detection. This code (shown in Listing 2.8) did not detect the debugger's presence on our system.

Listing 2.8: 3rd Simplest debugger detection code.

```
1 OutputDebugStringA(OUTPUT_MSG)
      ;
2 __asm {mov result, eax;}
3 if(result==DEBUGGED)
4         printf("debugged\n");
5 else
6         printf("NO DBG\n");
```

We also tested our solution in a real scenario, by inspecting an application protected with an unknown trick. We inspected the `Uplay`[17] binary, a game-launcher, since games are usually protected [211]. Figure 2.14 shows how the binary refuses to run under an ordinary debugger, whereas Figure 2.15 shows the inspection under our solution.

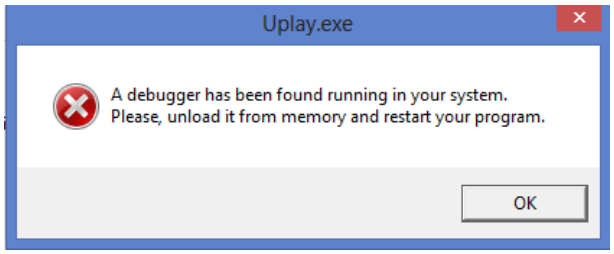

Figure 2.14: Uplay execution under an ordinary debugger.

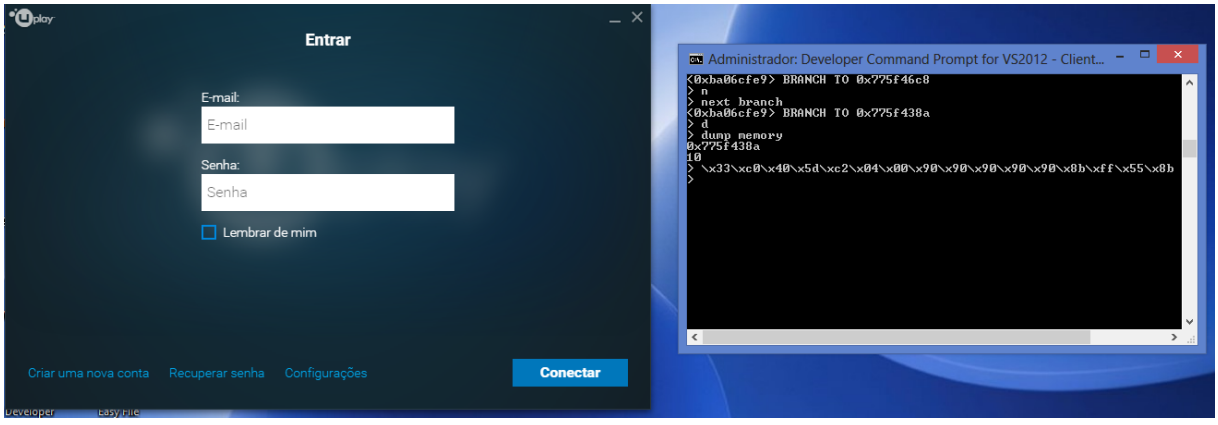

Figure 2.15: Uplay execution under our solution.

ROP Detector

Given our solution is based on a mechanism that provides branch data, addressing the ROP problem is an immmediate follow-up. Indeed, other authors have already leveraged branch monitors for such purposes, such as KBouncer, ROPecker and CFIMon, tools that

[17] `www.uplay.com`

were presented in Section 2.16. These approaches, however, are not based on a general framework, as proposed here. Our framework allows inspecting applications with no code injection while solutions like Kbouncer require hooking APIs for each process aimed to be monitored. Although such injection requirement does not impose a working restriction for these tools, it restricts the usage scenario. On a general way, such protections are suitable for known vulnerable unpatched applications in which the ROP protector can be injected. On a broader scenario, where no particular application should be protected but the whole system instead, such injection must occur on all running process. On this scenario, our injection-free, system-wide monitoring approach is a more suitable candidate. In addition, Kbouncer and ROPecker make use of a limited number of LBR entries whereas we can use unlimited memory space as we rely on BTS instead.

Implementing ROP policies on our framework is a straightforward task, since it provides us all required information (branch data) and capabilities (process identification, introspection and disassembling). The user-land client can store data in its memory and make use of libraries and data structures, therefore reducing implementation efforts.

In order to assure our approach's correctness, we opted for not developing any new ROP heuristic. Instead, we relied on verified ones. More specifically, we implemented the same two methods used by Kbouncer, `CALL-RET matching` and `gadget length`. The `CALL-RET policy` detects a ROP attack by enforcing that each `RET` call should be followed by a `RET` one. Since ROP attacks are based on `RET` instruction chaining, they can be detected.

The gadget length policy is based on the principle that ROP gadgets are usually smaller than legitimate ones. This policy defines a window of the last executed gadgets and their lengths, triggering the detection if a specified number of small gadgets occur. In our solution, we defined the same limits as KBouncer's. When any of the previous policies are violated, an alert is raised, as shown in Figure 2.16.

Figure 2.16: Alert raised by our solution when an attack is detected.

To evaluate our ROP detector, we executed some exploits against vulnerable applications, verifying whether the detection heuristics were triggered or not. The results are presented in Table 2.14.

Table 2.14: ROP exploit test results

Exploit	Vulnerability	Result	CALL-RET	GADGET-SIZE
[181]	CVE 2011-0065[18]	Exploited/Crashed	✓	✗
[90]	N/A	Exploited	✓	✗
[1]	N/A	Crashed	✗	✓

From the tests, we notice that even the exploits which failed due to an unknown reason were detected by the second stage heuristics, the gadget-size policy. In order to provide a more qualitative view on ROP detection, we present some more details about Nguyen's

exploit. Its execution triggered the gadget length policy; a snippet of the branch window is shown in Table 2.15.

Table 2.15: Excerpt of the ROP payload's branch window.

FROM	TO
—-	0x7c346c0a
0x7c346c0b	0x7c37a140
0x7c37a141	—-

The instruction disassembly of this code region, from the *MSVCR71.dll 7.10.3052.4 - 32bits* library, is presented in Listing 2.9.

Listing 2.9: Static disassembly of the `MSVCR71.dll` library.

```
1 7c346c08:      f2 0f 58 c3              addsd  %xmm3,%
    xmm0
2 7c346c0c:      66 0f 13 44 24 04        movlpd %xmm0,0x4
    (%esp)
```

The static disassembly provides aligned words. The exploit, however, makes use of an unaligned one, as indicated by the branch to `0x7c346c0a`. If we look to the dynamic disasm of the corresponding bytes (\x58\xc3), as shown in Listing 2.10, we identify the actual executed ROP gadget. As expected, our solution detects even unaligned branches.

Listing 2.10: Dynamic disassembly of the `MSVC71.dll` executed code.

```
1 0x7c346c0a (byte=0x58)    pop
    rax
2 0x7c346c0b (byte=0xc3)    ret
```

Anti-Analysis tricks detection

Along the previous sections, we have presented our solution's transparency claims, making anti-analysis trick detection a natural candidate to be addressed, since it is a straighforward application. In this section, we present some trick detections using our developed tools (mostly the tracer, but also the debugger). We have implemented a static detector that matches the executed code blocks.

Listing 2.11 presents an identified example of the `Fake Conditional` trick. In order to confuse solutions that follow the executed paths, this trick tries to purposefully trigger the path explosion problem [94]. Notice that in practice the branch will always be taken, given the `xor` instruction always yields zero.

Listing 2.11: Fake Conditional.

```
1 0x190 xor eax, eax
2 0x192 jnz 0x19c
```

A variation of this technique consists in changing the unconditional jump to a distinct instruction. Listing 2.12 shows a trick that changes the control flow by pushing a value to the stack and then returning to it.

Listing 2.12: Control Flow Change.

```
1 0x180 push 0x10a
2 0x185 ret
```

Some samples try to detect the presence of a hook, which may indicate it is under analysis. This is done by checking the presence of the `JMP` instruction (byte `0xe9`). A real example is shown in Listing 2.13.

Listing 2.13: Hook Detection

```
1 0x340 cmp eax,0xe9
2 0x345 jnz 0x347
```

Some samples perform a similar detection in order to detect the presence of a hardware debugger. The example in Listing 2.14 shows the presence checking of the debugger register 0 (`0x4`) inside the debugger context struct (`0xc`).

Listing 2.14: Hardware Debugger Detection

```
1 0x400 QWORD PTR fs:0x0,rsp
2 0x409 mov rax,QWORD PTR [
    rsp+0xc]
3 0x40e cmp rbx,QWORD PTR [
    rax+0x4]
```

Execution deviation detection at branch-level

Despite identifying the use of known anti-analysis tricks through a pattern matching procedure, as previously presented, we can also apply our solution for dynamic trick identification. When a trick leads to an evasion, a branch is taken in order to not execute the malicious payload. If this happens on the emulator but not on the bare metal setup, we can identify the divergence point by comparing the traces. The branch block which has led to the divergence point may present an anti-analysis trick.

This idea is exemplified in Figure 2.17. The block `0x2` presents an anti-analysis trick. When running on bare metal, the execution proceeeds to the `0x3` block whereas it proceeds to the `0x4` block when running on the emulator. For the sake of simplicity, we assume the execution flow will consolidate onto a single block (`0x5`). It can be understood as a common cleanup routine, for instance.

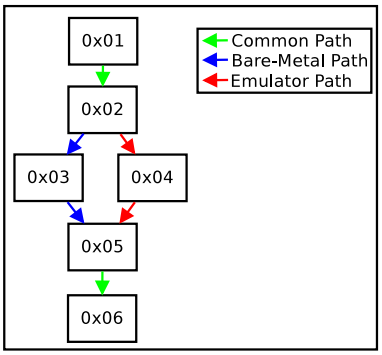

Figure 2.17: Example of a flow divergence between the code running on bare metal and on the emulated monitor.

If we assume this property and consider the execution on bare metal as the groundtruth, we can implement an algorithm for deviation detection, as presented in Algorithm 2. The first step consists of discarding the base image addresses and considering just the offsets (line 3), due to ASLR. This way, the traces are now comparable and since they will differ, the second step consists of finding an alignment (line 4) using a global sequence alignment algorithm[19]. After that, given the expected CFF structure we have defined, the aligned traces will be aligned in the beginning (blocks `0x1 and 0x2`) and in the end (blocks `0x5 and 0x6`). This way, the blocks in between are the deviating ones and the last aligned block is the one possibly having the anti-analysis trick. The algorithm proceeds by traversing the blocks, taking the bare metal trace as reference (line 5). When the blocks are aligned (line 6), they are just printed (line 7). When they are not aligned (line 8), we iterate one of the sides (line 11) in order to achieve another aligned block (block `0x5`). Algorithm 2's execution, using the inputs for the example presented in Figure 2.17 (`0x1 0x2 0x3 0x5 0x6` and `0x1 0x2 0x4 0x5 0x6`, respectively), resulted in the output presented in Listing 2.15.

Listing 2.15: Flow deviation identification by applying the alignment algorithm.

```
1 0x01 | 0x01
2 0x02 | 0x02
3      / \
4 0x03 | 0x04
5      \ /
6 0x05 | 0x05
7 0x06 | 0x06
```

We have evaluated the proposed approach on real samples, by comparing their executions under our solution and others. The compared solutions were our branch monitor solution built upon Intel PIN, presented in Section 2.17.10, and OllyDbg[20]. We manually

[19]Python's alignment library
[20]`http://www.ollydbg.de/`

ALGORITHM 2: Flow deviation detection

```
   Data: Bare Metal trace (BM), Emulated trace (E)
   Result: Deviated Block
1  = 0
   /* Calculate branch offsets                                    */
2  Bare Metal,Emulated = get_offsets(BM,E) /* Align traces        */
3  seq1, seq2 = align(Bare Metal,Emulated)
   /* Bare Metal trace is reference                               */
4  while seq1[i] not EOF do
5      if seq1[i]==seq2[j] then
6          emit_aligned(seq1[i],seq2[j])
7      else
8          emit_unaligned(seq1[i])
9          i++
10         while seq1[i]!=seq2[j] do
11             emit_unaligned(seq2[j])
12             j++
```

inspected the diverging points in order to find possible anti-analysis tricks. Figure 2.18 illustrates a divergence case due to an anti-analysis trick—checking for the `NtGlobalFlag` (offset `0x68`) in the PEB structure (`fs:0x30` offset)—in the instruction block right before the diverging branch. Some cases, however, are just false positives, since we could not identify any anti-analysis trick. As an example, Figure 2.19 shows a diverging behavior related to some kind of random decision (`<rand>` call). As future work, an automated decision mechanism may be implemented.

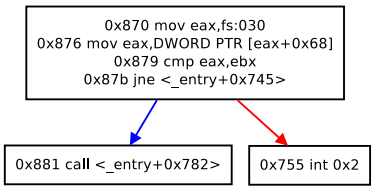

Figure 2.18: True divergence.

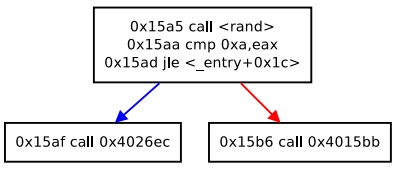

Figure 2.19: False divergence.

We analyzed 15 random samples from our dataset which presented divergent-like behavior. As a general result, some anti-analysis tricks were found, as shown in Table 2.16. The remaining samples turned out to be false positives.

Table 2.16: Anti-analysis tricks found due to branch-diverged behavior.

# of samples	Trick	Description
2	PUSH-RET	Replacing a `CALL` by a stack-pushed value
2	Fake Conditional	`XOR` itself to trigger branch-related flags
1	NtGlobalFlag	Checking data related to the process heap
1	Hook Detection	Check for a `JMP` instruction
1	Hardware Breakpoint	Debugger detection by checking context flags

The proposed approach does present limitations, such as the ones related to the CFG

format. However, our main goal is not to fully develop a tool for behavior divergence detection but to suggest how this kind of solution can benefit from using a branch-monitoring-assisted solution.

Discussion, limitations and future work

In this section, we provide a general overview of our contributions, current limitations and open opportunities on branch monitoring development.

Framework advances. The proposed solution differs from previous work by not only looking at specific branch data, but also proposing a complete framework to handle this data. Unlike previous work, our solution makes use of the BTS mechanism instead of the LBR one, which allows us new constructs, used to develop a complete analysis framework. This framework is characterized by not requiring any code injection and as such relies on a less intrusive approach than other monitoring tools. Our solution is a lightweight alternative to the state-of-the-art ones, since it requires less development efforts—no BIOS rewriting or hypervisor implementation is required—and presents a smaller overhead—only the monitored core is interrupted and most actions can be offloaded to other cores in a current multicore system or processed offline. Our implementation does not apply any system patch, being able to run on modern OSs, even if it has KPP[21], for instance.

Achieving Transparency. Besides the practical evaluation of our solution's transparency, we present here a discussion on how the formal transparency requirements were fulfilled. The `higher privilege requirement` was fulfilled since we restricted our threat model to cover only userland threats and our collection mechanisms operate at the kernel level and use hardware resources. The `no privileged side effects` and the `identical basic instruction set semantics` requirements are fulfilled by using a physical, real processor on a bare metal system instead of relying on emulation. The `transparency exception handling` requirement is fulfilled by relying on an external data capture mechanism (BTS), so no trapping or hooking is required. The `Identical measurement of time` requirement is achieved by our proposed performance solutions through performing offline disassembly and/or performing core-offloading for runtime processing. This way, no process time is taken on the core on which the monitored sample is running, thus, identical time should be observed.

