Virtualisation-based security countermeasures in software runtime systems

Francesco GADALETA

Jury :
Em. Prof. Dr. ir. Carlo Vandecasteele, chair
Prof. Dr. ir. Wouter Joosen, promotor
Prof. Dr. ir. Frank Piessens, co-promotor
Prof. Dr. ir. Vincent Rijmen,
Prof. Dr. Danny Hughes,
Prof. Dr. Nicola Zannone
Dr. Lieven Desmet,
Dr. Pieter Philippaerts

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

September 2013

ISBN 978-94-6018-714-8
D/2013/7515/100

Abstract

The appearance of global Internet services like social networking, remote storage and mobile computing into marketplace, is consistently influencing the way of using the computing infrastructure. As systems become larger and more complex, the need to optimise the infrastructure in favour of reliability and redundancy becomes an imperative. Virtualisation technology seems to have partially fulfilled the needs dictated by growth - first of all physical space and energy consumption - by redefining the concept of infrastructure and paving the way for new business models such as cloud computing.

One consequence of the aforementioned highly connected environment is that software bugs and malicious attacks can spread much faster and more effectively than it was in the past. Attacks to operating systems are facilitated by the existence of devices that are permanently connected, such as smart phones, tablets and mobile devices in general. In such conditions, infections can be easily taken at a global scale.

Security researchers have been looking at virtualisation technology as an approach that can potentially find the solutions to the well known security problems of operating system kernels. As a matter of fact, successful low level attacks can circumvent or disable many of the traditional countermeasures in place within the same target system.

Another trend that, according to the security research community, might be a cause for concern in the near future, is the tendency to shift current computer use to remote Internet services. This is making the web browser one of the most considerable actors of today's computer usage. As a consequence, the web browser is gaining more and more attention from attackers, due to its prominent position within user's experience.

Despite the active contribution of researchers to mitigate the aforementioned security issues, one major challenge to focus in the immediate future consists in minimising the performance overhead, while guaranteeing the highest degree of security. Such a task seems achievable only by the puzzling tradeoff between performance and security that usually sacrifices the former in favour of the latter or vice versa.

This dissertation contributes security mitigation techniques that address the aforementioned challenges. First, we focus on virtualisation technology to tackle the problem of operating system security. A countermeasure that relies on the cooperation between the target system and the virtualisation architecture, protects those critical memory locations within the target system that can be potentially compromised. Within the same field, a more general framework that protects operating systems by enforcing the execution of trusted code is presented.

Secondly, a security measure that improves web browser security against memory corruption attacks is provided. We also argue in favour of the role that virtualisation technology can play within such environments and discuss a realistic

scenario for integrating our security countermeasure into similar software architectures delivered on demand, as in current cloud computing settings.

Abstract - Dutch

De introductie van globale Internetdiensten zoals sociale netwerken, externe opslag en mobile computing in de markt, benvloedt voortdurend de manier waarop men de beschikbare computerinfrastructuur gebruikt. Naarmate systemen groter en complexer worden, neemt ook het belang om de infrastructuur te optimaliseren voor betrouwbaarheid en redundantie toe. Virtualisatietechnologie lijkt de behoeften voor groei - in de eerste plaats de benodigde fysieke plaats en het energieverbruik - gedeeltelijk te hebben vervuld, door het concept van infrastructuur te herdefiniren en door de basis te leggen voor nieuwe bedrijfsmodellen zoals cloud computing.

En van de gevolgen van een dergelijke alom verbonden omgeving is dat softwarefouten en -aanvallen zich veel sneller en effectiever kunnen verspreiden dan in het verleden. Aanvallen op besturingssystemen worden vergemakkelijkt door de aanwezigheid van apparaten die voortdurend verbonden zijn, zoals smartphones, tablets en mobiele apparaten in het algemeen. In zulke omstandigheden kunnen besmettingen op wereldschaal zich gemakkelijk voordoen. Omdat succesvolle laag-niveau aanvallen vele van de traditionele beveiligingsmaatregelen voor computersystemen kunnen omzeilen of uitschakelen, hebben beveiligingsonderzoekers zich op virtualisatietechnologie gericht om oplossingen te vinden voor de welbekende beveiligingsproblemen van besturingssystemen.

Een andere trend die volgens de gemeenschap van beveiligingsonderzoekers verontrustend is, is het toenemend gebruik van externe Internetdiensten. Deze trend maakt van de webbrowser een van de belangrijkste actoren van het hedendaags computergebruik. Bijgevolg krijgt de browser steeds meer aandacht van aanvallers, wegens zijn prominente positie in de gebruikerservaring.

Ondanks de actieve bijdragen van onderzoekers om de eerdergenoemde beveiligingsproblemen tegen te gaan, blijft het een grote uitdaging om de performantiekost van beveiligingsmaatregelen te minimaliseren, zonder afbreuk te doen aan hun effectiviteit. Een dergelijke taak lijkt enkel haalbaar door een afweging tussen performantie en beveiliging te maken, waarbij het ene meestal ten koste gaat van het andere, of vice versa.

Dit proefschrift handelt over beveiligingsmaatregelen die een antwoord kunnen bieden op de eerdergenoemde uitdagingen. Ten eerste concentreren we ons op virtualisatietechnologie om het probleem van de beveiliging van besturingssystemen aan te pakken. Een beveiligingsmaatregel die steunt op de samenwerking tussen het te beveiligen systeem en de virtualisatiearchitectuur beschermt kritieke geheugenlocaties binnen het systeem die potentieel kunnen worden misbruikt door een aanvaller. Binnen hetzelfde domein wordt een meer algemeen raamwerk voorgesteld dat besturingssystemen beschermt door het afdwingen van de uitvoering van vertrouwde code.

Ten tweede wordt een beveiligingsmaatregel die de webbrowser beschermt tegen geheugencorruptieaanvallen voorgesteld. We bespreken de rol die virtual-

isatietechnologie kan spelen binnen dergelijke omgevingen en we behandelen een realistisch scenario voor het integreren van onze beveiligingsmaatregel in gelijkaardige softwarearchitecturen die op aanvraag worden geleverd, zoals in opkomende cloud computing omgevingen.

Acknowledgements

The work you are going to read would not have been possible without the support and the contribution of several people to whom I express my deepest gratitude.
I am extremely indebted to my advisors Prof. Wouter Joosen and Prof. Frank Piessens, who accepted me to be part of the DistriNet Research Group and who supported me until the end of this beautiful journey.
This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, the Research Fund KU Leuven and the EU FP7 project NESSoS.
Not only to my advisors but also am I grateful to my examiners, Dr. Lieven Desmet, Prof. Nicola Zannone, Dr. Pieter Philippaerts, Prof. Danny Hughes, Prof. Vincent Rijmen and Prof. Dr. ir. Carlo Vandecasteele, chair of the PhD committee, for having accepted to review my dissertation and who made it a better scientific document.
I will always remember my colleagues Dr. Yves Younan, Nick Nikiforakis, Dr. Jan Tobias Muehlberg, Milica Milutinovic, Pieter Agten, Job Noorman, Willem De Groef, Dr. Frederic Vogel, Raoul Strackx, Zubair Wadood Bhatti, Dr. Davy Preuveneers, Dr. Dimiter Milushev Dr. Aram Hovsepyan, Dr. Riccardo Scandariato and many others from the Department of Computer Science, with whom I have been sharing liters of coffee, kilos of chocolates and an indefinite amount of hours to discuss our ideas.
I address my sincere gratitude to the project office managers Katrien Janssens, Ghita Saevels and Annick Vandijck for their great support to our research at DistriNet.
I would also like to thank all members of the administration task force for their professional approach in solving the numerous issues and troubles that a PhD student can provide: Marleen Somers, Karen Verresen, Esther Renson, Liesbet Degent, Karin Michiels, Margot Peeters, Katrien Janssens and Ghita Saevels.
A special thank goes to Sajjad Safaei, who reviewed the first draft of this work, always surprising me with his personal and stylish English idioms. He did a great job even in hiding the tremendous boredom of the subject, not really suitable for an anthropologist.
My deepest gratitude is addressed to my parents Marta and Domenico, my sister Jlenia and her family Antonio and Fabrizio - who will read this work as soon as he fixes the common issues of his first words - who have been patiently waiting for me to achieve my goals, accepting the compulsory and sometimes dreary distance that is separating us for such a long time. Thank you. This dissertation is dedicated to you.
I must, of course, thank Valeria Contarino, Camilla Fin, Tomohiro Kosaka, Audrey Casier, Laura, Gwen, Stephanie, and the long list of people I am most certainly forgetting, who unconsciously contributed to my work by convincing me of the fact that hills can only be climbed.
My journey continues. This time with a bigger luggage. Thank you.

Francesco
Leuven, August 2013

Contents

1 Introduction **1**
- 1.1 Problem statement . . . 3
- 1.2 Contributions . . . 4

2 Virtualisation: a new paradigm **7**
- 2.1 Virtualisation technology . . . 8
  - 2.1.1 Performance . . . 14
- 2.2 Benefits and drawbacks . . . 17
- 2.3 Drawbacks of virtualisation: a case study . . . 19
  - 2.3.1 Evaluation . . . 22
  - 2.3.2 Discussion . . . 23
- 2.4 Rethinking security . . . 24

3 Hypervisor-based invariance-enforcing framework **25**
- 3.1 Motivation . . . 27
  - 3.1.1 Rootkits . . . 27
  - 3.1.2 Threat Model . . . 28
- 3.2 Approach . . . 29
- 3.3 Evaluation . . . 34
- 3.4 Limitations . . . 39
- 3.5 Related work . . . 44
  - 3.5.1 Hardware-based countermeasures . . . 44
  - 3.5.2 Kernel code integrity . . . 46
  - 3.5.3 Analysis and profiling systems . . . 49
- 3.6 Summary . . . 52

4 Hypervisor-enForced Execution of Security-Critical Code **55**
- 4.1 Problem description . . . 56
- 4.2 Approach . . . 57
- 4.3 Evaluation . . . 60
  - 4.3.1 Macro-benchmarks . . . 61

4.3.2 Micro-benchmarks 62
4.4 Related work 65
4.4.1 Security Agent Injection 65
4.4.2 Hardware-based techniques 66
4.4.3 Formally verified systems 69
4.5 Summary 71

5 Secure web browsers with virtualisation **73**
5.1 Motivation 73
5.2 Problem description 75
5.3 Approach 78
5.4 Implementation 80
5.5 Evaluation 82
5.5.1 Performance benchmarks 82
5.5.2 Memory overhead 85
5.5.3 Security evaluation 86
5.6 Hypervisor integration 90
5.7 Related work 91
5.7.1 Heap-spraying defences 91
5.7.2 Alternative countermeasures 93
5.7.3 Virtualisation-based countermeasures 95
5.8 Summary 97

6 Conclusions **99**
6.1 Future work 101

List of Figures

2.1 Schema of Type 1 (native) hypervisor executing directly on physical hardware and Type 2 (hosted) hypervisor running on top of a commodity operating system 9
2.2 Lifecycle of a general virtual environment with two guests. When the guest executes a privileged instruction, control returns to the hypervisor by VM exit. The hypervisor executes the instruction on behalf of the guest, updates its VMCS and returns to guest's space using VM entry. 10
2.3 Schema of two-level translation from LPA (Logical Page Address) to PPA (Physical Page Address) and MPA (Machine Page Address). Red arrows indicate the mappings from guest to host that the hypervisor must keep synchronised in its shadow pages. 15
2.4 Intel's Extended Page Table hardware-supported translation mechanism to map guest addresses to host physical addresses without shadow paging. An equivalent method has been developed by AMD, with the name of NPT (Nested Page Tables) 16
2.5 Latency of VM exit, VMREAD, VMRESUME instructions across Intel processors 18

3.1 High level view of trusted module-hypervisor interaction 33
3.2 Schema of integrity checking 15000 objects in 150 VMExit trapped events 36
3.3 Monte carlo simulations for the analysis of the probability of attack, compromising from 1 to 11 objects and restoring their original content after 100 task switches 42
3.4 Probability of successful attack compared to restoring rate of 75, 90 and 120 task switches 43

4.1 Schema of HyperForce. Highlighted components indicate parts of the system that need instrumentation/modification. The trusted module within the guest is never unloaded. 58

4.2 Selection of Interrupt Gate via Interrupt Descriptor Table Register in the IA32 architecture. Dashed rectangles indicate that any attempt to write to these areas is trapped and handled by the hypervisor. 60
4.3 Schema of in-hypervisor approach implemented in Linux KVM. Highlighted components indicate parts of the system that need instrumentation/modification. The trusted module installed in the guest is unloaded after boot. 61

5.1 Schema of NOP sled and shellcode appended to the sequence . . . 76
5.2 A heap-spraying attack: heap is populated of a large number of $NOP-shellcode$ objects. The attack may be triggered by a memory corruption. This could potentially allow the attacker to jump to an arbitrary address in memory. The attack relies on the chance that the jump will land inside one of the malicious objects. 77
5.3 Javascript engine's JSString type is considered a threat for a heap-spraying attack since member *chars* is a pointer to a vector of size $(length + 1)$* *sizeof(jschar)* 80
5.4 Representation of the transformed string in memory: characters at random positions are changed to special interrupting values. The potential execution of the object's contents on the heap would be interrupted by the special value (in blue). 81
5.5 How metadata is stored to array *rndpos*: index i within the array contains the value of the position in the string; index $(i+1)$ contains its original value 82
5.6 Probability of detection versus size of injected code. 88
5.7 Probability of detection compared to detection rates for injected code of several sizes 89
5.8 Schema of hypervisor integration: upon loading malicious content from malicious.com (1) and local detection of heap-based attack, the guest notifies the hypervisor (2) sending the malicious URL that caused the attack. The hypervisor will deliver this information to all virtual machines running on top (3) or, alternatively will update a blacklist in its private space. 92

List of Tables

2.1 Intel VT-x hardware-supported virtualisation instruction set 12
2.2 List of privileged instructions that cause a VMExit event 13
2.3 SPEC CPU2000 benchmark results of Xen implementation 23

3.1 Hooking methods of common Linux rootkits 28
3.2 Testing machine specification 34
3.3 Performance overhead of our countermeasure in action measured with lmbench benchmark suite. IO tasks and file system operations can show better performance when the countermeasure is active, due to the longer time spent in hypervisor space. While user experience is generally slower, DMA operations can show slightly better throughput. 37
3.4 Results of ApacheBench sending 100000 requests, 50 concurrently on local lighttpd webserver 38
3.5 Performance overhead of our countermeasure in action on compression/decompression (bzip/bunzip) and compilation of kernel code . 38
3.6 Performance of our countermeasure running SPEC2000 benchmarks 38
3.7 Comparison of existent countermeasures that protect commodity operating systems against kernel mode rootkits. Column *protect nc data* indicates whether non-control data are protected; *special hw* indicates whether special-purpose hardware is required; *protect ROP* indicates whether the countermeasure protects against Return-Oriented Programming attacks; *Overhead* indicates whether the overhead of the countermeasure is low or high 49

4.1 Macro benchmarks (*in-host* OS and *in-guest* OS) evaluating the integrity checking method implemented with and without HyperForce 62
4.2 Overhead of kernel integrity checking using the HyperForce framework (HF) is measured against in-hypervisor alternative with LMbench micro-benchmarks within the host machine. Operations are measured in microseconds. 63

4.3 Overhead of HyperForce is measured against in-hypervisor approach with LMbench micro-benchmarks within the guest machine. Operations are measured in microseconds. 63

5.1 Performance overhead of countermeasure in action on 8 popular memory demanding web sites. 83
5.2 Microbenchmarks performed by SunSpider Javascript Benchmark Suite 84
5.3 Peacekeeper Javascript Benchmarks results (the higher the better). 84
5.4 V8 Benchmark Suite results (the higher the better). 85
5.5 Memory overhead of countermeasure in action on three Javascript benchmarks suites. 86

List of Listings

2.1 A simple system call in traditional UNIX 16
2.2 A function that is vulnerable to buffer overflow 20
2.3 The standard prologue and epilogue of vuln_foo() 20
2.4 Instrumented assembly code of vuln_foo() 20
5.1 A Javascript code snippet to perform a basic heap-spraying attack usually embedded in a HTML web page 77

Chapter 1

Introduction

A scientific truth does not triumph by convincing its opponents and making them see the light, but rather because its opponents eventually die and a new generation grows up that is familiar with it.

Max Planck

The last years have witnessed a rapidly growing number of complex services running on an increasing number of machines connected across the globe. Social networking, remote storage, media services and mobile computing are only a few examples of the enormous networked systems that made their appearance recently into marketplace [1–3].

An immediate consequence of such a connected world is that damage caused by bugs or malicious attacks can spread faster and more effectively than in the past [4–6]. In a more connected world, attackers have the option of resorting to far more viable means of materialising their goals without the need to design attacks specific to the different types of existing machines.

Despite the appearance of new Internet services and client side applications, modern computing devices and the radical change of infrastructure technology, the operating system is still the main target of attackers, due to its crucial role in providing the needed interaction between a running process and the external world. Essentially, the operating system controls directly hardware like the keyboard and other human interaction devices, video or network cards and any other chipset via device drivers; it executes critical operations like memory allocation and process scheduling; it initialises network connections on behalf of the running processes and provides the infrastructure required by multiple applications in order to be

executed on the same physical hardware. Moreover, the operating system provides the security mechanisms aimed at protecting the private address space of each application running in a less privileged level. Being, by necessity, part of the trusted computing base (TCB) of a system, it is not surprising that the number of attacks targeting such a critical component has never decreased over the years.

As a matter of fact, the increasing number of successful attacks on operating system kernels is yet another indicator of a more suitable environment for malicious activities. A software bug in the kernel of a widely used operating system can give rise to activities such as spamming, keylogging or stealth of private data, all of which can have immense global impact, as was the case with *Storm Worm* [7] and other bank fraud cases [8] or as in a more recent attack deployed with the cooperation of the kernel and the web browser [9, 10].

Moreover, in the era of mobile computing, devices tend to be permanently connected. For mobile phone users constantly connected to the internet, it is at times easy to forget that they are, in effect, always on the Internet. A mobile device that has been compromised, and is thus capable of executing malicious code locally, could take advantage of this permanent connection and infect other peers much more easily than ever before [11–13].

Another effect of the growth of internet services is the influence they have on the computing infrastructure. Modern Internet services need a greater level of reliability and redundancy [14–16].

As expected, the constant growth of complex internet services is followed by another phenomenon: a greater demand for improvement in reliability and optimisation of physical space and energy consumption [17–19]. As systems become larger and more complex, the need to optimise resources becomes an imperative [20]. New technologies appear to have partially, but nonetheless efficiently, addressed the needs dictated by growth [21–25] .

Virtualisation is one such technology which arrived in the late 1990s[1]. The technology, however, started to gain popularity only in the mid 2000s, when vendors like Intel and AMD extended the instruction set of their processors in order to provide hardware support with the purpose of lowering the performance overhead of the new technology. This fact allowed an extensive deployment to production systems. Moreover, hardware support allowed the parallel execution of a number of operating systems on the same physical hardware in complete isolation from each other, a feature that gave rise to an entirely new computing era. Virtualisation has, in fact, redefined the concept of infrastructure and is paving the way for new business models, one of which is *cloud computing* [27].

In this new paradigm, computing resources or even entire virtual infrastructures can be purchased and allocated at a moment's notice. A branch of cloud

[1]Virtualisation was first developed in the 1960s to better utilise mainframe hardware. In the late 1990s virtualisation was introduced to the x86 architecture as it became the industry standard [26] and was designed to execute several operating systems simultaneously on the same physical processor.

computing, usually referred to as *desktop virtualisation*, exploits the benefits of this new technology to deliver entire desktop environments, applications and user data on demand. Requirements like performance, easy management and mobility can be fulfilled in a straightforward way since they are fully supported by the design.

Yet the numerous benefits of virtualisation are not limited to the industry and the IT business world. The field has also captured the attention of the security research community. Security researchers have been looking at virtualisation technology as an alternative approach for finding solutions to the familiar problems of the past decade and to provide mitigations to unresolved challenges. One such challenge involves the protection of operating system kernels. When a suspicious program is running at a privilege level as high as that of the operating system kernel, it may be extremely difficult to detect its malicious intent. In such a scenario, both trusted and malicious code are granted the same access to available resources. Therefore the likelihood of malicious code disabling or circumventing any countermeasure becomes extremely high.

One possible way to overcome the execution of trusted and malicious code with the same privileges, involves the isolation of the system in need of protection from the code that implements the countermeasure. As we shall demonstrate in Chapter 2, hardware-supported virtualisation technology accommodates such a demand.

1.1 Problem statement

Based on our belief that virtualisation technology will increasingly take over traditional systems, starting from internet services and desktop computing to mobile devices, we examine the possibility of using the new technology to solve issues that are closer to the security world rather than energy and space optimization. Here, we briefly discuss the different types of attacks that security researchers have been responding to thus far and for which we provide security mitigations. Although the research community has responded quite actively with many solutions that use different technologies, far greater effort should be dedicated to incorporating these solutions for production systems. Despite the aforementioned solutions, kernel attacks are still common and effective against the protection mechanisms that are usually in place in commodity operating systems. On the other side, considerable usage of the web browser in modern digital life is making its architecture more complicated, increasing the chances of discovering vulnerabilities that might be exploited. In the course of this thesis we provide security mitigations for two kinds of attacks: attacks that compromise operating system kernels and attacks through modern web browsers and applications delivered on demand.

Attacks to the kernel Due to their prominent position amongst user applications and hardware, operating system kernels have been a common target for

attackers attempting to circumvent and modify protections to the best of their advantage. Such attacks appear quite often in the form of device drivers that are supposed to extend the kernel with a new feature or to control a particular type of device connected to the rest of the hardware, but are revealed to be malicious [28–31].

Even a software bug in the kernel can be exploited to inject and execute malicious code. Regardless of the way in which the kernel is compromised, the result is malicious code running at the highest privilege level. This can not only make the attack stealthy but also has the potential to circumvent any countermeasure in place.

Attacks through web browsers In order to provide better web experience, modern web browsers are supported by a richer environment and more complex software architecture. It is common to extend the functionality of modern web browsers with plug-ins or script language engines that interact via an API. This higher complexity has also led to numerous security problems [32, 33]. Moreover, the browser is often written in unsafe languages for performance reasons. This exposes it to the memory corruption vulnerabilities that can occur in programs written in these languages, such as buffer overflows and dangling pointer references. The presence of interpreters, plugins and extensions that can easily be embedded to the core of the browser made these environments an appealing target for the most recent attacks [33–36].

1.2 Contributions

The first part of our work tackles the problem of attacks to operating system kernels. Virtualisation technology fits very well in such scenarios due to a number of interesting features that come by design, such as isolation and hardware support. Therefore, we argue that a logical place where security ought to be increased is at the level of the *hypervisor*, the layer that is interposed between the virtual machine and the physical hardware.

In Chapter 2 we describe virtualisation technology and we provide details regarding the interposition of the hypervisor during execution of the operating systems running on top. In the same chapter we also explain the drawbacks of virtualisation technology - in terms of performance overhead - with the aid of a case study that involves the mitigation of *stack based buffer overflows* - and provide our conclusion about the reasons of such a performance impact.

A countermeasure against kernel attacks, which relies on the cooperation between the target system and the hypervisor, is described in Chapter 3. A mitigation that we found to be effective against the corruption of kernel code at runtime consists in checking the integrity of potential targets, independently from the execution of the virtual machine, namely the target system. We identify the

hypervisor as a suitable place in which implementing the aforementioned checkings.

A more general purpose framework that protects an operating system kernel running within a virtual machine is explained in Chapter 4. This protection is achieved by enforcing the execution of secure code into the target machine. While maintaining target code and secure code isolated, our solution also provides a minimal overhead, due to the fact that secure code executes within the same system to be protected.

Another environment that needs attention from the security research community, due to its prominent position in everyday computer usage is the web browser and applications delivered on demand. We focus on these other environments in the second part of our work. Although security of web browsers is a very active field of research, very little has been done to protect browsers with virtualisation in mind.

Therefore, we provide a security measure that improves web browser security and we explore the benefits of virtualisation technology in this area. A recent heap-based attack, by which attackers can allocate malicious objects to the heap of a web browser, by loading a specially forged web page that contains Javascript code, has drawn the attention of security researchers who operate in the field of browser security. We explain the details of a lightweight countermeasure against such attacks, known as *heap-spraying*. We also argue in favour of the role that virtualisation technology can play within this environment and discuss a possible strategy for integrating such a countermeasure into web browsers and applications with a similar architecture, delivered on demand. Details are provided in Chapter 5.

Conclusions are given in Chapter 6. Future work and research opportunities are discussed in the same chapter.

The work presented in this dissertation has led to the following publications:

- Francesco Gadaleta, Yves Younan, Bart Jacobs, Wouter Joosen, Erik De Neve, Nils Beosier, Instruction-level countermeasures against stack-based buffer overflow attacks, Eurosys, Nuremberg, 1-3 April 2009

- Francesco Gadaleta, Yves Younan, Wouter Joosen, Bubble: a Javascript engine level countermeasure against heap-spraying attacks, ESSoS, Pisa, 3-4 February 2010

- Francesco Gadaleta, Nick Nikiforakis, Yves Younan, Wouter Joosen, Hello rootKitty: A lightweight invariance-enforcing framework, ISC Information Security Conference, Xi'an China, 2011

- Francesco Gadaleta, Raoul Strackx, Nick Nikiforakis, Frank Piessens, Wouter Joosen, On the effectiveness of virtualization-based security, IT Security, Freiburg (Germany), 07-10 May 2012

- Francesco Gadaleta, Nick Nikiforakis, Jan Tobias Mhlberg, Wouter Joosen, Hyperforce: hypervisor-enforced execution of security-critical code, IFIP Advances in Information and Communication Technology, Heraklion, 04-06 June 2012

Chapter 2

Virtualisation: a new paradigm

Without deviation progress is not possible.

Frank Zappa

The *cloud* is probably the most widely used technological term of the last years [27]. Its supporters present it as a complete change in the way that companies operate that will help them scale on-demand without the hardware-shackles of the past. CPU-time, hard-disk space, bandwidth and complete virtual infrastructure can be bought at a moment's notice. Backups of data are synced to the cloud and in some extreme cases, all of a user's data may reside there. Cases of this kind already occur in frameworks such as Chromium OS, IBM Smart Business Desktop Cloud and other commercial products like eyeOS, CloudMyOffice, Dincloud, etc.

On the other hand, opponents of the *cloud* treat it as a privacy nightmare that will take away the users control over their own data and place it in the hands of corporations, resulting in great risk to the privacy, integrity and availability of user data [37, 38].

Regardless of one's view of the cloud, one of the main technologies that makes the cloud-concept possible is virtualisation [39]. Unsurprisingly, the concept of virtualising hardware resources is not new. It first made its debut back in the 70s. However, all the conditions for efficient system virtualisation, such as those introduced by Popek and Goldberg in [40], could be fulfilled only with virtualisation-enabled processors. A more detailed explanation of hardware-supported virtualisation is given in Section 2.1.

Once newer hardware met acceptable requirements of efficiency and performance, the new virtualisation paradigm found its way in several scenarios that shared common needs: decoupling services from the infrastructure and optimising utilisation of physical resources.

The trend of server consolidation, virtualisation of storage, networking and entire machines, paved the way for an entirely new business model in which changing, adding, removing, or migrating infrastructure components could be achieved easily with a dramatic drop in operational costs. Considering the costs required to switch to a new technology such as hiring and training, a cost reduction of approximately 55% can be obtained [41]. From another perspective, virtualisation technology reduces data center energy consumption by 10% to 40% [42].

The reduced performance impact that comes with hardware support is constantly accelerating the adoption of virtualisation technology and facilitating the migration from traditional infrastructure to the new paradigm.

The success of virtualisation technology, which was first noticed in the large computing environments of the industry, is influencing other types of users with more use cases every day, such as those provided by desktop virtualisation. Bringing virtualisation technology to mobile devices seems to be a logical step that might come next in a world with a constantly growing need to innovate.

The important features of virtualisation technology, which were discovered to be unavoidable for hardware design, also attracted security researchers whose interests fall in areas such as the study of malware, application-sandboxing and protection of operating system kernels, as explained more extensively in Chapter 3.

2.1 Virtualisation technology

Virtualisation is the set of software and hardware technologies that together allow for the existence of more than one running operating system on top of a single physical machine. While initially all of the needed mechanisms for virtualisation were emulated by software, the sustained popularity of the new technology and the desire for speed and reduced performance impact led to their implementation in hardware [43, 44]. Today, both Intel[1] and AMD[2] support a set of instructions whose sole purpose is to facilitate the virtualisation of operating systems.

We report the list of instructions added to the standard Intel instruction set to enable hardware-supported virtualisation in Table 2.1. Although we will refer to the Intel architecture, as this is the hardware used for our prototypes, throughout

[1] Intel-VT architecture `http://www.intel.com/technology/virtualization/technology.htm`

[2] AMD-SVM architecture `http://sites.amd.com/us/business/it-solutions/virtualisation`

this work, the concepts introduced here can be applied to equivalent hardware, supported by AMD and other vendors.

Generally speaking, the main components of a virtualisation system are:

- the *Hypervisor* (also referred to as Virtual Machine Monitor - VMM), which directly controls the hardware and offers an abstraction of the physical processor, memory, interrupt mechanisms, etc., to a higher level where the guest software usually executes. Despite the differences between hypervisors for commercial, experimental and academic purposes, we will refer to their common architecture and to the general aspects of hardware support.

- the *Guest* (also referred to as Virtual Machine - VM) represents the operating system and its applications that are running on top of a hypervisor. In the virtualisation setting we will be referring to in this work, the guest operating system normally executes without any modification. The hypervisor exposes an abstract version of the real hardware that matches[3] the physical one. This stratagem is essential to the execution of commodity operating systems that have been designed to execute on *bare metal*, for instance, on real physical hardware.

A former classification of hypervisors, provided in [40], makes a distinction between hypervisors that run directly on physical hardware, called *Type 1 hypervisors* or *native*, from those that run within a commodity operating system environment, called *Type 2 hypervisors* or *hosted*. The classification is depicted in Figure 2.1.

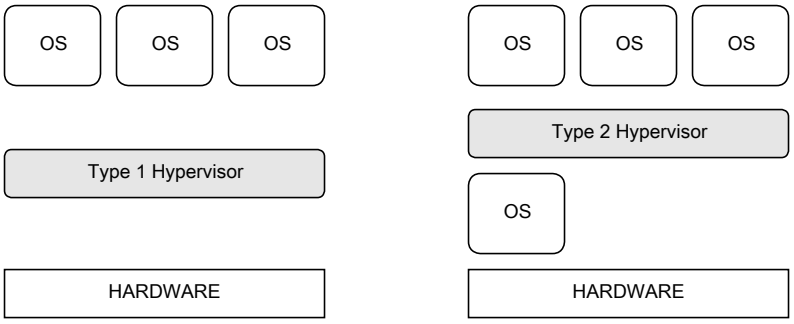

Figure 2.1: Schema of Type 1 (native) hypervisor executing directly on physical hardware and Type 2 (hosted) hypervisor running on top of a commodity operating system

[3] Although the guest will execute as it were running on bare metal there are some mechanisms that cannot be reproduced and therefore will act differently, such as timers.

Despite the place where an hypervisor is executing, a separation mechanism is required to run the guest operating system in a distinct software level.

In order to guarantee separation between the kernel and regular applications, traditional processors support two execution modes: operating system code runs in root mode and applications usually run in non-root mode. In the same way, the guest and the hypervisor need to be completely isolated from each other. This isolation is guaranteed by the virtualisation-enabled processor architecture. To allow the execution of the hypervisor at a higher privilege level and of the guest without modifying its code with *hypervisor awareness*, virtualisation-enabled processors have been extended with a new mode, called *VMX mode*. In this newer architecture, the hypervisor will execute in VMX-root mode and the guest kernel will run in VMX non-root mode.

The processor executing in VMX non-root has a restricted behaviour. Consequently, if the guest kernel executes specific instructions, a trap will be generated and control will be returned to the hypervisor. The transition from VMX non-root (guest) to VMX root (hypervisor) is usually referred to as *VM exit*. The transition in the other direction, from VMX root to VMX non-root, is called *VM entry*.

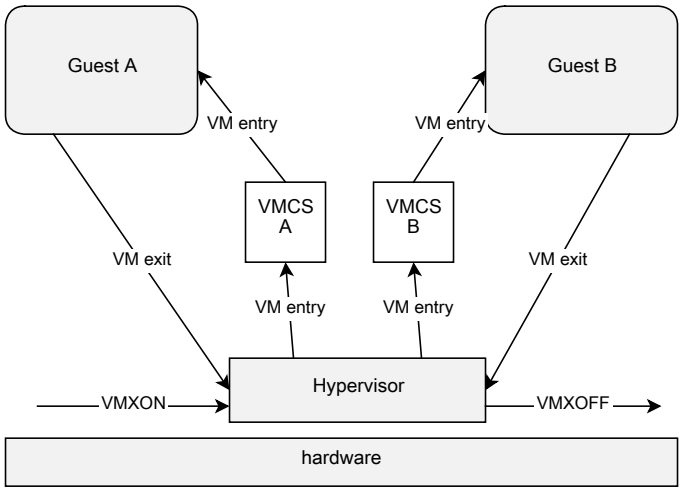

Figure 2.2: Lifecycle of a general virtual environment with two guests. When the guest executes a privileged instruction, control returns to the hypervisor by VM exit. The hypervisor executes the instruction on behalf of the guest, updates its VMCS and returns to guest's space using VM entry.

Both Intel and AMD architectures do not provide any hardware setting that is visible from the guest and that might reveal the current processor's mode. This is an effective solution to prevent guest code from detecting whether it is running on

physical or virtual hardware. However, measuring the delay of specific operations with external timers[4] has been found to be a viable way to detect the execution of code in virtualised environments rather than on real hardware [45, 46].

As a recurrent mechanism exploited by the countermeasures described in the proceeding chapters, we will briefly explain the lifecycle of a virtualised operating system running on top of a general purpose hypervisor.

Upon execution of the VMXON instruction, which enables the hardware-assisted virtualisation instruction set, the hypervisor enters the guest using a VM entry. At this point guest code executes until the next VM exit, and thus transfers control to a specific entry point in hypervisor space. An appropriate action is taken depending on the reason that caused the VM exit. The hypervisor usually performs the requested operation on behalf of the guest. After the execution of the privileged instruction, the hypervisor updates the context of the guest and returns using VM entry. The aforementioned mechanism is illustrated in Figure 2.2.

Another event that requires the hypervisor's intervention occurs when executing several virtual machines at the same time. In this case the hypervisor must restore the context of the next virtual machine to execute using instructions like $VMRESUME$ and $VMLAUNCH$. The procedure resembles the save-and-restore mechanism typical of task switching in traditional operating systems.

In both scenarios the context of the virtual machine is saved into a special data structure called Virtual Machine Control Structure (VMCS). The aforementioned data structure is defined *special* in the sense that it is not saved into memory that is normally accessible from the guest. Moreover, special hardware instructions are needed for reading and writing to this memory area.

In general the VMCS data are organised as follows[5]:

- **Guest-state area** is the location in which the guest processor's state (i.e. control registers, debug registers, segment selectors, instruction pointer, but also activity and interruptibility state) is saved before returning control to the hypervisor and restored upon VM entry.
- **Host-state area** is the location where the processor state of the host is loaded from, upon VM exit
- **VM-execution control fields** determine the causes of VM exits and limit processor behaviour when the guest is executing, in VMX non-root mode
- **VM-entry control fields** govern the behaviour of VM entries by specifying the list of MSR to be loaded or determine event injection by specifying the type of interrupt, length of instruction etc.

[4] Within a virtual machine, time is shared with the hypervisor and a number of virtual machines. Each virtual machine can be preempted at any time, even when interrupt sources are disabled. This is possible because in reality only virtual interrupts are disabled. This preemption can cause a desynchronisation between virtual time and real time.

[5] This layout is specific to the Intel-VT architecture

Instruction	Opcode	Description
VMXON	0xF30FC7	Enter VMX Operation
VMXOFF	0x0F01C4	Leave VMX Operation
VMCALL	0x0F01C1	Call to VM Monitor
VMLAUNCH	0x0F01C2	Launch Virtual Machine
VMRESUME	0x0F01C3	Resume Virtual Machine
VMPTRLD	0x0FC7	Load Pointer to Virtual-Machine Control Structure
VMPTRST	0x0FC7	Store Pointer to Virtual-Machine Control Structure
VMREAD	0x0F78	Read Field from Virtual-Machine Control Structure
VMWRITE	0x0F79	Write Field to Virtual-Machine Control Structure
VMCLEAR	0x660FC7	Clear Virtual-Machine Control Structure

Table 2.1: Intel VT-x hardware-supported virtualisation instruction set

- **VM-exit information fields** provide basic information about VM exits such as the exit reason to be handled accordingly by the hypervisor

The hypervisor is the only component that can shut down the virtualisation machinery and leave *VMX mode* by calling the $VMXOFF$ instruction. After such an event, the processor will operate with the standard instruction set.

As mentioned before, when the guest operating system is executing and the processor is in *VMX non-root* mode, several events can lead to control being returned to the hypervisor. These events are treated like faults. As for any type of fault, the instruction that caused it is not executed, the processor state is not updated and an action is taken by a fault handler, depending on a flag that determines the fault reason.

It should be clear that in a virtualisation setting, the fault handler is represented by code running in hypervisor space. Therefore the hypervisor is responsible for the execution of additional code on behalf of the guest and for updating the guest's processor state.

A list of instructions that cause VM exit when the processor is in *VMX non-root* mode is reported in Table 2.2. Other events that are handled with the same trapping mechanism are exceptions, external interrupts (otherwise served by the guest Interrupt Descriptor Table), non-maskable (NMI) and system-management (SMI) interrupts and task switches. For a more detailed description about how to handle VM exits and other architecture specific features of virtualisation-enabled processors we encourage the reader to examine the architecture developer's manual, usually provided by the hardware vendor.

Instruction	**Opcode**	**Description**
`Unconditionally VMExit`		
CPUID	0x0A20F	Returns processor type and features
INVD	0x0F08	Flushes CPU internal cache
MOV from CR3		Move from Control Register 3
VM* insn		All instructions in the VM extended instruction set
`Conditionally VMExit`		
CLTS	0x0F01C2	Clear Task-Switched Flag in CR0
HLT	0x0F01C3	Halt
IN,INS*		Input from Port to String
OUT,OUTS*		Output String to Port
INVLPG	0x0F017	Invalidate TLB Entry
LMSW	0x0F016	Load Machine Status Word
MONITOR	0x0F01C8	Set Up Monitor Address
MOV from CR8		Move from Control Register 8
MOV to CR0/CR3/CR4/CR8		Move to Control Register 0,3,4,8
MOV DR	0x660FC7	Move to Debug Register
MWAIT	0x0F01C9	Monitor Wait
PAUSE	0xF390	Spin Loop Hint
RDMSR	0x0F32	Read from Model Specific Register
RDPMC	0x0F33	Read Performance-Monitoring Counters
RDTSC	0x0F31	Read Time-Stamp Counter
RSM	0x0FAA	Resume from System Management Mode
WRMSR	0x0F30	Write to Model Specific Register

Table 2.2: List of privileged instructions that cause a VMExit event

2.1.1 Performance

The high complexity of the trapping mechanism has an impact on the overall performance of the system. Even for hardware-assisted virtualisation, VM exits are expensive. If regular instructions are affected by a latency in a range from 1 to 50 clock cycles, transitions between a virtual machine and the hypervisor can take thousands of cycles each [47].

Para-virtualisation has been introduced as a way to reduce the latencies of the trapping mechanism. Para-virtualisation is a technique in which the hypervisor provides an API to the guest operating system. A guest using the API, instead of the regular trapping mechanism, will result in an increase in the overall performance due to the reduced number of traps. In fact, some mechanisms that usually need the interposition of the hypervisor could be executed directly from the guest. In work such as [48–50] it is shown how to para-virtualise device drivers in order to considerably improve I/O-intensive tasks.

Para-virtualisation, however, usually requires guest kernel code to be modified. This last requirement cannot always be fulfilled, especially when source code is not available, as is the case of proprietary operating systems. On the other hand, hardware-supported virtualisation can rely on entirely unmodified guest operating systems, something that usually involves many more VM traps and thus higher CPU overheads.

One of the main issues of memory virtualisation is that in order to guarantee isolation, guests cannot access physical memory directly. Therefore, in addition to virtualising the processor unit, memory virtualisation is also required. This is a critical component that influences the performance impact of memory-intensive virtualised applications.

In a standalone operating system, the hardware Memory Management Unit (MMU) is used to map logical page addresses (LPA) to the physical page addresses (PPA). To achieve faster lookups, the Translation Lookaside Buffer (TLB) caches the most recently used $LPA \rightarrow PPA$ mappings, for future access.

This mechanism holds in a virtualised environment, but an additional layer is necessary in order to map a PPA to a machine page address (MPA). The two-level translation mechanism is illustrated in Figure 2.3.

Prior to the advent of hardware-supported virtualisation, the hypervisor had to maintain $PPA \rightarrow MPA$ mappings and had to store $LPA \rightarrow MPA$ mappings in *shadow page tables*[6]. Faster lookups could be achieved by caching $LPA \rightarrow MPA$ mappings into the TLB. Unfortunately, whenever the guest re-mapped its memory

[6]Shadow paging is a memory protection mechanism performed by the hypervisor to isolate guests' memory space and keep guest memory accesses updated. Specifically the hypervisor keeps the real $LPA \rightarrow MPA$ mapping updated in order to maintain a representation of the page tables that the guest thinks it is using. The update occurs whenever a page faut is generated within the guest and handled by the hypervisor. Due to the usually high number of page faults, the aforementioned mechanism is affected by consistent overhead.

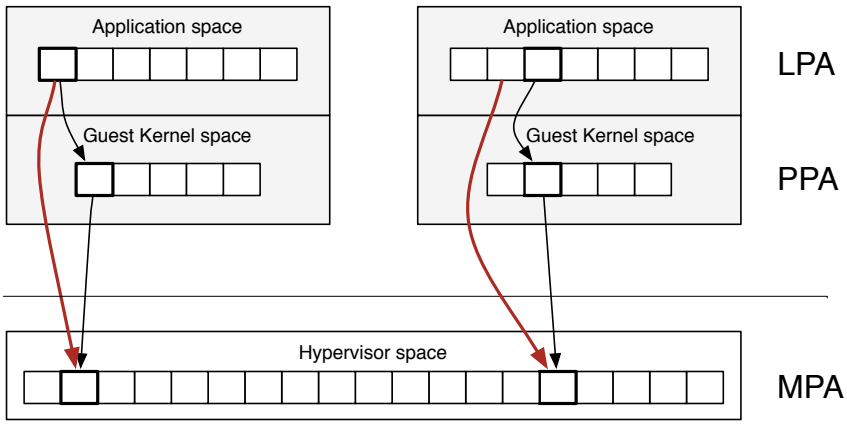

Figure 2.3: Schema of two-level translation from LPA (Logical Page Address) to PPA (Physical Page Address) and MPA (Machine Page Address). Red arrows indicate the mappings from guest to host that the hypervisor must keep synchronised in its shadow pages.

addresses, the hypervisor had to keep the shadow pages synchronised. This task was recognised as the factor responsible for most of the performance impact of the technology.

Lately, vendors like AMD and Intel included hardware support for memory virtualisation, called AMD Nested Page Tables (NPT) and Intel Extended Page Tables (EPT) respectively.
In this second scenario (Figure 2.4), any access to a logical page from within the guest triggers the composite translation of both $LPA \rightarrow PPA$ and $PPA \rightarrow MPA$. Therefore no shadow pages are needed and data will be kept synchronised without additional overhead. Clearly, the cost of a page walk needed for the double translation is slightly higher with respect to the one performed with traditional page tables[7].

The non-negligible performance impact introduced by the technology should always be taken into consideration for correctly designing infrastructures that rely heavily on virtualisation.

A mechanism that allows the guest operating system to return control to the hypervisor synchronously is referred to as *hypercall*. A hypercall is the equivalent of a system call in traditional operating systems. For instance, in order to invoke

[7]The use of large pages when NPT/EPT is enabled reduces the overall impact inflicted by the higher cost of page walks from 50% to 70% (depending on the type of benchmark), as it has been measured in [51, 52]

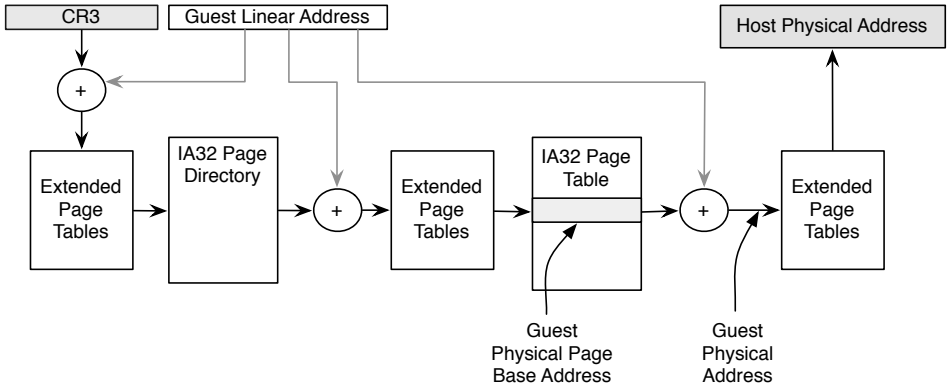

Figure 2.4: Intel's Extended Page Table hardware-supported translation mechanism to map guest addresses to host physical addresses without shadow paging. An equivalent method has been developed by AMD, with the name of NPT (Nested Page Tables)

a system call on UNIX systems, the program in user space pushes the value of the system call in register EAX and then raises an interrupt. A sequence of instructions that explain the mechanism is shown in Listing 2.1

Listing 2.1: A simple system call in traditional UNIX

```
mov eax, 1
push eax
int 80h
```

When interrupt *80h* is raised the kernel interrupt handler will read the value of register EAX, which in turn jumps to the handler of the system call (in the example above it will be system call 1) and executes it with the parameters popped from the stack. The mechanism of hypercalls is very similar to the one of a system call. The main difference is that an interrupt different from *80h* is raised. The interrupt number depends on the hypervisor's design. Hypercalls are commonly used in *para-virtualised* systems in which execution jumps directly from the application (Ring 3) to the hypervisor, which then passes control to the guest kernel (Ring 0).

However, hypercalls are also used in systems that are not hypervisor aware, whenever it is required to return control to the hypervisor explicitely. In fact, hypercalls are handled by the hypervisor synchronously, which means that execution of the guest operating system will be paused until the handler terminates. Despite the performance penalty of the hypercall mechanism, it revealed to be extremely useful in several cases presented throughout this work.

2.2 Benefits and drawbacks

The numerous advantages introduced by virtualisation are influencing several aspects of designing modern computing infrastructure. Hardware engineers are improving virtualisation-supported processors at a constant pace, reducing what used to be a considerable gap between the performance of software running on real hardware and those in virtualisation environments.

Despite ample room for the improvement of the latency of complex virtualisation-related instructions, server consolidation, one of the most frequent applications of virtualisation technology in the industry, is contributing to a much more efficient usage of computing resources. Server consolidation addresses the reduction of existing infrastructure [53]. Therefore, significant reduction of energy consumption is a benefit that comes as a direct consequence. Moreover, less physical hardware also leads to lower management costs, such as the costs of regular wearing, faulty hardware, etc. It goes without saying that it is the reduction of these types of costs that triggered the migration to virtualised data centres in the early days and encouraged other users to follow later on.

The natural course of hardware development and the popularity of virtualisation technology promoted the replacement of traditional processors with new hardware. Today, virtualisation-supported processors are off-the-shelf components regularly installed on general purpose computers. Therefore, virtualisation-friendly solutions meet their requirements with much more ease than in the past when they could be deployed only when accompanied with special hardware.

Despite the benefits described above, which might be interpreted more as business-related advantages, a feature that is rendering virtualisation technology even more attractive from a more technical view point, is the inherent isolation of virtual machines from themselves and the hypervisor. The need for executing several guests at the same time and preventing any type of interference are requirements that cannot be fulfilled without isolation.

As explained in Section 2.1, it should be clear that hardware-support is not only beneficial to the overall performance impact[8] but also to security. In a hardware-assisted virtualisation framework it is substantially hard - sometimes not practically feasible - to break the isolation constraint from within a virtual machine. This particular feature, which comes by design, is broadening the horizon of those researchers who provide security solutions based on sandboxes or similar isolation environments.

Apart from the use of virtualisation as a way to host different operating systems on a single physical machine, virtualisation is currently also being used to provide greater security guarantees for operating system kernels. This specific scenario

[8]The benefit becomes consistent when hardware-supported virtualisation is compared to software emulation. But compared to a native system, a virtualisation solution continues to have non-negligible impact in the range between 10% and 30%, depending on the type of benchmark [54].

will be described extensively in Chapter 3 and Chapter 4.

However, moving the direct control of physical hardware to the lower level of the hypervisor, and delegating to this only component all critical operations with the purpose of arbitrating the execution of the guests, might lead to security issues. Obviously, a bug within hypervisor code might affect the entire virtualisation platform. A general strategy to avert such an issue involves keeping the hypervisor's code size as small as possible in order to increase the chances to discover bugs and provide fixes at the earliest.

Formal verification techniques can be considered, in order to prevent possible faults or unexpected behaviour from hypervisor space. However, these solutions are feasible only under simple assumptions, most of the times when hypervisor code has been written with verification in mind. The challenging part of the verification task becomes more evident since hypervisors are usually written in unsafe languages (such as C and assembly code for performance reasons) with an easily circumvented type system and explicit memory allocation. For such systems, memory safety has to be verified explicitly. Moreover, the hypervisor is usually formed by code that runs concurrently and address translations occur asynchrounously and non-atomically [55, 56]. Last but not least, virtualisation strategies that can fully exploit the hardware to execute a multitude of guest operating systems with high performance, require the hypervisor to execute directly on physical hardware. This can increase the size of the hypervisor (a large code base is usually represented by device drivers) and make verification a non-tractable task. However, there have been attempts to formally verify hypervisors with a small codebase, a task that is more accessible than verifying commodity operating systems [56, 57].

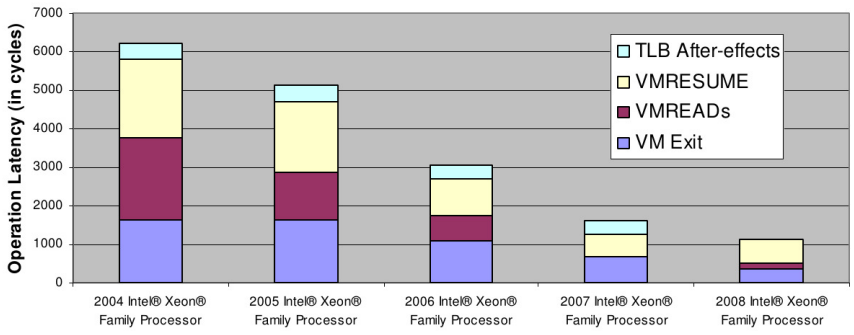

Figure 2.5: Latency of VM exit, VMREAD, VMRESUME instructions across Intel processors

Despite the performance improvements claimed by works like [58, 59] and by

Intel itself, as shown in Figure 2.5, the gap of the performance between native and virtualised operating systems remains consistent. It is clear that the performance impact is mainly caused by the latency of virtualisation-based instructions, which is mainly architecture dependent. The frequency of trapped events from the guest to the hypervisor is also important and it has been measured to have a considerable impact on the overall performance of a virtualised system, specially affecting Input/Output intensive tasks. In [59], this frequency has been minimised using software techniques. In the same work it is claimed that the overall transition costs are reduced up to 50%. However, the aforementioned latency seems to be improvable up to a lower bound imposed by the technology.

2.3 Drawbacks of virtualisation: a case study

The types of countermeasures proposed in this work have been designed to work together with the operating system. The very nature of virtualisation technology relies on the regular mechanisms of guest operating system kernel such as task switching, changes to control registers and other specific events that trigger the hypervisor's intervention. According to the benchmarks of each virtualisation-based countermeasure explained in this work, their performance impact is, in general, relatively low. One of the keys of this limited performance impact consists in the fact that the countermeasure's code is executed at a specific time, taking advantage of the regular delays imposed by virtualisation technology. Setting the performance penalty of the countermeasure aside, for instance, the VM exit - VM entry mechanism contributes to maintain the overall performance impact close to the lower bound introduced by the technology.

As we show later in this section, when such a strategy is not applied, the performance impact of the countermeasure will likely be summed up as the performance impact of the virtualisation technology. We observe that if hypervisor's intervention is triggered whenever required, without synchronizing with the operating system, the overall performance impact will be dramatically increased. One such scenario occurred when we examined the possibility of using virtualisation technology to implement a countermeasure that protects against buffer overflows, specifically *return-address attacks*.

Despite a plethora of available research in the field, the buffer overflow is still one of the most insidious vulnerabilities affecting software nowadays. According to the NIST's National Vulnerability Database [60], 587 (10% of all reported vulnerabilities) buffer overflow vulnerabilities were reported in 2008. Almost 90% of those vulnerabilities had a high severity rating. A buffer overflow is the result of stuffing more data into a buffer than it can handle and may allow an attacker to control the execution flow of the attacked program. In a *return-address attack* the attacker exploits a buffer overflow vulnerability to change the return address of a function. It is often performed together with code injection through *shellcode*.

The execution of arbitrary code is what results in the high severity rating of most of the reported vulnerabilities. These types of attacks are known as stack-based buffer overflow attacks.

A typical function that is vulnerable to a buffer overflow is given in Listing 2.2

Listing 2.2: A function that is vulnerable to buffer overflow

```
char* vuln_foo(char *msg) {
    char *p;
    char buffer[30];
    p=buffer;
    strcpy(p, msg);
}
```

The compiler translates this code and provides a standard prologue that saves the frame pointer (FP) to the stack and allocates space for the local variables and a standard epilogue that restores the saved frame and stack pointer (SP), as shown in Listing 2.3.

Listing 2.3: The standard prologue and epilogue of vuln_foo()

```
prologue:
    pushl %ebp
    mov %esp, %ebp
    // local variables

   (vuln_foo body)

epilogue:
    leave  // copies %ebp into %esp
           // restores %ebp from stack
    ret
            // jump to address on
            // top of the stack
```

Our approach to prevent the exploitation of such a vulnerability consists in extending the architecture with few extra instructions, which are emulated by the hypervisor.

These instructions are designed to save and restore the return address from a protected memory area.

Listing 2.4: Instrumented assembly code of vuln_foo()

```
main:
```

```
    call init_callretx
    ...

vuln_foo:
 prologue:
     pushl %ebp
     mov %esp , %ebp
    // local variables
    callx

          (vuln_foo body)

 epilogue:
     retx
     leave  // copies %ebp into %esp
           // restores %ebp from stack
     ret     // jump to address on
             // top of the stack
```

At the beginning of every program, from its main function, a 4KB page is created and protected via the *mprotect* system call. For each function to be protected the two hardware instructions are called at a specific time in order to prevent an attacker to tamper with the return address of the function.

In the proof-of-concept we provide the hardware[9] instruction

- `callx`

  has been added before the call instruction, that will execute the body of the function. It will save the return address onto the protected memory page

  while instruction

- `retx`

  has been added right before the assembler `leave` instruction in the function's epilogue. It will restore the return address from the protected memory page onto the stack.

Return addresses of nested functions are stored at higher addresses within the page with the aid of a counter that permits to handle return addresses in a *Last-In-First-Out* order. This order will be preserved until the maximum number of nested functions is reached. Clearly this number depends on the size of the *mprotected* page, which is 4KB in our implementation. Since the x86 architecture handles

[9]This is hardware instruction with respect to the program to be protected. From a more general view point the added instructions are not implemented by special purpose hardware since they will be emulated by the hypervisor.

32-bit addresses and a counter of the same size is required, our countermeasure can handle up to 1023 nested functions.

We implemented this concept in the Xen hypervisor [61] and optimized the most time consuming task of writing to the protected memory area. Our idea consists in clearing the write protection bit (WP) in Control Register 0 (CR0) [10] before any write operation to a read-only memory and then set it again. The Xen hypervisor, which runs in supervisor mode, needs to be able to write to a read-only page from the user space memory. By unsetting the WP in CR0, the memory management unit (MMU) does not check whether the page is read-only or not, allowing the new instruction to write directly. This strategy leads to a performance impact that is dramatically reduced when compared to the usual mechanism that rely on the MMU[11].

Since we need to save the return address from the current stack to memory (callx) and from memory back to the stack (retx), we need two functions that move data from one space to the other. As in a regular Linux kernel the `copy_to_user` and `copy_from_user` functions perform this task. A counter is needed to handle nested functions. This variable is incremented in `callx` and copied to the read-only memory and decremented in `retx` and copied back to the stack, in order to preserve a LIFO order.

A check if the return address has been altered may be performed before overwriting it with the saved value. However this will lead to a higher overhead in the overall test result.

2.3.1 Evaluation

To test the performance overhead we ran several integer benchmarks from the suite SPEC CPU2000 [62]. We collected results running programs instrumented with the code that implements the countermeasure and without.
All tests were run on a single machine with the hardware specifications reported in Table 3.2. As mentioned before, the hypervisor used for our implementation is Xen 3.3.0. The GCC 4.2.3 compiler has been modified to instrument assembler code with the new instructions.

[10] CR0 has control flags that modify the basic operation of the processor. WP bit is normally set to prevent supervisor from writing into read-only memory.

[11] Although Xen has the necessary code to capture illegal instructions, some setup is required to handle the new instructions' opcodes. New code that checks if the opcode we want to emulate occurred has been added. When the new instruction's opcode occurs, the user space program context (`ctxt` structure) is updated. This is required before calling `x86_emulate` which will take the context structure as parameter and performs the emulation. Before calling this function, the WP bit of `CR0` must be unset. Thus when `x86_emulate` is called, all writes to memory can happen without any fault. New code to emulate the `callx` and `retx` instructions in the hypervisor has been added to `x86_emulate.c`.

Program	Base r/t(s)	Instr. r/t(s)	Overhead
164.gzip	223	3203	+1336.32%
175.vpr	372	2892	+677.42%
176.gcc	225	2191	+873.78%
181.mcf	640	3849	+501.41%
186.crafty	114	3676	+3124.56%
256.bzip2	307	5161	+1581.11%
300.twolf	717	4007	+458.86%

Table 2.3: SPEC CPU2000 benchmark results of Xen implementation

Despite the aforementioned optimization strategy, the benchmarks show that this implementation experiences a slow-down between 5x and 30x, depending on the number of functions to be protected in the program. This latency is definitely not acceptable to consider this type of countermeasure in production systems (Table 2.3.1).

Moreover, our countermeasure does not detect if a buffer overflow has occurred since it overwrites the return address of the protected function, without checking if it has been altered.
Our implementation allows the protected function to recover its caller's return address and continue its normal execution flow. We are aware that there are cases in which it is better to terminate the attacked program and log that a buffer overflow has occurred. Checking that the protected return address has been tainted on the stack might be implemented with more overhead.