Bare metal and transparency. Some transparency requirements are fulfilled by using a physical machine, a way that might look like sufficient. However, as previously discussed, transparency is achieved by not introducing side-effects, a feature which is provided by bare metal, and by **not** perfoming code injection/interference. The second requirement comes from the fact that data collection mechanisms usually require interposing binary calls, a task often performed using hooks or debug attachment. Thus, there are many reported research about anti hooking and hooking detection [43, 157, 34] as well as debugger detection [30, 15]. In this sense, we have presented a real example of an interference detection occurrence. The `Uplay` executable refused to run under a debugger even on a bare metal machine. In turn, it executed under our solution, since we addressed

[21] Kernel Patch Protection

the *non-injection/non-interference* requirement.

Solutions Comparison. In order to better position our solution among other proposals, we here provide a comparison between our developments and consolidated techniques, tools and products.

Our malware tracer can be directly compared to public available and state-of-the-art sandbox solutions. When evaluated against solutions like `Cuckoo` and `CWSandbox`, for instance, our solution is more transparent, since no code injection is performed (these solutions rely on DLL injection for API hooking)—processor data is used instead—and no virtual machine is used (hypervisor side effects are often used as analysis environment indicators by evasive samples), since our solution is bare metal based. In this sense, ours is closer to the HVM-based ones, such as `Ether` and `MAVMM`, presented on related work. When compared to these solutions, ours presents the same transparency for user land threats, given that on all approaches the malware code is run on a real processor. Unlike such solutions, our approach is not able to handle kernel malware. This limitation, however, is due to the fact that we used a kernel driver to implement our solution and as such we must assure kernel integrity. This implementation choice, however, gives us advantages when compared to competing solutions: 1) Developing a kernel driver requires less development effort than developing a whole hypervisor, which makes our solution simpler to be implemented; 2) Recompiling a kernel driver is much more portable than reinstrumenting hypervisors, making our solution much more accessible; 3) Trapping only branch deviations at kernel level is less costly than trapping each instruction at hypervisor level, contributing to a much smaller overhead.

Our solution might also be compared to other approaches, such as disassemblers, like `Capstone` (used in our framework), Plasma [149] and `Udis86` [187]. These solutions are not analyzers by themselves, rather mere translators of given byte sequences into instruction opcodes. More importantly, the instruction byte sequence data acquisition procedure comes first. Although such solutions can be directly applied to original binaries (a naive static approach), they are vulnerable to anti-disassembly techniques, used by malicious samples to evade analysis [30]. Conversely, our dynamic instruction address collection solution, by relying on processor branch data, is able to provide such disassembly tools over unpacked, ready-to-run code, rendering uneffective most anti-disassembly techniques, such as instruction misalignment.

A commercial disassembly solution which can also be compared in some way to our solution is `IDA Pro`[22]. IDA disassembler performs static code disassembling and also allows for CFG and CG reconstruction. In order to tackle anti-analysis techniques, it relies on dynamic emulation of statically unsolvable pieces of code, therefore mitigating some of them. In fact, many approaches tried to solve the anti-analysis problem by relying on rules to defeat known anti-analysis tricks, such as the ones in Cobra [197] and Vampire [196]. The major drawback of such approaches is that as soon a new anti-analysis trick is discovered, the software has to be updated and/or recompiled. bare metal-based solutions like ours, however, are able to handle such new tricks naturally, since the code is executed on a real processor. Nonetheless, running huge amounts of samples on real machines does not scale well. Besides being a disassembler, `IDA Pro` also presents a

[22]https://www.hex-rays.com/products/ida/index.shtml

complete debugger, whose frontend can be attached to `GDB`, `VMWare`, `QEMU`, `BOCHS` and others. In this sense, our framework could be extended to provide branch data to the IDA frontend the same way as the aforementioned tools do.

As for the proposed debugger, our solution can be directly compared to the HVM-based `HyperDbg` and the SMM-based `MALT`, presented in Sec. 2.16 (Related work). Our solution presents the same functionalities of such systems, such as register and memory inspection. The most notable difference is that our solution operates at the branch level, due to the branch monitor inherent working characteristic, whereas the other ones operate at the instruction level. Despite not being able to stop at every instruction, only at every block, our debugger is able to reconstruct every executed instruction sequence by making use of the same introspection procedure used for CFG reconstruction.

Just like in the case of the tracer, our debugger is also restricted to the userland level, contraty to other solutions like HyperDbg that are able to analyze at even the kernel land. As previously discussed, this implementation choice gives us many advantages when compared to such solutions. The same discussion is also valid for MALT, since handling code at BIOS level is more expensive than using a kernel driver. When compared to the popular solution GDB, our solution is more transparent, since it does not rely on ptrace and also provides the same user friendliness, since it is integrated to the GDB frontend.

Finally, our ROP detector solution can be directly compared to Kbouncer and ROPecker, since the same detection heuristics were implemented. The most significant difference is the way they are implemented, making our usage scenario broader in two ways:

1. Since we rely on the BTS mechanism instead of the LBR one, we are able to handle larger ROP chains—the BTS mechanism relies on O.S. memory page storage, which is theoretically unlimited, whereas the LBR one is limited to the number of MSR registers present in the processor. Our Haswell processor presents 16 of such registers, which would limit detection to a 16-gadget-length ROP exploit at most;

2. Our approach does not require code injection, allowing us to monitor the whole system at a time; competing solutions require injecting DLLs on each specific code one intends to monitor. It allows us to monitor the whole system without knowing in advance that a specific application is vulnerable.

Implementation limitations. The main limitation of our solution is the process context inspection mechanism—notably the memory reading mechanism—which is implemented as a userland component, making it less protected from subversion than kernel components. We considered this project decision as acceptable since we are developing a proof of concept application. If more protection is required, these mechanisms can be moved to kernel without significant side effects, apart from the development effort. Another limitation of our solution is related to the ROP scenario. Although we are able to detect its occurrence, we currently cannot block it, since no active component is injected into the process. An external blocking procedure should be implemented if the user is concerned about it. This task is eased due to the fact that our framework is constructed in a modular and independent way, allowing such kind of extensions. Our solution does not handle some code constructions, such as the use of POP+RET as a replacement for a

CALL instruction. This is a frequent assumption with many monitors due to implementation constraints, although no theoretical limitation is observed. We also targeted only single-core threats, since they are the most frequently observed ones. The monitoring platform, however, does not present any limitation to work on a multi-core scenario. In this case, the framework should be extended to work with multiple sources of data, since a malicious action could be spread through many different cores in order to avoid an ordinary pattern matching process. Currently, in order to prevent a process from migrating to a different core than the one where the mechanism was enabled into—which would break data capture—, we attach the process to a specific core by setting processor affinities, an O.S. functionality [122].

The BTS mechanism is configured to capture data only in the userland ring—despite being able to collect data at the kernel level—since we are targeting only userland processes in our threat model. Targeting kernel threats would require a more privileged ring in order to provide the required isolation for the data collection mechanism. This choice leads us to lose execution control when a syscall is invoked, which is not a problem for tracing binaries that only call libraries, but is otherwise a problem when tracing libraries that do perform such calls[23].

Introspection limitations. A sample which employs an external or static library may bypass our introspection procedure, since function names will not be recovered in this case. The execution of these libraries, however, will still be logged by the BTS monitor, allowing post-analysis by a human analyst. Another corner case is about function arguments. As BTS provides only instruction addresses, we are only able to directly get function calls, not their arguments. This is not a limitation *per se*, since some solutions, such as some malware variant detection tools [223], rely on function call structures. Solutions which require function arguments to enrich their usefulness may instrument the function calls indicated by our solution, such as in the modular malware case presented in Section 2.18.4. Notice that, in this case, there will be an overhead penalty according to the added instrumentation mechanism.

Malware Analysis Limitations. Our malware analysis solution suffers from the same limitations others do regarding stimulation, which is directly related to the reached code coverage. In order to overcome such limitation, user interactions can be simulated by using AutoIT scripts [24] or similar ones. However, in the scope of this work, we are concerned about reaching code hidden by anti-analysis techniques, for which transparent solutions like this play a crucial role. Our solution, as a sandbox, is also subject to fingerprinting, an open problem for all monitoring systems, thus outside the scope of this work.

Sandbox Restore. As our approach is bare metal based, we have to restore the system to a clean state after running a malicious sample. In virtualized environments, it is usually done by reverting to a VM snapshot. On a bare metal system, as automatic snapshots are not available, it has to be manually done. As a way of automating the task, PXE boot or LVM volumes may be used.

[23] Distinctly from Linux, Windows applications do not call the O.S. directly, rather they use O.S. libraries.

[24] `https://www.autoitscript.com/site/autoit/`

Evasion Scenarios. Every new proposed solution will be targetted by attackers in order to defeat it and so keep their stealthyness. Our solution, as is, relies on PID tracking for process filtering, a feature we believe is the most probable target for attackers: by faking a PID, a malware could make the analysis produce no result. As a countermeasure, we could filter actions not by PID but by the address itself, since each process is mapped to a unique memory region. This change is straightforward in our solution since the addresses are exactly the data provided by the BTS mechanism.

Solution Portability.

An important consideration regarding the proposed solution is about its application on other systems and architectures. In the first case, the system is portable since its main component is a hardware-resource, so we have to port only the introspection procedure to the target O.S. Porting our system from Windows to Linux, for instance, would require only changing DLL imports to a system call table when interpreting target addresses. In fact, this port is a current work-in-progress. Regarding the architectural support, our solution depends on a branch monitor mechanism able to provide source and target adress data. Despite relying on Intel's facilities, we are aware that similar mechanisms are also present in the AMD and ARM platforms. Further investigation is required to develop a port to these architectures.

Portability and Linux. The Linux kernel provides an interface for accessing many performance counters, including branch monitors. These interfaces are used by some tools, such as the `perf` profiler. The simple, direct use of these interfaces, however, does not answer many of the stated questions in this work, such as introspection or code reconstruction. Besides, these interfaces present some limitations, such as being disabled in the kernel [180]. This way, it is fully justifiable to develop/port a framework like the one hereby presented to the Linux environment.

ROP Scenario. This solution is not intended to be the definitive one, since new ways of constructing gadgets have been constantly presented [169] and new deviation attacks have been developed, such as Jump Oriented Programming (JOP) [22] and Loop Oriented Programming (LOP) [96]. In addition, ROP can be seen as only the tip of an iceberg in the code-reuse attack scenario, since other constructions, like for instance the weird machines ([191, 14, 174]), may arise in the near future as practical and widespread attacks. However, monitoring ROP will still be a required task for security purposes, such as countermeasure development or forensic procedures. Our solution presented advances by monitoring the whole system without requiring injection and providing a framework which allows monitoring unorthodox constructs. Therefore, building tools relying on previous assured characteristics, such as data collection transparency, is now an easier task.

Overhead.

The branch monitor mechanism theoretically presents zero overhead, since it is a hardware component that runs independently from the main CPU processing. Some work, however, suggested some considerable impact [180]. We confirmed a negligible overhead by measuring the activation overhead[25]—less than 1%—for both LBR and BTS. Despite the low impact of this stage, data collection and analysis add overhead to the system,

[25]With no data handling.

since an interrupt is raised and memory access and I/O are performed. This overhead is application dependent: a delayed collection for malware tracing adds less overhead than the real-time ROP detection approach. This way, we opt to split the overhead by tasks.

In order to do so, we developed a tiny program—compiled with no optimizations—which takes a million branches and measured its execution on a dedicated core with and without the running framework, using synchronous I/O. Data collection in the client, including interrupt and I/O, adds a 14% overhead; the introspection process adds another 26% overhead; the total overhead when both are combined may go up to 43%. When the introspection handling was moved to the same core as the monitored application, the overhead grew up to 75% on the small test program. These results are still smaller than the related tools Ether and MAVMM, for example, which present, in some cases, overheads of around 72% and 100%, respectively. Another evaluated scenario is changing the delayed execution collection to a real time monitor using asynchronou I/O. On these tests, we measured overheads of 100%. As a comparison, the PIN tool used for validation purposes presented overheads of 400% in the same scenario. This 4x higher overhead matches Paleari's findings on his Fuzztrace solution.

An approach to speed monitoring up is to move the analysis to another core/processor whenever possible, such as in related approaches [150]. As for disasm, we can build a disasm database from relevant and trustable portions of code, such as system libraries, avoiding the cost of a dynamic disasm. This approach would be similar to what ROPecker applies to its gadgets.

We also evaluated the impact of our solution in real scenarios. Table 2.17 presents the results of running a benchmark tool[26] with and without the monitor enabled. The `Base Value` column refers to the values obtained by running the system without the monitor. The `System Monitoring` column refers to the values obtained by running the monitor in a system-wide way, without disassembling instructions. The `Benchmark Monitoring` column refers to the data obtained by introspecting **and** disassembling benchmark instructions. All results were obtained by using the delayed data collection mode and running the monitor on a distinct core, the best usage scenario possible.

Table 2.17: Benchmarking the system with and without the monitor.

Task	Base value	System monitoring	Penalty	Benchmark monitoring	Penalty
Floating-point operations (op/s)	101530464	99221196	2.27%	97295048	4.17%
Integer operations (op/s)	285649964	221666796	22.40%	219928736	23.01%
MD5 Hashes (hash/s)	777633	568486	26.90%	568435	26.90%
RAM transfer (MB/s)	7633	6628	13.17%	6224	18.46%
HDD transfer (MB/s)	90	80	11.11%	75	16.67%
Overall (benchm. pt)	518	470	9.27%	439	15.25%

[26] `https://novabench.com/`

These results show us that distinct operations are affected in distinct ways, due to the distinct incidence of branch deviations. An example of such difference is observed between the floating-point tests and the integer ones. The MD5 calculation is also affected when the monitor is enabled, since it encompases many branches due to algorithm's inner loops. We also notice the monitor imposes higher overheads when monitoring specific applications instead of the whole system. This result is expect since, besides the additional processing, some operations, such as memory read, may block. Additionally, in this scenario we observe a penalty in disk usage, due to log files being written.

Apart from these evaluations, we performed some tests to compare BTS and LBR in distinct scenarios. Firstly, we evaluated the impact of the data collection procedure on the test program. As mentioned, the BTS use imposed a 14% overhead. When using a high-rate[27], software interrupt-based polling approach for LBR collected data, the overhead grows to 26%. These results match CERN's results [20], which reported overheads from 16% to 25%, depending on applications. We tried to vary the polling interval time from 1ms to 1000ms. The overhead started to decrease after the 200ms threshold, possibly causing data loss, thus showing our correct choice for BTS instead. We also performed the same experiment of threshold variation for the BTS hardware interrupt threshold. We measured a decreased performance impact only after a 50-instruction threshold, which shows the ISR handling itself as the most performance-expensive event. We also tried to evaluate the instruction filtering effect provided by the LBR mechanism. We noticed an overhead decrease of 6% when handling only `CALL` data compared to the general case. The result was 3% when handling only `JMPs`. This impact, however, is application/system-dependent, since it is impacted by the frequency of such instructions in the executed code. In order to better demonstrate this point, we implemented the basic handling mechanisms on Linux, so that we could compare both OSs. The Linux base performance value is 4% lower than on Windows, which reflects system differences; the same result is seen when handling the BTS interrupt. The Linux overhead is 6% lower than on Windows. This way, we conclude that the performance impact should be evaluated in each usage scenario by considering distinct OSs, applications and architectures.

Hardware-Assisted Approaches for ROP-CFI The main advantage of the proposed branch-monitor-assisted approach when compared to software-dependent solutions is that no recompilation or binary-rewriting is required. However, if it is not the usage case, other hardware-assisted approaches are alternative candidates. HAFIX [48] extends the instruction set to add CFI instructions which implement the same CALL-RET policy here presented. As no instruction-level monitoring is required, the overhead is significantly smaller. However, the usage of such instructions depends on (re)compiling the code with the newly added CFI instructions. The official proposal to extend the x86 architecture to implement a CFI policy was presented by Intel in its Control Flow Enforcement technology [80], whose CFI policy is implemented through a shadow stack, a distinct yet related approach to the ones previously presented.