2.3.2 Discussion

The main reason for which virtualisation technology inflicts such a significant overhead is given by the high number of context switches from the guest to the hypervisor and back. These transitions are needed for the hypervisor to perform the emulation of the special instructions *callx* and *retx*. While, at operating system level, these transitions occur a limited number of times (ie. I/O operations, privileged instructions like the ones reported in Table 2.1), in this specific case they will occur an additional number of times that depends on how many functions the program is formed by (or equivalently how many times the function to be protected is called).

Therefore, while our implementation is technically feasible and even faster than RAD [63], a compiler-based countermeasure that provides an equivalent protection, we conclude that it does not have a realistic chance of deployment, except in higher security environments.

2.4 Rethinking security

One of the immediate effects of the advent of virtualisation into marketplace is the new way of thinking about security. The desire to exploit this new environment and deal with challenging and unsolved problems - mainly related to operating system kernel security - is so intense that several security solutions promptly appeared in the literature [64–67]. However, protecting operating system kernels from being compromised is challenging and still an open problem. Virtualisation offers viable ways to mitigate these types of threats. However, due to the significant overhead introduced by the technology, any security solution needs to be designed with performance in mind.

We provide a realistic model of attack to operating system kernels and a strategy to detect attacks of this type when the operating system has been virtualised, in Chapter 3.

Still in the field of kernel security, a more general framework that enforces the execution of monitoring code within a virtualised operating system is described in Chapter 4. It will be shown that the performance impact of our solutions is negligible and limited by the lower bound of the technology.

The role of virtualisation technology to improve the security of web browsers and applications delivered on-demand is described in Chapter 5.

Chapter 3

Hypervisor-based invariance-enforcing framework

Prediction is very difficult, especially if it's about the future.

Niels Bohr

Operating systems consist of trusted code that executes directly on top of a host's hardware. This code usually provides several functionalities [68] such as

- abstraction, essential to develop programs that use the system functions offered by the operating system
- arbitration, to allocate resources to more than one concurrently running program in a collision-free fashion and
- isolation, that prevents one program from addressing the memory space of another

Due to their prominent position, operating systems become a common target of attackers who regularly try to circumvent their protection mechanisms and modify them to their advantage. In the past, a malicious program that allowed a user to elevate his access and become a system administrator, or `root`, was called a `rootkit`. Today the meaning of rootkits has changed and is used to describe software that hides the attacker's presence from the legitimate system's administrator. Kernel-mode rootkits target the core of an operating system. It

goes without saying that they are the hardest to detect and remove. Such rootkits appear very often in the form of device drivers (Windows platform) or Loadable Kernel Module (Linux kernel). In other cases, a kernel-mode rootkit may be introduced by a software bug in the kernel and triggered by a malicious or a benign but-exploitable process. Regardless of the way the rootkit is introduced, the result is malicious code running with operating system privileges which can add and execute additional code or modify existent kernel code. The activities resulting from a successful attack can range from spamming and key-logging to stealing private user-data and disabling security software running on the host. In the past, rootkits have also been used to turn their targets into nodes of a botnet as with Storm Worm [7] or to perform massive bank frauds [8]. An even more subtle type of rootkits does not introduce new code at all, but rather makes use of existing fragments of operating system code - that will be considered trusted by any active monitor - to fabricate their malicious functionality. The execution of these fragments in a specific order, chosen by the attacker has give birth to an entirely new way of compromising the kernel called `return-oriented rookits` [69].

Generally speaking, changing the control flow of the operating system kernel involves either changing the content of specific kernel objects such as existing fragments of code with new code or overwriting kernel function pointers. Several approaches that mitigate rootkits have been developed by security researchers who consider modified kernel-data structures an evidence of attack. Unfortunately, many of these approaches are affected by substantial overhead [70, 71] or miss a fundamental security requirement such as isolation [72]. Isolation is needed to prevent a countermeasure in the target system from being disabled or crippled by a potential attack. When malicious code is running at the same privilege level as the operating system kernel, no isolation mechanism is in place and the attacker's code is capable of accessing arbitrary memory locations or hardware resources. The property that makes virtualisation particularly attractive in the field of security research is that isolation is guaranteed by the current virtualisation-enabled hardware.

Other countermeasures have been presented in which operating system kernels are protected against rootkits by executing only validated code [66, 71, 73]. But the aforementioned type of rootkit [69] that doesn't introduce new kernel code and basically re-uses fragments of validated code can bypass such countermeasures. In [74] a countermeasure to detect changes of the kernel's control flow graph is presented; Anh et al. [75] uses virtualisation technology and emulation to perform malware analysis and [76] protects kernel function pointers. Another interesting work is [77] which gives more attention to kernel rootkit profiling and reveals key aspects of the rootkit behaviour by the analysis of compromised kernel objects. Determining which kernel objects are modified by a rootkit not only provides an overview of the damage inflicted on the target but is also an important step to design and implement systems to detect and prevent rootkits. In this chapter

we present an invariance-enforcing framework that mitigates rootkits in common operating system kernels. Our protection system runs inside a hypervisor and protects operating systems that have been virtualised. Some practical constraints that led to the relaxation of the original problem, impose negligible performance overhead on the virtualised system. Moreover it does not require kernel-wide changes, making it an attractive solution for production systems.

The remainder of this chapter is structured as follows. Section 3.1 describes the problem of rootkits and presents the attacker model. In Section 3.2, we present the architectural details of our countermeasure, followed by its implementation. We evaluate our prototype implementation in Section 3.3 and present its limitations in Section 3.4. We discuss related work on rootkit detection in Section 3.5 and conclude in Section 3.6.

3.1 Motivation

In this section we describe common rootkit technology and we also present the model of the attacker that our system can detect and neutralise.

3.1.1 Rootkits

Rootkits are pieces of software that attackers deploy in order to hide their presence from a system. Rootkits can be classified according to the target and consequently to the privilege-level which they require to operate. The two most common rootkit classes are: a) user-mode and b) kernel-mode.

User-mode rootkits run in the user-space of a system without the need of tampering with the kernel. In the Microsoft Windows platform, user-mode rootkits commonly modify the loaded copies of the Dynamic Link Libraries (DLL) that each application loads in its address space [78]. More specifically, an attacker can modify function pointers of specific Windows APIs and execute their own code before and/or after the execution of the legitimate API call. In Linux, user-mode rootkits hide themselves mainly by changing standard Linux utilities, such as `ps` and `ls`. Depending on the privileges of the executing user, the rootkit can either modify the default executables or modify the user's profile in a way that their executables will be called instead of the system ones (e.g. by changing the `PATH` variable in the Bash shell).

Kernel-mode rootkits run in the kernel-space of an operating system and are thus much stronger and much more capable. The downside, from an attacker's perspective, is that the user must have enough privileges to introduce new code in the kernel-space of each operating system. In Windows, kernel-mode rootkits are loaded as kernel extensions or device-drivers and target locations such as the `call gate` for interrupt handling or the System Service Descriptor Table (SSDT). The rootkits change these addresses so that their code can be executed

Rootkit	**Description**
Adore, afhrm, Rkit, Rial, kbd, All-root, THC, heroin, Synapsis, itf, kis	Modify system call table
SuckIT	Modify interrupt handler
Adore-ng	Hijack function pointers of `fork()`, `write()`, `open()`, `close()`, `stat64()`, `lstat64()` and `getdents64()`
Knark	Add hooks to /proc file system

Table 3.1: Hooking methods of common Linux rootkits

before specific system calls. This capability is usually referred to as `hooking`. In Linux, rootkits can be loaded either as a Loadable Kernel Module (LKM), the equivalent of Windows device-drivers, or written directly in the memory of the kernel through device files that provide access to system's memory such as `/dev/mem` and `/dev/kmem` [79]. These rootkits target kernel-data structures in the same way that their Windows-counterparts do. Although we focus on Linux kernel-mode rootkits, the concepts we introduce here apply equally well to Windows kernel-mode rootkits. An empirical observation is that kernel-mode rootkits need to corrupt specific kernel objects, in order to execute their own code and add malicious functionality to the victim kernel. Studies of common rootkits [74, 80] show that most dangerous and insidious rootkits change function pointers in the system call table, interrupt descriptor table or in the file system, to point to malicious code. The attack is triggered by calling the relative system call from user space or by handling an exception or, in general, by calling the function whose function pointer has been compromised. We report a list of rootkits which compromise the target kernel in Table 3.1. Due to the high number of variant rootkits that attackers design in order to evade current security mechanisms, the following list is not meant to be exhaustive.

3.1.2 Threat Model

In order to describe the environment in which our mitigation technique will operate, we make some assumptions about the scenario that an attacker might create. We assume that the operating system to be protected and which is being attacked is virtualised. This means that it runs on top of a hypervisor which executes at a higher privilege level than the operating system itself. Virtualisation-enabled hardware guarantees isolation. Thus we assume that the guest operating system cannot access the memory or code of the hypervisor. Moreover, no physical access to the host machine is granted. We have designed a system that detects the presence of a rootkit after it has been deployed, a fact that allows our model to

include all possible ways of introducing a rootkit in a system.

A rootkit can be introduced either by:

- A privileged user loading the rootkit as a Loadable Kernel Module
- A privileged user loading the rootkit by directly overwriting memory parts through the `/dev/` memory interfaces
- An unprivileged user exploiting a vulnerability in the kernel of the running operating system which will allow him to inject and execute arbitrary code

Finally, our system does not rely on secrecy. Therefore, our threat model includes the attacker being aware of the protection system.

3.2 Approach

In this section we describe the architectural details of the countermeasure and explain our choices in the implementation of our proof of concept. By studying the most common rootkits and their hooking techniques one can realize that they share at least one common characteristic. In order to achieve execution of their malicious code, rootkits overwrite locations in kernel memory which are used to dictate, at some point, the control-flow inside the kernel. Most of these locations are very specific (see Table 3.1) and their values are normally invariant, i.e. they don't change over the normal execution of the kernel. Since these objects are normally invariant, any sign of variance can be used to detect the presence of rootkits. We use the terminology of `critical kernel objects` to name objects that are essential to change the control-flow of the kernel and thus likely to be used by an attacker.

Given a list of invariant critical kernel objects, our approach consists of periodically checking them for signs of variance. When our countermeasure detects that the contents of an invariant critical kernel object have been modified, it will report an ongoing attack. Such a strategy might be affected by false positives, when variable objects are erroneously considered invariants. Considering the modification of invariant critical kernel objects as evidence of rootkit attack is not entirely novel to the literature. These types of objects have been identified in several contributions, such as [76, 81–83]. The need for special hardware and the performance overhead of such contributions motivated our research in this area.

The methods to detect invariance differ depending on the type of critical kernel object and are summarized in the following list:

1. Static kernel objects (type 1) at addresses that are hardcoded and not dependent on kernel compilation

2. Static kernel objects (type 2) dependent on kernel compilation (e.g., provided by `/boot/System.map` in a regular Linux kernel)

3. Dynamic kernel objects (type 3) allocated on the heap by kmalloc, vmalloc and other kernel-specific memory allocation functions

Identifying and protecting static kernel objects (type 1 and type 2) is straightforward. During the installation of the operating system to be monitored, a virtual machine installer would know in advance whether the guest is of Windows or Unix type. This is the minimal and sufficient information required to detect kernel objects whose addresses have been hardcoded (type 1). Moreover, the Linux operating system, that we used for our prototype, provides `System.map`, where compilation-dependent addresses of critical kernel objects are stored (type 2).

In contrast, identifying dynamic kernel objects (type 3) needs much more effort since the heap changes at runtime depending on the program's execution. The identification of invariant objects of type 3 relies on the invariance detection algorithm in place. Part of our countermeasure is trusted code which operates in the guest operating system at boot time. Boot time is considered our root of trust. We are confident this to be a realistic assumption that does not affect the overall degree of security of the approach. [1]

From this point on, the system is considered to operate in an untrusted environment and a regular integrity checking of the protected objects is necessary to preserve the system's safety. Given a list of invariant kernel objects, the trusted code communicates this data (virtual address and size) of the kernel objects to observe after boot, and stores them in the guest's address space. When data of the last object have been gathered, the trusted code will raise a hypercall in order to send the collected entries to the hypervisor. The hypervisor will checksum the contents mapped at the addresses provided by the trusted code and will store their hashes in its address space, which is isolated and not accessible to the guest. The trusted code is then deactivated via a *end-of-operation* message sent by the hypervisor. No objects are accepted after the kernel has booted. In fact, at this point, an attacker who is aware of the presence of our security measure could give rise to a Denial-Of-Service attack.

It is important to point out that our system must be provided with a list of kernel objects on which it will enforce invariance. This list can be either generated by invariance detection systems like the ones described in [76, 81–83] or manually compiled by kernel and kernel-module developers (for objects of type 1 and type 2).

As already stated, the most beneficial aspect of implementing countermeasures in a separated virtual machine or within the hypervisor is the increased degree of

[1] Specifically related to the Linux kernel, boot time ends right before calling *kernel_thread* which starts *init*, the first userspace application of the kernel. At this stage the kernel is booted, initialized and all the required device drivers have been loaded.

security via isolation. Unfortunately, it often leads to higher performance overhead than the equivalent implementation in the target system. A challenging task is that of checking integrity outside of the target operating system while limiting the performance overhead. We achieve this by exploiting the regular interaction of a hypervisor and the guest operating system.

As described in Chapter 2.1, in a virtualised environment the guest's software stack runs on a logical processor in VMX non-root operation [84]. This mode differs from the ordinary operation mode because certain instructions executed by the guest kernel may cause a `VMExit` that returns control to the hypervisor.

Our countermeasure performs integrity checks every time the guest kernel writes to a control register (MOV_CR* event) which is a privileged instruction of the type listed in Table 2.2 that in-turn causes a `VMExit`. Figure 3.1 depicts a high-level view of the approach described above. Trapping this type of event is strategic because if virtual addressing is enabled (as in the case of modern commodity operating systems), the upper 20 bits of control register 3 (*CR3*) become the page directory base register (PDBR). This register is fundamental to locate the page directory and the page tables for the current task. Whenever the guest kernel schedules a new process, a task usually referred to as `process switching`, the guest *CR3* is modified. The mechanism explained above is typical of the Intel IA-32 and higher architectures. Since other processor architectures are equipped with equivalent registers, most of our descriptions can be easily applied. We found that performing integrity checks on the MOV_CR* events is a convenient way to keep detection time and performance overhead to a minimum while guaranteeing a high level of security on protected objects. Moreover, this choice revealed to be suitable for constraint relaxations that dramatically improve performance with a small cost in terms of detection time. More details of the aforementioned relaxation are provided further in this section. We are aware of an alternative approach in which integrity-checking code running in hypervisor's space is randomly interposed to code running in the guest, without relying on the trapping mechanism. According to our experiments, ignoring modifications to guest control registers would not scale with the guest system load as our current approach does.

Another instruction that, when executed by the guest kernel, causes a VMExit and that might have been used as an alternative to the MOV_CR* instruction is INVLPG. This instruction, invalidates a single page table entry in the TLB (Intel architecture). Invalidating part of the TLB is an optimization that replaces the complete flushing when the number of entries that need to be refreshed is below a certain treshold. In the Linux kernel this does not occur often. Therefore trapping on this instruction would have resulted in very few VMExit events, giving an attacker enough time to compromise an object and set its value back to the original, without being detected. Moreover, a MOV_CR* instruction is also executed upon raising a system call from userspace. Although the number of VMExits can increase consistently, we believe that system calls, as task switches represent a

sound heuristic measure of system activity.

We developed a prototype of our countermeasure by extending BitVisor, a tiny Type-I hypervisor [85] which exploits Intel VT and AMD-V instruction sets. Our target system runs a Linux kernel with version 2.6.35. The trusted code has been implemented as a loadable kernel module for the Linux kernel [86].

What made BitVisor our choice to implement a *proof-of-concept* is not only the availability of the source code but also its memory address translation mechanism. In BitVisor, the guest operating system and the hypervisor share the same physical address space. Hence, in this specific case, the hypervisor does not need any complex mechanism to provide translations from guest to host virtual addresses. As a consequence, its size results consistently reduced, when compared to other Type-I hypervisors. The guest operating system will rely on its private guest page table to perform translations from virtual to physical addresses. The only drawback of this software architecture is that the hypervisor can not directly use the guest page table, otherwise the guest could access hypervisor's memory regions by mapping physical addresses to the page table. BitVisor uses shadow paging to verify page table entries right before they are used by the processor, to prevent such attacks.

Although hypervisor's and guest's memory is shared on the same physical address space, isolation is still guaranteed by hiding hypervisor's memory regions from the guest via BIOS functions (function e820h) that fake that the memory region is reserved. Being on the line to this memory architecture, translations of guest virtual addresses to host virtual addresses are performed by the cooperation of the trusted module and the hypervisor, as explained later in this section.

Our system detects illegal modifications of invariant critical kernel objects in three distinct phases which are described below.

Communicating phase The trusted module executes in the guest's address space and communicates the addresses and sizes of critical kernel objects to be protected. In order to test and benchmark our system in a realistic way, we created an artificial list of critical kernel objects by allocating synthetic kernel data. For each critical object the trusted module in the guest will retrieve its physical address by calling $\_\_pa(virtual\_address)$, a macro of the Linux kernel. If the kernel object is stored in one physical frame the trusted module will immediately collect the start address and the size. Otherwise, if the kernel object is stored on more than one physical page frame the trusted module will store the relative list of physical addresses. For some critical kernel objects it is possible to communicate their original content for the reasons explained in paragraph `Repairing phase`. A memory area shared with the hypervisor is allocated for storing the object-related data and control flags. When the virtual addresses of all objects have been translated, a hypercall is raised which signals the hypervisor to start the integrity checking.

Detection phase In order to detect changes the hypervisor needs to access

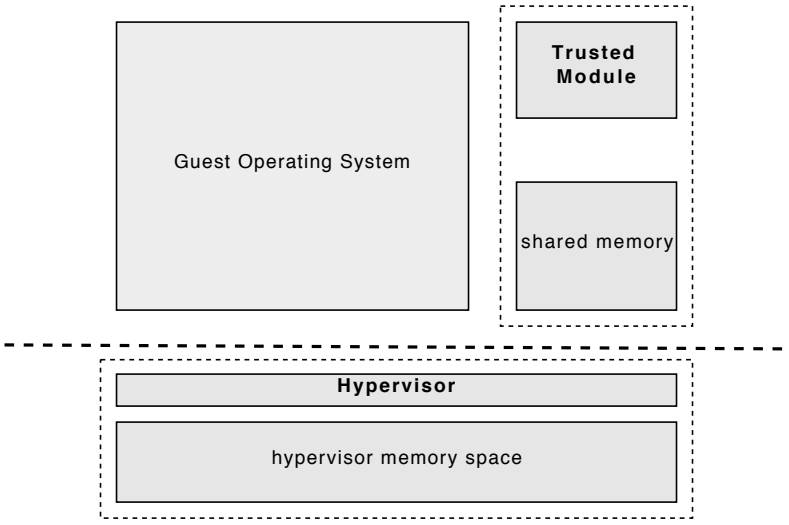

Figure 3.1: High level view of trusted module-hypervisor interaction

the contents at the guest physical addresses collected by the trusted module. This is achieved by mapping the physical address and size of each object in its private memory in order to compute the signature of its actual contents. When all objects have been checksummed an end-of-operation flag is set in the memory area shared with the trusted module, which in turn will be unloaded. The checksum is performed by a procedure which implements the MD5 algorithm [87]. This cryptographic hash function provides the integrity guarantees needed for our purposes. While stronger hash functions exist, we believe that the security and collision rate provided by MD5 adequately protect our approach from mimicry attacks.

Repairing phase When the hypervisor detects that the signature of a protected object is different from the one computed the first time, two different behaviours are allowed:

a) the system will report an ongoing attack or

b) the system attempts to restore the contents of the compromised object if a copy has been provided by the trusted module.

Since the hypervisor and the guest share the same physical address space, the hypervisor can restore the original content by mapping the physical address of the compromised object in its virtual space. The untampered value is then copied and control returns to the guest. The restoration of modified critical data structures means that, while the rootkit's code is still present in the address space of the kernel, it is no longer reachable by the kernel control-flow and thus it is neutralised. One drawback of the repairing phase is data inconsistency: if a process in the guest is using the data structure that is being repaired by the hypervisor, on the next

Component	Description
Processor	Intel Core 2 Duo
Model	E6750 @ 2.66GHz
Speed [MHz]	2667.000
Virtualisation	enabled Intel VT
Memory	4GB
NPT/EPT	not enabled
OS	Linux 2.6.36

Table 3.2: Testing machine specification

VM Entry an inconsistency might be raised. However, switching from VMX-root to VMX-non-root causes a flush of the Translation Lookaside Buffer (TLB). Any code in the guest that was using the compromised object will perform the address translation and memory load again and will thus load the restored value.

As previously mentioned, whenever task switching, the CR3 register's contents are changed. A trap of the MOV_CR* event occurs and integrity-checking is performed. This checking occurs outside of the guest operating system and thus cannot be influenced nor delayed by any other event that occurs in the guest.

Since the number of kernel objects is usually high, a relaxation of the former problem consists in performing the integrity checking of only a subset of objects. Therefore, the overall list of objects will be checked in number of MOV_CR* events that depends on the total number of objects to be protected and the number of objects checked each time. Finally, control is returned to the guest kernel and another subset of critical kernel objects will be checked at the next MOV_CR* event. While considerably improving the performance overhead, this relaxation obviously comes at a cost in terms of security and detection time. We do believe however, that the resulting detection ability of our method remains strong, a belief which we explore further in Section 3.3.

3.3 Evaluation

In order to evaluate whether our approach would detect a real rootkit we installed a minimal rootkit [88] which hijacks a system-call entry, specifically the `setuid` systemcall, from the system-call table. Although the aforementioned rootkit is a simplified version of real-world kernel rootkits, it belongs to the family of rootkits that target system calls and other operating system invariant function pointers.

Whenever the `setuid` system call is invoked with the number `31337` as an argument, the rootkit locates the kernel structure for the calling process and elevates its permissions to `root`. The way of hijacking entries in the system-call table is very common among rootkits (see Table 3.1) since it provides the rootkit a convenient and reliable control of sensitive system calls.

The critical kernel object that the rootkit modifies is the system-call table

which normally remains invariant throughout the lifetime of a specific kernel version. The Linux kernel developers have actually placed this table in read-only memory, however the rootkit circumvents this by remapping the underlining physical memory to new virtual memory pages with write permissions.

Before installing the rootkit, we gave as input to our trusted module, the address of the invariant system-call table and its size. In a real-life scenario, our mitigation would work in concert with an external source that detects the memory locations of the invariant critical kernel-objects and their size. This source can either be automatic invariance-discovering systems or kernel programmers who wish to protect their data structures from malicious modifications. Once our system was booted we loaded the rootkit in the running kernel. When the next `MOV` to control-register occurred, the system, trapped into the hypervisor, was capable of detecting changes on the invariant system-call table. After reporting the attack, the system repaired the system-call table by restoring the system-call entry with the original memory address. In this specific case although the rootkit's code was still loaded in kernel-memory it was no longer reachable by any statement and thus inactive.

A list of synthetic kernel objects has been created to run a set of benchmarking utilities and evaluate the performance impact. The size of each of these objects has been chosen according to *slabtop*, a Linux utility which displays kernel slab cache information. Approximately 15,000 kernel objects are allocated during system's lifetime, 75% of which smaller than 128 bytes. Moreover, these numbers are never exceeded in other detection systems. The trusted module has been instrumented to create a set of objects with the aforementioned characteristics.

Checking the integrity of 15,000 kernel objects at a time, and only then returning control to the guest operating system is not a practical solution and it is affected by a considerable overhead that dramatically penalises the usability of the overall system. Moreover this solution would not scale with the number of objects that can arbitrarily grow if the goal is to minimise the attack surface. Therefore at each `VMExit` we check each time a different subset of the object set. This parameter is configurable and its value depends on the priorities of each installation (performance versus detection time). In our proof of concept the hypervisor will check the integrity of 100 objects of 128 bytes each every time a MOV_CR event is trapped. Needless to say, 150 `VMExit` events must occur in order to check the integrity of the entire list. The choice of the aforementioned numbers of objects to be protected is based on empirical observations that try to keep the user's experience within the guest as smooth as possible for the general tasks of every day computing.

Measuring timings in a virtualised environment differs from the usual procedure used with traditional systems. The guest's timers might be paused when the hypervisor is performing any other operation. Therefore we measure real (wall-clock) timings in the guest to compensate for any inaccuracy that might occur.

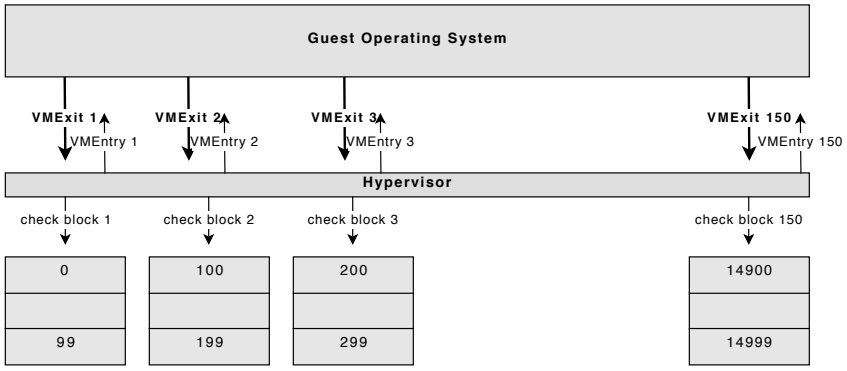

Figure 3.2: Schema of integrity checking 15000 objects in 150 VMExit trapped events

For the sake of completeness, three distinct benchmarks have been performed on our system, that measure three different aspects for a better evaluation of the performance impact. We collected results from ApacheBench [89] sending requests on a local webserver running lighttpd (Table 3.4) and from SPECINT 2000 as macrobenchmarks in order to estimate the delay perceived by the user (Table 3.6). Lastly, we collected accurate timings of microbenchmarks from lmbench (Table 4.3.2). The macrobenchmarks show that our system imposes neglibible overhead on the SPEC applications (0.005%) allowing its widespread adoption as a security mechanism in virtualised systems.

As expected, microbenchmarks show a consistent overhead on process forking. In Table 4.3.2 we do not report measurements of context switching latencies because the numbers produced by this benchmark are inaccurate [90, 91]. An improvement of local communication bandwidth is due to the slower context switching which has the side effect of slightly increasing the troughtput of file or mmap re-reading operations.

Processes - times in microsecs - smaller is better				
	open clos	slct TCP	sig inst	sig hndl
w/o counterm.	16.6	3.08	0.48	2.41
w counterm.	16.5	3.09	0.48	2.47
overhead (%)	0.6%	0.3%	0%	2.5%
	fork proc	exec proc	sh proc	
w/o counterm.	1222	4082	16.K	
w counterm.	1724	5547	18.K	
overhead	41.0%	35.8%	12.5%	
File and VM system latencies in microsecs - smaller is better				
	0K File create	0K File delete	10K File create	10K File delete
w/o counterm.	26.0	21.5	99.9	28.2
w counterm.	26.4	21.3	99.8	27.8
overhead (%)	1.53%	-0.93%	-0.1%	-1.43%
	Mmap latency	Prot fault	Page fault	
w/o counterm.	62.2K	4.355	9.32010	
w counterm.	66.5K	4.444	9.84780	
overhead (%)	6.9%	2.0%	5.5%	
Local Communication bandwidth in MB/s - bigger is better				
	TCP	File reread	Mmap reread	Bcopy(libc)
w/o counterm.	2401	313.0	4838.1	617.5
w counterm.	2348	313.2	4885.0	619.7
overhead (%)	2.2%	-0.06%	-0.93%	-0.32%
	Bcopy (hand)	Mem read	Mem write	
w/o counterm.	616.1	4836	698.7	
w counterm.	618.8	4842	697.8	
overhead (%)	-0.43%	-0.12%	0.12%	

Table 3.3: Performance overhead of our countermeasure in action measured with lmbench benchmark suite. IO tasks and file system operations can show better performance when the countermeasure is active, due to the longer time spent in hypervisor space. While user experience is generally slower, DMA operations can show slightly better throughput.

Benchmark	**w/o counterm.**	**w counterm.**	**Perf.overh.[%]**
Time [sec]	7.153	7.261	1.50%
Req/sec[num/sec]	13981.10	13771.43	1.52%
Time/req [ms]	3.576	3.631	1.54%
Time/concurrent req [ms]	0.072	0.073	1.4%
Transfer rate [KB/sec]	52534.36	51746.51	1.52%

Table 3.4: Results of ApacheBench sending 100000 requests, 50 concurrently on local lighttpd webserver

Benchmark	**w/o counterm.[ms]**	**w counterm.[ms]**	**Perf. overh.[%]**
compression	21.32	21.40	0.4%
decompression	6.73	7.33	8.9%
compiling	394.3	421.2	7.0%

Table 3.5: Performance overhead of our countermeasure in action on compression/decompression (bzip/bunzip) and compilation of kernel code

Benchmark	**w/o counterm.[sec]**	**w counterm.[sec]**	**Perf. overh.[%]**
164.gzip	204	204	0%
175.vpr	138	142	2.8%
176.gcc	88.7	89.0	0.3%
181.mcf	86.4	86.7	0.34%
197.parser	206	207	0.5%
256.bzip2	179	179	0%
300.twolf	229	229	0%
Average	161.6	162.4	0.005%

Table 3.6: Performance of our countermeasure running SPEC2000 benchmarks

Memory overhead Naturally, memory overhead is proportional to the number of objects to be protected. The data structure needed to store information for integrity checking is 32 bytes long for each object (64-bit kernel object physical address, 32-bit kernel object size, 128-bit checksum, 32-bit support flags used by the hypervisor)[2]. Therefore, protecting 15,000 objects costs 469KB in terms of memory space when the original content has not been provided and 2344 KB otherwise.