[27] Using Windows kernel timers

Suggestions for Branch Monitoring improvement

The BTS and LBR mechanisms were originally developed aiming at profiling issues. However, as researchers tried to turn them onto a security-oriented monitoring platform, some resource gaps are apparent. In this section, we pinpoint some missing features we wish were present on the monitoring platform.

As our work is BTS-based, we would like to see some enhancements to it. Although BTS supports some kind of filtering, such as userland/kernel capture, it does not support all the filters of the LBR mode, such as the branch-type-based one. The implementation of such feature in BTS would allow for more granular policy implementations, such as those which relies on indirect branches (JOP, for instance).

Despite using the same interrupt vector and O.S. pages, the PEBS mechanism supplies richer context information than the BTS one. For instance, PEBS is able to provide register value information on its data units. If such data were provided also on BTS units, solutions like our debugger proposal would be easier to implement, since data acquisition would be straightforward.

Having more context data could also allow for fast data processing. If some process isolation information were available, the introspection procedure would be simplified. The concept of an O.S. process is not defined at the processor level, but having register information, such as the CR3 (page directory base register - PDBR), unique for each process, would ease the filtering task.

We are aware that many of the proposed features might be unfeasible or hard to implement in a mechanism originally not intended for such tasks, due to either design constraints or increased costs. As such, developing an independent monitoring platform which works in a similar way to BTS and LBR might be a better choice for processor improvement. This kind of proposal tends to look more attractive as computer systems get more complex to instrument. We see Intel's `Processor Tracer` [153] as a first step toward such direction.

Future Work

An immediate extension of our framework is the implementation of new policies and monitors, since it provides complete support for such developments. In addition, the Intel platform brings other opportunities, such as extending our framework to work with PEBS data, which would allow one to develop distinct policies in the userland client. These policies include, for instance, malware detection through side effect measurements. The monitoring platform also provides other implementation opportunities: changing the interrupt delivery mode to SMI, for instance, could provide SMM-based systems another triggering mechanism.

Conclusion

In this paper, we have introduced an extensible framework for transparent software analysis, which is based on performance monitoring hardware features. We have shown how the

framework can be applied to convey dynamic malware analysis, debugging facilities and a ROP detector tool. Our work is intended to be a stealth, lightweight solution compared to other state-of-the-art developments.

Solution's Availability

The framework's source is available on github[28]. Media is available on youtube[29].

Acknowledgments

This work was supported by the Brazilian National Counsel of Technological and Scientific Development (CNPq, Universal 14/2014, process 444487/2014-0) and the Coordination for the Improvement of Higher Education Personnel (CAPES, Project FORTE, Forensics Sciences Program 24/2014, process 23038.007604/2014-69).

[28] https://github.com/marcusbotacin/BranchMonitoringProject
[29] https://www.youtube.com/watch?v=BguVzqMt_j0&list=PLVYZ2jULLUDvqFVpU3pCZGlY9gCzYoyXP

Chapter 3

Discussion

In the previous chapter, I have presented two papers which encompass my contributions regarding the hardware-assisted security scenario and my proposal of using performance counters for security purposes. In this chapter, I link the ideas from both papers in order to build a landscape of my contributions and the current state-of-the-art. I also discuss goal achievements and limitations. Along this chapter, I answer the questions stated in Section 1.2, *Objectives*.

Contributions

The first contribution presented in this dissertation is the comprehensive review of tools, techniques and hardware-assisted solutions aiming at assuring a certain level of security on modern systems. Through a critical evaluation, I compared these solutions and pointed to development gaps.

The second contribution was that, more than pointing to gaps, I bridged one of them by leveraging branch-related capabilities to implement security solutions, which has never been done before to the best of our knowledge. As presented in Section 2.17, I have developed a general framework in order to provide the basic mechanisms required for collecting, filtering, and enriching data needed for deployment of such security solutions.

An extra important contribution is that the framework's code was publicly released which, beyond allowing for reproducibility, also allows for further work on the approach itself, its inner security, exhaustive testing and more comprehensive comparison by community members.

Solutions comparison

When compared to related approaches, our framework presents advantages and disadvantages, thus the best usage scenarios for the solution must be preceeded by a criterous evaluation. In order to clarify some points, we summarize some usage scenarios in the Table 3.1.

As can be noticed, my solution is the best choice when the goal is to reduce the development cost. As a drawback, the model must be restricted to userland threats. My

Table 3.1: Solutions Comparison. Comparing our solution to other approaches regarding distinct usage scenarios.

Solution	Development	Monitoring (Ring-level)	Protection	Overhead
HVM	Hard (Hypervisor)	0/3	Hypervisor-isolated	High (HVM exits)
SMM	Hard (BIOS)	-1/0/3	BIOS-isolated	High (SMIs)
SGX	Easy (API)	—	Fully-isolated	Native
Branch Monitor	Easy (Driver)	3	Processor collection/ No software isolation	Native + policy selection

solution is also a good candidate for low overhead monitoring in general. As a general limitation, our solution is not protected from system-level subversion, which can be achieved by moving userland components to the kernel.

The Framework

The proposed framework is modular, thus allowing separate tool implementations on top of it, so the tools may benefit from the underlying security properties and capabilities assured by the framework.

Process Isolation

A required capability for any malware analysis solution is to isolate actions according to the process that originated them, since many processes may be running simultaneously at a given time. The BTS mechanism captures data in a system-wide way and has no process filtering mechanism, therefore capturing actions generated by any running process. As a consequence, I had to develop a mechanism to do this filtering.

As presented in Section 2.17.2, an interrupt is raised when the user-defined, BTS buffer threshold is reached. From that moment onwards, the process execution context is swapped out so that the processor can handle the interrupt. This way, one may inspect the process state from within the interrupt routine.

However, in the general case, many branch instructions may happen to be stored in the BTS buffer when the interrupt routine is called. Moreover, these branch instructions may refer to different processes, especially in the case of a context switch happening right before the last branch instruction execution. In order to overcome this scenario, I have set the BTS threshold for a single instruction, which causes an interrupt at every executed branch instruction. By doing so, I can ensure that only one process is tracked at a time and that this one is exactly the last process scheduled by the kernel. Afterwards, the PID filtering proceeds the usual way, by retrieving data from ordinary OS structures. I consider this insight—and thus the process filtering—as the most important achievement towards enabling BTS-based process analysis.

This achievement allows me to answer the stated question of "*Could I isolate process actions?*". Not only it is answered, but the framework also offers this filtering capability

to every tool built on top of it.

Transparency

This work claims transparency as a requirement to handle evasive malware. More specifically, I tackled the anti-analysis problem from the perspective of not introducing side effects nor performing code-injection.

Considering the stated definitions of transparency (Section 2.15.3), I fulfill those requirements by: i) performing data capture at a more privileged level; ii) performing transparent instruction handling and time measurement; and iii) providing identical instruction semantics.

The first achievement derives from the BTS use, which captures data at the processor level, thus not being tampered by any software. The second one comes straighforward from the use of a bare-metal machine whereas the third is a consequence of not injecting code to interpose malware actions, which might change their behavior.

This way, the framework ensures the transparency property, as evaluated in Section 2.18.9. This evaluation allows me to answer the stated question of *Is the final solution transparent?*. As with the previous case, this property is inherited by all solutions built on top of the framework.

Implementation efforts

It is important to evaluate the development aspects of the proposed solution, which is also one of the stated questions, *Is the solution easy to implement?*. As discussed in Section 2.19, *Framework Advances*, the solution has the advantage of requiring only a driver to run, whereas other approaches require writing a hypervisor (Section 2.5) or even the system BIOS (Section 2.6). This is a significant enhancement over competing solutions, making it easily portable, upgradeable and fixable in the field. It is also easily testable, whereas an SMM approach requires a cumbersome BIOS flashing to install each new version of the tool.

The advantages brought about by the driver requirement stem from the fact that BTS use requires only handling the hardware-raised interrupts while the OS itself takes care of system management. Conversely, when developing an HVM or SMM-based solution, the developer has to tackle management issues himself. The drawback, however, is the need to rely on a much larger code base, thus limiting analysis capabilities. This fact is reflected in my threat model definition (presented in Section 2.15), since I am not able, for example, to perform kernel-level analysis. However, this was considered as an acceptable trade-off for purposes of developing and validating this research idea.

Portability

Another important concern regarding the solution is its portability, as stated in question *Is the solution portable?*, since this capability would allow us to monitor distinct OSs and their versions, broadening the solution use cases and thus expanding its impact. The discussion from Section 2.19 shows that, as a hardware mechanism, the monitor can be

enabled in an OS-independent way, only requiring writing a driver to properly handle the hardware interrupts.

Key to the solution, however, is how to handle the BTS-captured data, which is why introspection procedures had to be developed, in an OS-dependent way. The latter feature comes as library offsets change, thus requiring introspection into each OS version prior to its first use. Nevertheless, the solution's concepts and its algorithms may be applied to any BTS-powered system.

Tracer

Previous work on performance counters have addressed the side-effect analysis and the ROP attack detection problem (Section 2.9.1), but not the malware tracing problem. Based on the transparency property provided by the framework, my initial feeling was that such application was possible. So, in order to answer the stated question "*Could I develop a performance-counter-based malware analyzer?*", a prototype was developed, as presented in Section 2.18.1. The analyzer repeatedly makes use of the branch-collected data introspection procedure to follow the execution of a given process until its exit.

The possibility of gathering binary information from branch-retrieved data is somehow straightforward. However, we need to evaluate in a more fine-grained way these possiblities in order to assure the malware analysis utility. This way, I present below two examples on how these data can be applied for malware analysis: CG and CFG reconstruction.

CG Reconstruction

Given the developed application could follow processes' actions, the remaining question was the analysis granularity I could achieve using this technique. The first question I aimed to answer was: *Is CG reconstruction possible?*.

When an interrupt occurs and the filtering mechanism marks a given process to be monitored, all corresponding branches are captured. These include branches inside library code, since these are mapped in the binary space. This way, our first task is to identify when a given branch refers to a library. Fortunately, there are system APIs which provide library address information.

The second task is to identify which function is being called inside that given library. Due to the library organization, all functions are placed on fixed offsets from the function entry point. As such, we can compute such offsets by subtracting the library base address from the branch target address. As each branch may refer to a distinct function call, I have to repeat this process for each entry. In order to speed up this process, I have built an offset database which is loaded at the solution's startup.

As presented in Section 2.18.2, this reconstruction allows my solution to provide the same information as the state-of-the-art ones (HVM in Section 2.5 and SMM in Section 2.6). A drawback from my approach is that no function argument can be retrieved, which requires additional information gathering procedures. In the case of a `CreateProcess` call, for instance, no PID information is retrieved. In order to overcome this limitation, a callback was used, as shown in Section 2.18.4.

CFG Reconstruction

Beyond the call graph, one of the stated questions was *Is CFG reconstruction possible?*. This would be an important achievement since I would be able to reconstruct every executed instruction, even though the processor feature provides only the branch information. In addition, it would increase my solution's analysis capabilities, since execution-level is the most fine-grained analysis level possible for malware analysis.

As presented in Section 2.18.3, I developed a code block identification procedure based on two consecutive branches. By repeating this process at every two branches, all executed code can be retrieved.

It is important to notice that two consecutive branches is the minimum amount of data required to reconstruct a block. Given BTS data is composed by only a source and a target addresses, we need to combine two of them in order to confine a generic portion of code. In addition, as the mechanism provides branch data, we can ensure all instructions inside this given code section were effectively executed; otherwise, they would be branch-preceeded, thus being BTS-captured. With that in mind, this code block may be named as a basic block, since it consists of a code section surrounded by two branch instructions. As this process repeats until its exit, a collection of basic blocks is presented, thus being named CFG[1].

It is important to notice that the developed solution of two consecutive branches-block identification is platform-independent, making the approach portable.

Trace example

In previous sections, I have presented the immediate results which can be retrieved from the solution from a high-level perspective. In order to make clearer how an analysis proceeds from branch-collected data from a lower level, I present here a detailed, step-by-step analysis of an execution trace that can be obtained by looking to branch data. What follows is an interpretation of the data collected by the mechanism when analyzing the execution of the simple program presented in Listing 3.1.

Listing 3.1: Sample code. The identification of the function calls performed by this code is used as the solution's validation test.

```
1 int main()
2 char name[MAX_NAME];
3 scanf("%s",name);
4 printf("Hello, %s",name);
```

Before analysis begins, we have to collect addresses and sizes of the main binary and its libraries, which is performed through an introspection procedure, as shown in Listing 3.2.

[1] Control Flow Graph

Listing 3.2: Introspection process. Code image addresses should be enumerated before starting.

```
1 Binary:  Sample.exe                                7f7ccac0000 e000
2 Library: C:\Windows\System32\msvcr110d.dll         7fc05e30000 1e2000
3 Library: C:\Windows\System32\KernelBase.dll        7fc24050000 f3000
4 Library: C:\Windows\System32\kernel32.dll          7fc252e0000 136000
5 Library: C:\Windows\System32\ntdll.dll             7fc26f90000 1c0000
```

The data collection starts when the branch monitor is enabled. In the example shown in Listing 3.3, the code execution had already reached a system library (`ntdll.dll`).

Listing 3.3: Starting monitoring. The code is already in execution at a given library.

```
1 Started analyzing at 7fc26f92c4a:
2 C:\Windows\System32\ntdll.dll
```

Since the mechanism collects data in a system-wide way, without any filtering, we are able to capture libraries' internal codes as well as transitions from/to them. Listing 3.4 shows one of these transitions, from `KernelBase.dll` to `MSVCR110D.dll`.

Listing 3.4: Monitoring Execution. A library internal code is analyzed.

```
1 Code swapping from 7fc240561b9:
2 C:\Windows\System32\KernelBase.dll
3 (Unknown Function)
4 TO 7fc05f3f11b:
5 C:\Windows\System32\msvcr110d.dll
6 (_read+0xf4b)
```

By Capturing `RET` deviations, we are able to identify when a given function reaches its end and thus when the flow returns to the calling binary. Listing 3.5 shows the end of execution of the `scanf` function, at an offset of `0x3f` from the `msvcr110d.dll`, and the flow transition to our sample binary.

Listing 3.5: Monitoring Execution. Function scanf reaches its end and returns.

```
1 LIB C:\Windows\System32\msvcr110d.dll
2 at 7fc05e5d4af (scanf+0x3f)
3 returned to Binary Sample.exe
4 at 7f7ccac1037
```

Since the buffer was read, `printf` is called. Listing 3.6 shows this call.

Listing 3.6: Monitoring Execution. Print string function called.

```
1 Binary Sample.exe at 7f7ccac1079 called lib
2 C:\Windows\System32\msvcr110d.dll
3 at 7fc05e5c7b0 (printf)
```

After the string is printed, the function returns to the calling binary, as shown in Listing 3.7. In this case, internal `printf` calls were omitted.

Listing 3.7: Monitoring Execution. Printed string, now returning.

```
1 LIB C:\Windows\System32\msvcr110d.dll
2 at 7fc05e5c924 (printf+0x174)
3 returned to Binary Sample.exe
4 at 7f7ccac107f
```

Finally, as shown in Listing 3.8, the `exit` function is called, finishing program execution. After this point, no instruction is captured by the mechanism.

Listing 3.8: Monitoring Execution. The execution is finished when exit is called.

```
1 Binary Sample.exe at 7f7ccac15d2 called lib
2 C:\Windows\System32\msvcr110d.dll
3 at 7fc05e35520 (exit)
```

Code Coverage

Despite all the claims regarding malware analysis capabilities, I have to make clear that my proposal is only a first step towards branch monitor-based analysis solutions, being my tool only a proof-of-concept (PoC). Hence, more research effort has to be made in order to improve analysis capabilities. As insights into these future improvements, I present here ideas of how code coverage may be evaluated using the developed solution. To do so, I present a case study of a simple program used to classify input numbers as odd or even.

Figures 3.1 and 3.2 present graphical visualizations of executed code. The blue values indicate branch targets whereas green values indicate branch sources. The intermediate gray values indicate non-branch executed instructions. Providing such kind of view is straightforward from collected branch data and increases analysis power for the analyst.

0x1010	callq *0x110a(%rip)
0x1016	lea 0x119f(%rip),%rcx
0x101D	callq *0x10ed(%rip)
0x1023	xor %eax,%eax
0x1025	add $0x28,%rsp
0x1029	retq

Figure 3.1: Code Coverage. Example A. The blue values come from the green CALLs. The last green value is the function return.

0x1778	mov %rbx,0x8(%rsp)
0x177D	push %rdi
0x177E	sub $0x20,%rsp
0x1782	lea 0xb57,(%rip),%rbx
0x1789	lea 0xb50,(%rip),%rdi
0x1792	jmp 0x1400017a0

Figure 3.2: Code Coverage. Example B. The blue value is the target of an external function call. The green value is an unconditional branch.

Figures 3.3 and 3.4 show the same even-odd code execution for two distinct inputs. Green values indicate executed code whereas gray values indicate non-executed code. This data view procedure comes straightforward from branch data and may enhance the analyst's capabilities. In a real scenario, these data could be two distinct malware executions, such as on bare-metal and on VM-based machines, allowing for evasive behavior identification.

mov -0x4(%ebp),%eax
add $0x8,%esp
and $0x80000001,%eax
ins 0x401025
dec %eax
or $0xfffffffe,%eax
inc %eax
jne 0x40103b
push $0x402104
call *0x402090
add $0x4,%esp
xor %eax,%eax
mov %ebp,%esp
pop %ebp
ret
push $0x40210c
call *0x402090
add $0x4,%esp
xor %eax,%eax
mov %ebp,%esp
pop %ebp
ret

Figure 3.3: Code Coverage. Even Values. The gray instructions correspond to the non-executed odd function.

mov -0x4(%ebp),%eax
add $0x8,%esp
and $0x80000001,%eax
jns 0x401025
dec %eax
or $0xfffffffe,%eax
inc %eax
jne 0x40103b
push $0x402104
call *0x402090
add $0x4,%esp
xor %eax,%eax
mov %ebp,%esp
pop %ebp
ret
push $0x40210c
call *0x402090
add $0x4,%esp
xor %eax,%eax
mov %ebp,%esp
pop %ebp
ret

Figure 3.4: Code Coverage. Odd Values. The gray instructions correspond to the non-executed even function.

The same visualization tool could also be used for detecting dead-code, for instance, since dead code insertion is a popular malware anti-analysis trick. Figure 3.5 shows the identification of a dead code fragment I have inserted in the previous example's code.

Debugger

After tracing process actions, it is a natural step ahead to think about interrupting execution at certain places of interest. In a general way, the stated question here is: *Could I*

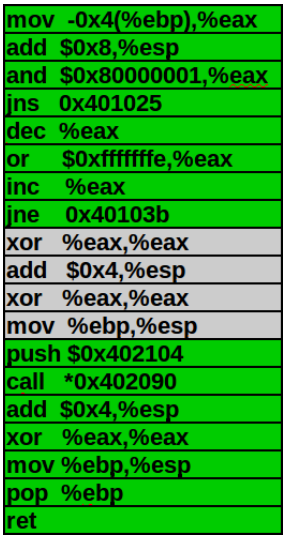

Figure 3.5: Dead Code Identification. The gray instructions were not executed.

develop a Debugger?. It is important to notice that this task is significantly different from tracing, since there is no hardware support for process suspension, yet I can still collect branch data through a hardware interrupt.

As shown in Section 2.18.6, the solution for this problem relies on a combination of OS process suspension support with an inverted I/O call generated from within an interrupt. I consider this the key point of the debugger development, since the inspection procedure itself is based on the previously presented techniques.

For the task of debugging, beyond just collecting branch data, I need to offer the user the ability to inspect the application at that given moment. For that, I need to immediately transfer him control whereas this could be done at any time in the tracer scenario. In order to allow coherent process data inspection, processes must be in the suspended state. This is what happens during the interrupt routine handler, but the execution resumes as soon as the interrupt is released. So, a suspension mechanism is required.

The OS provides a suspension routine, but it requires communicating between two system components: the ISR and the suspension module. As the ISR is watchdog-limited, I could not wait for a polling-based I/O request, so an inverted I/O was used instead. In this mode, the ISR calls the suspension module, which immediately handles the request and suspends the process. This way, as the interrupt is released, the process is already suspended.

An additional contribution claim is the novelty of this approach, since I am unaware of other debugging solutions which make use of a similar technique, given most of them are based on external hardware support or trap flags (Section 2.5.3). The same aforementioned PoC principle may be applied to my debugger proposal, which provides the basic mechanisms for binary inspection but may also be enhanced for specific-purpose actions. My contribution towards those extensions was given by integrating the solution to GDB, the most popular debugger solution today.

ROP Detection

The first performance counters-based approaches showed up in the context of ROP attacks, as I have shown in Section 2.9.1. In this sense, it is natural to compare my solution to those, so that we can check whether the developed solution presents, at least, the same capabilities, thus answering the stated question of *Does the solution handle ROP attacks?*. It is important to notice that handling ROP attacks was not my first concern. However, proving such capability demonstrates my solution's extent.

As presented in Section 2.18.10, the solution does handle ROP attacks. Moreover, it does that naturally, since checking returns is straightforward from branch-collected data. As the proposed framework is modular, I was able to retrive the same information used for malware analysis in the context of ROP attacks. The ROP detector is implemented as a policy in a framework client module.

In my solution, I have not implemented any new detection heuristics, but the same ones proposed in the state-of-the-art solutions. This way, the branch-retrieved data is submitted to the detection processing rules. My solution's contribution resides in the ROP detection heuristics being implemented in a system-wide way, due to the intrinsic framework characteristics. By doing this, broader usage scenarios are opened, since the system can be globally monitored, not just on a per application basis.

ROP Detection Policies

Since in the included paper I had space limitation issues, I opt to present here some more details about the ROP heuristics.

CALL-RET Policy. In order to provide a better understanding of the practical application of the `CALL-RET CFI` policy, I present an excerpt of a simple program execution. The policy is stack-based, pushing values as they appear. Listing 3.9 presents stack states since execution began. When a function returns to its caller, a `CALL-RET MATCH` is detected and the value is `pop`'ed from the stack.

Listing 3.9: ROP CALL-RET CFI. CALLs are `push`'ed on the stack as they appear and `pop`'ed from when a matching RET appears. The flow is legitimate when all CALLs and RETs match.

```
1  ('CURRENT STACK ', [['call', 'NewToy', 'printf']])
2  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','__iob_func']])
3  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','__iob_func'], ['ret', '__iob_func', 'printf']])
4  CALL-RET MATCH, REMOVING
5  ('CURRENT STACK ', [['call', 'NewToy', 'printf']])
6  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_lock_file']])
7  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_lock_file'], ['call', '_lock_file', '_lock']])
8  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_lock_file'], ['call', '_lock_file', '_lock'], ['
       call', '_lock','RtlEnterCriticalSection']])
9  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_lock_file'], ['call', '_lock_file', '_lock'], ['
       call', '_lock','RtlEnterCriticalSection'], ['ret', '
       RtlEnterCriticalSection', '_lock']])
10 CALL-RET MATCH, REMOVING
11 ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_lock_file'], ['call', '_lock_file', '_lock']])
12 ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_lock_file'], ['call', '_lock_file', '_lock'], ['ret
       ', '_lock','_lock_file']])
13 CALL-RET MATCH, REMOVING
```

We should take care of some corner cases, such as those generated by exception handling. Listing 3.10 presents one such case, where the `SetLastError` matches the `RtlRestoreLastWin32Error`.

Listing 3.10: ROP CALL-RET CFI. Exception handlers are corner cases and should match according to more flexible rules.

```
1  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_vcwprintf_s_l'], ['call', '_vcwprintf_s_l', 'void
       __cdecl Concurrency'], ['call', 'void __cdecl Concurrency', '
       _getptd'], ['call','_getptd', '_getptd'], ['call', '_getptd',
        'SetLastError'], ['ret', 'RtlRestoreLastWin32Error', '
       _getptd']])
2  CALL-RET MATCH, REMOVING
```

Listing 3.11 shows execution finish. When the stack is empty, we can assure flow integrity.

Listing 3.11: ROP CALL-RET CFI. An integer execution flow matches all CALLs with RETs.

```
1  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_unlock_file'], ['call', '_unlock_file', '_unlock
       ']])
2  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_unlock_file'], ['call', '_unlock_file', '_unlock'],
        ['ret', '_unlock','_unlock_file']])
3  CALL-RET MATCH, REMOVING
4  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_unlock_file']])
5  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['call', '
       printf','_unlock_file'], ['ret', '_unlock_file', 'printf']])
6  CALL-RET MATCH, REMOVING
7  ('CURRENT STACK ', [['call', 'NewToy', 'printf']])
8  ('CURRENT STACK ', [['call', 'NewToy', 'printf'], ['ret', '
       printf', 'NewToy']])
9  CALL-RET MATCH, REMOVING
10 ('CURRENT STACK ', [])
```

Gadget-Size In order to provide a better understanding of the practical application of the `Gadget-Size` policy, we present an excerpt of a simple program execution. Listing 3.12 presents distinctly evaluated windows. When a sequence of small-sized gadgets is found, it is indicated by the `Detect` flag.

Listing 3.12: ROP GADGET-SIZE CFI. Sequences of small gadgets are detected by using a moving window.

```
1  [34, 107, 34, 107, 34, 107, 34, 107, 34, 107]
2  [107, 34, 107, 34, 107, 34, 107, 34, 107, 34]
3  [34, 107, 34, 107, 34, 107, 34, 107, 34, 107]
4  [107, 34, 107, 34, 107, 34, 107, 34, 107, 34]
5  [34, 107, 34, 107, 34, 107, 34, 107, 34, 183]
6  [107, 34, 107, 34, 107, 34, 107, 34, 183, 63]
7  [34, 107, 34, 107, 34, 107, 34, 183, 63, 166]
8  [107, 34, 107, 34, 107, 34, 183, 63, 166, 29]
9  ('Detected in', [20, 7, 4, 4, 60, 8, 21, 37, 62, 1])
```

Performance

An important aspect of my contribution is evaluating the performance impact imposed by the developed solution, thus answering the question *Is the conceived solution's overhead acceptable?* As a hardware feature, the activation overhead itself is negligible, but the data collection and introspection procedures add overhead to the system. My main concern regarding overhead was to not add so much overhead in these steps so that I would loose

any advantage of using a hardware feature. As briefly discussed in Section 2.19, the measured overhead was significantly smaller than other solutions' for some scenarios. I considered having overhead measurements smaller than or equal to the state-of-the-art figures as acceptable.

The main reason for the acceptable overhead is the framework architecture, discussed below in details. Since data analysis may be performed on a time/core different from the data capture one, the performance does not suffer any penalty. Additionaly, it is important to notice that the overhead is application-dependent. The more branches are executed, the more interrupts will be raised. The smaller the code blocks, the more often the interrupts will occur. Moreover, the system load may also affect the performance, since BTS captures data in a system-wide way.

Framework Architecture and performance

In order to answer the question *Could the solution run in real-time?*, I developed strategies to minimize the performance penalty in the general case. The most significant strategy is to offload execution to a separate core, which is supported by the developed decoupled system architecture. Also for overhead impact purposes, a decoupled I/O data acquisition procedure was implemented. As shown in Figure 3.6, the interrupt is released as soon as data is enqueued. This is further asynchronously dequeued.

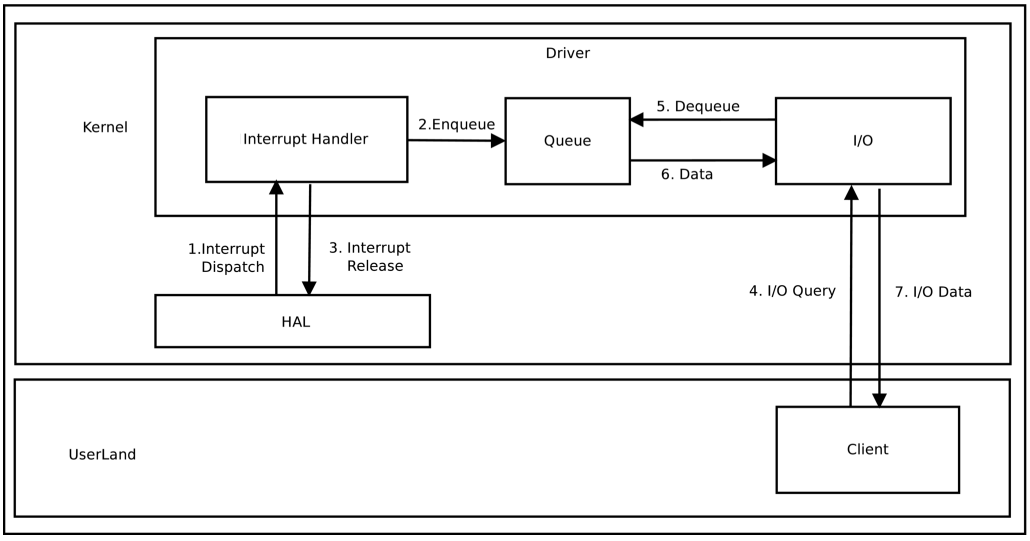

Figure 3.6: System Architecture. The data acquisition procedure in user-land is decoupled from kernel-land, allowing lower overheads by core-offloading client processing.

In a more detailed way:

1. **Interrupt Dispatch Step**: The interrupt is generated on the processor and forwarded through the Hardware Abstraction Layer (HAL) to the interrupt handler in the driver.

2. **Enqueue Step**: The interrupt handler stores captured data in a global queue.

3. **Interrupt Release Step**: The interrupt is released immediately after data enqueue, without the need for sending data to userland.

4. **I/O Query Step**: The userland client periodically queries the driver for new data availability. The driver I/O routine is interrupt-independent.

5. **Dequeue Step**: The driver I/O routine checks whether the queue is empty or not. Queue content check may be performed with no locking, since the interrupt only adds data to the queue.

6. **Data Step**: When new data is available, it is dequeued by the driver.

7. **I/O Data Step**: The driver I/O routine supplies data to the client or returns "no data available".

Other branch monitor-based solutions

Given the presented, I think I have effectively contributed for improving the state-of-the-art scenario, since no other work evaluated or applied the use of branch monitor for security purposes. In fact, after this work had started, I noticed some work related to branch monitor use, though not geared towards security. A brief view on these is presented below.

The most famous application using branch monitor is the Intel `Vtune`, a performance profiler. As expected, they use such processor features for their original purpose: evaluating execution time of code segments. Intel does not provide much public information about Vtune's internals, but a forum post[2] indicates they hook the `HalpPerfInterruptHandler` function in order to handle interrupts.

Another profiler making use of branch monitor is the Linux built-in tool `perf`. As it may be checked in the source code[3], interrupts are handled by NMI routines, as implemented in the work of this dissertation.

Despite profilers, I also found a gaming cheat-engine, a framework for modifying gaming software behavior, using branch monitors. The `gamecheat`[4] employs branch monitor for finding specific addresses and thus stopping the monitored process at specific points. The implementation choice was to hook the interrupt handler.