Moreover, every time a subset of the list of objects is checked the hypervisor needs to map each object from the guest physical space to its virtual space. As explained, the hypervisor will map 100 objects of 128 bytes each every time a MOV_CR event is trapped. This adds an additional cost of 13KB. Thus the overall cost in terms of memory is approximately 306KB (2181KB if a copy of the original content is provided for each object). Since the regular hypervisor allocates 128MB at system startup, the overall memory impact amounts to 1.7%. The trusted module needs exactly the same amount of memory. After raising the hypercall and sending object data to the hypervisor, that memory will be freed and will be made available to the guest kernel. Moreover guest virtual machines are equipped with an amount of memory that is higher than the one allocated by the hypervisor. For the reasons explained above we consider the memory overhead negligible for the guest operating system.

Detection Time Due to the relaxation of integrity checking introduced in the earlier sections, it is possible that the modified critical kernel object will not be in the current subset of objects to be checked. In the considered case study, the list of objects will be checked in 150 process switches. In the worst case scenario a compromised object will be detected after 149 trapped events. This delay has been measured to approximately 5 seconds of wall-clock time, on a machine with hardware specification reported in Table 3.2.

Although this is a considerable time lag, we believe that it is an acceptable security trade-off for the performance benefits that the relaxation offers. An analysis of the chances for an attacker to circumvent the countermeasure with resorting to randomness is provided in the next section.

3.4 Limitations

In this section we describe the limitations and possible weak points of our countermeasure. One possible way to circumvent the countermeasure is compromising the scheduler of the operating system in order to avoid task switching, the regular mechanism that triggers integrity checking. The problem with such an attack is

[2]In order to repair the compromised object, the hypervisor needs to store the object's original content too. This may increase the memory overhead.

that it effectively freezes the system, since the control cannot be returned back from the kernel to the running applications. A rootkit's main goal is to hide itself from administrators. Thus, any rootkit behaving this way will clearly reveal that there is something wrong in the kernel of the running operating system. Moreover, a kernel compromised in this way will never be able to intercept system calls of running processes. These facts suggest that while the attack is possible, it is not probable.

A more probable attack might occur as a consequence of the relaxation explained in Section 3.3. Since only a subset of objects will be checked at any MOV_CR event, a group of malicious processes in the guest might compromise the kernel and restore the original contents before the hypervisor performs the checking (Figure 3.2). We consider such an attack hard to accomplish because, although the list of objects is checked until completion, in a deterministic fashion, the attacker has no knowledge of the position of the compromised object in the hypervisor's memory space nor within the list. An immediate mitigation for this kind of attack might be the randomisation of the sequence in which blocks are checked.

An analytical explanation of the scenario in which our method operates is explained by solving a simple combinatorial problem.

Given N the total number of objects and k the number of objects checked after one trapped event, the number of subsets of k objects containing the compromised one is given by

$$s = \binom{1}{1}\binom{N-1}{k-1}$$

and the probability that the compromised object will be detected after t trapped events is given by

$$P_t(detect) = s\binom{N}{k}^{-1} = t\frac{(N-1)!}{(N-k)!(k-1)!} = t\frac{k}{N}$$

In the scenario described above, where $N = 15000$, $k = 100$ this probability is

$$P_{t=1} = \frac{100}{15000} = 0.6\%$$

Although it seems there might be a very small chance for the hypervisor to detect whether one object has been compromised after one task switch, some considerations must be made to show that, from a practical point of view, this is not a consistent limitation. Rootkits that are capable of inflicting a high damage to their target system, usually have a quite complex behaviour and need to compromise more than just one object. Moreover the effectiveness of a rootkit depends on whether or not it can stay resident and keep target objects in their compromised state.

A 64-bit Linux rootkit [9] that attacks browsers of clients to do iFrame injections has been found recently. After loading into memory via loadable kernel module, the rootkit retrieves kernel symbols with their physical address and writes them into a hidden file

```
cat /proc/kallsyms > /.kallsyms_tmp
cat /boot/System.map-`uname -r` > /.kallsyms_tmp
```

After extracting the memory addresses of several kernel functions and variables it hides the created file and the startup entry in order to make the attack stealthy. At least five kernel functions are needed to be compromised and complete the attack successfully. The addresses of the functions reported below are replaced by pointers to their malicious equivalent [9, 10]

- vfs_readdir
- vfs_read
- filldir64
- filldir
- tcp_sendmsg

These kernel objects must stay compromised within the attacked system to hide files that otherwise would make it observable and for the time needed to connect to a Command and Control server. This delay will definitely allow our countermeasure to detect this rootkit. Moreover, the number of objects to be compromised at the same time reduces the window of detection time so much that it becomes extremely difficult for an attack of this type to stay unnoticed to our mitigation technique.

The probabilistic nature of selecting a subset of kernel objects to be checked each time, makes our technique susceptible to probabilistic attacks. To confirm what we claimed about the effectiveness of the described approach, we provide a Monte Carlo simulation to study the probability of successful attacks in several conditions.

In Figure 3.3 we plot 1000 runs of our method when 1 to 11 objects are compromised and restored after 100 task switches. If an object stays compromised for a time longer than the one required for 150 task switches, it will certainly be detected ($Pr(detect) = 1$) by the hypervisor. Therefore we provide an analysis under condition that is favourable to the attacker.

In Figure 3.4 the probability of successful attacks has been estimated for three cases that consist in restoring the original content of the compromised objects after 75, 90 and 120 task switches respectively. This probability rapidly decreases as the number of compromised objects increases, as in rootkits with more complex

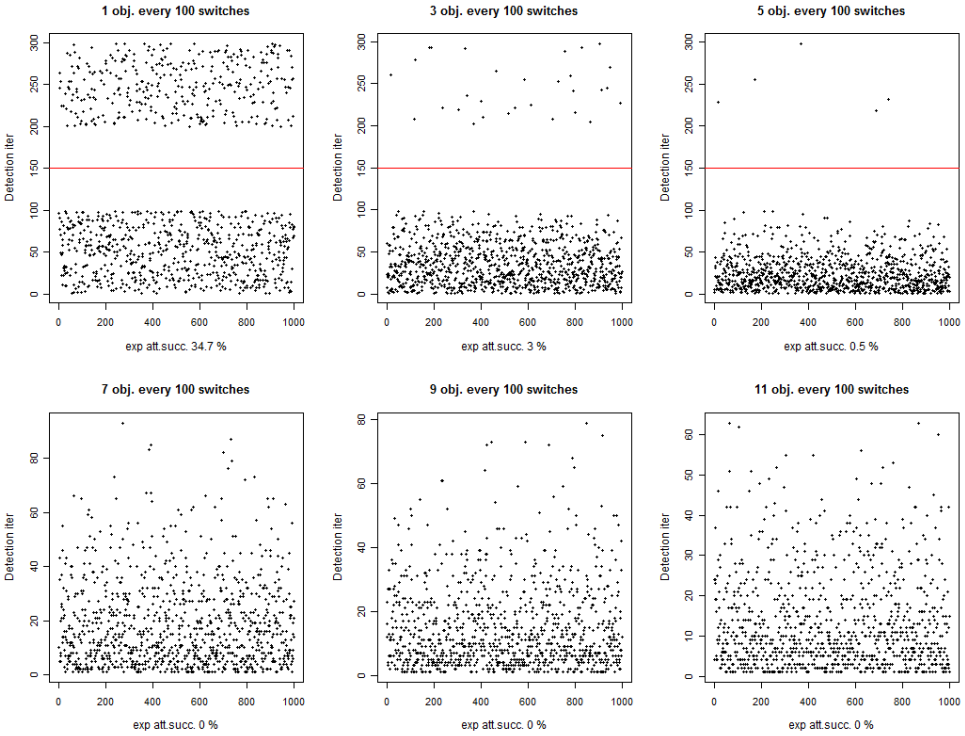

Figure 3.3: Monte carlo simulations for the analysis of the probability of attack, compromising from 1 to 11 objects and restoring their original content after 100 task switches

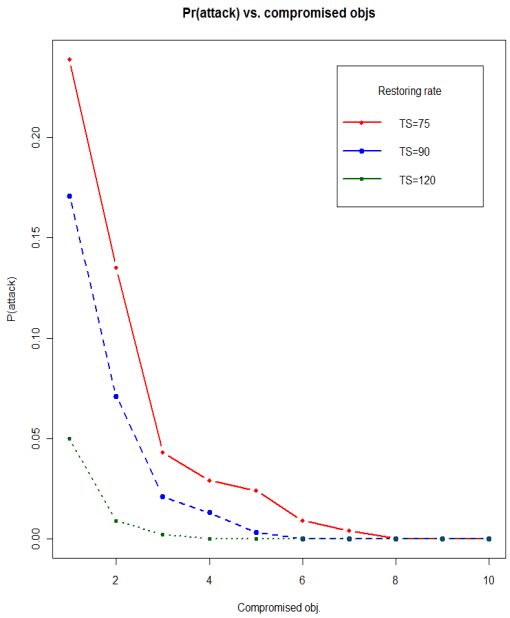

Figure 3.4: Probability of successful attack compared to restoring rate of 75, 90 and 120 task switches

behaviour. As expected, this probability is very small ($<< 0.65\%$) when the number of compromised objects is greater than 4 and the number of task switches $TS > 100$.

A third possible way to compromise the guest kernel would be by corrupting critical kernel objects whose values legitimately change during the kernel's lifetime. Such objects are not invariant and thus cannot be included in the list of objects that this approach checks since this system is unable to differentiate legitimate from non-legitimate changes. The majority though of existing kernel-mode rootkits modify invariant data structures. Thus our system reduces considerably the rootkit attack surface and prevents most rootkits from performing a successful attack.

Lastly, our method relies on invariance inference engines to provide an accurate list of invariant critical kernel objects. In the worst case scenario, if the designated inference engine does not provide all the invariant critical kernel objects that are interesting targets for attackers (false negatives), this approach will be unable to detect attacks that occur in the non-reported kernel objects.

3.5 Related work

Due to the constant development of malicious software and active research in the field, a number of efforts exist on detecting and preventing kernel-mode rootkits. In this section we explore related work that attempts to protect a kernel using different technologies such as virtualisation and special-purpose hardware in Section 3.5.1, kernel code integrity in Section 3.5.2 and code profiling in Section 3.5.3.

3.5.1 Hardware-based countermeasures

Copilot Copilot [92] is a kernel integrity monitor which detects illegal modifications to a host kernel by fetching the physical memory pages where kernel data and code have been stored. The detection strategy of the monitor is based on checking the integrity of kernel data structures with the aid of MD5 hashes. The above detection is performed by a dedicated PCI card, equipped of a co-processor that fetches kernel pages at regular intervals. Although the measured performance impact is relatively negligible, this mitigation technique is affected by several limitations. Since the detection mechanism is running within the same kernel to protect, the aforementioned detection will fail when the target kernel is sufficiently modified. As the authors claim in their work, when the kernel itself lies about its integrity, all other system utilities will receive this information and consider it trustworthy. Even in the presence of dedicated hardware, the fact that it is controlled by the target operating system can influence the overall effectiveness of the countermeasure. In a testbed similar to the one we propose in our solution, which consists in performing checking of kernel data every 5 seconds, the overall performance impact has been measured to 3.80% (WebStone throughput results)

compared to the performance of a native system. Another shortcoming of Copilot is represented by access limited to main memory: there is no way for Copilot to inspect CPU registers, since it is not possible to pause the CPU's execution of the target system. In our solution, whenever control is returned to the hypervisor, which performs kernel integrity checking, the guest system is paused. A snapshot of the register set of a paused system can be inspected in hypervisor space relatively easily. In our prototype we check the integrity of the Interrupt Descriptor Table Register (IDTR). It goes without saying that the entire register set of the guest machine can be checked with the same mechanism. Because Copilot accesses host memory via the PCI bus, there are chances to find kernel data structures in an inconsistent state whenever the monitor performs a DMA access (direct access to physical memory) and a process is modifying them at the same time. Our solution avoids race conditions of this type by invalidating the virtual to physical mapping in the TLB (Translation Lookaside Buffer) upon returning control to the guest operating system. Specifically, when the hypervisor is repairing (writing) a compromised kernel structure, a process that was using that data before will find an inconsistency. However, invalidating the virtual-physical address mapping will force the process to remap the new content stored at the physical address. The strategy of checking integrity every 30 seconds makes the Copilot monitor susceptible to timed attacks. A rootkit that compromises kernel data and rapidly repairs it will stay unnoticed. Although our mitigation technique is affected by the same problem, we provide a probabilistic analysis that shows the chances of successful timed attacks under several conditions.

Gibraltar A system that, similarly to the one described in the previous sections, detects violation of kernel data structures integrity is Gibraltar [81]. Gibraltar executes on a separate machine, called the observer, and monitors the integrity of the kernel running in another machine, physically separated, called the target. Both the observer and the target are connected by the Myrinet PCI intelligent network card. Within such an architecture the observer can remotely access the physical pages of the target kernel. Gibraltar is formed by three main components: the page fetcher, the invariant generator engine and the monitor. The page fetcher, which executes on the observer, fetches the page of a given memory address of the target, where the PCI card has been installed. The PCI device initiates a DMA request for the requested page and send the content back to the observer. The invariant generator engine, which executes in the observer, generates a list of invariants that conform to several templates, such as membership ($x \in a, b, c$), nonzero ($x \neq 0$), bounds ($x \geq c$, $x \leq c$), etc. The monitor ensures that the list of observed data structures in the memory of the target system satisfy the invariants inferred by the invariant generator engine. One limitation of Gibraltar is represented by the fact that pages are fetched and monitored asynchronously. Due to the consistent delay between fetching and monitoring, the chances of timed

attacks are considerably high. However, the main limitation is represented by the isolation requirement which is fulfilled by executing the monitor and the target systems within two physically separated machines. The need of special hardware and the overall performance penalty, make Gibraltar not suitable for production systems.

3.5.2 Kernel code integrity

SecVisor A countermeasure specifically designed to prevent the execution of unauthorized code is described in [73]. For the reasons explained in the previous sections, malicious code executing in kernel mode can be extremely dangerous. SecVisor ensures that only approved code can be executed in kernel mode. It comes in the form of a tiny hypervisor that operates at VMX root mode, exploiting the virtualisation capabilities of modern hardware architectures like AMD and Intel. By validating kernel code before executing it, SecVisor prevents the execution of injected code, as is the case of kernel rootkits. Existing kernel code is also protected in order to prevent any illegal modification from unauthorised parties. In order to achieve the aforementioned goals, SecVisor virtualises the physical memory to set CPU-based memory protections over the kernel, the IO Memory Management Unit (IOMMU) to protect code from DMA writes, and the CPU Memory Management Unit (MMU) in order to check any modification to the MMU from the protected kernel. The very limited amount of changes that need to be applied to port an operating system like Linux to execute on SecVisor makes it an attractive solution against rootkits. Unfortunately, the overhead of virtualising memory and shadowing several components of the CPU is very high, compared to other solutions that can achieve the same degree of security. Another limitation arises from the security evaluation of SecVisor: since control flow integrity is not guaranteed, *return-to-libc*[3] type attacks are still possible. For instance, an attacker can pass control to a kernel function of his choice by overwriting the return address of another function within the same kernel space. Such a behaviour is considered totally legal by SecVisor.

NICKLE A system that prevents the execution of rootkits by detecting the presence of malicious code before its execution is NICKLE [66]. It comes in the form of a minimal hypervisor which prevents unauthorised kernel code execution. One of the main advantages of such a solution consists in the fact that no changes to the guest OS kernel are required. Therefore commodity OSes can be supported without recompilation or reinstallation. To overcome the problem of single memory space for kernel code and user code, NICKLE implements a memory shadowing

[3]This type of attacks usually overwrites the return address on the call stack, replacing it with the address of a function that is already loaded in the binary and that provides the functionality required by the attacker, otherwise implemented by injecting new code.

system that is controlled by the virtual machine monitor and that is not accessible to the guest. The hypervisor maintains a shadow physical memory to store authenticated guest kernel code. Upon the startup of the VM, that is assumed to be in a untampered trusted state, authenticated guest kernel instructions will be copied from the guest's standard memory to the shadow memory controlled by NICKLE. Another specific event that triggers NICKLE's intervention is the loading/unloading of loadable kernel modules. Loading a module is considered injecting code, that needs to be authenticated and validated before execution. At system startup the guest's shadow memory is expected to be empty. At this stage, the system bootstrap code will be verified and copied into the shadow memory. After the kernel has been loaded and decompressed, a cryptographic hash is used to verify its integrity. The kernel code is then copied from standard memory to shadow memory. If the hash values do not match with the values computed previously, the code will not be copied to the shadow memory. An administrator or distribution maintainer is supposed to perform such computations beforehand. The actual protection occurs after validation, when the virtual machine is about to execute a kernel instruction. At this point NICKLE will redirect the instruction fetch to the shadow memory rather than standard memory, after verifying that the current instruction is authenticated. Memory accesses to user code, user data and kernel data will proceed without any intervention. Kernel rootkit attacks would be detected and prevented since invalidated code would attempt to run in kernel mode. A recent attack to bypass countermeasures against code injection attacks, such as the Non-Executable stack countermeasure, is Return Oriented Programming (ROP) [93]. This method of attacking has been used to create return-oriented rootkits which re-use fragments of authorized kernel code for malicious purposes [69]. Such rootkits can bypass countermeasures like the ones proposed in [66, 73].

Hookscout Yin et al. [72] focus on kernel function pointers and protect them from being compromised by rootkits. They developed a function pointer protection system called Hookscout. The main observation of the authors is that, in many cases, access to the source code of the operating system to be protected is not available. Therefore, they provide a binary-centric solution that can generate a hook detection policy without accessing the OS kernel source code. Their approach is also context-sensitive. This allows detection of kernel hooks even in the presence of polymorphic data structures. The approach consists of an analysis and a detection phases. The idea of Hookscout consists in performing dynamic binary analysis of the target system in order to monitor kernel memory objects and keep track of function pointers propagating in kernel space. Since kernel function pointers are the main target of kernel rootkits, the aforementioned analysis can detect how function pointers are created, distributed and used. Moreover, all memory objects allocated statically or dynamically are monitored. The analysis

is performed by TEMU, a dynamic binary analysis platform based on QEMU. Therefore, the target system needs to be run within an emulator. This is the main cause of the overall performance impact. During the analysis, the emulated operating system is tested with common test cases, while the monitor engine collects system information such as the state of memory objects and function pointers. Given this information as input to an inference engine, context-sensitive analysis is performed in order to generate the policy for hook detection. During the detection phase, the detection engine, located within the target system, enforces the policy generated during the analysis phase and detects hooks in the kernel space at runtime. It goes without saying that Hookscout's effectiveness can be compromised by a rootkit capable of disabling the protection since the detection system resides within the same target machine.

HookSafe Another system that can protect thousands of kernel hooks from being compromised is HookSafe [76]. Commodity hardware can provide protection with page-level granularity, meaning that protection flags can be set for a page, not for a byte. Since protection of kernel objects needs byte-level granularity, the authors of HookSafe consider the former type of protection not sufficient against rootkits. HookSafe is a hypervisor-based system that relocates kernel hooks to be protected to a dedicated page and then exploits the regular page-level protection of the MMU to protect that page. This is doable due to the fact that a kernel hook, once initialised, will be frequently read-accessed and rarely write-accessed. Therefore, a relocation of those pointers will not affect the functionality of the overall system. The concept of shadow memory is used to copy the kernel hooks in a centralised location. Attempts to modify the shadow copy will be trapped by the hypervisor which will verify if the caller has enough permissions to modify the protected area. Read accesses will be redirected without any intervention to the shadow copy. A layer between the guest and the hypervisor will handle read and write accesses accordingly. Since the hypervisor is the only component that can write to the memory pages of the protected kernel, control needs to be transferred from the guest to the hypervisor, which will modify on guest's behalf, and then back to the guest kernel. Kernel rootkits can target areas different from regular memory-based function pointers, such as hardware registers like GDTR, IDTR, SYSENTER and MSR. Being HookSafe a hypervisor-based countermeasure, access to hardware registers can be regulated. Any attempt to write to these registers is intercepted and validated. The main limitation of HookSafe consists in the fact that it does not protect non-control kernel data. Therefore, rootkits that target this type of data structures would not be prevented.

SBCFI An approach that dynamically monitors operating system kernel integrity is described in [94]. In this work the authors argue that enforcing control flow integrity can effectively protect against a large class of kernel rootkits. The

Hardware-based countermeasures				
	protect nc data	**special hw**	**protect ROP**	**Overhead**
Copilot	Yes	Yes	No	High
Gibraltar	No	Yes	No	High
Kernel code integrity				
	protect nc data	**special hw**	**protect ROP**	**Overhead**
SecVisor	No	No	No	High
NICKLE	No	No	No	Low
Hookscout	No	No	No	High
Hooksafe	No	No	No	Low
SBCFI	No	No	No	Low

Table 3.7: Comparison of existent countermeasures that protect commodity operating systems against kernel mode rootkits. Column *protect nc data* indicates whether non-control data are protected; *special hw* indicates whether special-purpose hardware is required; *protect ROP* indicates whether the countermeasure protects against Return-Oriented Programming attacks; *Overhead* indicates whether the overhead of the countermeasure is low or high

main observation consists in the fact that, despite the specific behaviour of rootkits (such as injecting code, compromising function pointers, changing the value of specific registers, etc.), the execution flow of a compromised system will differ from the execution flow determined before the attack. Therefore, control flow integrity can be considered a valuable measure of the integrity of the overall system. Due to the persistent nature of rootkits, monitoring control flow integrity continuously is not required. Infrequent monitoring can still effectively protect the system and reduce the performance overhead consistently. SBCFI enforces an approximation of control flow integrity, called state-based control flow integrity. While control flow integrity checks each branch of a precomputed control flow graph in step with the program's execution, SBCFI periodically examines the kernel's state and validates it as a whole. The target kernel code is first checked for any modification, then all static branches are validated. In a subsequent phase, all usable function pointers of the heap are verified that they target valid code. The approximated nature of control flow integrity makes the overhead of SBCFI negligible. Moreover, the control flow integrity monitor can be isolated from the target operating system. Both the implementations of SBCFI on top of the Xen hypervisor and the VMware Workstation virtual machine monitor have a similar performance impact.

3.5.3 Analysis and profiling systems

Although not specifically designed to protect against rootkits, malware analysis can reveal important information about the way rootkits can compromise kernel data structures or how private information is stolen, allowing researchers to understand rootkit's behaviour and design effective countermeasures against them.

Most state-of-the-art analysis approaches share the same execution space of the malware that is being analysed. This can reduce the value and effectiveness of the analysis because some malware can detect the analysis platform and behave differently than in the real target system. The isolation capability of virtualisation technology and the ability to save and restore the state of a guest operating system are offering new tools to shed light on the complex techniques of malware development. General purpose hypervisors are usually not suitable for the purpose of analysing malware because they provide access to an emulated version of the hardware making the entire virtualisation platform visible to complex malicious software. Recent malware like Storm and Conficker are known to detect they are running inside a virtual machine and behave differently [95, 96].

MAVMM MAVMM [75] is a virtual machine monitor specifically designed for malware analysis. It can extract useful information for a complete profile of the investigated malware such as memory pages, system calls and accesses to disks and network devices. Contrarily to other hypervisors, MAVMM lets most hardware access requests go through without interception. This feature makes the entire platform invisible to the malware while still controlling accesses to the aforementioned areas of the system. The analysis platform only intercepts guest execution at a few places, in order to protect its own integrity and log the guests' behaviours for further analysis. Hardware supported Nested Page Tables (NPT) are used to protect its memory from being compromised by the guest, while an IOMMU unit is used to prevent any tamper by external hardware devices via DMA. The execution trace of a guest program is recorded by single stepping its execution at each instruction. This is implemented by virtualising the TF flag that will raise a debugger exception, which will in turn be intercepted by MAVMM. A log of executed system calls can give a good picture of what the malware under investigation is trying to do. MAVMM can record all system calls that a guest system invokes during its lifetime. Selective analysis of specific processes in the guest is also possible by notifying MAVMM each time a process switch occurs in the guest. A known issue to deal with consists of choosing the safest location to which log files can be stored and subsequently analysed. Using the same disk device would expose collected data to the attacker. Therefore, an USB flash drive or a separate disk unit controlled by different device drivers is an approach preferred by the authors.

Hookmap Wang et al. [97] proposed Hookmap, an analysis tool against persistent rootkits. Hookmap is based on the observation that rootkits try to hide their presence within the target kernel and they usually install kernel hooks on the corresponding kernel-side execution path invoked by the target program. Therefore, given a security program, the authors argue that an analysis of its kernel-side execution path would be sufficient to detect and investigate a rootkit's behaviour.

As a consequence, the user-side execution path might be neglected. Linux utility programs like *ls*, *ps* and *netstat* have been analysed and their relative kernel hooks identified. For instance, Hookmap found 35 kernel hooks mapped to the *ls* utility, 85 kernel hooks for *ps* and so forth. As a result, this approach uncovers those kernel hooks that can be potential targets of attacks. By recording possible control-flow transfer instructions in the kernel-side execution paths, Hookmap is able to derive all related kernel hooks and search for abnormalities at runtime. The authors exploit virtualisation technology and system emulation software QEMU to perform their analysis in complete isolation.

HookFinder An approach that does not rely on any prior knowledge of hooking mechanisms is HookFinder [98]. Therefore it represents a suitable tool to identify novel hooking mechanisms. To overcome the limitation of static analysis, that can be easily circumvented by malicious programs equipped with code obfuscation techniques, HookFinder relies on dynamic analysis. This means that malware programs are analysed while being executed within a controlled environment. A whole-system emulator is used for the purpose. All the information collected by the three main components of HookFinder, namely the impact engine, the hook detector and the semantics extractor, is used to derive how the rootkit implants the hook and how the hook is activated by the operating system. When the malware starts executing, the initial impacts, that is data written directly by malicious code (and by external components, on behalf of malicious code) are added to a list to be monitored. Then HookFinder keeps track of the impacts propagating through the whole system. For instance, if it is observed that the instruction pointer is loaded with the address of a monitored impact, and the execution jumps immediately into the malicious code, a hook has been identified. Once a hook has been identified, HookFinder logs into a trace the details about its behaviour within the system. By combining the information of the trace and operating system level semantics, HookFinder can provide an intuitive graphical representation that might be useful to malware analysts for a better understanding of the hooking mechanism under investigation.

PoKer Another virtualisation-based rootkit profiler is described in [77]. PoKer can log the rootkit hooking behaviour, it can monitor targeted kernel objects, extract kernel rootkit code and infer the potential impact on user-level programs. Poker is deployed in systems that can tolerate non negligible performance impact such as honeypots, for which correctly profiling a rootkit has a higher priority than performance. The overall system with PoKer enabled executes 3.8 times slower then the equivalent system without PoKer. The profiler will be enabled after the first malicious instruction has been executed. The techniques described by NICKLE and SecVisor are considered by PoKer to ensure that all actions are logged after this specific event. At this point, PoKer switches to a rootkit

profiling mode, by activating a separate virtual machine, and starts to determine the static and dynamic objects that are being targeted by the rootkit. In order to accurately profile compromised objects, PoKer needs to access the source code of the operating system kernel or debugging symbols and type information of a compiled kernel binary. Gathered information is stored outside the target virtual machine, to prevent any tamper with the rootkit under investigation. Interpreting the traces and resolving read and write target addresses is considered a challenging task, due to the difficulty of identifying the corresponding kernel object, given an arbitrary memory address. Since virtualisation technology does not support such a reverse lookup, an address-to-dynamic object table map is used to translate memory addresses into kernel objects. An initial map of static objects, combined to the rootkit's reads is a sufficient information to build the aforementioned map at runtime.

The countermeasure described in the previous sections can be integrated with the systems described above in order to perform integrity checking and detect illegal changes to those kernel objects collected by the aforementioned analysis tools.

3.6 Summary

In this chapter we showed the effectiveness of the isolation between a hypervisor and a guest operating system and demonstrated how to build a non-bypassable invariance-enforcing framework that takes advantage of this feature. We realised our idea by designing and implementing a lightweight countermeasure that mitigates the problem of kernel-mode rootkits in common operating system kernels.

Upon detection of a change of an invariant kernel object, our countermeasure alerts the administrator of the guest operating system and proceeds to repair the kernel by restoring the data structure to its original values. Due to this change, the rootkit's injected code is usually no longer reachable by any statements in the kernel and thus can no longer affect the running kernel's operations.

The performance impact that usually affects virtualisation technology has been consistently reduced by relaxing the problem of protecting a high number of kernel data structures, without limiting the security degree of the countermeasure. A probabilistic analysis has been provided in order to assess the impact of the aforementioned relaxation, in terms of security. Moreover, our prototype has been evaluated on synthetic and real rootkits (such as the ones described in [88, 99][4]) and it has been showed that it can detect control-flow changes with negligible performance and memory overhead, making it a viable countermeasure for protecting operating systems in virtualised environments.

[4]The rootkits considered in the aforementioned prototype belong to a family of rootkits designed to compromise the system call table and kernel function pointers that perform operations like process forking and opening/closing/writing file descriptors, among other tasks.

The limit of 15000 critical kernel objects holds only in the prototype. However we consider this number large enough to protect a large surface that will make the system much harder to compromise with the aid of those kernel mode rootkits known at date. Although the attack surface results dramatically reduced, attacks to variant data structures are still possible. However, a different strategy and system's design must be investigated in order to overtake such a limitation. We believe that research in that direction is needed as more virtualisation-based systems are making their appearance on the scene.