Finally, in mid-2016, a year after this research had started, I found claims from the Checkpoint company which suggest they might be using branch monitors in their products in some way. As posted in their blog[5], *"SandBlast CPU-level protection uses Intel's HW debugging and profiling features (which were not originally designed for security), extracts the raw data out of it and implements a sophisticated software logic layer inside a customized hypervisor. This allows it to detect the existence of an ROP exploit during emulation, and therefore block the malicious content before it is delivered to the end user".*

[2] `https://software.intel.com/en-us/forums/intel-vtune-amplifier-xe/topic/594762`

[3] `https://github.com/torvalds/linux/blob/08328814256d888634ff15ba8fb67e2ae4340b64/arch/x86/events/intel/bts.c`

[4] `https://github.com/cheat-engine/cheat-engine/blob/master/DBKKernel/ultimap2.c`

[5] `http://blog.checkpoint.com/2016/06/22/intel-spot-on-with-cet/`

In addition, a product benchmark[6] reports a *"1% Threat Extraction Overhead"*, similar to this work's measurements. The product description suggests branch monitor use for helping VM introspection and semantic gap bridging, instead of direct data collection, as hereby presented. However, no further information is provided.

Future Directions

Given the presented scenario and contributions, I present here some insights on future directions for Branch Monitor development.

Multi Process

Besides implementing CFI policies, branch monitor could be used to detect attacks by checking deviations from a baseline. Figure 3.7 shows instant branch rates of `avgnt`, `dwm`, `svchost` and `explorer` processes.

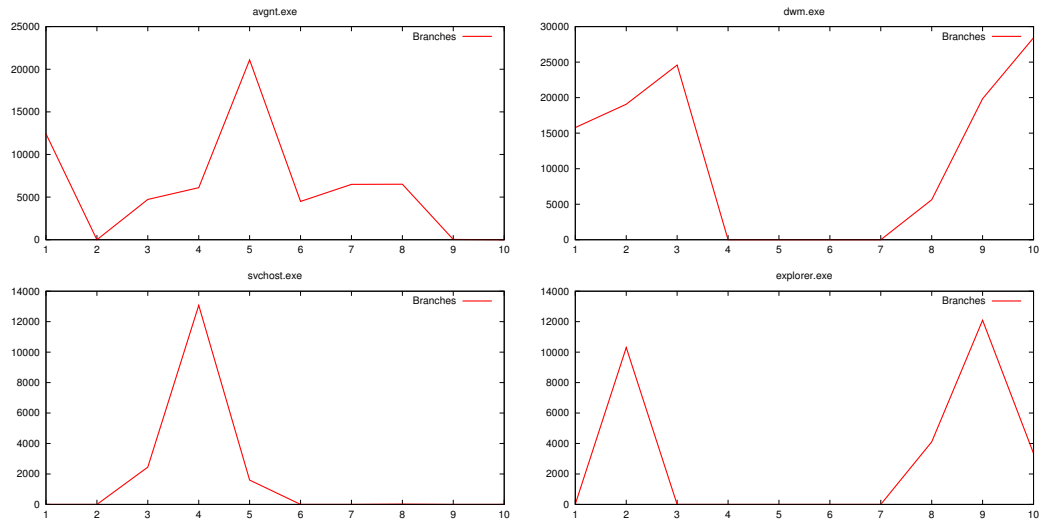

Figure 3.7: Simultaneous multi-process monitoring: Accumulated branches for each 1-second interval. The branch-rate for each process may be used for system profiling and side-channel attack detection.

As a first step, if we monitor these rates for a reasonable time and under ordinary user activities, we could define a baseline. As a follow-up, the same mechanism running as a daemon could detect any deviations from the previously established baseline. It is important to notice that this monitoring procedure should be done in a process basis, since each process presents distinct profiling metrics regarding their CPU-bound or I/O-bound behavior.

[6] `https://www.she.net/content/produktbl%C3%A4tter/sandblast-datenblatt.pdf`

Multi-core

In a similar way, the same approach could be used for whole-system profiling. As modern systems are multi-core, profiling should be considered for each core. Although my system is not able to fully support multi-core yet, I could retrieve preliminary data in order to illustrate this possibility, as shown in Figure 3.8.

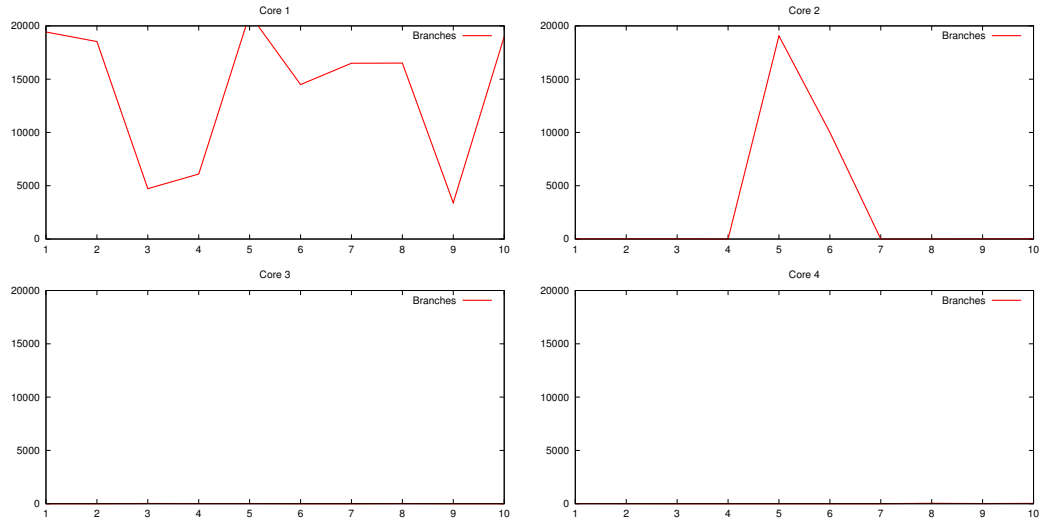

Figure 3.8: Multi-core monitoring: Accumulated branches for each 1-second interval. Enabling the mechanism on all system cores eases whole-system profiling.

We should notice some cores present higher branch rates than others, thus core migration is a corner case to be handled for this kind of policy.

Reproducibility

Finally, I am concerned about this work's reproducibility, since many technical details are involved with developing a branch tracer. In order to provide an overview on such details, I present explanations on how to set the proper flags as well as some code examples showing how to handle data at the kernel level. These examples are presented in Appendix A.1. Besides, I released the framework's source code on Github[7];

[7] `https://github.com/marcusbotacin/BranchMonitoringProject`

Chapter 4

Conclusion

In this work, I have presented an extensive review of hardware-assisted security solutions aimed to handle evasive malicious pieces of code and modern threats. I have discussed the role of special-purpose hardware on modern systems security, such as HVM, SMM mode, GPU and others. This review has led to a critical analysis of current tools and solutions.

In my evaluation, I was able to identify development gaps related to solutions which could be improved by using a hardware-supported mechanism. More specifically, I have identified a gap in the usage of performance counters, which I further discussed inn this dissertation.

My proposal for performance monitor use resulted in a transparent, modular, branch monitor-based framework able to reconstruct binary execution paths (CG and CFG) through an introspection procedure.

Based on the proposed framework, I developed three distinct tools to address security problems: a malware tracer, able to handle evasive malware; a debugger, able to inspect protected applications; an injection-free, system-wide ROP attack detector.

I believe I have contributed to the field since no other branch-based proposal addressed these issues.

Future Work

My proposed solution opens broad possibilities for developing security solutions. Although I have developed some applications, many others were left as future work. Below I present some extension possibilities.

The framework was developed as a modular architecture, so extensions are straightforward tasks. A natural first step on enhancing the solution is to develop more security policies, such as active agents, flow monitors and so on. A special interest one is to leverage branch monitor use to cluster similar malware samples. Since the CG reconstruction approach allows the analyst to retrieve the really executed functions, there are possibilities of overcoming the data collection limitation of current state-of-the-art approaches, usually based on static disassembly.

Another natural thought regarding the solution's portability is to develop framework versions for distinct OSs. In fact, while writing this document, I am already developing

a Linux version of my solution.

Finally, the framework could be enriched with additional monitor data. The PEBS monitor (discussed on Section 2.15.7) is a good candidate to provide event data to the framework, allowing an extension to the monitoring model to include side effects-based malware detection. A possible analysis extension towards side channel attack detection is to profile process branch rates and/or even whole-system branch rate.

Bibliography

[1] Julien Ahrens. Easy file management web server 5.3 - userid remote buffer overflow (rop). `https://www.exploit-db.com/exploits/33610/`, 2014. Access Date: 2017.

[2] Y. Akao and T. Yamauchi. Proposal of kernel rootkits detection method by monitoring branches using hardware features. In *2015 IIAI 4th International Congress on Advanced Applied Informatics*, pages 721–722, Okayama,Japan, July 2015. IEEE.

[3] Erdem Aktas and Kanad Ghose. Run-time control flow authentication: An assessment on contemporary x86 platforms. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1859–1866, New York, NY, USA, 2013. ACM.

[4] Malak Alshawabkeh, Byunghyun Jang, and David Kaeli. Accelerating the local outlier factor algorithm on a gpu for intrusion detection systems. In *Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, pages 104–110, New York, NY, USA, 2010. ACM.

[5] AMD. *AMD64 Architecture Programmer's Manual Volume 2*. AMD, 2013.

[6] AMD. Amd secure processor (built-in technology). `http://www.amd.com/en-gb/innovations/software-technologies/security`, 2016.

[7] ARM. *Cortex-A Series Programmer's Guide*.

[8] ARM. *ARM Security Technology - Building a Secure System using TrustZone Technology*. ARM, 2009.

[9] Warwick Ashford. Malware growth reaches record rate. `http://www.computerweekly.com/news/1280094367/Malware-growth-reaches-record-rate`, 2010.

[10] JP Aumasson and Luis Merino. Sgx secure enclaves in practice: Security and crypto review, 2016.

[11] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. Hypersentry: Enabling stealthy in-context measurement of hypervisor integrity. In *Proc. 17th ACM Conf. on Computer and Communications Security*, CCS '10, pages 38–49, New York, NY, USA, 2010. ACM.

[12] M.B. Bahador, M. Abadi, and A. Tajoddin. Hpcmalhunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *2014 4th Intl. Conf. on Computer and Knowledge Engineering (ICCKE)*, pages 703–708, Oct 2014.

[13] Davide Balzarotti, Marco Cova, Christoph Karlberger, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. Efficient detection of split personalities in malware. In *NDSS 2010, 17th Annual Network and Distributed System Security Symposium, February 28th-March 3rd, 2010, San Diego, USA*, pages 1–17, San Diego, UNITED STATES, 02 2010. EURECOM.

[14] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W. Smith. The page-fault weird machine: Lessons in instruction-less computation. In *Proc. of the 7th USENIX Conf. on Offensive Technologies*, WOOT'13, pages 13–13, 2013.

[15] Gabriel Negreira Barbosa and Rodrigo Rubira Branco. Prevalent characteristics in modern malware. `http://www.kernelhacking.com/rodrigo/docs/blackhat2014-presentation.pdf`, 2014. Access in May 11, 2016.

[16] U. Bayer, C. Kruegel, and E. Kirda. Ttanalyze: A tool for analyzing malware. In *15th European Inst. for Computer Antivirus Research (EICAR 2006) Annual Conf.*, pages 1–12, Hamburg, Germany, 2006. EICAR.

[17] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proc. USENIX Annual Technical Conf.*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[18] Xavier J. A. Bellekens, Christos Tachtatzis, Robert C. Atkinson, Craig Renfrew, and Tony Kirkham. Glop: Enabling massively parallel incident response through gpu log processing. In *Proc. 7th Intl. Conf. on Security of Information and Networks*, SIN '14, pages 295:295–295:301, New York, NY, USA, 2014. ACM.

[19] Arnar Birgisson, Mohan Dhawan, Úlfar Erlingsson, Vinod Ganapathy, and Liviu Iftode. Enforcing authorization policies using transactional memory introspection. In *Proc. 15th ACM Conf. on Computer and Communications Security*, CCS '08, pages 223–234, New York, NY, USA, 2008. ACM.

[20] Georgios Bitzes and Andrzej Nowak. The overhead of profiling using pmu hardware counters. `https://zenodo.org/record/10800/files/TheOverheadOfProfilingUsingPMUhardwareCounters.pdf`, 2014. Access Date: May/2017.

[21] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 353–362, New York, NY, USA, 2011. ACM.

[22] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: A new class of code-reuse attack. In *Proceedings of the 6th ACM*

Symposium on Information, Computer and Communications Security, ASIACCS '11, pages 30–40, New York, NY, USA, 2011. ACM.

[23] Marcus Botacin, Paulo de Geus, and André Grégio. Análise transparente de malware com suporte por hardware. In *SBSeg 2016 - Artigos completos ()*, Niterói, nov 2016.

[24] Marcus Botacin, Paulo de Geus, and André Grégio. Detecção de ataques por rop em tempo real assistida por hardware. In *SBSeg 2016 - Artigos completos ()*, Niterói, nov 2016.

[25] Marcus Botacin, Paulo de Geus, and André Grégio. Voidbg: Projeto e implementação de um debugger transparente para inspeção de aplicações protegidas. In *SBSeg 2016 - Artigos completos ()*, Niterói, nov 2016.

[26] Marcus Felipe Botacin, Paulo Lício de Geus, and André Ricardo Abed Grégio. The other guys: automated analysis of marginalized malware. *Journal of Computer Virology and Hacking Techniques*, pages 1–12, 2017.

[27] Marcus Felipe Botacin, Paulo Lício de Geus, and André Ricardo Abed Grégio. Enhancing branch monitoring for security purposes: From control flow integrity to malware analysis and debugging. *ACM Transactions On Privacy and Security*, pages 1–29, Submitted in 2017.

[28] Marcus Felipe Botacin, Paulo Lício de Geus, and André Ricardo Abed Grégio. One thousand and one nights: Brazilian malware stories. *Journal in Computer Virology and Hacking Techniques*, pages 1–28, Submitted in 2017.

[29] Marcus Felipe Botacin, Paulo Lício de Geus, and André Ricardo Abed Grégio. Who watches the watchmen: A review of techniques, tools and methods to counterfeit anti-analysis techniques on modern platforms. *ACM Computer Surveys*, pages 1–38, Submitted in 2017.

[30] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti- vm technologies. `http://www.kernelhacking.com/rodrigo/docs/blackhat2012-paper.pdf`, 2012. Access in May 11, 2016.

[31] Michael Brengel, Michael Backes, and Christian Rossow. Detecting hardware-assisted virtualization. In *Proc. 13th Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 207–227, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[32] BSDaemon, coideloco, and D0nad0n. System management mode hack - using smm for "other purposes". `http://phrack.org/issues/65/7.html`, 2008.

[33] Capstone. The ultimate disassembly framework. `http://www.capstone-engine.org/`, 2016. Access Date: July/2016.

[34] Alexander Chailytko and Stanislav Skuratovich. Vb2016 paper: Defeating sandbox evasion: how to increase the successful emulation rate in your virtual environment. `https://www.virusbulletin.com/virusbulletin/2016/12/vb2016-paper-defeating-sandbox-evasion-how-increase-successful-emulation-rate-your-virtual-environment/`, 2016. Access Date: 2017.

[35] Ping Chen, Hai Xiao, Xiaobin Shen, Xinchun Yin, Bing Mao, and Li Xie. Drop: Detecting return-oriented programming malicious code. In *Proceedings of the 5th International Conference on Information Systems Security*, ICISS '09, pages 163–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[36] 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. *SIGPLAN Not.*, 43(3):2–13, March 2008.

[37] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *2008 IEEE Intl. Conf. on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 177–186, Anchorage, Alaska, USA, June 2008. IEEE.

[38] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, Huijie DENG, et al. Ropecker: A generic and practical approach for defending against rop attack. *NDSS Symposium 2014*, 2014.

[39] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Huijie Robert Deng. Ropecker: A generic and practical approach for defending against rop attacks. In *Symp. on Network and Distributed System Security (NDSS*, pages 1–15, San Diego, CA, USA, 2014. Internet Society.

[40] Andrei Chiş, Marcus Denker, Tudor Gîrba, and Oscar Nierstrasz. Practical Domain-specific Debuggers Using the Moldable Debugger Framework. *Comput. Lang. Syst. Struct.*, 44(PA):89–113, December 2015.

[41] CHIPSEC. Chipsec platform security assessment framework. `https://www.blackhat.com/docs/us-14/materials/arsenal/us-14-Bulygin-CHIPSEC-Slides.pdf`, 2014.

[42] CHIPSEC. Chipsec. `https://github.com/chipsec/chipsec`, 2016.

[43] CloudBurst. Reverse engineering for malware: Shellcodes and av/api hook evasion. `https://www.cloudburstsecurity.com/2016/06/10/reverse-engineering-for-malware-shellcodes-and-avapi-hook-evasion/`, 2016.

[44] CoreBoot. Coreboot. `http://www.coreboot.org/`, 2015.

[45] Paul Crowley. Pixel security: Better, faster, stronger. `http://security.googleblog.com/2016/11/pixel-security-better-faster-stronger.html`, 2016.

[46] CVE. Cve-2011-0065. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0065`, 2011. Acessado em junho/2016.

[47] Shaun Davenport and Richard Ford. Sgx: the good, the bad and the downright ugly. `https://www.virusbulletin.com/virusbulletin/2014/01/sgx-good-bad-and-downright-ugly`, 2014.

[48] L. Davi, M. Hanreich, D. Paul, A. R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. Hafix: Hardware-assisted flow integrity extension. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, San Francisco, CA, USA, June 2015. ACM/IEEE.

[49] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*, STC '09, pages 49–54, New York, NY, USA, 2009. ACM.

[50] S. Debray and J. Patel. Reverse engineering self-modifying code: Unpacker extraction. In *2010 17th Working Conference on Reverse Engineering*, pages 131–140, Beverly, MA, USA, Oct 2010. IEEE.

[51] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *SIGARCH Comput. Archit. News*, 41(3):559–570, June 2013.

[52] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Spider: Stealthy binary program instrumentation and debugging via hardware virtualization. In *Proc. 29th Annual Computer Security Applications Conf.*, ACSAC '13, pages 289–298, New York, NY, USA, 2013. ACM.

[53] Steve Dent. Microsoft's edge browser stays secure by acting as a virtual pc. `https://www.engadget.com/2016/09/27/microsofts-edge-browser-stays-secure-by-acting-as-a-virtual-pc/`, 2016.

[54] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proc. 15th ACM Conf. on Computer and Communications Security*, CCS '08, pages 51–62, NY, USA, 2008. ACM.

[55] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.

[56] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proc. 2011 IEEE Symp. on Security and Privacy*, SP '11, pages 297–312, Washington, DC, USA, 2011. IEEE Computer Society.

[57] L. Duflot, D. Etiemble, and O. Grumelard. Using cpu system management mode to circumvent operating system security functions. `http://fawlty.cs.usfca.edu/~cruse/cs630f06/duflot.pdf`, 2007.

[58] DynamoRIO. Dynamorio - dynamic instrumentation tool platform. `http://dynamorio.org/home.html`, 2001.

[59] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008.

[60] Shawn Embleton, Sherri Sparks, and Cliff Zou. Smm rootkits: A new breed of os independent malware. In *Proc. 4th Intl. Conf. on Security and Privacy in Communication Netowrks*, SecureComm '08, pages 11:1–11:12, New York, NY, USA, 2008. ACM.

[61] Aristide Fattori, Roberto Paleari, Lorenzo Martignoni, and Mattia Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 417–426, New York, NY, USA, 2010. ACM.

[62] FEBRABAN. FEBRABAN dá dicas de segurança eletrônica. `http://www.febraban.org.br/Noticias1.asp?id_texto=2758`, December 2015. Access in May 22, 2016.

[63] David Fitzpatrick and Drew Griffin. Cyber-extortion losses skyrocket, says fbi. `http://money.cnn.com/2016/04/15/technology/ransomware-cyber-security/`, 2016.

[64] Yangchun Fu and Zhiqiang Lin. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. *ACM Trans. Inf. Syst. Secur.*, 16(2):7:1–7:29, September 2013.

[65] Yuxin Gao, Zexin Lu, and Yuqing Luo. Survey on malware anti-analysis. In *Intelligent Control and Information Processing (ICICIP), 2014 Fifth International Conference on*, pages 270–275, Aug 2014.

[66] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: Vmm detection myths and realities. In *Proc. 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 6:1–6:6, Berkeley, CA, USA, 2007. USENIX Association.

[67] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 417–432, San Diego, CA, August 2014. USENIX Association.

[68] Mariano Graziano, Davide Balzarotti, and Alain Zidouemba. ROPMEMU: A framework for the analysis of complex code-reuse attacks. In *ASIACCS 2016, 11th ACM Asia Conference on Computer and Communications Security, May 30-June 3, 2016, Xi'ian, China*, 2016.

[69] Groundworkstech. A python interface to the gnu binary file descriptor (bfd) library. `https://github.com/Groundworkstech/pybfd`, 2016. Access Date: July/2016.

[70] Grsecurity. Grsecurity. `https://grsecurity.net/`, 2013.

[71] Claudio Guarnieri. Cuckoo sandbox. `http://www.cuckoosandbox.org/`, 2013.

[72] Neha Gupta, Smita Naval, Vijay Laxmi, M.S. Gaur, and Muttukrishnan Rajarajan. P-spade: Gpu accelerated malware packer detection. In *Privacy, Security and Trust (PST), 2014 Twelfth Annual Intl. Conf. on*, pages 257 – 263, Toronto, Canada, 2014. IEEE.

[73] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. Ilr: Where'd my gadgets go? In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 571–585, Washington, DC, USA, 2012. IEEE Computer Society.

[74] A. Ho, S. Hand, and T. Harris. Pdb: pervasive debugging with xen. In *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 260–265, Nov 2004.

[75] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proc. 2nd Intl. Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 11:1–11:1, NY, USA, 2013. ACM.

[76] Owen S. Hofmann, Alan M. Dunn, Sangman Kim, Indrajit Roy, and Emmett Witchel. Ensuring operating system kernel integrity with osck. In *Proc. 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 279–290, New York, NY, USA, 2011. ACM.

[77] Joel Hruska. Report claims intel cpus contain enormous security flaw. `http://www.extremetech.com/computing/230342-report-claims-intel-cpus-contain-enormous-security-flaw`, 2016.

[78] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel, 2013.

[79] Intel. Pin - a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`, 2015.

[80] Intel. Control-flow enforcement technology preview. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`, 2016. Access Date: January/2017.

[81] Intel Security. Net losses: Estimating the global cost of cybercrime. `http://www.mcafee.com/br/resources/reports/rp-economic-impact-cybercrime2.pdf`, 2014.

[82] Alex Ionescu. Battle of the skm and ium: How windows 10 rewrites os architecture. `https://www.blackhat.com/us-15/briefings.html#battle-of-the-skm-and-ium-how-windows-10-rewrites-os-architecture`, 2015.

[83] iseclab. Anubis - malware analysis for unknown binaries. `https://anubis.iseclab.org/`, 2010.

[84] Grégoire Jacob, Hervé Debar, and Eric Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3):251–266, 2008.

[85] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han. Opensgx: An open platform for sgx research. In *Proc. 2016 Annual Network and Distributed System Security Symp.*, pages 1–16, San Diego, CA, USA, 2016. Internet Society.

[86] Jun Jiang, Xiaoqi Jia, Dengguo Feng, Shengzhi Zhang, and Peng Liu. Hypercrop: A hypervisor-based countermeasure for return oriented programming. In *Proceedings of the 13th International Conference on Information and Communications Security*, ICICS'11, pages 360–373, Berlin, Heidelberg, 2011. Springer-Verlag.

[87] Min Gyung Kang, Heng Yin, Steve Hanna, Stephen McCamant, and Dawn Song. Emulating emulation-resistant malware. In *Proc. 1st ACM Workshop on Virtual Machine Security*, VMSec '09, pages 11–22, New York, NY, USA, 2009. ACM.

[88] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barebox: Efficient malware analysis on bare-metal. In *Proc. 27th Annual Computer Security Applications Conf.*, ACSAC '11, pages 403–412, NY, USA, 2011. ACM.

[89] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. Barecloud: Bare-metal analysis-based evasive malware detection. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 287–301, Berkeley, CA, USA, 2014. USENIX Association.

[90] Knaps. Easy file sharing web server 7.2 - remote buffer overflow (seh) (dep bypass with rop). `https://www.exploit-db.com/exploits/38829/`, 2015. Access Date: 2017.

[91] Kompalli and Sarat. Using existing hardware services for malware detection. In *Proc. 2014 IEEE Security and Privacy Workshops*, SPW'14, pages 204–208, Washington, DC, USA, 2014. IEEE Computer Society.

[92] S. Kompalli. Using existing hardware services for malware detection. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 204–208, May 2014.

[93] Lazaros Koromilas, Giorgos Vasiliadis, Elias Athanasopoulos, and Sotiris Ioannidis. *GRIM: Leveraging GPUs for Kernel Integrity Monitoring*, pages 3–23. Springer International Publishing, Cham, 2016.

[94] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *2010 19th IEEE Asian Test Symposium*, pages 59–64, Shanghai, China, Dec 2010. IEEE.

[95] Evangelos Ladakis, Lazaros Koromilas, Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. You can type, but you can't hide: A stealthy gpu-based keylogger. `http://www.cs.columbia.edu/~mikepo/papers/gpukeylogger.eurosec13.pdf`, 2013.

[96] Bingchen Lan, Yan Li, Hao Sun, Chao Su, Yao Liu, and Qingkai Zeng. Loop-oriented programming: A new code reuse attack to bypass modern defenses. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 190–197, Aug 2015.

[97] Hojoon Lee, HyunGon Moon, DaeHee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. Ki-mon: A hardware-assisted event-triggered monitoring platform for mutable kernel object. In *22nd USENIX Security Symposium*, pages 511–526, Washington, D.C., 2013. USENIX.

[98] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *20th Annual Network and Distributed System Security Symp., NDSS 2013, San Diego, California, USA, February 24-27, 2013*, page 16, San Diego, CA, USA, 2013. Internet Society.

[99] Tamas Lengyel, Thomas Kittel, George Webster, and Jacob Torrey. Pitfalls of virtual machine introspection on modern hardware. In *1st Workshop on Malware Memory Forensics (MMF)*, pages 1–7, New Orleans, Louisiana, USA, December 2014. ACM.

[100] LibVMI. Introduction to libvmi. `http://libvmi.com/docs/gcode-intro.html`, 2015.

[101] Jari-Matti Mäkelä, Ville Leppänen, and Martti Forsell. Towards a parallel debugging framework for the massively multi-threaded, step-synchronous replica architecture. In *Proc. 14th Intl. Conf. Computer Systems and Technologies*, CompSysTech '13, pages 153–160, NY, USA, 2013. ACM.

[102] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are hardware performance counters a cost effective way for integrity checking of programs. In *Proc. 6th ACM Workshop on Scalable Trusted Computing*, STC '11, pages 1–6, New York, NY, USA, 2011. ACM.

[103] Tarjei Mandt, Mathew Solnik, and David Wang. Demystifying the secure enclave processor, 2016.

[104] Eli Marcus. RSA Uncovers Boleto Fraud Ring in Brazil. https://blogs.rsa.com/rsa-uncovers-boleto-fraud-ring-brazil/, July 2014. Access in May 11, 2016.

[105] J.A.P. Marpaung, M. Sain, and Hoon-Jae Lee. Survey on malware evasion techniques: State of the art and challenges. In *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, pages 744–749, Feb 2012.

[106] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection*, RAID'10, pages 297–316, Berlin, Heidelberg, 2010. Springer-Verlag.

[107] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing cpu emulators. In *Proc. 18th Intl Symp. on Software Testing and Analysis*, ISSTA '09, pages 261–272, NY, USA, 2009. ACM.

[108] Microsoft. Keregisternmicallback routine. https://msdn.microsoft.com/en-us/library/windows/hardware/ff553116%28v=vs.85%29.aspx, 2013. Access Date: July/2016.

[109] Microsoft. Kernel patch protection for x64-based operating systems. https://technet.microsoft.com/en-us/library/cc759759%28v=ws.10%29.aspx, 2015.

[110] Microsoft. Createprocess function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425%28v=vs.85%29.aspx, 2016. Access Date: July/2016.

[111] Microsoft. Createprocessnotifyex routine. https://msdn.microsoft.com/en-us/library/windows/hardware/ff542860%28v=vs.85%29.aspx, 2016. Access Date: July/2016.

[112] Microsoft. Debugactiveprocess function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms679295%28v=vs.85%29.aspx, 2016. Access Date: July/2016.

[113] Microsoft. Debugging functions. https://msdn.microsoft.com/en-us/library/windows/desktop/ms679303%28v=vs.85%29.aspx, 2016. Access Date: July/2016.

[114] Microsoft. Enumprocessmodules function. https://msdn.microsoft.com/en-us/library/windows/desktop/ms682631%28v=vs.85%29.aspx, 2016. Access Date: July/2016.

[115] Microsoft. Getmodulehandle function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms683199%28v=vs.85%29.aspx, 2016. Access Date: July/2016.

[116] Microsoft. Getthreadcontext function. https://msdn.microsoft.com/pt-br/library/windows/desktop/ms679362%28v=vs.85%29.aspx, 2016. Access Date: July/2016.

[117] Microsoft. I/o request packets. `https://msdn.microsoft.com/en-us/library/windows/hardware/hh439638%28v=vs.85%29.aspx`, 2016. Access Date: July/2016.

[118] Microsoft. Isdebuggerpresent. `https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680345%28v=vs.85%29.aspx`, 2016. Access Date: July/2016.

[119] Microsoft. Psgetcurrentprocessid routine. `https://msdn.microsoft.com/en-us/library/windows/hardware/ff559935%28v=vs.85%29.aspx`, 2016. Access Date: July/2016.

[120] Microsoft. Readfile function. `https://msdn.microsoft.com/en-us/library/windows/desktop/aa365467%28v=vs.85%29.aspx`, 2016. Access Date: July/2016.

[121] Microsoft. Readprocessmemory function. `https://msdn.microsoft.com/pt-br/library/windows/desktop/ms680553%28v=vs.85%29.aspx`, 2016. Access Date: July/2016.

[122] Microsoft. Setprocessaffinitymask function. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms686223%28v=vs.85%29.aspx`, 2016. Access Date: January/2017.

[123] Microsoft. Singly and doubly linked lists. `https://msdn.microsoft.com/en-us/library/windows/hardware/ff563802%28v=vs.85%29.aspx`, 2016. Access Date: July/2016.

[124] Microsoft. Suspendthread function. `https://msdn.microsoft.com/pt-br/library/windows/desktop/ms686345%28v=vs.85%29.aspx`, 2016. Access Date: July/2016.

[125] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. Vigilare: Toward snoop-based kernel integrity monitor. In *Proc. 2012 ACM Conf. on Computer and Communications Security*, CCS '12, pages 28–37, New York, NY, USA, 2012. ACM.

[126] Hyungon Moon, Jinyong Lee, Dongil Hwang, Seonhwa Jung, Jiwon Seo, and Yunheung Paek. Architectural supports to protect os kernels from code-injection attacks. In *Proc. Hardware and Architectural Support for Security and Privacy 2016*, HASP 2016, pages 5:1–5:8, New York, NY, USA, 2016. ACM.