Chapter 4

Hypervisor-enForced Execution of Security-Critical Code

La semplicitá é l'estrema perfezione (Simplicity is the ultimate sophistication)

Leonardo da Vinci

Leveraging virtualisation technology to mitigate attacks to operating system kernels with a very low impact is a challenging task because of the high overhead caused by the context switch between the guest and the hypervisor. An important goal for any framework employing virtualisation as a security tool, is to guarantee the execution of critical code in the kernel-space of the virtualised operating system regardless of the state of the kernel, i.e. code that will run identically in both clean and compromised kernels.

By critical code we refer to code that, in general, monitors the state of the system and that it is desirable, mainly from a security point of view, to maintain its execution. Examples of such code include the integrity checking of sensitive kernel-level data structures that are usually abused by rootkits or the scanning of files and memory areas for known malware signatures, as in the case of common antivirus systems. Given our assumption of a kernel-level attacker, it is also needed to ensure the integrity of the critical code itself to protect it from malicious modifications which might compromise its efficacy or completely deactivate its operations. As explained in Chapter 3, a way of achieving such a goal is to

implement and execute security-critical code within the hypervisor. Alternative approaches monitor the target system from a separate virtual machine in order to take advantage of isolation between the hypervisor and any virtual machine as well as isolation among multiple virtual machines [70, 100, 101]. Unfortunately, both approaches are known to be affected by a consistent performance overhead. Moreover, when integrity checking methods are considered, the amount of guest memory to be checked can be so large that the overhead might become prohibitive for adoption in production systems. Designing approaches in which guest operating systems cooperate with the hypervisor can lead to high degrees of security, even comparable to the ones guaranteed by completely isolated systems, while significantly reducing the overall performance impact.

4.1 Problem description

Despite the number of publications in the field of kernel level security ([76, 102, 103]), the consistent presence of kernel-level malware is a sign of how complicated kernel protection really is. Fortunately, virtualisation technology may facilitate the design of countermeasures that usually reveal to be effective.

The chief difference between these systems that operate within the hypervisor, (*in-hypervisor*), and those that operate within the target system, (*in-guest*), is that the latter are part of the system's attack surface. Researchers have already proposed various systems that use virtualisation primitives that all fall in this category [66, 92, 104–106].

In-hypervisor security systems, in contrast, can utilize the isolation guarantees of virtualisation technology to make sure that they will be active regardless of the state of the system that they protect. Unfortunately, these security benefits do not come for free. The constant transition from the virtualised operating system to the hypervisor (known as `VMExit`) and back (`VMEntry`), negatively affects the overall performance of the guest forcing one to choose between better security or better performance.

In this chapter we present a framework that facilitates the implementation of countermeasures to protect virtualised operating systems, with security and integrity comparable to those provided by *in-hypervisor* systems but at the performance cost of *in-guest* systems. The system described here follows a hybrid approach by maintaining the security-critical code within the guest, protecting its instructions and data and forcing its execution from the hypervisor. For the reasons explained above we called this framework `HyperForce`.

Taking advantage of the aforementioned framework, we re-implement the countermeasure described in Chapter 3 since that functions as a typical *in-hypervisor* rootkit-detection system. An evaluation of the countermeasure in `HyperForce` shows that it significantly outperforms the original version while maintaining comparable security guarantees.

The rest of the chapter is structured as follows. In Section 4.2 we explain the core idea of our approach followed by its design details. In Section 4.3 we evaluate the performance benefits introduced by this framework and measure its performance impact. Related work is discussed in Section 4.4 and Section 4.5 concludes.

4.2 Approach

The core idea of HyperForce is to combine the best features of the *in-guest* and *in-hypervisor* defence systems into a hybrid solution which performs as an *in-guest* countermeasure while providing security comparable to *in-hypervisor* countermeasures. We achieve this by deploying the functional part of the countermeasure within the guest operating system while maintaining its integrity and enforcing its execution with the assistance of the hypervisor. Since the functional part of the security-critical code, i.e. its instructions and data, is running within the virtualised operating system, the main advantage of such approach is that it has native access to the resources of the virtualised operating system such as the memory, disk and API of the virtualised kernel. This provides a great performance benefit for code that needs to access a high number of memory locations within the virtualised operating system since it alleviates the costly need of introspection usually required by in-hypervisor systems, i.e. the discovery of the corresponding physical memory pages of the guest's virtual memory pages and their remapping within hypervisor's space or another virtual machine's.

Enforcement of Execution. Given an arbitrary piece of security-critical code, HyperForce needs to ensure its execution at regular time intervals. A complete reliance for its execution on the virtualised operating system, could potentially allow a kernel-level attacker to intervene and inhibit the code's execution through the modification of the appropriate kernel-level data-structures. For instance, an attacker could locate the function address pointing to the security-critical code and overwrite it with a pointer towards their own code.

From a high-level view, HyperForce changes the execution flow of the guest kernel whenever the installed monitoring code has to be executed and restores the original execution flow upon code termination. The advances of virtualisation technology allows one to implement this transition in a multitude of ways. Our decision has been influenced by the desire of minimizing the amount of instrumentation code in the hypervisor and of keeping performance overhead to a minimum.

In our framework, the security-critical code is encapsulated within a function that is loaded in the virtualised operating system in the form of Loadable Kernel

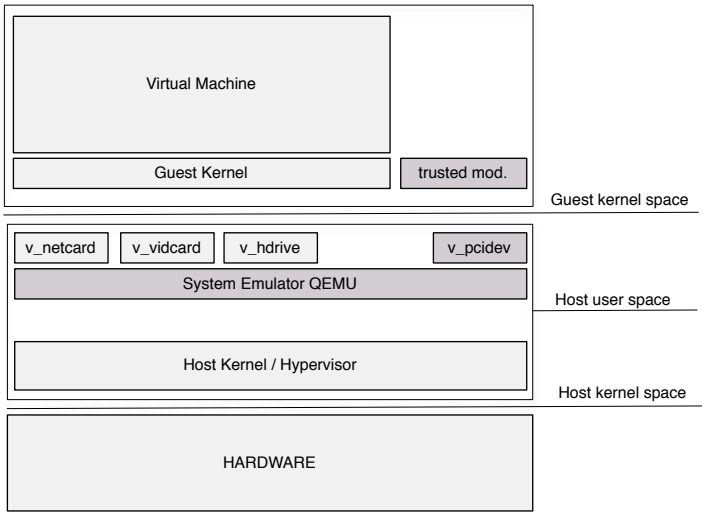

Figure 4.1: Schema of HyperForce. Highlighted components indicate parts of the system that need instrumentation/modification. The trusted module within the guest is never unloaded.

Module (LKM)[1]. This allows the code to have native access to all of the guest operating system's resources. The key component of HyperForce is a virtual device, created by the hypervisor's infrastructure, that is detected and treated as a regular hardware device by the guest operating system. Virtual devices are supported by all modern hypervisors to simulate real hardware, such as sound-cards, graphic cards, storage units, etc. and to allow the execution of unmodified guest code, as it has been designed to run on bare metal. Once the aforesaid virtual device has been created and loaded into the virtualised operating system, HyperForce registers the address of the memory location where the security-critical code has been stored to as an interrupt handler for the aforementioned virtual device. A schema is illustrated in Figure 4.1 in which the additional virtual device within the system emulator QEMU [107] and the trusted module within the guest are highlighted.

The interrupt handler, and consequently the security-critical code, will be called whenever the virtual device generates an Non-Maskable Interrupt (NMI). Hence the cooperation of the hypervisor and the trusted module is based on this call-and-respond mechanism. Since the virtual device is fully controlled by the hypervisor, it is the hypervisor that decides when interrupts must be generated

[1]Although the proof-of-concept has been developed specifically for the Linux kernel, the method can be easily applied to other commodity operating systems such as Windows

and not the virtualised operating system. Due to this fact, the eventually compromised guest kernel has no possibility to anticipate or delay the execution of the security-critical code. The logic behind is hidden from it through the isolation mechanism guaranteed by virtualisation-enabled processors. This fact stops any attackers' efforts to evade detection by mimicking a non-compromised operating system just before the execution of the critical-code and restoring their malicious activities after it.

Integrity of Code. Since the code is loaded in the guest as a LKM, it executes with the privileges of the virtualised kernel. While this is desired, it also opens up the code to attacks, e.g. modifications of its code and data, from an attacker who is in control of the virtualised kernel. Traditionally, the module could not be protected from the rest of the kernel since they both operate within the same protection ring, namely Ring 0. Due to virtualisation however, the hypervisor has more power than the virtualised operating system's kernel (signified as Ring -1) and can thus protect any resources from the virtualised kernel, including memory pages.

HyperForce takes advantage of the paging system provided by the Linux kernel (extended with hypervisor capabilities) to write-protect the memory pages holding the instructions and data of the security-critical code. In order to allow the code to make changes to its data, HyperForce can unlock the memory pages before it triggers an interrupt of its virtual device and lock them back immediately after the code's execution. Most of the performance penalty is due to the task of trapping an access violation from the guest to the hypervisor. As explained in Section 2.1.1, hardware-supported extended page tables (Intel EPT or AMD NPT) for the memory management unit consistently reduces this performance impact.

Another way to circumvent the protection mechanism in place consists in compromising the Interrupt Descriptor Table (IDT), where pointers to interrupt handlers are stored. When an exploitable kernel-level vulnerability is found, it can be relatively easy to compromise the IDT. Therefore HyperForce write-protects the memory page holding the Interrupt Descriptor Table (IDT) of the protected guest. Lastly, HyperForce protects the Interrupt Descriptor Table Register (IDTR) that contains the address of the IDT, by checking its integrity at regular intervals. The aforementioned protection will prevent any attempt to use an Interrupt Descriptor Table that is different from the one observed within the untampered system. The task of protecting the IDTR is straightforward since the IDTR is not supposed to change during operating system lifetime. A schema of the selection of the interrupt gate via the IDTR is showed in Figure 4.2.

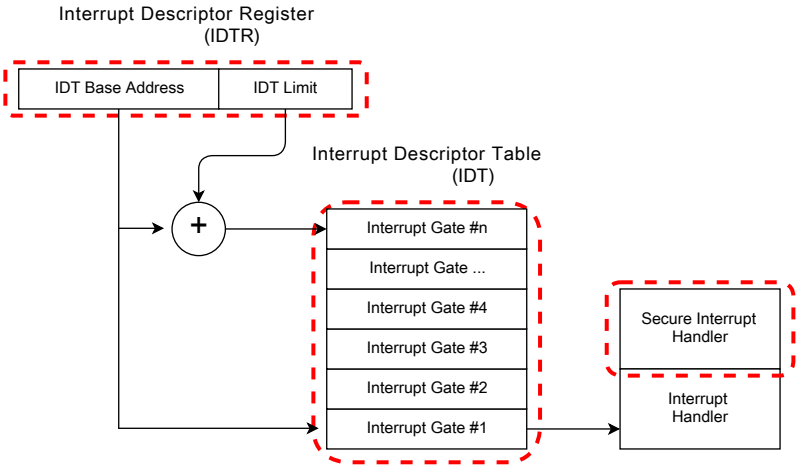

Figure 4.2: Selection of Interrupt Gate via Interrupt Descriptor Table Register in the IA32 architecture. Dashed rectangles indicate that any attempt to write to these areas is trapped and handled by the hypervisor.

4.3 Evaluation

A proof-of-concept of HyperForce has been implemented in Linux-KVM [108], an extension of the Linux kernel that adds hypervisor capabilities. KVM is formed by a system emulator (formerly QEMU [107]) and a Linux device driver. The former emulates all hardware devices that the guest operating system expects to detect after boot and executes as a regular process in user space. The latter executes in kernel-space and uses the new instructions of virtualisation-enabled processors to run virtual machines with increased performance. A virtual device has been added in the form of a PCI card to the set of devices emulated by QEMU. As a physical PCI card, our virtual device can raise interrupts and can access physical memory within the guest system. A device driver executing in the guest will handle the interrupts raised by the virtual device and will execute the code implemented in the interrupt handler. Since the device driver is an extension of the target kernel, the monitoring code will be executed with (guest) kernel privileges.

In order to show the improvements that have been claimed above we re-implemented a pure *in-hypervisor* countermeasure, namely the one described in Chapter 3. A re-implementation has been necessary also due to the different hypervisor technology used in the original version. This allowed us to fairly compare the two.

A schema of the aforementioned integrity checking method implemented in Linux/KVM is provided in Fig. 4.3. It can be observed that while the implementation of the in-hypervisor approach needs instrumentation code to be added

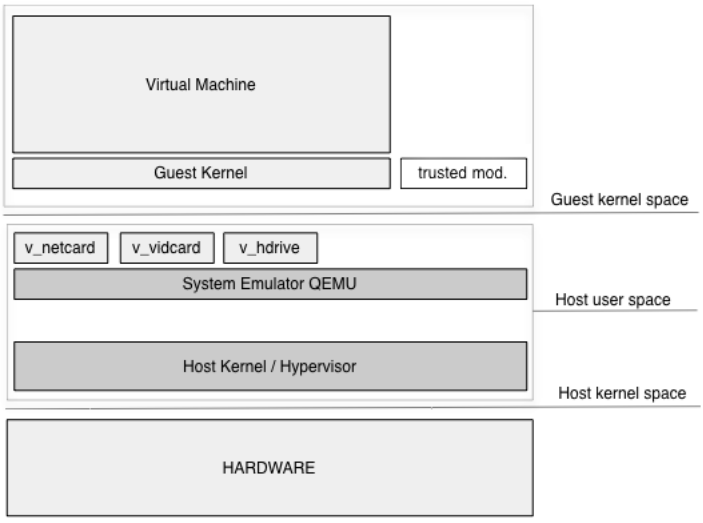

Figure 4.3: Schema of in-hypervisor approach implemented in Linux KVM. Highlighted components indicate parts of the system that need instrumentation/modification. The trusted module installed in the guest is unloaded after boot.

to the host kernel, the HyperForce framework requires only the system emulator to be modified (Fig. 4.1). In Linux/KVM, each virtual machine runs a regular process in user space. An instance of QEMU provides the required emulation of the overall system. A trusted module within the guest kernel to be protected is needed, regardless the choice of the hypervisor.

Results of macro and micro-benchmarks have been collected from the guest and from the host machine and an explanation of these is given in Section 4.3.1 and Section 4.3.2. In order to provide reliable results, all tests have been repeated 10 times and their mean value has been reported. Experiments have been performed on a machine with the hardware specification reported in Table 3.2.

4.3.1 Macro-benchmarks

We run two macro benchmarks with the *iperf* utility, that measures TCP and UDP bandwidth performance and *bunzip* that extracts the Linux kernel source code from a compressed file. The original in-hypervisor version of the integrity checking method is denoted as `in-hyper` while the version using the HyperForce framework is denoted as `HF`.

While the in-hypervisor approach, due to the slower context switching, has a slightly better throughput of network performance in the host machine Ta-

(a) In-host measurements

	iperf [Gb/s]	overhead
in-hyper	6.36	-
HF	6.29	+1.1%

(b) In-guest measurements

	iperf [Gb/s]	bunzip [sec]
native KVM	5.97	32.04
in-hyper	5.26 (+12%)	33.73 (+5%)
HF	5.71 (+4.3%)	32.88 (+2.5%)

Table 4.1: Macro benchmarks (*in-host* OS and *in-guest* OS) evaluating the integrity checking method implemented with and without HyperForce

ble 4.1(a), benchmarks in the guest machine show a considerably better performance with HyperForce. *iperf* and *bunzip* have also been executed on a native KVM system and compared against the same system running in-hypervisor and then in HyperForce. The performance overhead of our approach is about half of the in-hypervisor approach, as shown in Table 4.1(b). The original approach implemented using HyperForce performs with 4.3% overhead compared to a native KVM guest while when implemented within hypervisor shows 12% overhead. The second column reports overhead of *bunzip* measured in seconds. Again, HyperForce outperforms the in-hypervisor approach, showing an overhead of only 2.5% compared to the native KVM guest.

4.3.2 Micro-benchmarks

Micro benchmarks show a more detailed picture of the two approaches. We use *LMbench* [109][2] to measure the overhead of operating system specific events such as context switch, memory mapping latency, page fault, signal handling and fork latency. Within the host machine, HyperForce shows substantial improvement against the in-hypervisor alternative. In Table 4.3.2 we report only the tests where this improvement is consistent. In all other tests the in-hypervisor approach and the one using HyperForce show negligible performance overhead.

The picture in the guest machine shows a similar trend in which HyperForce outperforms the original in-hypervisor method in every test (Table 4.3.2).

To interpret the results shown in Table 4.3.2, we recall that the original in-hypervisor approach performs integrity checks whenever the guest kernel writes to a control register (`MOV_CR*` event). In contrast, the implementation within the HyperForce framework employs interrupt events to trigger in-guest integrity

[2]We use version 3 of LMbench as available at `http://lmbench.sourceforge.net/`.

	ctx switch	mmap lat	page flt	mem lat
in-hyper	2.020	6148	1.57	114.7
HF	1.48	4950	1.46	101.7
speedup	+26%	+19%	+7%	+11%

Table 4.2: Overhead of kernel integrity checking using the HyperForce framework (HF) is measured against in-hypervisor alternative with LMbench micro-benchmarks within the host machine. Operations are measured in microseconds.

	null call	null IO	open/close	sig inst
in-hyper	0.30	0.32	2.32	0.74
HF	0.14	0.21	2.10	0.45
Speed increase	+53%	+34%	+10%	+39%
	sig handl	fork proc	exec proc	ctx switch
in-hyper	5.37	1923	4087	5.58
HF	2.60	1788	3984	5.00
Speed increase	+51%	+8%	+2.5%	+10%

Table 4.3: Overhead of HyperForce is measured against in-hypervisor approach with LMbench micro-benchmarks within the guest machine. Operations are measured in microseconds.

checks. This eliminates overheads with respect to switching execution context and address mapping between the hypervisor and the guest OS, while the remaining computational overhead affects guest operations more evenly. As can be seen in Table 4.3.2, the above changes imply significant speedups on system call invocations (53%) and context switches (10%). Although LMbench is often considered as insufficient for evaluating system performance [91], our example shows that the benchmark suite can be used to neatly distinguish the actual increase of speed on system call invocations ("null call") from the impact on a particular system call execution (e.g. "open/close").

One may think that our approach to trigger security checks through interrupts in HyperForce reduces the security of the protected system compared to the original in-hypervisor: in the latter case an attacker increases their chance of being detected with every system call raised or process switch. However for a total of 15,000 protected kernel objects, the worst-case detection time reported in Chapter 3.3 is about 6 seconds. HyperForce improves on that by checking the same amount of kernel objects in less time. An essential difference between the two approaches is that while the original method relies on the activity of the system as a trigger that checks the integrity of protected objects, HyperForce performs the checking independently of system activity every 4 seconds. This is the time lag of HyperForce measured while checking the entire list of objects in the guest.

Our results indicate that the HyperForce framework could be used to reimplement other in-hypervisor applications, enhancing their performance and maintaining their effectiveness.

4.4 Related work

In this section we review related work in the domain of kernel code integrity assurance. We identify three main areas of active research. Two methods that, similarly to the approach described above, inject security agents within a target system are described in Section 4.4.1. Protection of commodity operating systems by secure code that is executed in isolation with the aid of the regular protection mechanisms, is described in Section 4.4.2. Finally, work related to formal verification of commodity operating systems and hypervisors is provided in Section 4.4.3.

4.4.1 Security Agent Injection

Closely related to the approach described in this chapter is work by Lee et al. [110] and Chiueh et al. [111] on deploying agents by means of code injection from a hypervisor. Both approaches are applicable to guest operating systems that have not been previously prepared by loading a special driver or similar. In [110], Lee et al. proposes to protect agent code that is executing in a compromised guest OS kernel by the use of cryptography and by injecting this code on demand from the hypervisor. As there is no implementation and no experimental evaluation given, a comparison with our solution is not feasible.

SADE Similarly, work on SADE [111] by Chiueh et al. uses VMWare's ESX server API to inject and execute code in a guest OS so as to disable and remove a previously detected malware infection from that guest. In difference to our approach, the agent code in SADE is not protected from malicious interference on the guest. Chiueh et al. argue that on-demand injection leaves a relatively short time span for such interference. SADE is used by a virtual appliance that implements out-of-guest monitoring of VMs' memory, scanning for malware signatures. The paper presents experimental data on the code injection process but does not discuss the overhead implied by mapping memory pages between the virtual appliance and the VMs. We expect in-guest memory inspection, as implemented by our kernel integrity detection method, to outperform SADE.

Kernel heap buffer overflow monitoring The primary goal of Kruiser, a security countermeasure explained in [112], consists in detecting heap based buffer overflows in kernel space. The authors argue that those countermeasures that try to detect a buffer overflow before any write operation are usually affected by considerable performance overhead. As a consequence, the monitored process will slow down consistently whenever the monitoring system is under heavy load. Other systems perform detection occasionally, for instance whenever the buffer to be protected is deallocated. This strategy usually allows the attacker to compromise the system relatively easily, due to the large time window between corruption and

detection. If a buffer is deallocated much later its corruption has occurred there are no viable ways for the countermeasure to detect and interrupt any malicious intent coming from that buffer. Finally, approaches that rely on special hardware are effective but less practical for wide deployment, as we discuss in Section 3.5.1.

Due to the challenging nature of protecting operating system kernels, the authors take advantage of virtualisation technology. The high-level idea of Kruiser consists in placing canaries into kernel heap buffers and lately check their integrity. When a canary is found to be tampered, an alarm, indicating the detection of an overflow, is raised. In order to guarantee isolation between the monitoring and target systems, a separate process, running concurrently with the kernel, performs the checking of the canaries. While canaries are attached at the end of each heap buffer via an in-guest interposition mechanism, the task of checking their integrity is decoupled and it is performed by a process running in a separate virtual machine. This is an essential requirement to strengthen self protection.

We argue that using our framework in such a scenario would allow the execution of the integrity checking process inside the guest, with native performance and without the requirements of allocating resources for any additional virtual machine. The hypervisor can be instrumented from the trusted module about the checking frequency and the physical memory address of the protected area where canaries will be stored at runtime. The non-maskable interrupt mechanism of Hyperforce, triggered by the virtual PCI device installed within the guest, will ensure that there are no chances for the attacker to interfere with the security measure.

As a result, Kruiser implemented in Hyperforce would become a security countermeasure that executes entirely in the target kernel, still keeping the same degree of security of the equivalent out-of-guest approach.

4.4.2 Hardware-based techniques

Flicker An infrastructure that allows the execution of security critical code, in complete isolation from the rest of the system is described in [113]. In order to guarantee such isolation, Flicker takes advantage of hardware support such as AMD SVM or the equivalent Intel Trusted eXecution Technology. These chips allow the launch of a virtual machine monitor or a security kernel at any time, still protecting their code from malicious software. Contrarily to hypervisor's development, that even in the case of minimal hypervisors, usually adds thousands of lines of code to the trusted computing base (TCB), the TCB of Flicker is as small as 250 lines of code. The programmer can add only the code that implements her particular application in a bottom-up approach. One of the most advantageous peculiarities of Flicker consists in the fact that isolation and code attestation can be guaranteed even in the presence of compromised BIOS, DMA devices and the operating system itself. When the processor receives an *init* instruction, it disables DMA, system interrupts and debugging access, enters at 32-bit protected mode and jumps to the provided entry point in order to start the execution of the

isolated application. There is no possibility for other software executing before Flicker to monitor or interfere with Flicker's code. An example that sheds light on the potential of the aforementioned isolation property consists in a piece of code that handles a user's password, that will stay isolated from all other software running on the same machine. In the same scenario, the hardware can guarantee that the secrecy of the password has been preserved. The authors implemented a rootkit detector in the Flicker architecture. In this other scenario, the hardware can guarantee that no other software, the rootkit included, could tamper with the detection code. Any piece of code that needs to be executed in isolation should be included in a Piece of Application Logic (PAL). A Flicker session will then be initialised to begin execution of the Secure Loader Block (SLB). At the end of PAL's execution, a cleanup procedure will erase its secret values and control will return to the operating system or other untrusted components. Flicker offers strong security and isolation properties. But current hardware performance offered by commodity processors is not enough to consider this architecture for every day computing.

TrustVisor Another work aimed at providing the same goals of Flicker with reduced performance penalty is TrustVisor [114]. TrustVisor is a specially crafted hypervisor, designed to provide code and data integrity, secrecy of portions of an application (PAL) in isolation from a legacy untrusted operating system and DMA-capable devices. The granularity of the protected code is as fine as in Flicker, although the code base is one order of magnitude higher, but still small compared to general-purpose hypervisors. The two main capabilities of TrustVisor that use the trusted computing mechanisms offered by AMD and Intel are sealed storage and remote attestation. The former allows a particular Piece of Application Logic to encrypt data according to a policy such that the ciphertext can only be decrypted by the PAL specified in the policy. The latter is the mechanism that allows a remote party to be guaranteed that a particular PAL ran on a specific platform protected by TrustVisor. TrustVisor can operate in host mode, which is the highest privilege level that controls hardware devices; legacy guest mode, where a commodity x86 operating system and its applications usually execute; secure guest mode, dedicated to the execution of PALs in an isolated environment. The TrustVisor architecture is implemented taking advantage of hardware virtualisation support, in order to provide memory isolation and DMA protection for each PAL. The virtualisation mechanisms are also used by TrustVisor to protect its own code. Specifically, TrustVisor virtualises the guest operating system's memory using Nested Page Tables (NPT) and IOMMU to prevent DMA-capable devices to access arbitrary physical memory addresses. PALs are isolated from each other with the same memory virtualisation mechanisms. The execution of PALs follows a registration event that must be requested explicitely by the developer. Both registration and unregistration are implemented by using a hypercall

that is intercepted and handled directly by TrustVisor. The performance penalty of TrustVisor is much smaller than the one of Flicker and the security guarantees are comparable. The code base, around 6000 lines of code, is small enough to verify TrustVisor implementation using software model checking methods, that is what authors plan to achieve in the future.

Fides The goal of Fides [115] is to protect trusted modules from malware capable of exploiting vulnerabilities in the surrounding components of an operating system. The isolation properties of Fides make a vulnerable module exploitable only if other modules explicitly place trust in the former. In all other cases in which, for instance an attacker introduces malicious modules that are disconnected from the rest of the system, as is the case of rootkits, there are no viable ways to compromise the entire operating system. Fides combines a run-time security architecture that protects fine-grained software modules within a commodity operating system and a compiler that compiles modules written in the C language to exploit the security capabilities of the architecture on which they will execute. The software components that execute in isolation from the rest of the system are called Self Protecting Modules (SPM). They are basically a chunk of memory, formed by a secret section, that contains the module's sensitive data and a public section that contains information for which only integrity must be guaranteed. The three main components that form Fides are: the legacy kernel with its applications that run without interruption and intervention of the security system; the hypervisor that manages the hardware and provides coarse grained memory isolation of the legacy and the secure virtual machines; the security kernel that manages the isolated software entities (SPM). The legacy kernel and the security kernel execute respectively in two separated virtual machines. Isolation and scheduling of the virtual machines are performed by the hypervisor, as in a regular virtualisation system. The isolation among SPMs is guaranteed by the security kernel. When a module is invoked, the security kernel receives a request in which the virtual address of the entry point of the module is specified. This address is translated to its physical equivalent with the aid of Nested Paging (NPT) in the legacy virtual machine. According to the Fides architecture, when the processor is executing outside the boundaries of the SPM, it has limited access to the module's memory. Also the destruction of the SPM follows a protocol that will leave no trace of its secret values to other components. The overall performance impact has been measured to 3.22%, but for applications that make heavy usage of SPMs it can grow up to 14%. The abilities of Fides such as confidentiality and integrity of module data, authentication and secure communication between modules, makes this architecture a suitable solution only for high security environments.

4.4.3 Formally verified systems

Formally verifying software means that it becomes possible to guarantee the absence of bugs and, taking security even further, to predict exactly how the verified piece of software will behave under any condition. Formal verification has always been an appealing goal for security researchers since, in theory, it can lead to operating system kernels that never crash and that will never perform unsafe operations [116].

The verification process takes place in the form of a mathematical proof that the software to be verified is bug-free and consistent with its specification [117]. Unfortunately, the complexity of the verification, the code size of commodity operating systems and the complexity of the hardware mechanisms on which they are executed, make the complete verification of such systems more an illusion rather than a goal that can be achieved in the immediate future. However, consistent simplifications and initial assumptions can make the task of verification slightly easier.

In light of virtualisation technology, seeking for a formal verification of hypervisors makes more sense than verifying the single guest operating systems because 1) hypervisors have a much simpler architecture than commodity operating systems and 2) the hypervisor is becoming the bottleneck of a virtualisation platform in terms of performance and security. In order to shed light on the two different fields in which formal verification practitioners have to deal with, we report a formal verification of a simplified operating system and two studies conducted towards the verification of a hypervisor.

seL4 The first formal proof of functional correctness of a general-purpose operating system kernel is achieved by [118]. The authors provide a machine-checked verification of the seL4 microkernel from an abstract specification to its implementation in the C programming language. Correctness of the compiler, assembly code and hardware are some of the assumptions that anticipate the formal proof. Although the seL4 kernel is directly usable on ARMv6 and x86 architectures, a proof on the latter is missing. The model assumes that all memory allocations occurring in the kernel are explicit and authorised. While operating system development follows a bottom-up approach to kernel design, formal verification practitioners have the tendency to verify in a top-down fashion, in order to control the level of complexity of the overall system. seL4 is verified using an approach that is somewhere in between the two extremes. The functional programming language Haskell is used to provide a programming interface for operating system developers and, at the same time, provide code that can be translated directly into the theorem proving tool. The technique used to formally verifying the microkernel is interactive, namely it needs human intervention. Each unit of proof has a set of preconditions that must hold prior to execution and a set of post-conditions that must hold afterwards. The level of complexity depends on the statement being

verified. Strong simplifications are required to lower the difficulty of the verification process. Many of them do not apply to operating systems that are commonly used. Proofs about concurrent systems are extremely hard. The authors do not consider verification of their kernel executing on multiprocessor systems. Moreover the complexity of interrupts, in the form of non-deterministic execution of handlers and preemption, is *solved* by disabling interrupts most of the time except in a small number of interrupt points. Although exceptions occur synchronously, their effects are similar to the ones of interrupts. Exceptions are avoided completely in the verification of seL4. Due to the aforementioned oversimplifications, it is hard to believe that commodity operating systems that executes on architectures like x86, will be formally verified soon.

Hyper-V One of the first attempts to verify an hypervisor has been conducted in [56] on the Microsoft Hyper-V hypervisor. Hyper-V is relatively small and runs directly on x64 hardware. The code is divided in the *kernel stratum* that includes a minimal operating system, a hardware abstraction layer, the kernel, the memory manager and the scheduler (device drivers are not included) and the *virtualisation stratum*, that simulates the x64 machine for the guest operating systems. The hypervisor is written in C, which has a weak type system. But the main challenge in verifying such a piece of code is due to the presence of assembly code snippets, which leads to an integrated verification of the two languages. The authors use VCC, an industrial-strength verification suite that specifically addresses the formal verification of low-level C code. Due to the static modular analysis that VCC can perform, each function is verified in isolation. Moreover, before the verification process, the original codebase is annotated. One of the authors' goals is to let these annotations evolve with the core development and to maintain them within the codebase. An interesting stratagem to tackle with the problem of verifying programs that deal with shared knowledge (e.g. a program that is acquiring a spin lock should know that the spin lock has been initialised and it still exists) consists in using ghost objects, called *claims*, rather then preconditions. Claims are associated with many objects and they guarantee, among other properties, that the objects stay allocated as long as a claim to them exists. Ghost data and code are visible only to the verifier. The verification chain consists in translating the code with annotations into the Boogie language, then the Boogie tool generates the verification conditions to be passed to the theorem prover. The model of the processor core is entirely implemented with ghost functions. Specific ghost functions specify the effects of executing single x64 instructions. The aforementioned model is used to provide verification of assembly code and to prove that Hyper-V simulates x64 instructions correctly for the guest operating systems. Despite the relatively small codebase size of Hyper-V and the expected complexity of the verification process, the authors argue that several hundred functions have been verified and that the VCC-based approach is powerful enough to verify them

all.

Xenon A more recent attempt to formally verify a widely used hypervisor like Xen [61] is Xenon [57]. In the aforementioned work, a formal security policy model is defined, starting from the original version of the Xen hypervisor. The authors argue they provide a robust information flow control, tamper-resistance and self-protection of the virtual machines (called execution environments) running on top of their hypervisor. Their basic approach consists in confining each user community in its own execution environment and limiting the information flow between two environments to a restricted form of replication. Namely, an execution environment that is reading data from another environment, will only get access to a local copy of the data. The policy that allows or denies information flow between two execution environments is called *security domain lattice*. The Xenon formal security model is written in CSP, an algebra used to model non-interference and other trace-based information flow security policies. Moreover, CSP is well suited for reasoning about communication patterns between multiple threads of computation. These patterns are defined as sets of traces of instantaneous events and their duration is not modelled. The Xenon model has been decomposed in subsystems that are modelled separately. Three subsystems have been identified, namely the boundary controller, the guest process and the virtual machine monitor process. The complete model that results from the parallel combination of the guest and the virtual machine monitor is deadlock free. Some peculiarities of the Xenon model consist in the fact that hypercalls, interrupts, traps, privileged and unprivileged instructions are considered Xenon events. Moreover the virtual machine monitor interface provides data structures such as virtual memory, virtual machine control structures (VMSC) and device driver front ends the effects of which can be mapped as Xenon events. Processes are used to model data structures, as a conventional CSP technique. When data have been transformed into CSP processes, the overall complexity of the verification process is reduced because it is possible to interchangeably refer to the interface data structures as though they were processes. Although the formal model does not support non-deterministic interrupts, a simplification represents them as signal events that arise deterministically within the virtual machine monitor. Due to the structure of the policy model that results close to the structure of an implementation, the authors of Xenon argue that they will implement it on the 64-bit x86 architecture.

4.5 Summary

In this chapter we briefly discuss the differences between security mechanisms deployed within an operating system (*in-guest*) and the ones deployed within a hypervisor (*in-hypervisor*) and bring attention to the, seemingly exclusive, choice

between the performance benefits of the former versus the security benefits of the latter.

We tackle this choice by developing a hybrid framework, allowing security mechanisms to be developed in a way that provides them with performance analogous to *in-guest* systems while maintaining the security of *in-hypervisor* systems.

Using this new concept, we re-implemented an *in-hypervisor* rootkit detection system and show how the new version significantly outperforms the original without compromising the security or integrity of the detection system.

We conclude that hybrid security systems that are built on top of the described framework can provide effective and efficient alternatives to mitigate the overhead of techniques that exclusively operate in-hypervisor. An interesting type of applications that might be designed with our framework in mind, is represented by malware detection and removal software, for which both isolation from the target system and performance must be guaranteed, in order to take full advantage of the protection capabilities that these systems usually claim.

Another constraint we tackle with consists in the size of instrumentation code required by our approach. We believe that smaller instrumentation code size will ease the adoption of that security countermeasure and facilitate its deployment within production systems.

Chapter 5

Secure web browsers with virtualisation

Computers are useless. They can only give you answers.

Pablo Picasso

Although virtualisation technology appeared to fulfil the needs of server consolidation and to optimise energy consumption within data centers, another growing trend that has been observed into marketplace is known as *virtual desktop*. It seems that the tendency of centralising data has been extended to the desktop environment. Desktop virtualisation is taking advantage of hypervisor-based technology for delivering applications, data and entire desktop environments to users. Traditional countermeasures that have been designed for those applications that are now delivered on-demand do not benefit of the diverse execution context. We are confident that hypervisor-awareness can lead to stronger security measures, better user experience and, above all, minimal performance impact.

5.1 Motivation

One of the most considerable actors in today's computer use is the web browser. Companies like Google™and Yahoo are evidence of this trend since they offer full-fledged software inside the browser. This has resulted in a very rich environment that is being used by web programmers too. Unfortunately, an immediate side effect of this tendency is the growth of security problems like cross site scripting and cross site request forgeries (CSRF) [119–121]. Moreover, the browser is

often written in unsafe languages. As in any other software of this type, the web browser is exposed to the various vulnerabilities that can affect programs written in such languages, such as buffer overflows, dangling pointer references, format string vulnerabilities, memory corruption, etc. One of the most often exploited type of C/C++ vulnerability of the last decade has been the stack-based buffer overflow [122]. Countermeasures like StackGuard [123], ProPolice [124] and other tools like those explained in [125, 126] that protect areas of potential interest for attackers from being modified, have made buffer overflows harder to exploit. As a consequence, in the constantly animated attacker-researcher world, attackers have focussed on other types of vulnerabilities, targeting the heap rather than the stack. This has massively increased the number of heap-based buffer overflows and memory corruption attacks.

However, due to the changing nature of the heap, these types of vulnerabilities are notoriously harder to exploit. Specifically to the web browser, the heap can look completely different depending on which and how many web sites the user has been visiting. It will be hard for attackers to figure out where in the heap space an exploitable overflow has occurred in order to locate and execute their injected code. Countermeasures like ASLR (Address Space Layout Randomisation) [127] specifically designed to protect the heap have made it even harder to reliably exploit these types of vulnerabilities. In fact, ASLR is a technique by which positions of key data areas in a process's address space, such as the heap, the stack or libraries, are re-arranged at random locations in memory. All attacks based on the knowledge of target addresses (e.g. *return-to-libc* attacks in the case of randomised libraries or attacks that execute injected *shellcode* in the case of a randomised heap/stack) may fail if the attacker cannot guess the exact target address.

Recently a new attack emerged that combines the richer environment found in the browser to facilitate exploits of vulnerabilities based on unsafe languages, sometimes even resulting in the successful bypass of countermeasures like ASLR, which are supposed to protect against these types of vulnerabilities. One such attack is known as *heap-spraying*. The name depicts its nature of using the Javascript engine, usually embedded within modern web browsers, to replicate the code that attackers want executed, a large number of times inside the heap memory. As a result, the chances that a particular memory location in the heap will contain their code will dramatically increase.

Back in 2009 a cyber attack called Operation Aurora [128] targeted several organisations such as Google, Adobe, Yahoo, Symantec, Morgan Stanley and others, with the sole purpose of stealing intellectual property. The attack achieved many of its goals by a coordinated set of operations that included the execution of malicious Javascript which included a zero-day exploit within the Internet Explorer web browser. The exploit downloaded a binary, executed its payload to set up a backdoor and connected to command and control servers. After such attack was

publicly disclosed, code snippets with similar behaviour affected a series of applications that, equally to the web browser, are equipped with script environments and are thus vulnerable to the same type of attack [35, 36, 129].

In the first part of this chapter we focus on modern web browsers and describe a countermeasure against a memory corruption attack such as heap-spraying. In the second part we discuss how the aforementioned countermeasure can be integrated and benefit of the virtualisation-based environment that delivers the protected application on-demand.

The rest of the chapter is organised as follows: Section 5.2 discusses the problem of heap-based buffer overflows and heap-spraying in more detail. Section 5.3 discusses our approach while in Section 5.4 we describe our prototype implementation. We evaluate our approach in Section 5.5. A discussion about the integration within hypervisor-aware systems is given in Section 5.6. We compare our approach to related work in Section 5.7. Section 5.8 concludes.

5.2 Problem description

Heap-based buffer overflows The first step to deploy a heap-spraying attack successfully consists in the injection of malicious code at an arbitrary memory location. Designing security countermeasures assumes that an exploitable vulnerability might exist. This is considered the minimal condition for an attacker to change the execution flow of the program and jump to the injected code. Because a memory corruption is required, heap-spraying attacks are considered a special case of heap-based attacks. Exploitable vulnerabilities for such attacks normally deal with dynamically allocated memory. A general way of exploiting a heap-based buffer overflow is to overwrite management information that the memory allocator stores next to the actual data. General purpose memory allocators, as the ones present in commodity operating systems, allocate memory in chunks. These chunks are usually located in a doubly linked list and contain both memory management information (referred to as *chunkinfo*) and real data (referred to as *chunkdata*). Several allocators have been attacked by overwriting the *chunkinfo* section [130].

Since the heap memory area is much less predictable than the stack it would be difficult to predict the memory address to jump to and execute the injected code. Some countermeasures have contributed to making these vulnerabilities even harder to exploit [131, 132].

Heap-spraying attacks As mentioned before, an effective countermeasure against attacks based on heap-based buffer overflow is Address Space Layout Randomisation (ASLR) [127]. ASLR is a technique which randomly arranges the positions of key areas in a process's address space. This would prevent the attacker from easily predicting target addresses. However, attackers have developed more effective

strategies that can even bypass these countermeasures. Heap spraying [34] is a technique that will increase the probability to land on the desired memory address even if the target application is protected by ASLR. Heap spraying is performed by populating the heap with a large number of objects containing the attacker's injected code. The act of spraying simplifies the attack and increases its likelihood of success. This strategy has been widely used by attackers to compromise security of web browsers, making attacks to the heap more reliable than in the past while opening the door to bypassing countermeasures like ASLR [35, 36, 129, 133, 134].

A heap-spraying attack attempts to increase the probability to jump to the (`shellcode`). To achieve this, a basic block of NOP[1] instructions is created. The size of this block is increased by appending the block's contents to itself, building the so called *NOP sled*. Alternatively, a code semantically equivalent to `No Operation` might be used and have the same effect of increasing the size of the block. This technique can also circumvent trivial countermeasures that rely on simple NOP code detection. Finally `shellcode` is appended to the block of instructions. Therefore a jump to any location within the *NOP sled* will sooner or later transfer control to the `shellcode` appended at the end. Clearly, the bigger the sled (or, equally, the higher the number of sleds) the higher the probability to land in it (or to land in one of them) and the attack to succeed.

A schema of the object described above is provided in Fig.5.1.

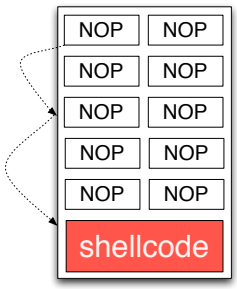

Figure 5.1: Schema of NOP sled and shellcode appended to the sequence

The second phase of the attack consists in populating the heap by exploiting the legal constructs provided by the scripting language embedded in the browser. Figure 5.2 shows the schema of a heap-spraying attack while populating the heap.

Although we will refer to spraying the heap of a web browser, this exploit can be used to spray the heap of any process that leaves to the user the ability to allocate objects into memory. For instance a popular PDF viewer has been

[1]Short for No Operation Performed, is an assembly language instruction that effectively does nothing at all [135]

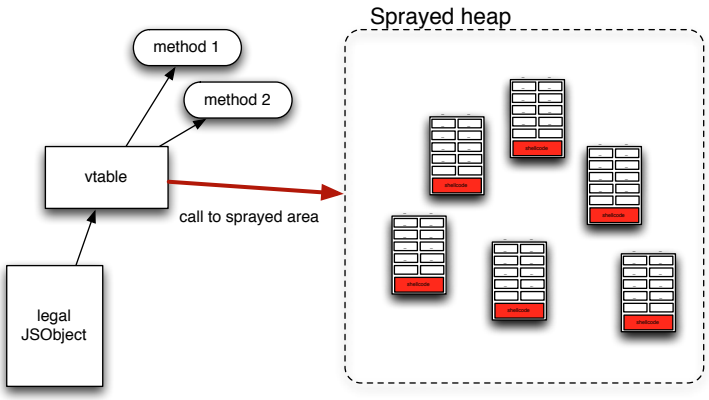

Figure 5.2: A heap-spraying attack: heap is populated of a large number of $NOP-$ *shellcode* objects. The attack may be triggered by a memory corruption. This could potentially allow the attacker to jump to an arbitrary address in memory. The attack relies on the chance that the jump will land inside one of the malicious objects.

found vulnerable to heap-spraying attacks. In that case, a forged PDF file was the medium to execute arbitrary code [136].

What makes heap-spraying an unusual security exploit is the fact that the action of spraying the heap is considered legal and should be permitted by the application. In our specific scenario, memory allocation may be the regular behaviour of the browser while surfing benign web pages. Modern web sites that use AJAX (Asynchronous Javascript And XML) technology or plain Javascript (i.e. facebook.com, economist.com, ebay.com, yahoo.com and many others) appear as they are spraying the heap since a large number of objects is usually downloaded and allocated during their normal operation. We are aware that a security countermeasure should not prevent an application from allocating memory. This makes heap-spraying detection a challenging problem to deal with.

Listing 5.1: A Javascript code snippet to perform a basic heap-spraying attack usually embedded in a HTML web page

```
1. var sled;
2. var spraycnt = new Array();
3. sled = <NOP_instruction>;
4. while(sled.length < _size_)
5.   {
6.     sled+=sled;
```

```
7.    }
8.  for(i=0; i< __very_large__; i++)
9.    {
10.    spraycnt[i] = sled+shellcode;
11. }
```

Because the layout of the heap depends on how often the application has allocated and freed memory, triggering an attack without any knowledge of the exact location to jump to, is utterly difficult. The attacker's strategy would be reduced to merely guessing the address where the malicious objects have been injected to. Unfortunately, by using a client-side scripting language, such as Javascript, it is possible to arrange the heap to the desired layout, making the attack more reliable as described in [137,138].

Code in Listing 5.2 shows a basic heap-spraying attack in Javascript (without any attempt of arranging the heap layout in order to better locate the injected code).

5.3 Approach

An important property of a heap-spraying attack is that it relies on homogeneity of memory. This means that it expects large parts of memory to contain the same information. It also relies on the fact that landing anywhere in the `nopsled` will cause the shellcode to be executed. The key of our countermeasure consists in introducing diversity on the heap, breaking the assumption of homogeneity and making it much harder to reach the shellcode and thus trigger the attack.

The first known heap-spraying attack used the Javascript string type to build sleds. We explain our approach accordingly. The assumption is broken by inserting special interrupting values at random positions whenever the string is stored into memory and removing them whenever the string is used by the application. If these special interrupting values are executed as an instruction, the program will generate an exception that will be served by an interrupt handler that has been previously installed.

Because these special values interrupt the strings inside the memory of the application, the attacker can no longer depend on the NOP sled or even the shellcode being intact. If these values were placed at fixed locations, the attacker could attempt to bypass the code by inserting jumps over specific possible locations within the code. Such an attack, however is unlikely, because the attacker does not know exactly where inside the shellcode control has been transferred.

To make the attack even harder, the special interrupting values are placed at random locations inside the string. Since an attacker does not know at which locations in the string the special interrupting values are stored, he can not jump over them in his NOP-shellcode. This lightweight approach thus makes heap-

spraying attacks significantly harder at very low cost. We have implemented this concept in the Javascript engine of Mozilla Firefox, an open source web browser.

The internal representation of Javascript strings has been modified in order to add the interrupting values to the contents when in memory and remove them properly whenever the string variable is used or when its value is read. The amount of interrupting values can be tuned via a parameter that is set at browser build time. The highest degree of security of our approach can be guaranteed by setting to $s = 25$ bytes the maximum interval at which interrupting. The smallest useful shellcode found in the wild [139] at the time of writing would not fit in this interval. For less strict security requirements, larger values can be assigned to the parameter s.

Given the length n of the string to transform $i = \lceil \frac{n}{25} \rceil$ intervals are generated (where $n > 25$). A random value is selected for each interval. These numbers will represent the positions within the string to modify. The parameter sets the size of each interval, thus the number of positions that will be modified per string. Obviously by choosing a lower value for the parameter s the amount of special interrupting values i that are inserted will be increased and so will the overall performance impact. However, setting the size of each interval to the length of the smallest shellcode does not guarantee that the positions will be at a distance of 25 bytes. It may occur that a position p is randomly selected from the beginning of its interval i_p and the next position q from the end of its interval i_q. In this case $(q - p)$ could be greater than 25, allowing the smallest shellcode to be stored in between. Nevertheless what makes heap-spraying attacks reliable is the large amount of homogeneous data, not simply the insertion of shellcode. Thus being able to insert shellcode will not simply allow an attacker to bypass this approach.

When the string characters at random positions have been changed, a support data structure is filled with *metadata* in order to keep track of the original values and their locations within the string. The modified string is then stored into memory. Whenever the string variable is used, the engine will perform an inverse function, to restore the original content of the string and return it to the caller. This task is achieved by reading the metadata from the data structure bound to the current Javascript string and replacing the special interrupting values with their original content on a copied version of the string. With this approach different strings can be randomised differently, giving the attacker even less chances to figure out the locations of the interruptions. When the function processing the string stores the result back to memory, the new string is again processed by our countermeasure. If the function discards the string, it will simply be freed. Moreover the Javascript engine considered here implements strings as immutable type. This means that string operations do not modify the original value. Instead, a new string with the requested modification is returned.

5.4 Implementation

In this section we discuss the implementation details of our countermeasure and the strategy we considered to tailor it on the Javascript engine of Mozilla Firefox web browser. An attacker performing a heap-spraying attack attempts to arrange a contiguous block of values of his choice in memory. This is required to build a sled that would not be interrupted by other data. To achieve this, a monolithic data structure is required. Javascript offers several possibilities to allocate blocks in memory. The types supported by the Javascript engine in our prototype are numbers, objects and strings. An overview about how the Javascript engine represents Javascript objects in memory is given in [140].

The string type represents a threat and can be used to perform a potentially dangerous heap-spraying attack. Figure 5.3 depicts what a `JSString`, looks like. It is a data structure composed of two members: the `length` member, an integer representing the length of the string and the `chars` member which points to a vector having byte size `(length + 1) * sizeof(jschar)`. When a string is created, `chars` will be filled with the real sequence of characters, representing that contiguous block of memory that the attacker can use as a sled.

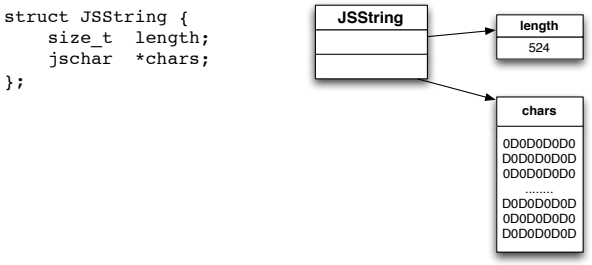

Figure 5.3: Javascript engine's JSString type is considered a threat for a heap-spraying attack since member *chars* is a pointer to a vector of size $(length + 1)$* *sizeof(jschar)*

We have instrumented the `JSString` data structure with the fields to store the metadata: a flag $transformed$ will be set to 1 if the character sequence has been transformed and an array *rndpos* is used to store the random positions of the characters that have been modified within the sequence.

Our countermeasure will save the original value of the modified character to *rndpos*, change its value (at this point the string can be stored into memory) and will restore the original value back from *rndpos* whenever the string is read.

This task is performed respectively by two functions:
`js_Transform(JSString*)` and `js_Restore(JSString*)`.

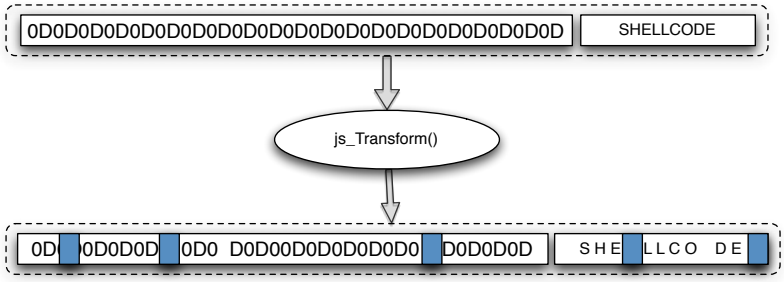

Figure 5.4: Representation of the transformed string in memory: characters at random positions are changed to special interrupting values. The potential execution of the object's contents on the heap would be interrupted by the special value (in blue).

The value of the character to modify is changed to the 1-byte value `0xCC`. This is the assembly language instruction for x86 processors to generating a software breakpoint. If a heap-spraying attack has been successfully triggered[2] and the byte `0xCC` at a random position has been executed, an interrupt handler will detect and report the attack. This alert usually results in exposing a visible popup that encourages the user to close the application and notify the browser's vendor of the detected issue.

As mentioned in Section 5.3, the number of characters to randomise depends on the s parameter. This parameter was chosen based on the length of the smallest shellcode found (to date, 25 bytes long[3]), but can be tuned to select the level of security and the overhead that will be introduced by the countermeasure. If $size$ is the length of the string to transform, the number of intervals is given by $\lceil \frac{size}{24} \rceil$. A random value for each interval will represent the position of the character that will be changed. For reducing the impact of the countermeasure, 50 random values in the range $(0, 24)$ are generated at browser startup. These values are used as offsets to add to the first index of each interval to compute the random position within that interval. The random values are regenerated whenever the Garbage Collector reclaims memory. This prevents an attacker from learning the values over time as they may already have changed.

The value of the i^{th} random position is stored at $rndpos[2i]$, while the original value of the i^{th} character is stored at $rndpos[2i+1]$, with $i = 0, 1, 2 \ldots$. Figure 5.5 provides a schema of how these (position, character) pairs are stored.

[2]The instruction pointer has been forwarded to an arbitrary location within one of the sprayed NOP-sleds

[3]The smallest setuid and execve shellcode for GNU/Linux Intel x86 to date can be found at `http://www.shell-storm.org/shellcode/files/shellcode-43.php`

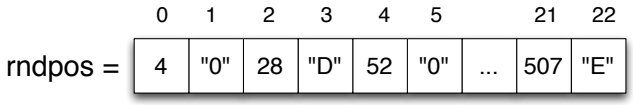

Figure 5.5: How metadata is stored to array *rndpos*: index i within the array contains the value of the position in the string; index $(i + 1)$ contains its original value

Finally, function `js_Transform(str)` will use the values stored in the $str \rightarrow rndpos[]$ array to restore the string to its original content.

5.5 Evaluation

In this section we discuss the performance overhead measured by executing our proof-of-concept on a machine with hardware specifications reported in Table 3.2. The reference environment consists of Mozilla Firefox (Ver. 3.7 Beta 3) browser [141] equipped with ECMA-262-3-compliant Javascript engine, Tracemonkey (Ver. 1.8.2) [142].

We report results of macro benchmarks in order to assess the impact on user experience and micro benchmarks for measuring the real overhead. An analytical study of the memory overhead in the worst case scenario is provided too. By *worst case* we mean the case in which it is guaranteed that the smallest shellcode cannot be stored on the heap without being interrupted at least once by the `0xCC` byte. Last but not least, we made a probabilistic analysis to evaluate the chances of an attacker who can execute shellcode without being detected.

5.5.1 Performance benchmarks

Macro-benchmarks To collect timings of the overhead in a real life scenario, a performance test similar to the one used in Nozzle [143] has been used. Nozzle is the first countermeasure that has been specifically designed to mitigate heap-spraying attacks. We downloaded and instrumented the HTML pages of eight popular web sites by adding the Javascript `newDate()` routine at the beginning and at the end of the page, and computed the difference between the two values. This number represents the time it takes to load the page and execute the Javascript code. Since the browser caches the contents of the web page, that value is a good approximation of the time needed for executing the Javascript. We perform the benchmark 20 times for each website, 10 times with countermeasure disabled and 10 times with countermeasure in place. Table 5.1 shows that the average performance impact over these websites is below 5%.

We consider the aforementioned performance overhead small and the overall

Site URL	Load w/o [ms]	Load w [ms]	Perf. overh.
economist.com	17304	18273	+5.6%
amazon.com	11866	12423	+4.7%
ebay.com	7295	7601	+4.2%
facebook.com	8739	9167	+4.9%
maps.google.com	15098	15581	+3.2%
docs.google.com	426	453	+6.3%
cnn.com	12873	13490	+4.8%
youtube.com	12953	13585	+4.9%
Average			**+4.82**

Table 5.1: Performance overhead of countermeasure in action on 8 popular memory demanding web sites.

countermeasure fast enough for every day computing, due to the fact that the time for loading a web page is usually relatively short to have a noticeable impact on user experience.

Micro-benchmarks Micro-benchmarks can offer a better evaluation of the real performance impact. We performed three different benchmarks: the SunSpider Javascript Benchmarks [144], the Peacekeeper benchmarks [145] and the V8 benchmarking suite [146].

SunSpider is used by the Mozilla Firefox developers team to benchmark specifically the core Javascript language, without the DOM or other browser dependent APIs. The tests are divided into multiple domains: testing code that performs 3D calculations, math, string operations, etc. Table 5.2 contains the runtime in milliseconds of running the various benchmarks that are part of the SunSpider suite. The results for each domain are achieved by performing a number of subtests. We do not report those subtests in which the overhead is close to 0%. However, as we modify the way strings are represented in memory and perform a number of transformations of these, we consider only the subtests that specifically measure the performance of string operations.

The results in Table 5.2 show that the overhead in tests other than string manipulation are negligible. The overhead for string operations on the other hand vary from 3% to 27%. This higher overhead of 27% for base64 is due to the way the base64 test is written: the program encodes a string to base64 and stores the result. When the program starts, it generates a character by adding a random number, multiplying it by 25 and adding 97. This character is converted to a string and added to an existing string. This is done until a string of 8192 characters is created. Then to do the encoding, it will loop over every 3rd character in a string and perform the encoding of those three characters to 4 base64 encoded characters. In every iteration of the loop, it will do 7 accesses to a specific character in the original string, 4 access to a string which contains the valid base64 accesses and finally it

Test	w/o (ms)	w counterm (ms)	Perf. overh.
3d	568.6 +/- 1.4%	569.0 +/- 1.2%	**+0.17%**
bitops	66.4 +/- 1.8%	67 +/- 1.8%	**+0.89%**
controlflow	13.8 +/- 1.9%	14.0 +/- 1.6%	**+1.44%**
math	63.2 +/- 1.0%	63.6 +/- 1.7%	**+0.62%**
regexp	84.2 +/- 2.0%	84.4 +/- 2.9%	**+0.23%**
string ops			
base64	74.8 +/- 2.9%	102.2 +/- 1.9%	+27.3%
fasta	280.0 +/- 1.5%	283.4 +/- 0.7%	+1.24%
tagcloud	293.2 +/- 2.6%	299.6 +/- 0.8%	+2.20%
unpack-code	352.0 +/- 0.8%	363.8 +/- 3.1%	+3.24%
validate-input	119.8 +/- 2.4%	132.2 +/- 1.0%	+9.30%
	1119.8 +/- 0.9%	1181.2 +/- 1.0%	**+5.19%**

Table 5.2: Microbenchmarks performed by SunSpider Javascript Benchmark Suite

will do 4 more operations on the result string. Given that our countermeasure will need to transform and restore the string multiple times, this causes a noticeable slowdown in this application.

Benchmark	Score w/o	Score w	Perf. overh.
Rendering	1929	1919	+0.5%
Social Networking	1843	1834	+0.5%
Complex graphics	4320	4228	+2.2%
Data	2047	1760	+14.0%
DOM operations	1429	1426	+0.2%
Text parsing	1321	1298	+2.0%
Total score	1682	1635	**+2.8**

Table 5.3: Peacekeeper Javascript Benchmarks results (the higher the better).

Peacekeeper is currently used to tune Mozilla Firefox. It will assign a score based on the number of operations performed per second. The results of the Peacekeeper benchmark are located in Table 5.3: for most tests in this benchmark, the overhead is negligible, except for the Data test which has an overhead of 14%. The Data test is a test which will do all kinds of operations on an array containing numbers and one test which performs operations on an array containing strings of all the countries in the world. The operations on the strings of the aforementioned array heavily contribute to slowdown the program: whenever a country is read, the string is restored, whenever one is modified the resulting new string is transformed.