[127] Asit More and Shashikala Tapaswi. Virtual machine introspection: towards bridging the semantic gap. *Journal of Cloud Computing*, 3(1):1–14, 2014.

[128] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430, Dec 2007.

[129] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Annual Computer Security Applications Conf.*, pages 1–10, Miami Beach, FL, USA, 2007. ACM.

[130] mseaborn. gdb-debug-stub. `github.com/mseaborn/gdb-debug-stub`, 2014. Access Date: July/2016.

[131] Marius Muench, Fabio Pagani, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna, and Davide Balzarotti. *Taming Transactions: Towards Hardware-Assisted Control Flow Integrity Using Transactional Memory*, pages 24–48. Springer International Publishing, 2016.

[132] Igor Muttik, Alex Nayshtut, and Roman Dementlev. Creating a spider goat: using transactional memory support for security, 2014.

[133] Michael Myers and Stephen Youndt. An introduction to hardware-assisted virtual machine (hvm) rootkits. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.90.8832&rep=rep1&type=pdf`, 2007.

[134] Matthias Neugschwandtner, Christian Platzer, PaoloMilani Comparetti, and Ulrich Bayer. danubis – dynamic device driver analysis based on virtual machine introspection. In Christian Kreibich and Marko Jahnke, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 6201 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin Heidelberg, Bonn, Germany, 2010.

[135] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. *″″*, ""(""):17, 2005.

[136] Anh M. Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T. King, and Hai D. Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 441–450, Washington, DC, USA, 2009. IEEE Computer Society.

[137] NirSoft. Dll export viewer. `http://www.nirsoft.net/utils/dll_export_viewer.html`, 2016. Access Date: July/2016.

[138] Nist.gov. National vulnerability database. `https://web.nvd.nist.gov/view/vuln/search-results?query=firmware&search_type=all&cves=on`, 2017.

[139] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-free: Defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 49–58, New York, NY, USA, 2010. ACM.

[140] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In *Proc. 3rd USENIX Conf. on Offensive Technologies*, WOOT'09, pages 2–2, Berkeley, CA, USA, 2009. USENIX Association.

[141] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 601–615, Washington, DC, USA, 2012. IEEE Computer Society.

[142] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 447–462, Berkeley, CA, USA, 2013. USENIX Association.

[143] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proc. 2008 IEEE Symp. on Security and Privacy*, SP '08, pages 233–247, Washington, DC, USA, 2008. IEEE Computer Society.

[144] Michael Pearce, Sherali Zeadally, and Ray Hunt. Virtualization: Issues, security threats, and solutions. *ACM Comput. Surv.*, 45(2):17:1–17:39, March 2013.

[145] Gábor Pék, Boldizsár Bencsáth, and Levente Buttyán. nether: In-guest detection of out-of-the-guest malware analyzers. In *Proc. 4th Eur. Wksp on System Security*, EUROSEC '11, pages 3:1–3:6. ACM, 2011.

[146] Gábor Pék, Levente Buttyán, and Boldizsár Bencsáth. A survey of security issues in hardware virtualization. *ACM Comput. Surv.*, 45(3):40:1–40:34, July 2013.

[147] Nick L. Petroni, Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. 13th Conf. on USENIX Security Symp. - Volume 13*, SSYM'04, pages 13–13, Berkeley, CA, USA, 2004. USENIX Association.

[148] Cody Pierce, Matthew Spisak, and Kenneth Fitch. Capturing 0day exploits with perfectly placed hardware traps. `https://www.blackhat.com/docs/us-16/materials/us-16-Pierce-Capturing-0days-With-PERFectly-Placed-Hardware-Traps-wp.pdf`, 2016.

[149] Plasma. Plasma. `https://github.com/plasma-disassembler/plasma`, 2015. Access Date: January/2017.

[150] Rian Quinn. *Detection of malware via side channel information.* PhD thesis, Binghamton University, 2012.

[151] Daniel Quist, Lorie Liebrock, and Joshua Neil. Improving antivirus accuracy with hypervisor assisted analysis. *J. Comput. Virol.*, 7(2):121–131, May 2011.

[152] Nguyen Anh Quynh and Kuniyasu Suzaki. Virt-ice: Next-generation debugger for malware analysis. `https://media.blackhat.com/bh-us-10/whitepapers/Anh/BlackHat-USA-2010-Anh-Virt-ICE-wp.pdf`, 2010.

[153] James R. Processor tracing. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`, 2013. Access Date: May/2017.

[154] Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. When hardware meets software: A bulletproof solution to forensic memory acquisition. In *Proc. 28th Annual Computer Security Applications Conf.*, ACSAC '12, pages 79–88, New York, NY, USA, 2012. ACM.

[155] Junghwan Rhee, Ryan Riley, Dongyan Xu, and Xuxian Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. *2012 7th Intl. Conf. on Availability, Reliability and Security*, 0:74–81, 2009.

[156] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proc. 11th Intl. Symp. on Recent Advances in Intrusion Detection*, RAID '08, pages 1–20, Berlin, Heidelberg, 2008. Springer-Verlag.

[157] Thomas Roccia. An overview of malware self-defense and protection. `https://securingtomorrow.mcafee.com/mcafee-labs/overview-malware-self-defense-protection/`, 2016.

[158] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[159] Christian Rossow, Christian J. Dietrich, Christian Kreibich, Chris Grier, Vern Paxson, Norbert Pohlmann, Herbert Bos, and Maarten van Steen. Prudent Practices for Designing Malware Experiments: Status Quo and Outlook . In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)* , pages 1–15, San Francisco, CA, May 2012. IEEE.

[160] Rutkowska. Subverting vista kernel for fun and for profit. `https://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf`, 2006.

[161] Rutkowska. Qubes os project. `https://www.qubes-os.org/`, 2010.

[162] Joanna Rutkowska. Intel x86 considered harmful. `https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf`, 2015.

[163] Joanna Rutkowska and Rafał Wojtczuk. Preventing and detecting xen hypervisor subversions. `http://invisiblethingslab.com/resources/bh08/part2-full.pdf`, 2008.

[164] Alireza Saberi, Yangchun Fu, and Zhiqiang Lin. Hybrid-bridge: Efficiently bridging the semantic gap in virtual machine introspection via decoupled execution and training memoization. In *Proc. 21st Annual Network and Distributed System Security Symp. (NDSS'14)*, San Diego, CA, USA, 2014. Internet Society.

[165] Samsung. Samsung knox. `https://www.samsungknox.com/en`, 2017.

[166] J. Schiffman and D. Kaplan. The smm rootkit revisited: Fun with usb. In *Availability, Reliability and Security (ARES), 2014 9th Intl. Conf. on*, pages 279–286, Fribourg, Switzerland, Sept 2014. IEEE.

[167] Christian Schneider, Jonas Pfoh, and Claudia Eckert. A universal semantic bridge for virtual machine introspection. In *Proc. 7th Intl. Conf. on Information Systems Security*, ICISS'11, pages 370–373, Berlin, Heidelberg, 2011. Springer-Verlag.

[168] Daniel Schulz and Frank Mueller. A thread-aware debugger with an open interface. In *Proc. 2000 ACM SIGSOFT Intl. Symp. Software Testing and Analysis*, ISSTA '00, pages 201–211, USA, 2000. ACM.

[169] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the effectiveness of current anti-rop defenses. In *Research in Attacks, Intrusions and Defenses: 17th International Symposium*, RAID 2014, pages 88–108, 2014.

[170] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331, Berkley, CA, 2010. IEEE, IEEE.

[171] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clementine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. `https://arxiv.org/abs/1702.08719`, 2017.

[172] SeaBIOS. Seabios. `http://www.seabios.org/SeaBIOS`, 2015.

[173] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proc. 21st ACM SIGOPS Symp. on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA, 2007. ACM.

[174] Rebecca Shapiro, Sergey Bratus, and Sean W. Smith. "weird machines" in elf: A spotlight on the underappreciated metadata. In *Proc. of the 7th USENIX Conf. on Offensive Technologies*, WOOT'13, pages 11–11, 2013.

[175] Ahmad Sharif and Hsien-Hsin S. Lee. Total recall: A debugging framework for gpus. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, GH '08, pages 13–20, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[176] Joseph Sharkey. Breaking hardware-enforced security with hypervisors. `https://www.blackhat.com/docs/us-16/materials/us-16-Sharkey-Breaking-Hardware-Enforced-Security-With-Hypervisors.pdf`, 2016.

[177] Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. Cardinal pill testing of system virtual machines. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 271–285, San Diego, CA, August 2014. USENIX Association.

[178] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. Bitvisor: A thin hypervisor for enforcing i/o device security. In *Proc. ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments*, VEE '09, pages 121–130, 2009.

[179] Michael Sikorski and Andrew Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, San Francisco, CA, USA, 1st edition, 2012.

[180] M. L. Soffa, K. R. Walcott, and J. Mars. Exploiting hardware advances for software testing and debugging: Nier track. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 888–891, Honolulu, Hawaii, USA, May 2011. IEEE.

[181] Nguyen Hong Son. Rop chain for windows 8. `http://security.bkav.com/home/-/blogs/rop-chain-for-windows-8/normal`, 2011. Acessado em junho/2016.

[182] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proc. 4th Intl. Conf. on Information Systems Security*, ICISS '08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[183] Sherri Sparks and Jamie Butler. Shadow walker - raising the bar for windows rootkit detection. `http://phrack.org/issues/63/8.html`, 2005.

[184] Patrick Stewin, Jean-Pierre Seifert, and Collin Mulliner. Poster: Towards detecting dma malware. In *Proc. 18th ACM Conf. on Computer and Communications Security*, CCS '11, pages 857–860, NY, USA, 2011.

[185] Adrian Tang, Simha Sethumadhavan, and Salvatore J. Stolfo. *Unsupervised Anomaly-Based Malware Detection Using Hardware Features*, pages 109–129. Springer International Publishing, 2014.

[186] Alexander Tereshkin and Rafal Wojtczuk. Introducing ring -3 rootkits. `https://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf`, 2009.

[187] Vivek Thampi. Udis86 disassembler library for x86/x86-64. `http://udis86.sourceforge.net/`, 2009. Access Date: January/2017.

[188] The Register. Feds count Cryptowall cost: $18 million says FBI. `http://www.theregister.co.uk/2015/06/24/feds_count_cryptowall_cost_18_million_says_fbi/`, June 2015. Access in May 11, 2016.

[189] Kevin Townsend. Mobile malware shows rapid growth in volume and sophistication. `http://www.securityweek.com/mobile-malware-shows-rapid-growth-volume-and-sophistication`, 2016.

[190] Jeroen van Prooijen. The design of malware on modern hardware. `http://www.delaat.net/rp/2015-2016/p89/report.pdf`, 2016.

[191] J. Vanegue. The weird machines in proof-carrying code. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 209–213, May 2014.

[192] Giorgos Vasiliadis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Pixelvault: Using gpus for securing cryptographic operations. In *Proc. 2014 ACM SIGSAC Conf. on Computer and Communications Security*, CCS '14, pages 1131–1142, New York, NY, USA, 2014.

[193] Giorgos Vasiliadis and Sotiris Ioannidis. Gravity: A massively parallel antivirus engine. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection*, RAID'10, pages 79–96. Springer-Verlag, 2010.

[194] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Midea: A multi-parallel intrusion detection architecture. In *Proc. 18th ACM Conf. on Computer and Communications Security*, CCS '11, pages 297–308, New York, NY, USA, 2011. ACM.

[195] Giorgos Vasiliadis, Michalis Polychronakis, and Sotiris Ioannidis. Gpu-assisted malware. *Int. J. Inf. Secur.*, 14(3):289–297, June 2015.

[196] Amit Vasudevan and Ramesh Yerraballi. Stealth breakpoints. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 381–392, Washington, DC, USA, 2005. IEEE Computer Society.

[197] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 264–279, Washington, DC, USA, 2006. IEEE Computer Society.

[198] Amit Vasudevan and Ramesh Yerraballi. Spike: Engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ACSC '06, pages 311–320, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[199] Vassilios Ververis. *Security Evaluation of Intel's Active Management Technology*. PhD thesis, KTH Information and Communication Technology, 2010.

[200] Jack Wallen. Is the intel management engine a backdoor? `http://www.techrepublic.com/article/is-the-intel-management-engine-a-backdoor/`, 2016.

[201] Gary Wang, Zachary J. Estrada, Cuong Pham, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Hypervisor introspection: A technique for evading passive virtual machine monitoring. In *9th USENIX Wksp on Offensive Technologies (WOOT 15)*, pages 1–8, Washington, D.C., August 2015. USENIX Association.

[202] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Proc. 13th Intl. Conf. on Recent Advances in Intrusion Detection*, RAID'10, pages 158–177. Springer-Verlag, 2010.

[203] Jiang Wang, Fengwei Zhang, Kun Sun, and Angelos Stavrou. Firmware-assisted memory acquisition and analysis tools for digital forensics. In *Proc. 2011 6th IEEE Intl. Wksp on Systematic Approaches to Digital Forensic Engineering*, SADFE '11, pages 1–5, Washington, DC, USA, 2011. IEEE Computer Society.

[204] Xueyan Wang and Xiaofei Guo. Numchecker: A system approach for kernel rootkit detection and identification. `https://www.blackhat.com/docs/asia-16/materials/asia-16-Guo-NumChecker-A-System-Approach-For-Kernel-Rootkit-Detection-And-Identification.pdf`, 2016.

[205] Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, and Ramesh Karri. Confirm: Detecting firmware modifications in embedded systems using hardware performance counters. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design*, ICCAD '15, pages 544–551, NJ, USA, 2015. IEEE Press.

[206] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[207] Filip Wecherowski. A real smm rootkit: Reversing and hooking bios smi handlers. `http://phrack.org/issues/66/11.html`, 2009.

[208] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5:32–39, March-April 2007.

[209] Carsten Willems, Ralf Hund, Andreas Fobian, Dennis Felsch, Thorsten Holz, and Amit Vasudevan. Down to the bare metal: Using processor features for binary analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 189–198, New York, NY, USA, 2012. ACM.

[210] Carsten Willems, Ralf Hund, and Thorsten Holz. Cxpinspector: Hypervisor-based, hardware-assisted system monitoring. Technical report, Horst Gortz Institute for IT Security, 2012.

[211] Jiyoung Woo and Huy Kang Kim. Survey and research direction on online game security. In *Proceedings of the Workshop at SIGGRAPH Asia*, WASA '12, pages 19–25, New York, NY, USA, 2012. ACM.

[212] Yubin Xia, Yutao Liu, H. Chen, and B. Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, Boston, MA, June 2012. IEEE.

[213] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. Cfimon: Detecting violation of control flow integrity using performance counters. In *Proc. 2012 42nd Annual IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, DSN '12, pages 1–12, Washington, DC, USA, 2012. IEEE Computer Society.

[214] M. Xianya, Z. Yi, W. Baosheng, and T. Yong. A survey of software protection methods based on self-modifying code. In *2015 International Conference on Computational Intelligence and Communication Networks (CICN)*, pages 589–593, Jabalpur, India, Dec 2015. IEEE.

[215] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: Combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. In *Proc. 8th ACM SIGPLAN/SIGOPS Conf. on Virtual Execution Environments*, VEE '12, pages 227–238, NY, USA, 2012.

[216] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127, Miami, Florida, 2007. ACM, ACM.

[217] Liwei Yuan, Weichao Xing, Haibo Chen, and Binyu Zang. Security breaches as pmu deviation: Detecting and identifying security attacks using performance counters. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 6:1–6:5, New York, NY, USA, 2011. ACM.

[218] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun. Using hardware features for increased debugging transparency. In *2015 IEEE Symp. on Security and Privacy*, pages 55–69, San Jose, CA, USA, May 2015. IEEE.