The V8 Benchmark Suite is used to tune V8, the Javascript engine of Google

Benchmark	Score w/o	Score w	Perf. overh.
Richards	151	143	+5.6%
DeltaBlue	173	167	+3.6%
Crypto	110	99.6	+10.4%
Ray Trace	196	193	+1.5%
EarlyBoyer	251	242	+3.7%
RegExp	174	173	+0.6%
Splay	510	501	+1.8%
Total score	198	193	**+2.6**

Table 5.4: V8 Benchmark Suite results (the higher the better).

Chrome. The scores are relative to a reference system (where this score is 100) and as with Peacekeeper, the higher the score, the better. Again, most overheads are negligible except for Crypto, which has an overhead of 10.4%. Crypto is a test that encrypts a string with RSA. The application does a significant number of string operations, resulting in transformation and restoration occurring quite often. These benchmarks show that for any string intensive Javascript application that do little else besides just performing string operations, the overhead can be significant, but not a show stopper. In all other cases the overhead was negligible.

5.5.2 Memory overhead

This section discusses the memory overhead of our countermeasure. This is done by providing both an analytical description of the worst case scenario (in terms of memory requirements) and providing a measurement of the memory overhead that the benchmarks incur.

An analytical study of memory usage has been conducted in the case of the highest level of security. As mentioned before this is achieved when we want to prevent the execution of the smallest shellcode by changing at least one character every 24 bytes. Given s the length of the smallest shellcode, the `js_Transform()` function will change the value of a random character every $(s-k)$ bytes, $k = 1...(s-1)$. In a real life scenario $k = 1$ is sufficient to guarantee a lack of space for the smallest shellcode. If the length of the original string is n bytes, the number of positions to transform will be $i = \lceil \frac{n}{s} \rceil$. The array used to store the positions and the original values of the transformed characters will be $2i$ bytes long.

Memory usage: a numerical example Given the following data:

original string length: $n = 1MB = 1.048.576bytes$

smallest shellcode length: $s = 25bytes$

and the size for each field in the support data structure

number of interruptions $t = 2bytes$

position of changed character $p = 1byte$

injected sequence length: $r = 1byte$

The number of interruptions is $i = \lceil \frac{n}{s} \rceil = \lceil \frac{1MB}{25} \rceil = 43691$ and the total size ds of the support data structure, in the worst case, is given by Equation 5.1

$$ds = t + i \times (p + r) \tag{5.1}$$

Referring to the above numerical example, the total size of the data structure is given in Equation 5.2

$$ds = 2 + 43691 \times 2 = 87384[bytes] = 86[KB] \tag{5.2}$$

Therefore, the memory overhead is $(\frac{1024+86}{1024} - 1) * 100 = 8.3\%$

Benchmark	**Used mem w/o (MB)**	**Used mem w (MB)**	**Overh.**
Sunspider	88	93	+5.6%
V8	219	229	+4.2%
Peacekeeper	148	157	+6.5%
Average			**+5.3%**

Table 5.5: Memory overhead of countermeasure in action on three Javascript benchmarks suites.

Memory overhead for the benchmarks Table 5.5 contains measurements of the maximum memory in megabytes that the benchmarks used during runtime. These values have been measured by starting up the browser, running the benchmarks to completion and then examining the *VmHWM* entry in $/proc/ < pid > /status$. This entry contains the peak resident set size which corresponds to the maximum amount of RAM the program has used during its lifetime. Our tests were run with swap turned off, so this is equal to the actual maximum memory usage of the browser. These measurements show that the overhead is significantly less than the theoretical maximum overhead.

5.5.3 Security evaluation

In this section we give a security evaluation of the approach described in the previous section. The Javascript code snippet of Fig 5.2 can populate the heap with

malicious objects. As mentioned before, spraying the heap means allocating memory and, in our opinion, this should not be considered an action to be detected. However, when the countermeasure is in place and an exploitable memory corruption vulnerability exists, any attempt to execute the contents of the sprayed objects will be very likely to fail. As the instruction pointer lands within a sled, the execution of a byte instruction `0xCC` located at a random position, will call the interrupt procedure that will halt the program. The execution of the `0xCC` sequence is a sufficient condition to consider the system under attack. In fact, a legal access to the same object would purge all `0xCC` interruption bytes and de-randomise its content.

One limitation of the proof-of-concept we provided, not the approach in itself, is that heap-spraying attacks can still be performed by using languages other than Javascript (i.e. Java, C#, ActionScript, etc...). We are confident that the design in itself gives a reasonably strong security guarantee against heap-spraying attacks to be considered for other browser supported languages.

Another way to store malicious objects to the heap of the browser would be by loading images or media directly from the Internet. However, this strategy would make the attack clearly observable. The effectiveness of spraying the heap is due to the fact that large amounts of data are allocated. If, in order to circumvent content randomisation, this data has to be downloaded from the Internet, the attack would result discoverable because of traffic of hundreds of MBs.

Probabilistic analysis In order to quantify the chances of an attacker who injects shellcode under several conditions, we provide a simple probabilistic analysis and results obtained from numerical simulation. We recall that the highest level of security is guaranteed when an interruption is inserted every 25 bytes, which determines a lack of physical space in memory even for the smallest shellcode known at date. The probability of hitting one special interruption byte in a chunk of 25 bytes is $\frac{1}{25} = 0.04$.

In the numerical simulation which confirms this probability, sprayed memory is interrupted according to the parameters of the countermeasure described above. For each variation of parameters such as size of injected code and interruption rate, the simulation has been run several times and the final number of detections is obtained by averaging across 1000 experiments. We believe this strategy to be capable of returning robust results.

In Figure 5.6 we show a graph of the probability of detection versus the size of the injected code, at the highest level of security. When the injected code is smaller than 10 bytes the probability of interrupting it and detecting the attack upon code execution, is less than 40%. Fortunately, this probability increases very quickly for codes of larger sizes and goes practically to 100% (certain detection) when the attacker injects a code larger than 42 bytes.

Another variable that not only determines the level of security but also the

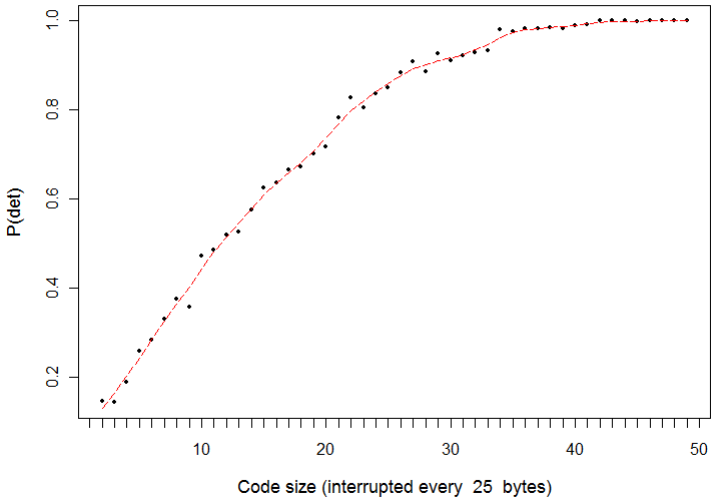

Figure 5.6: Probability of detection versus size of injected code.

overall performance impact of the countermeasure is the interruption rate. In order to achieve better performance this variable can be set to higher values, at the cost of decreasing the level of security. In Figure 5.7 we show the probability of detection for injected code of several sizes versus the interruption rate. We are confident this to represent a better picture that should be considered before tuning the interruption rate in order to have a countermeasure with a smaller performance impact.

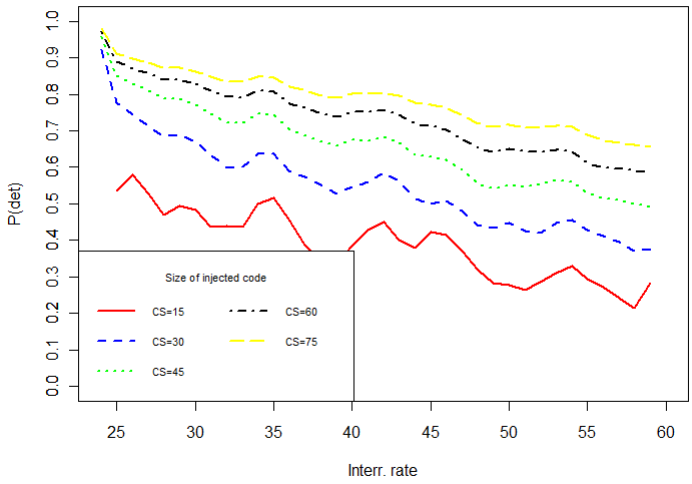

Figure 5.7: Probability of detection compared to detection rates for injected code of several sizes

An interesting scenario would arise if an attacker breaks the 25-byte shellcode in smaller chunks, not only to fit in the limited interrupted space but also to decrease the probability of detection. Therefore, he concatenates each chunk by using *JMP relx* instructions[4]. Clearly, each chunk can be stored in the memory space interrupted every 25 bytes and, according to the aforementioned simulation, the probability of successful execution of that code would be $1 - P(detection) = 1 - 0.4 = 0.6$. Assuming the attacker is concatenating chunks of 10 bytes using *JMP rel8* instructions (2 bytes on the Intel architecture), there will be space for only $10 - 2 = 8$ bytes of shellcode per chunk. Therefore, the number of chunks needed

[4]On the Intel architecture *JMP relx* indicates an instruction that sets the Instruction Pointer to a relative offset of 8,16 or 32 bits.

to form the complete 25 byte shellcode amounts to $\lceil \frac{25}{8} \rceil = 4$. The assumption of independency between chunks and special interruption bytes provides an estimate of the probability of executing 4 adjacent chunks equal to $0.6^4 = 0.13$. This represents the best case scenario for the attacker who has complete knowledge of the memory layout and can jump directly to the beginning of his shellcode without the aid of NOP sleds. In realistic conditions this knowledge is not present.

As already mentioned, the reliability of heap-spraying attacks is due to the usage of code sleds that drive the Instruction Pointer from inaccurate memory locations to the shellcode. The use of NOP sleds dramatically increases the size of injected code. Therefore, although we consider these attacks feasible from a probabilistic point of view, they are quite hard to be realised in practice with reasonable reliability.

5.6 Hypervisor integration

Despite the numerous possibilities that virtualisation technology can offer to the end user, we will focus on a specific case of virtualised desktop. The environment we refer to in this section is the one used to deliver web browsers on demand.

A virtual machine for each web browser is deployed in order to guarantee mobility, consistency of view and security. Mobility allows the user to access his own private desktop from anywhere, using any device and maintaining a consistent look and feel among them. It should be clear that, in such an environment, security by isolation is achieved by design. In fact, all the virtual machines are running on top of a hypervisor executing on virtualisation-enabled hardware.

However, considering an application more secure and shielded just because it is running within a virtual machine is a type of misunderstanding that is becoming more and more common, specially in the industry [147]. Specifically to the environment we described, isolating virtual machines would not prevent attacks to the single web browser. Executing an instance of the web browser within a virtual machine does not make it immune to common attacks. Therefore, if that web browser loads a malicious page where a heap-spraying attack has been implemented, the chances of successful attack are equivalent to those of a browser running within a physical machine.

However, if a countermeasure like the one described before is in place, precious information about the attack can be collected and used to reduce the risk of attacks to the rest of the infrastructure. The other virtual machines running on top of the same hypervisor can be informed of the malicious address that caused the attack.

As mentioned before, executing the $0xCC$ byte should be a convincing evidence of an on-going heap-based attack on the browser's heap. Moreover, when the operating system has been virtualised, the $0xCC$ interrupt handler implemented in the guest kernel can interact with the hypervisor in a straightforward way, via the raise of a hypercall as described in Chapter 2. This mechanism is used not

only to notify an attempt of execution of malicious code to the hypervisor, but also to update a blacklist of URLs that caused such an attack. A schema of such an environment is provided in Figure 5.8.

Once a heap-spraying attack has been detected, two main strategies might be considered:

- deliver the blacklist of malicious URLs to the virtual machines that will, in turn, update their network filters and prevent the local web browsers from navigating to these addresses or
- keep the aforementioned blacklist in the hypervisor's space.

Since the hypervisor has access to physical network devices and it opens network connections on behalf of the virtual machines running on top, we believe that the second choice is to be preferred. A centralised blacklist results easier to keep up to date. Moreover, the isolation mechanism will even protect this information from any attempt of compromising or deleting it from the guest kernel.

Clearly, in those cases in which virtual machines need to be migrated to other hypervisors, for the reasons that are out of the scope of this work, decentralising the blacklist and delivering it to the single guest machines will be essential to keep the guests protected even after their migration has occurred.

Regardless the choice of strategy, further access to malicious network address will be denied to browsers running within the virtualised environment.

5.7 Related work

In this section we provide some of the most effective countermeasures related to web browser security developed so far. Several countermeasures, like the ones described in Section 5.7.1, have been designed and implemented to specifically protect against heap-spraying attacks. Others have been designed to prevent memory corruption in general, like those in Section 5.7.2. Two architectures that take advantage of virtualisation technology to provide coarse grained isolation of user applications are described in Section 5.7.3.

5.7.1 Heap-spraying defences

Nozzle Nozzle is the first countermeasure specifically designed against heap-spraying attacks via web browsers [143]. It uses emulation techniques to detect the presence of malicious objects. This is achieved by the analysis of the contents of any object allocated by the web browser. The countermeasure is in fact implemented at the memory allocator level. This has the benefit of protecting against a heap-spraying attack by any scripting language supported by the browser. Each

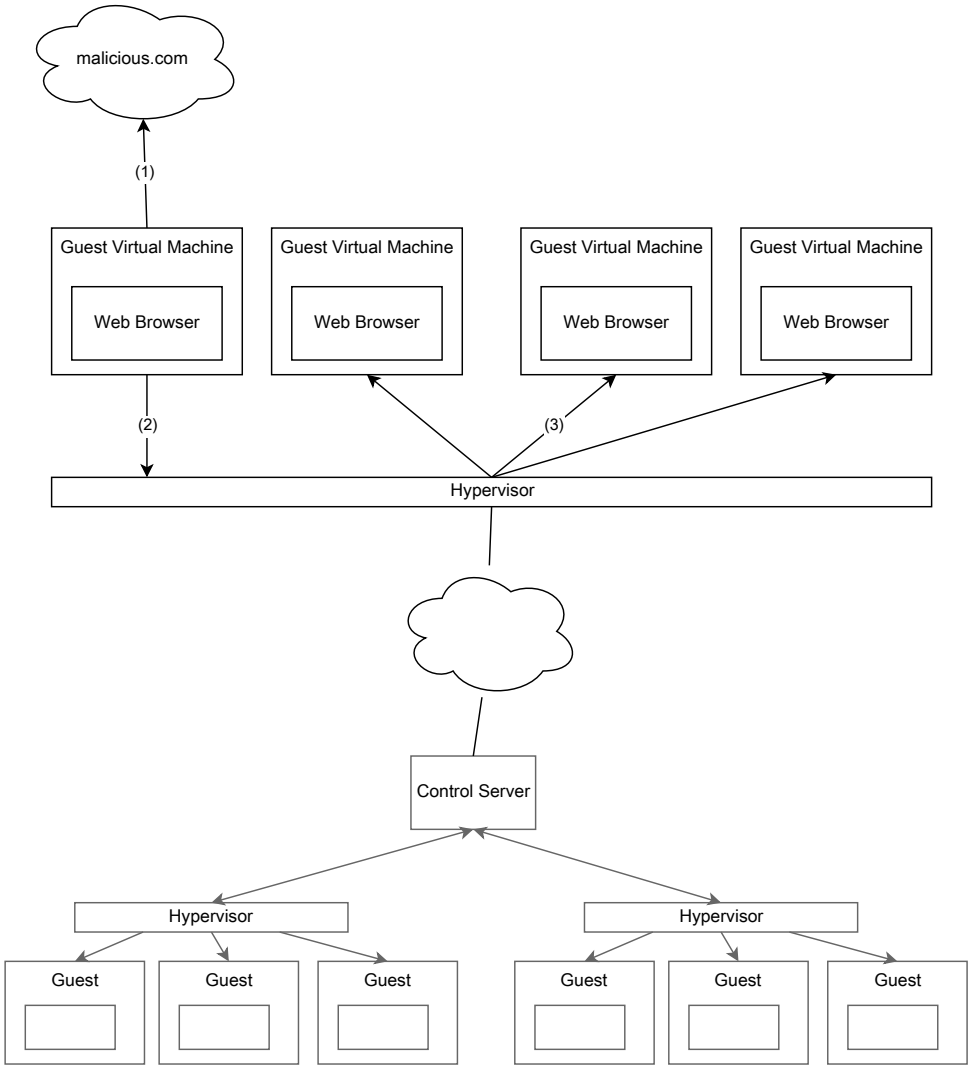

Figure 5.8: Schema of hypervisor integration: upon loading malicious content from malicious.com (1) and local detection of heap-based attack, the guest notifies the hypervisor (2) sending the malicious URL that caused the attack. The hypervisor will deliver this information to all virtual machines running on top (3) or, alternatively will update a blacklist in its private space.

block on the heap is disassembled and a control flow graph of the decoded instructions is built. A `NOP-shellcode` object may be easily detected because one basic block in the control flow graph will be reachable by several directions (other basic blocks). For each object on the heap a measure of the likelihood of landing within the same object is computed. This measure is called `attack surface area`. The surface area for the entire heap is given by the accumulation of the surface area of individual blocks. This metric reflects the overall heap *health*. This countermeasure is more compatible than DEP and would help to detect and report heap-spraying attacks by handling exceptions, without just crashing. This approach has although some limitations. Because Nozzle examines objects only at specific times, this may lead to the so called TOCTOU-vulnerability (Time-Of-Check-Time-Of-Use). This means that an attacker can allocate a benign object, wait for Nozzle to examine it, then change it to contain malicious content and trigger the attack. Moreover, Nozzle examines only a subset of the heap, due to performance reasons. But this approach will lead to a lower level of security. The performance overhead of Nozzle examining the whole heap is unacceptable for every day computing. Another limitation of Nozzle comes from the assumption that a heap-spraying attack allocates a relatively small number of large objects. A design based on this assumption would not protect against another type of heap-spraying attack which allocates a large number of small objects instead, which is known to have the same probability to succeed.

Shellcode detection Another countermeasure specifically designed against heap-spraying attacks to web browsers is proposed by [148]. This countermeasure is based on the assumptions that (1) a heap-spraying attack may be conducted by a special crafted HTML page instrumented with Javascript and (2) Javascript strings are the only way to allocate contiguous data on the heap. Thus all strings allocated by the Javascript interpreter are monitored and checked for the presence of shellcode. All checks have to be performed before a vulnerability can be abused to change the execution control flow of the application. If the system detects the presence of shellcode, the execution of the script is stopped. Shellcode detection is performed by *libemu*, a small library written in C that offers basic x86 emulation. Since *libemu* uses a number of heuristics to discriminate random instructions from actual shellcode, false positives may still occur. Moreover an optimised version of the countermeasure that achieves accurate detection with no false positives is affected by a significant performance penalty of 170%.

5.7.2 Alternative countermeasures

Probabilistic countermeasures Many countermeasures make use of randomness when protecting against attacks. Canary-based countermeasures [149,150] use a secret random number that is stored before an important memory location: if

the random number has changed after some operations have been performed, then an attack has been detected. Memory-obfuscation countermeasures [151, 152] encrypt (usually with XOR) important memory locations or other information using random numbers. The memory layout is usually randomised as in [127, 153, 154], by loading the stack and heap at random addresses and by placing random gaps between objects. Another type of randomisation consists in encrypting the entire instruction set of an architecture [155] before fetching the instructions and decrypting them before they are executed. While these approaches are often efficient, they rely on keeping memory locations secret. Unfortunately, programs that contain buffer overflows could also be affected by *buffer overreads* vulnerabilities (e.g. a string which is copied via *strncpy* but not explicitly null-terminated could leak information) or format string vulnerabilities, which allow attackers to print out memory locations. Such memory leaking vulnerabilities could allow attackers to bypass this type of countermeasure. Another drawback of these countermeasures is that, while they can be effective against remote attackers, they can be easily bypassed locally, via brute force attacks on the secret areas.

DEP Data Execution Prevention [125] is a countermeasure designed to prevent the execution of code in memory pages. It is implemented either in software or hardware, via the NX bit. With DEP enabled, pages will be marked non-executable and this will prevent the attacker from executing shellcode injected on the stack or the heap of the application. If an application attempts to execute code from a page marked by DEP, an access violation exception will be raised. This will lead to a crash, if not properly handled. While the aforementioned hardware-based countermeasure is recognised as effective against code injection attacks, its main limitation consists in the fact that several applications attempt to execute code from memory pages. Due to these types of compatibility issues, the deployment of DEP is less straightforward than it should be [156].

Separation and replication of information Countermeasures that rely on separation or replication of information will try to replicate valuable control-flow information [63, 157–159] or will separate this information from regular data. This makes it harder for an attacker to overwrite these critical memory areas using an overflow. Some countermeasures will simply copy the return address from the stack to a separate stack and will compare it to or replace the return addresses on the regular stack before returning from a function. These countermeasures are easily bypassed using indirect pointer overwriting by which an attacker overwrites a different memory location instead of the return address exploiting a pointer from the stack. More advanced techniques try to separate all control-flow data (like return addresses and pointers) from regular data, making it harder for an attacker to use an overflow to overwrite this type of data. While these techniques can efficiently protect against buffer overflows that try to overwrite control-flow

information, they do not protect against attacks where an attacker controls an integer that is used as an offset from a pointer, nor do they protect against non-control-data attacks.

Virtual Browser A work that borrows the concepts of virtualisation technology and applies them to the development of secure web browsers is provided in [160]. Virtual Browser is a browser-level virtualised environment that executes third party Javascript code in isolation from the rest of the system, namely other components of the web browser itself and the host system. The isolated components can still communicate with each other through data flows that have been carefully examined with security in mind. The idea of Virtual Browser is very similar to that of virtual machines. It provides its own HTML and CSS parsers and a Javascript interpreter. Therefore, untrusted Javascript code is parsed and executed within the isolated environment, preventing the exploitation of bugs in the main Javascript interpreter of the browser. For the same reason, trusted Javascript can execute within the native Javascript engine, improving the overall performance of the countermeasure. The performance overhead is comparable to the Microsoft Web Sandbox, but the degree of security is higher and a more complete Javascript language is supported. An important feature that makes Virtual Browser readily deployable is that it is written in a language that is supported by the native browser (a Javascript implementation is provided). Therefore, no modification to the browser codebase is required. Virtual Browser provides isolation of the Javascript interpreter, leaving all other means to attack web browsers unprotected. Although a direct comparison with Web Sandbox shows that Virtual Browser outperforms it in some cases, the overhead compared to web browsers running natively is not negligible.

5.7.3 Virtualisation-based countermeasures

QubesOS The development of a virtualisation-based Linux distribution like QubesOS [106] has been driven by the inability of traditional operating systems to provide isolation among different applications running within the same machine. This is the main reason for which current operating systems are usually not capable of protecting other user's applications and data from being compromised when an application, for instance the web browser, has been attacked by a malicious website. This concept has been extended to the other components of the operating system. Exploiting a bug in the network stack can affect the security of other applications and their data, without the operating system to notice and defend potential targets. Virtualisation is the chief technology QubesOS is based on, due to its security isolation properties. The main idea of QubesOS is to execute a number of guests, also called *disposable virtual machines*, in which applications that need to stay isolated during their lifetime are executed. The entire system is

based on disposable virtual machines such as the social virtual machine, in which applications like the email client or any other web social service will be executed, the shopping virtual machine, used for those applications to purchase goods from the Internet with a credit card, or the corporate virtual machine, where the corporate email client or VPN connections will take place. If one of the aforementioned virtual machines gets compromised, applications and data running in the other ones will stay isolated and their code untampered. Disposable virtual machines are not only used for regular applications but also for system services such as the network system, the graphical or the storage subsystems. A bug in one of those subsystems will stay isolated for the same reasons explained above. QubesOS is a hybrid architecture that takes the concept of isolation, typical of microkernel systems, in which device drivers are not part of the core kernel and run in unprivileged mode, and the flexibility of monolithic systems. The challenging task of QubesOS consists in allowing virtual machines to share data among each other and still maintain the overall system safe. For instance, those applications that are being run in different virtual machines might need their data to be shared; the web browser isolated in the social virtual machine will certainly use the network system, provided by the network virtual machine in which the NIC driver, the TCP/IP stack and 802.11 stack are running; most of the applications running in their own virtual machines will need the graphical subsystem for basic human-computer interaction, provided by the GUI virtual machine. All the required inter-connectivity is provided by the Xen hypervisor that takes advantage of hardware supported virtualisation technology. Despite the high demand in terms of hardware resources and the performance impact that depends on the number of virtual machines running simultaneously, QubesOS is one of the few systems that can guarantee strong isolation at application level.

Invincea A commercial product originally designed to isolate web browsers in virtual execution environments within a virtualisation platform is presented in [161]. Due to the proprietary nature of Invincea software, no details about its technical implementation are provided. However, the main goal of the product is to protect the host operating system from malware that targets mainly the web browser as a spreading medium. Whenever the browser protection detects a malware threat, the user is informed and the virtual environment in which the browser is executing is shut down. At this point a new disposable virtual machine is started in order to minimise user interruption. Hardware-supported virtualisation is used to provide the required separation. The main feature of Invincea consists in the fact that it relies on unusual behaviour of an application in the virtual environment rather than on malware signature. Moreover, the data collected during a malware attack is sent to centralised data servers to be analysed and to build a collective intelligence database that may protect other clients. This type of protection has been extended to PDF file readers, office suite, compressed and executable files,

in a more recent release of the product.

5.8 Summary

A recent heap-based attack to web browsers revealed to be effective and capable of circumventing countermeasures that have been specifically designed against these types of threats. Heap-spraying attacks expect to have large parts of the heap to be homogeneous, a requirement that is fulfilled a number of times in a complex application like the web browser. In this chapter we show that, by introducing heterogeneity where attackers expect this homogeneity, we can make heap-based buffer overflows a lot harder.

We provide a proof-of-concept, by modifying the Javascript engine internal representation of the string data type of a widely used web browser. We show how effective is introducing diversity against such attacks. This is done by inserting special values at random locations in the string, which will cause a breakpoint exception to occur if they are executed. Benchmarks show that this countermeasure has very limited impact, both in terms of performance at runtime and memory overhead.

In addition to the single-browser scenario we explain the case in which the target application is running in a virtualised operating system. We show how this countermeasure can be easily integrated with the virtualisation infrastructure and take advantage of hypervisor capabilities to protect other peers in a *attack-once-protect-everywhere* fashion.

Although we implemented a prototype for a widely adopted web browser, the concepts described in this chapter hold to many other scenarios like those in which interpreted language engines are embedded in user space applications and that are vulnerable to the same type of attack.

Chapter 6

Conclusions

People fear death even more than pain. It's strange that they fear death. Life hurts a lot more than death. At the point of death, the pain is over. Yeah, I guess it is a friend.

Jim Morrison

Protecting operating system kernels from malware that executes with the same privilege level is an extremely challenging task for which, at the moment, there is no winner either from the community of security researchers nor attackers. This fact is mainly due to the nature of the shared environment in which both trusted and malicious code operate. A mitigation to attacks with kernel level malware can be achieved when trusted code has been isolated and any external attempt to tamper with it will fail.

Virtualisation technology offers the aforementioned required isolation capabilities, but at the cost of an additional layer referred to as *hypervisor*, on top of which regular operating systems can be executed. Although we are aware of attacks to the hypervisor that can compromise the entire virtualisation infrastructure [162–165], we believe that such attacks can occur under assumptions that are stronger than the ones explained in this work, such as physical access or faulty hardware. We demonstrated how the isolation between a hypervisor and a guest operating system can be used to build a non-bypassable protection system against kernel-level malware. Despite some limitations of the described protection system, regarding the type of kernel code that can be protected, the overall attack surface of the target system results dramatically reduced, giving the attacker very few chances to launch malicious operations. In general, isolation is a common requirement of all protection strategies dealing with kernel security. In traditional

systems in which the countermeasure and the code to be protected share the same space and privileges, it is highly unlikely that trusted code will still be executed even after the kernel has been compromised. The claims made by researchers or secure software vendors about the effectiveness of these types of security measures can be achieved if and only if the isolation requirement is fulfilled.

We have contributed to this end by designing a framework that sets a protected environment within the target system and enforces the execution of trusted code from the hypervisor. The aforementioned enforcement of trusted code results decoupled from the target system. Therefore, there are no viable ways for an attacker to tamper with the countermeasure in order to postpone security checkings or to circumvent them completely. In our opinion, two important features, both present in the proposed framework, make it suitable for production systems: isolation and performance.

Isolation allows the execution of the trusted code even after the operating system has been compromised. Any countermeasure in place within a compromised kernel cannot be guaranteed to provide the functionality it has been designed for. This was not the case in our framework.

Moreover, by setting most of the secure environment within the target system, our framework can operate with an almost native performance impact. The countermeasures related to kernel attacks we propose in this work have been designed with performance and size of instrumentation code in mind. Despite the improvements in hardware supported virtualisation technology, we believe that another fundamental property that can impede a security measure's chances of competing in the marketplace is the complexity of integrating it into existing solutions. The hybrid approach - extensively used in the countermeasures presented in this work - of extending the target kernel with a trusted module that bridges communication to the hypervisor is revealed to be an effective strategy that can be considered even for those systems that cannot be modified for intellectual property reasons or because their source code is not available. Once the trusted module is no longer needed, it is unloaded from the target kernel in order to remove it from the attack surface and to reduce the chances of circumvention even further. Despite the recurrent mechanism provided by virtualisation technology by which the execution of the guest can be arbitrated by the hypervisor, finding a common pattern to the plethora of attacks that might compromise virtualised operating systems is challenging.