[219] F. Zhang, K. Leach, A. Stavrou, H. Wang, and K. Sun. Using hardware features for increased debugging transparency. In *IEEE Symp. Security and Privacy (SP)*, pages 55–69, May 2015.

[220] Fengwei Zhang. Iocheck: A framework to enhance the security of i/o devices at runtime. In *2013 43rd Annual IEEE/IFIP Conf. on Dependable Systems and Networks Wksp (DSN-W)*, pages 1–4, Budapest, June 2013.

[221] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. Spectre: A dependable introspection framework via system management mode. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, DSN '13, pages 1–12, Washington, DC, USA, 2013. IEEE Computer Society.

[222] Fengwei Zhang and Hongwei Zhang. Sok: A study of using hardware-assisted isolated execution environments for security. In *Proc. Hardware and Architectural Support for Security and Privacy*, HASP, pages 3:1–3:8, New York, NY, USA, 2016. ACM.

[223] Y. Zhong, H. Yamaki, and H. Takakura. A malware classification method based on similarity of function structure. In *2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet*, pages 256–261, Izmir,Turkey, July 2012. ACM.

Appendix A

Appendix

Branch Monitor Implementation

I present here details on how to enable the branch monitor and how to handle its internal structures.

Enabling monitors and interrupts

All monitoring mechanisms (PEBS, BTS, LBR) are activated by setting the proper flags on the `IA32_DEBUGCTL` MSR register, as shown on Figure A.1. `LBR` and `BTS` flags activate the LBR and BTS mechanisms, respectively, whereas `BTINT` flag enables the interrupt generation.

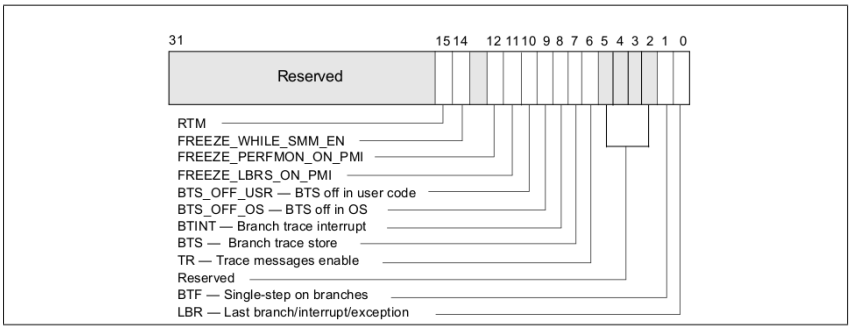

Figure 17-3. IA32_DEBUGCTL MSR for Processors based on Intel Core microarchitecture

Figure A.1: Enabling monitoring: Flags should be set in this MSR in order to activate LBR, BTS and interrupts. The bitmask also defines the data capture scope (user and/or kernel-land). Source: Intel manual [78]

If the `LBR` mechanism is enabled, the captured data is stored on MSR registers. Each register stores either the source or the target address of a given taken branch, thus being named `LAST_BRANCH_FROM` or `LAST_BRANCH_TO`, respectively, as shown on Figure A.2.

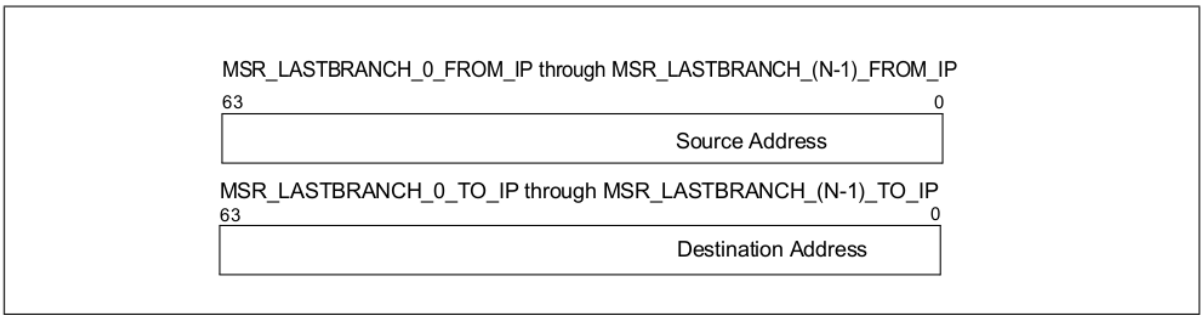

Figure 17-4. 64-bit Address Layout of LBR MSR

Figure A.2: LBR MSRs. When LBR is activated, data is stored on LBR MSRs. Branch source addresses are stored on *FROM* MSRs whereas branch target addresses are stored on *TO* MSRs. These MSR registers are numbered from 0 to N-1, according the number of MSR registers available on the processor. Source: [78]

If the `BTS` is used, an OS-allocated page must be supplied to the processor. This is made by writing the base page address to the `DS_AREA` MSR. The page supplied to the DS register must be filled with data in the pattern shown on Figure A.3. The first fields are related to the `BTS` data whereas the last are related to the `PEBS` mechanism.

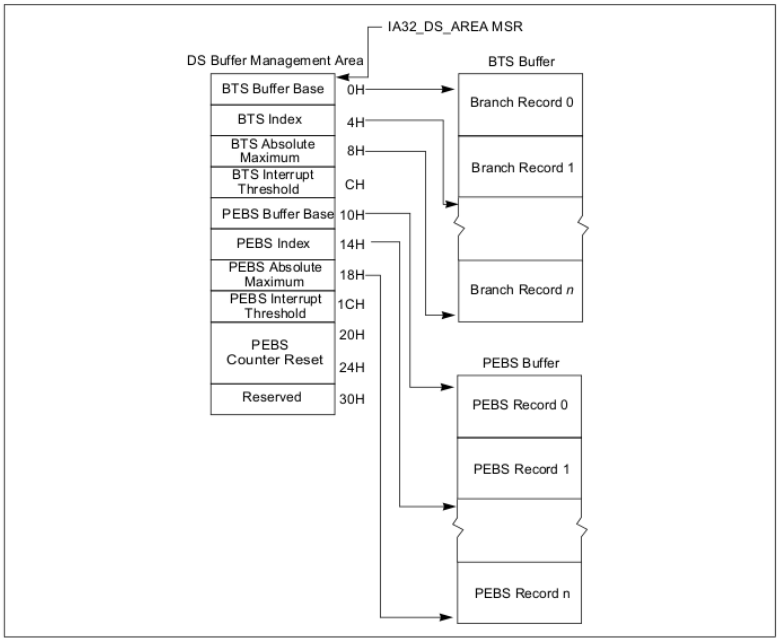

Figure 17-5. DS Save Area

Figure A.3: DS MSR. The address pointed by the DS MSR is an OS allocated page having pointers for the BTS and PEBS mechanisms. Source: [78]

A closer look to the fields, as shown on Figure A.4, shows we are required to supply some data. The `Buffer Base` address is a pointer to another OS allocated page which stores `Branch Records` sequentially. We are also required to provide `BTS Index` address, which is usually the page base address. The `BTS Absolute Maximum` is the address of the last allowed entry – which can be calculated as `buffer size - 1`. The `BTS Interrupt`

`Threshold` is the address of the `Branch Record` which raises an interruption when filled. The same fields are also present for the `PEBS` mechanism.

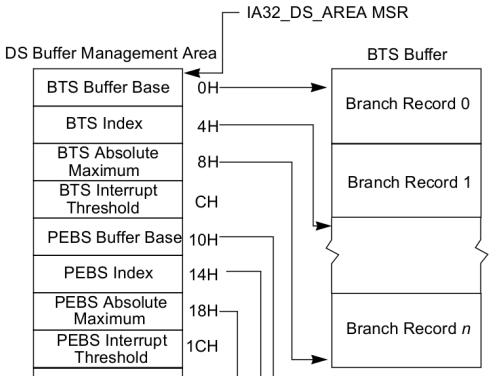

Figure A.4: DS fields. The BTS and/or PEBS fields should be filled with the base address of another OS allocated page, which will store the captured data itself. Besides, it should be filled with pointers to the current stored entry, maximum allowed entry, and threshold addresses. Source: [78]

The `BTS` mechanism allows us to filter some actions through the `CPL-Qualified BTS Encodings`, as shown on Figure A.5. By setting the proper flags, we are able to select, for instance, if we want the capture to occur at user-land (`BTS_OFF_USR`) or kernel-land (`BTS_OFF_OS`).

Table 17-6. CPL-Qualified Branch Trace Store Encodings

TR	BTS	BTS_OFF_OS	BTS_OFF_USR	BTINT	Description
0	X	X	X	X	Branch trace messages (BTMs) off
1	0	X	X	X	Generates BTMs but do not store BTMs
1	1	0	0	0	Store all BTMs in the BTS buffer, used here as a circular buffer
1	1	1	0	0	Store BTMs with CPL > 0 in the BTS buffer
1	1	0	1	0	Store BTMs with CPL = 0 in the BTS buffer
1	1	1	1	X	Generate BTMs but do not store BTMs
1	1	0	0	1	Store all BTMs in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	1	0	1	Store BTMs with CPL > 0 in the BTS buffer; generate an interrupt when the buffer is nearly full
1	1	0	1	1	Store BTMs with CPL = 0 in the BTS buffer; generate an interrupt when the buffer is nearly full

Figure A.5: BTS Filtering. By setting the proper flags, data is captured only when the capture condition is satisfied. Source: [78]

When the `BTINT` bit is set, an interruption is raised an must be handled at kernel-level. When it is raised, the processor looks to the interruption configuration on the `Local Vector Table` (LVT), presented on Figure A.6.

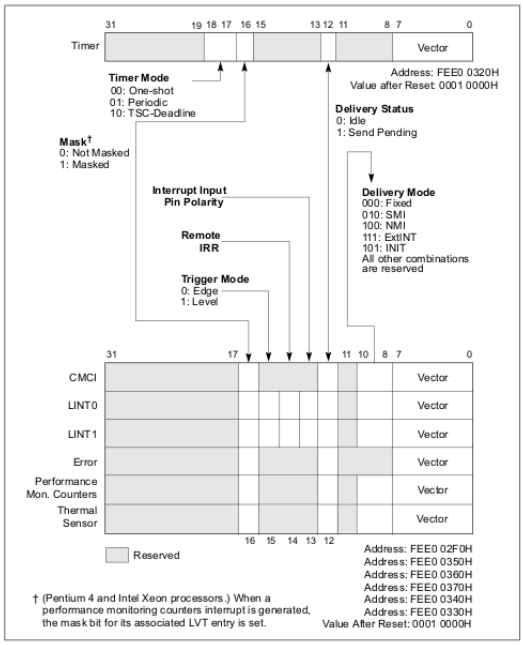

Figure A.6: LVT. A series of entries which control interrupts according their source. Source: [78]

As shown on Figure A.7, we can observe the system stores interrupt configurations for distinct interrupt-sources. On our case, the processor will look for interrupt configurations on the `Performance Mon. Counters` vector.

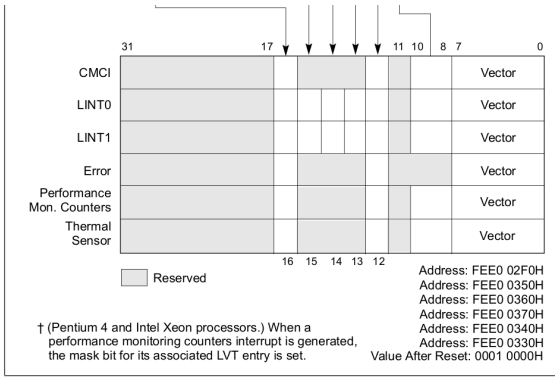

Figure A.7: Performance Counter at LVT. When an PMI occurs, the processor looks into the performance counter entry to identify how the interrupt has to be handled. Source: [78]

Figure A.8 details an LVT entry. Regarding the performance monitor, we have to look to the `Delivery Mode` and `Vector` fields. The first indicates what type of interrupt will be raised: An NMI, as we have used, handled by an special Kernel handler; an SMI, handled by BIOS; Fixed-Mode, as recommended by the vendor. In this case, the number supplied on the `Vector` field is used as IDT entry for ISR registration.

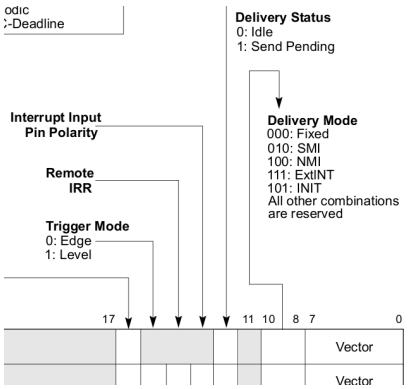

Figure A.8: Interrupt Configuration. For a given interrupt type, we should set delivery mode and the vector number, whose IDT entry points to the correct ISR. Source: [78]

Handling branch-related structures

In order to allow for reproducibility, I present some implementation details which make a developer's life easier.

Initially, we have to define flags, register and memory values, since most configurations depend on properly setting them. Listing A.1 shows definitions of MSR registers (`MSR_`), memory-mapped APIC register (`PERF_COUNTER_APIC`), and flags for enabling BTS (`BTS_FILTER`).

Listing A.1: BTS Definitions. Flag values and MSR numbers. These values are used to interact to the hardware resources.

```
1 #define MSR_LBR_SELECT 456
2 #define MSR_DEBUGCTL 473
3 #define MSR_DS_AREA 1536
4 #define MSR_FROM_BASE 1664
5 #define MSR_TO_BASE 1728
6 #define PERF_COUNTER_APIC 0xFEE00340
7 #define PERF_COUNTER_APIC_VALUE 0x400
8 #define BTS_FILTER_INTERRUPT 0x381
```

We also have to define structures to manipulate `DS` and `BTS` entries. Listing A.2 shows definitions for these structures.

Listing A.2: BTS Definitions. Structs definitions for BTS and DS structures.

```
typedef struct st_BTSBUFFER
{
        UINT64 FROM,TO,MISC;
}TBTS_BUFFER,*PTBTS_BUFFER;

typedef struct st_DSBASE
{
        PTBTS_BUFFER BUFFER_BASE,BTS_INDEX,MAXIMUM,THRESHOLD;
}TDS_BASE,*PTDS_BASE;
```

As we need to set the interruption vector properly, we have to map the APIC register into our driver space. Listing A.3 shows an example of such mapping.

Listing A.3: BTS Definitions. APIC register mapping into userspace.

```
pa.QuadPart=PERF_COUNTER_APIC;
APIC=(UINT32*)MmMapIoSpace(pa,sizeof(UINT32),MmNonCached);
if(APIC!=NULL)
{
        *APIC=PERF_COUNTER_APIC_VALUE;
        MmUnmapIoSpace(APIC,sizeof(UINT32));
}
```

We also have to set the DS area properly to define interrupt thresholds.Listing A.4 shows the definition of DS fields.

Listing A.4: BTS Definitions. DS entries should be filled with pointers to the BTS storage page.

```
void FILL_DS_WITH_BUFFER(PTDS_BASE DS_BASE,PTBTS_BUFFER
    BTS_BUFFER) {
  /* BTS BUFFER BASE */
  DS_BASE->BUFFER_BASE=BTS_BUFFER;
  /* BTS INDEX */
  DS_BASE->BTS_INDEX=BTS_BUFFER;
  /* BTS MAX */
  DS_BASE->MAXIMUM=(PTBTS_BUFFER)(((UINT_PTR)BTS_BUFFER)+(
      SIZE_BTS_BUFFER*sizeof(TBTS_BUFFER)));
  /* BTS Threshold */
  DS_BASE->THRESHOLD=(PTBTS_BUFFER)(((UINT_PTR)BTS_BUFFER)+(
      THRESHOLD_BTS_BUFFER*sizeof(TBTS_BUFFER)));
```