Another area we focus on belongs to the field of web browser security, a topic discussed in the second part of the thesis. Owing to the recent transformation of the web browser into one of the most important elements in today's computer usage and a recent *heap-based* attack that can circumvent some of the most effective countermeasures, we developed a lightweight security measure that not only revealed to be effective but it is also affected by negligible performance impact. We focused on *heap-spraying* attacks, by which objects of malicious content can

be allocated on the browser's heap via script languages such as Javascript. An essential requirement to deploy heap-spraying attacks successfully is homogeneity of data. When the malicious object has been allocated in the form of a homogeneous array, it becomes relatively easy for an attacker to forward the instruction pointer of the running system to a location within the aforementioned array and start execution of its content. By introducing diversity via random interruptions of special bytes we can successfully prevent the execution of malicious code stored on the heap.

Another challenging aspect when shifting to a newer technology involves designing countermeasures that can take advantage of the new features and can be easily integrated with the purpose of improving both performance and security. We contributed to this end by providing a strategy to integrate secure web browsers with hypervisor technology. This might have an impact in those cases in which applications are delivered on demand, as in a *desktop virtualisation* setting.

6.1 Future work

Despite the numerous contributions to tackling kernel malware, we believe that further research is needed in this area. In the virtualisation-based rootkit protection system described in Chapter 3, we are aware of a consistent limitation that restricts integrity checking to those kernel objects that stay invariant during the system lifetime. Although the proposed countermeasure provides an efficient protection against kernel mode rootkits, we believe that there is still room for improvement. We expect that rootkits with higher complexity might target *variable* data structures not only to circumvent a countermeasure like the one we described, but also to achieve a more complex behaviour and inflict further damage. Therefore, we suggest further research that leads to protecting critical kernel objects that are permitted to change during operating system lifetime, such as task structures created at runtime, structures that can be assigned to multiple values, dynamic kernel pointers, etc. as reported in [67]. The reader is likely to notice that protecting dynamic kernel data is a much more difficult task for which an approach different from the one proposed for static objects must be considered.

The trusted code enforcement framework proposed in this thesis can be used to perform the difficult task of rootkit prevention in a more dynamic way. This major flexibility is due to the fact that, despite the isolation layer, the trusted code is executing within the target operating system. The number of applications that may take advantage of such a framework is limited by the needs and reader's imagination. For instance, traditional signature-based anti-malware systems are effective only against known rootkits [166] and any attempt to protect against rootkits coded with a different style or targeting unprotected areas of the kernel might fail. Moreover, in case of attack, these systems can be deactivated by the malware itself since they are executing within the same system to be protected.

Such anti-malware systems can thus benefit from the secure framework we propose in order to operate within the target system and to stay isolated at the same time, making any attempt to be circumvented extremely difficult or not feasible at all.

To conclude, we identify an area that needs the attention of security researchers in light of virtualisation technology, namely *mobile computing*. The trend of mobile devices outselling traditional computers is a consistent evidence of the drift of computing experience in general [167–169]. Moreover, this popularity is stimulating the spread of malware specifically designed for operating systems that execute on mobile devices [13, 170–173].

We expect that virtualisation technology will affect the mobile arena in the near future. Hardware support to virtualisation might provide an entirely new way of thinking about security for mobile devices, similarly to what has been observed so far. For instance, the technology for executing corporate and personal identity on top of the same physical mobile device is already present [174]. The nature of mobile devices, usually equipped with limited hardware resources (such as less computational power, smaller storage capacity, finite battery life, etc), may place additional constraints on the design of those security countermeasures that will take advantage of mobile virtualisation technology.

Bibliography

[1] K. Musial and P. Kazienko, "Social networks on the internet," *World Wide Web*, vol. 16, no. 1, pp. 31–72, 2013.

[2] A. Datta, M. D. Dikaiakos, S. Haridi, and L. Iftode, "Infrastructures for online social networking services," *IEEE Internet Computing*, vol. 16, no. 3, pp. 10–12, 2012.

[3] W. Lehr, "Measuring the internet: The data challenge," OECD Digital Economy Papers 194, OECD Publishing, Apr. 2012.

[4] B. Edwards, T. Moore, G. Stelle, S. A. Hofmeyr, and S. Forrest, "Beyond the blacklist: Modeling malware spread and the effect of interventions," *CoRR*, vol. abs/1202.3987, 2012.

[5] M. R. Faghani, A. Matrawy, and C.-H. Lung, "A study of trojan propagation in online social networks," in *NTMS*, pp. 1–5, 2012.

[6] G. Yan, G. Chen, S. Eidenbenz, and N. Li, "Malware propagation in online social networks: nature, dynamics, and defense implications," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, (New York, NY, USA), pp. 196–206, ACM, 2011.

[7] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling, "Measurements and mitigation of peer-to-peer-based botnets: a case study on storm worm," in *LEET'08: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, USENIX Association, 2008.

[8] "Mcaffee: 2010 threat predictions." `http://mcafee.com/us/local_content/white_papers/7985rpt_labs_threat_predict_1209_v2.pdf`.

[9] Seclists.org, "Linux rootkit in combination with nginx."

[10] M. Janus, "New 64-bit Linux Rootkit Doing iFrame Injections." `https://www.securelist.com/en/blog/208193935/New_64_bit_Linux_Rootkit_Doing_iFrame_Injections`, 2012.

[11] "Avg community powered threat report 2012." http://www.scribd.com/doc/111016015/AVG-Community-Powered-Q3-Threat-Report-2012.

[12] T. Vidas, D. Votipka, and N. Christin, "All your droid are belong to us: a survey of current android attacks," in *Proceedings of the 5th USENIX conference on Offensive technologies*, WOOT'11, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2011.

[13] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, (Washington, DC, USA), pp. 95–109, IEEE Computer Society, 2012.

[14] R. Moreno-Vozmediano, R. Montero, and I. Llorente, "Key challenges in cloud computing to enable the future internet of services," *Internet Computing, IEEE*, vol. PP, no. 99, p. 1, 2012.

[15] K. K. Nguyen, M. Lemay, and M. Cheriet, "Enabling Infrastructure as a Service (IaaS) on IP Networks: From Distributed to Virtualized Control Plane," *IEEE Communications Magazine*, pp. 1+, Mar. 2012.

[16] C. Nielsen, "Towards a cloud computing strategy for europe: Matching supply and demand," 2011.

[17] T. Das, P. Padala, V. N. Padmanabhan, R. Ramjee, and K. G. Shin, "Litegreen: saving energy in networked desktops using virtualization," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2010.

[18] A. Beloglazov and R. Buyya, "Energy efficient resource management in virtualized cloud data centers," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, (Washington, DC, USA), pp. 826–831, IEEE Computer Society, 2010.

[19] R. Buyya, A. Beloglazov, and J. Abawajy, "Energy-efficient management of data center resources for cloud computing: A vision, architectural elements, and open challenges."

[20] D. M. Quan, R. Basmadjian, H. de Meer, R. Lent, T. Mahmoodi, D. Sannelli, F. Mezza, L. Telesca, and C. Dupont, "Energy efficient resource allocation strategy for cloud data centres," in *ISCIS*, pp. 133–141, 2011.

[21] S. Pandey, L. Wu, S. Guru, and R. Buyya, "A particle swarm optimization-based heuristic for scheduling workflow applications in cloud computing environments," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pp. 400 –407, april 2010.

[22] S. Chaisiri, B.-S. Lee, and D. Niyato, "Optimization of resource provisioning cost in cloud computing," *Services Computing, IEEE Transactions on*, vol. 5, pp. 164 –177, april-june 2012.

[23] S. Malakuti, S. te Brinke, L. Bermans, and C. Bockisch, "Towards modular resource-aware applications," in *Proceedings of the 3rd international workshop on Variability & Composition*, VariComp '12, (New York, NY, USA), pp. 13–18, ACM, 2012.

[24] Y. Zhang, G. Huang, X. Liu, and H. Mei, "Integrating resource consumption and allocation for infrastructure resources on-demand," in *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pp. 75 –82, july 2010.

[25] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, "Cost optimization of virtual infrastructures in dynamic multi-cloudscenarios," *Concurrency and Computation: Practice and Experience*, pp. n/a–n/a, 2012.

[26] M. Rosenblum, "The reincarnation of virtual machines," *Queue*, vol. 2, pp. 34–40, July 2004.

[27] R. Buyya, J. Broberg, and A. M. Goscinski, *Cloud Computing Principles and Paradigms*. Wiley Publishing, 2011.

[28] B. Blunden, *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. USA: Jones and Bartlett Publishers, Inc., 2009.

[29] S. Butt, V. Ganapathy, M. M. Swift, and C. cheng Chang, "Protecting commodity operating system kernels from vulnerable device drivers ."

[30] S. Boyd-Wickizer and N. Zeldovich, "Tolerating malicious device drivers in linux," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 9–9, USENIX Association, 2010.

[31] J. Navarro, E. Naudon, and D. Oliveira, "Bridging the semantic gap to mitigate kernel-level keyloggers," in *IEEE Symposium on Security and Privacy Workshops*, pp. 97–103, 2012.

[32] "Securing Legacy Firefox Extensions with SENTINEL," in *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 7 2013.

[33] Mozilla, "*Security Vulnerabilities Published in 2013 Mozilla Firefox and Thunderbird*," 2013.

[34] skypher.com, "Heap spraying," 2007. `http://skypher.com/wiki/index.php`.

[35] securiteam.com, "Heap spraying: Exploiting internet explorer vml 0-day xp sp2," 2009. `http://blogs.securiteam.com/index.php/archives/641`.

[36] www.milw0rm.com, "Safari (arguments) array integer overflow poc (new heap spray)," 2009. `http://www.milw0rm.com/exploits/7673`.

[37] Business Insider, "Amazon's Cloud Crash Disaster Permanently Destroyed Many Customers' Data." `http://articles.businessinsider.com/2011-04-28/tech/29958976_1_amazon-customer-customers-data-data-loss`, 2011.

[38] N. Nikiforakis, M. Balduzzi, S. Van Acker, W. Joosen, and D. Balzarotti, "Exposing the lack of privacy in file hosting services," in *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats (LEET)*, 2011.

[39] U. Gurav and R. Shaikh, "Virtualization: a key feature of cloud computing," in *Proceedings of the International Conference and Workshop on Emerging Trends in Technology*, ICWET '10, (New York, NY, USA), pp. 227–229, ACM, 2010.

[40] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974.

[41] L. Sgallari, "Reducing infrastructure costs through virtualization," 2009. `http://msdn.microsoft.com/en-us/architecture/aa902619.aspx`.

[42] D. Technologies, "The importance of virtualization's green benefits and cost savings." `http://www.derivetech.com/userfiles/files/Virtualizations_green_benefits_and_cost_savings_WP.pdf`.

[43] vmware Inc., "Software and hardware techniques for x86 virtualization," 2010.

[44] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, pp. 48–56, May 2005.

[45] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch, "Virtualize everything but time," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2010.

[46] X. Chen, J. Andersen, Z. Morley, M. Michael, and B. J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *In Proceedings of the International Conference on Dependable Systems and Networks*, 2008.

[47] Intel, *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Nov. 2007.

[48] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *SIGOPS Oper. Syst. Rev.*, vol. 42, pp. 95–103, July 2008.

[49] A. Gavrilovska, S. Kumar, H. Raj, K. Schwan, V. Gupta, R. Nathuji, R. Niranjan, A. Ranadive, and P. Saraiya, "Abstract high-performance hypervisor architectures: Virtualization in hpc systems."

[50] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, (New York, NY, USA), pp. 164–177, ACM, 2003.

[51] V. Inc., "Vmware. performance evaluation of intel ept hardware assist," 2008.

[52] V. Inc., "Vmware. large page performance: Esx server 3.5 and esx server 3i v3.5," 2008.

[53] vmware, "Virtualization overview whitepaper."

[54] VMWare, "A performance comparison of hypervisors." `http://www.cc.iitd.ernet.in/misc/cloud/hypervisor_performance.pdf`.

[55] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, pp. 19:1–19:36, Oct. 2009.

[56] D. Leinenbach and T. Santen, "Verifying the microsoft hyper-v hypervisor with vcc," in *Proceedings of the 2nd World Congress on Formal Methods*, FM '09, (Berlin, Heidelberg), pp. 806–809, Springer-Verlag, 2009.

[57] J. McDermott and L. Freitas, "A formal security policy for xenon," in *Proceedings of the 6th ACM workshop on Formal methods in security engineering*, 2008.

[58] S. Kumar, H. Raj, K. Schwan, I. Ganev, I. Sidecores, and S. H. For, "Ganev, re-architecting vmms for multicore systems: The sidecore approach."

[59] O. Agesen, J. Mattson, R. Rugina, and J. Sheldon, "Software techniques for avoiding hardware virtualization exits," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, (Berkeley, CA, USA), pp. 35–35, USENIX Association, 2012.

[60] "National institute of standards and technology, *National vulnerability database statistics*, `http://nvd.nist.gov/statistics.cfm`."

[61] "*The Xen hypervisor, the powerful open source industry standard for virtualization*, `http://www.xen.org`."

[62] L. John, "Spec cpu2000: Measuring cpu performance in the new millennium."

[63] T. Chiueh and F. H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *Proceedings of the 21st International Conference on Distributed Computing Systems*, (Phoenix, Arizona, USA), pp. 409–420, IEEE Computer Society, IEEE Press, April 2001.

[64] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-vm monitoring using hardware virtualization," in *Proceedings of CCS'09*, 2009.

[65] M. Grace, Z. Wang, D. Srinivasan, J. Li, X. Jiang, Z. Liang, and S. Liakh, "Transparent protection of commodity os kernels using hardware virtualization," in *6th International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, 2010.

[66] R. Riley, X. Jiang, and D. Xu, "Guest-transparent prevention of kernel rootkits with VMM-based memory shadowing," in *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, vol. 5230 of *LNCS*, pp. 48–67, Springer, 2008.

[67] J. Rhee, R. Riley, D. Xu, and X. Jiang, "Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring," in *ARES*, pp. 74–81, 2009.

[68] W. Stallings, *Operating Systems - Internals and Design Principles (7th ed.)*. Pitman, 2011.

[69] R. Hund, T. Holz, and F. C. Freiling, "Return-oriented rootkits: Bypassing kernel code integrity protection mechanisms," in *SSYM'09: Proceedings of the 18th conference on USENIX security symposium*, pp. 383–398, USENIX Association, 2009.

[70] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," 2003.

[71] D. A. S. d. Oliveira and S. F. Wu, "Protecting kernel code and data with a virtualization-aware collaborative operating system," in *ACSAC '09: Proceedings of the 2009 Annual Computer Security Applications Conference*, 2009.

[72] H. Yin, P. Poosankam, S. Hanna, and D. Song, "HookScout: Proactive binary-centric hook detection," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, vol. 6201 of *LNCS*, pp. 1–20, Springer, 2010.

[73] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "Secvisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity oses," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[74] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of CCS '07*, 2007.

[75] A. M. Nguyen, N. Schear, H. Jung, A. Godiyal, S. T. King, and H. D. Nguyen, "Mavmm: Lightweight and purpose built vmm for malware analysis," *Computer Security Applications Conference, Annual*, 2009.

[76] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of CCS '09*, 2009.

[77] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *In EuroSys 09: Proceedings of the 4th European Conference on Computer Systems*, 2009.

[78] Symantec, "Windows rootkit overview." `http://www.symantec.com/avcenter/reference/windows.rootkit.overview.pdf`.

[79] "Linux on-the-fly kernel patching without LKM, by sd and devik." Phrack Issue 58.

[80] PacketStorm. `http://packetstormsecurity.org/UNIX/penetration/rootkits/`.

[81] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants," 2010.

[82] M. Carbone, W. Lee, W. Cui, M. Peinado, L. Lu, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking," in *In ACM Conf. on Computer and Communications Security*, 2009.

[83] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin, "Robust signatures for kernel data structures," in *Proceedings of CCS '09*, 2009.

[84] Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B*, 2007.

[85] T. Shinagawa, H. Eiraku, K. Tanimoto, K. Omote, S. Hasegawa, T. Horie, M. Hirano, K. Kourai, Y. Oyama, E. Kawai, K. Kono, S. Chiba, Y. Shinjo, and K. Kato, "BitVisor: a thin hypervisor for enforcing I/O device security," in *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 121–130, ACM, 2009.

[86] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[87] R. Rivest, "The md5 message-digest algorithm," 1992.

[88] oblique, "setuid rootkit." `http://codenull.net/articles/kernel_mode_hooking.tar.gz`.

[89] "Apachebench: A complete benchmarking and regression testing suite."

[90] C. Staelin and L. McVoy, "lmbench manual page."

[91] O. K. labs, "Why lmbench is evil?." `http://www.ok-labs.com/blog/entry/why-lmbench-is-evil/`.

[92] N. L. Petroni, J. Timothy, F. Jesus, M. William, and A. Arbaugh, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *In Proceedings of the 13th USENIX Security Symposium*, 2004.

[93] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pp. 552–561, ACM, 2007.

[94] N. L. Petroni, Jr. and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, (New York, NY, USA), pp. 103–115, ACM, 2007.

[95] B. Zdrnja, "More tricks from conficker and vm detection," 2009.

[96] B. Zdrnja, "E-cards dont like virtual environments," 2007.

[97] Z. Wang, X. Jiang, W. Cui, and X. Wang, "Countering persistent kernel rootkits through systematic hook discovery," in *Proceeding of RAID '08*, 2008.

[98] H. Yin, Z. Liang, and D. Song, "Hookfinder: Identifying and understanding malware hooking behaviors," in *NDSS*, 2008.

[99] Stealth., "Adore-NG." `http://www.cs.dartmouth.edu/~sergey/cs258/rootkits/adore-ng`.

[100] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," pp. 233–247, May 2008.

[101] *Secure and Flexible Monitoring of Virtual Machines*, 2007.

[102] L. Liu, Z. Yin, Y. Shen, H. Lin, and H. Wang, "Research and design of rootkit detection method," *Physics Procedia*, vol. 33, no. 0, pp. 852 – 857, 2012. ¡ce:title¿2012 International Conference on Medical Physics and Biomedical Engineering (ICMPBE2012)¡/ce:title¿.

[103] Y. Chubachi, T. Shinagawa, and K. Kato, "Hypervisor-based prevention of persistent rootkits," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, (New York, NY, USA), pp. 214–220, ACM, 2010.

[104] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems," in *SOSP'07: Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pp. 351–366, ACM, 2007.

[105] P. Dewan, D. Durham, H. Khosravi, M. Long, and G. Nagabhushan, "A hypervisor-based system for protecting software runtime memory and persistent storage," in *SpringSim '08: Proceedings of the 2008 Spring simulation multiconference*, pp. 828–835, Society for Computer Simulation International, 2008.

[106] J. Rutkowska and R. Wojtczuk, "Qubes OS Architecture."

[107] F. Bellard, "QEMU open source processor emulator," 2012. `http://wiki.qemu.org/`.

[108] A. Kivity, "kvm: the Linux virtual machine monitor," in *OLS '07: The 2007 Ottawa Linux Symposium*, pp. 225–230, July 2007.

[109] L. McVoy and C. Staelin, "LMbench: Portable tools for performance analysis," in *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, (Berkeley, USA), pp. 23–39, USENIX Association, 1996.

[110] Y.-C. Lee, S. Rahimi, and S. Harvey, "A pre-kernel agent platform for security assurance," in *IEEE Symposium on Intelligent Agent (IA)*, pp. 1–7, IEEE, 2011.

[111] T. cker Chiueh, M. Conover, M. Lu, and B. Montague, "Stealthy deployment and execution of in-guest kernel agents," in *Proceedings of the Black Hat USA Security Conference*, 2009.

[112] D. Tian, Q. Zeng, D. Wu, P. Liu, and C. Hu, "Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring,"

[113] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the ACM European Conference in Computer Systems (EuroSys)*, pp. 315–328, ACM, Apr. 2008.

[114] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2010.

[115] R. Strackx and F. Piessens, "Fides: Selectively hardening software application components against kernel-level or process-level malware," in *Proceedings of the 19th ACM conference on Computer and Communications Security (CCS 2012)*, October 2012.

[116] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser, "Towards a practical, verified kernel," *usenix.org*.

[117] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, pp. 19:1–19:36, Oct. 2009.

[118] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: Formal verification of an OS kernel," in *22nd ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[119] F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-site scripting prevention with dynamic data tainting and static analysis," in *In Proceeding of the Network and Distributed System Security Symposium (NDSS07*, 2007.

[120] W. Zeller and E. W. Felten, "Cross-site request forgeries: Exploitation and prevention," *Bericht, Princeton University*, 2008.

[121] A. Barth, C. Jackson, and J. C. Mitchell, "Robust defenses for cross-site request forgery," in *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, (New York, NY, USA), pp. 75–88, ACM, 2008.

[122] M. Dalton, H. Kannan, and C. Kozyrakis, "Real-world buffer overflow protection for userspace & kernelspace," in *Proceedings of the 17th conference on Security symposium*, pp. 395–410, USENIX Association, 2008.

[123] P. Wagle and C. Cowan, "Stackguard: Simple stack smash protection for gcc," in *Proceedings of the GCC Developers Summit*, (Ottawa, Ontario, Canada), pp. 243–256, May 2003.

[124] H. Etoh and K. Yoda, "Protecting from stack-smashing attacks," tech. rep., IBM Research Divison, Tokyo Research Laboratory, June 2000.

[125] TMS, "Data execution prevention." `http://technet.microsoft.com/en-us/library/cc738483.aspx`.

[126] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," NDSS, 2003.

[127] S. Bhatkar, D. C. Duvarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proceedings of the 12th USENIX Security Symposium*, (Washington, D.C., U.S.A.), pp. 105–120, USENIX Association, August 2003.

[128] M. Labs, "Protecting your critical assets lessons learned from operation aurora," 2010. `http://www.mcafee.com/us/resources/white-papers/wp-protecting-critical-assets.pdf`.

[129] S. Berry-Bryne, "Firefox 3.5 heap spray exploit," 2009. `http://www.milw0rm.com/exploits/9181`.

[130] Y. Younan, W. Joosen, F. Piessens, and H. Van den Eynden, "Security of memory allocators for c and c+," tech. rep., Technical Report CW 419, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, 2005.

[131] Y. Younan, W. Joosen, and F. Piessens, "Code injection in c and c++ : A survey of vulnerabilities and countermeasures," tech. rep., Departement Computerwetenschappen, Katholieke Universiteit Leuven, 2004.

[132] U. Erlingsson, "Low-level software security: Attacks and defenses," Tech. Rep. MSR-TR-2007-153, Microsoft Research, November 2007.

[133] M. A. L. Blog, "New backdoor attacks using pdf documents," 2009. http://www.avertlabs.com/research/blog/index.php/2009/02/19/new-backdoor-attacks-using-pdf-documents/.

[134] F. M. I. Lab, "Heap spraying with actionscript," 2009. http://blog.fireeye.com/research/2009/07/actionscript_heap_spray.html.

[135] Intel, "Intel architecture software developer's manual. volume 2: Instruction set reference," 2002.

[136] Securitylab, "Adobe reader 0-day critical vulnerability exploited in the wild (cve-2009-0658)," 2009. http://en.securitylab.ru/nvd/368655.php.

[137] A. Sotirov, "Heap feng shui in javascript," 2007.

[138] M. Daniel, J. Honoroff, and C. Miller, "Engineering heap overflow exploits with javascript," in *WOOT'08: Proceedings of the 2nd conference on USENIX Workshop on offensive technologies*, (Berkeley, CA, USA), pp. 1–6, USENIX Association, 2008.

[139] www.packetstormsecurity.org, "25bytes-execve," 2009. http://www.packetstormsecurity.org/shellcode/25bytes-execve.txt.

[140] Jorendorff, "Anatomy of a javascript object," 2008. http://blog.mozilla.com/jorendorff/2008/11/17/anatomy-of-a-javascript-object.

[141] M. Foundation, "Firefox 3.5b4," 2009. http://developer.mozilla.org.

[142] E. C. M. A. International, *ECMA-262: ECMAScript Language Specification*. Geneva, Switzerland: ECMA (European Association for Standardizing Information and Communication Systems), third ed., December 1999.

[143] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: A defense against heap-spraying code injection attacks," tech. rep., Microsoft Research, Nov. 2008.

[144] www2.webkit.org, "Sunspider javascript benchmark," 2009. http://www2.webkit.org/perf/sunspider-0.9/sunspider.html.

[145] F. Corporation, "*Peacekeeper The Browser Benchmark*." http://service.futuremark.com/peacekeeper/.

[146] Google, "*V8 Benchmark Suite - version 5*." http://v8.googlecode.com.

[147] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith, "Vm-based security overkill: a lament for applied systems security research," in *Proceedings of the 2010 workshop on New security paradigms*, NSPW '10, (New York, NY, USA), pp. 51–60, ACM, 2010.

[148] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda, "Defending browsers against drive-by downloads : mitigating heap-spraying code injection attacks," in *DIMVA'09, 6th International Conference on Detection of Intrusions and Malware and Vulnerability Assessment, July 9-10, 2009, Milan, Italy, also published in Springer LNCS*, Jul 2009.

[149] A. Krennmair, "ContraPolice: a libc extension for protecting applications from heap-smashing attacks," Nov. 2003.

[150] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur, "Run-time detection of heap-based overflows," in *Proceedings of the 17th Large Installation Systems Administrators Conference*, (San Diego, California, U.S.A.), pp. 51–60, USENIX Association, Oct. 2003.

[151] S. Bhatkar and R. Sekar, "Data space randomization," in *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, vol. 5137 of *Lecture Notes in Computer Science*, (Paris, France), Springer, July 2008.

[152] C. Cowan, S. Beattie, J. Johansen, and P. Wagle, "PointGuard: protecting pointers from buffer overflow vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium*, (Washington, D.C., U.S.A.), pp. 91–104, USENIX Association, August 2003.

[153] S. Bhatkar, R. Sekar, and D. C. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *14th USENIX Security Symposium*, (Baltimore, MD), USENIX Association, August 2005.

[154] J. Xu, Z. Kalbarczyk, and R. K. Iyer, "Transparent runtime randomization for security," in *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, (Florence, Italy), pp. 260–269, IEEE Computer Society, IEEE Press, Oct. 2003.

[155] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS2003)*, (Washington, D.C., U.S.A.), pp. 281–289, ACM, October 2003.

[156] A. Anisimov, "Defeating microsoft windows xp sp2 heap protection and dep bypass." `http://www.ptsecurity.com`.

[157] Y. Younan, W. Joosen, and F. Piessens, "Efficient protection against heap-based buffer overflows without resorting to magic," in *Proceedings of the International Conference on Information and Communication Security (ICICS 2006)*, vol. 4307, Springer-Verlag, December 2006.

[158] Y. Younan, D. Pozza, F. Piessens, and W. Joosen, "Extended protection against stack smashing attacks without performance loss," in *Proceedings of the Twenty-Second Annual Computer Security Applications Conference (ACSAC '06)*, pp. 429–438, IEEE Press, December 2006.

[159] Vendicator, "Documentation for stackshield."

[160] Y. Cao, Z. Li, V. Rastogi, Y. Chen, and X. Wen, "Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security," in *ASIACCS*, pp. 8–9, 2012.

[161] I. Inc., "Protect your organization from its single largest threat: Cyber breach."

[162] R. Wojtczuk, "Sysret 64-bit operating system privilege escalation vulnerability on intel cpu hardware vu649219," 2012.

[163] A. Desnos, E. Filiol, and I. Lefou, "Detecting (and creating !) a hvm rootkit (aka bluepill-like)," *J. Comput. Virol.*, vol. 7, pp. 23–49, Feb. 2011.

[164] S. Embleton, S. Sparks, and C. Zou, "Smm rootkits: a new breed of os independent malware," in *Proceedings of the 4th international conference on Security and privacy in communication netowrks*, SecureComm '08, (New York, NY, USA), pp. 11:1–11:12, ACM, 2008.

[165] C. Gebhardt, C. I. Dalton, and R. Brown, "Hypervisors: Preventing hypervisor-based rootkits with trusted execution technology," *Netw. Secur.*, vol. 2008, pp. 7–12, Nov. 2008.

[166] C. Mahapatra and S. Selvakumar, "An online cross view difference and behavior based kernel rootkit detector," *SIGSOFT Softw. Eng. Notes*, vol. 36, pp. 1–9, Aug. 2011.

[167] D. Stone, "Mobile - more than a magic moment for marketers," 2012.

[168] A. Oulasvirta, T. Rattenbury, L. Ma, and E. Raita, "Habits make smartphone use more pervasive," *Personal and Ubiquitous Computing*, vol. 16, pp. 105–114, Jan. 2011.

[169] M. Ebling and M. Baker, "Pervasive tabs, pads, and boards: Are we there yet?," *Pervasive Computing, IEEE*, vol. 11, pp. 42 –51, january-march 2012.

[170] P. Traynor, M. Lin, M. Ongtang, V. Rao, T. Jaeger, P. McDaniel, and T. La Porta, "On cellular botnets: measuring the impact of malicious devices on a cellular network core," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, (New York, NY, USA), pp. 223–234, ACM, 2009.

[171] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, (New York, NY, USA), pp. 3–14, ACM, 2011.

[172] "Yet another android rootkit." `https://media.blackhat.com/bh-ad-11/Oi/bh-ad-11-Oi-Android_Rootkit-WP.pdf`.

[173] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications."

[174] "vmware inc., *vmware Horizon Mobile*, `http://www.vmware.com/products/desktop_virtualization/mobile/overview.html`."