
Securing the Virtual Infrastructure in the IaaS Cloud Computing Model

by
Amani S. Ibrahim

This thesis is submitted in fulfilment
of the requirements for
the degree of
Doctor of Philosophy.

Faculty of Science, Engineering and Technology
Swinburne University of Technology

June 2014

Abstract

With the advent of virtualization, the Infrastructure-as-a-Service (IaaS) cloud computing model has been widely deployed by organizations. One of the main issues with this model is the untrusted client virtual machines security problem. Unlike traditional physical servers that are directly managed and protected by their owners, the security of hosted virtual machines in the IaaS platform relies on the cloud customer, who might be a malicious hacker. Moreover, these hosted virtual machines are owned and controlled by a cloud consumer, and at the same time they are hosted by a cloud provider. Thus, both consumers and providers claim the right to provide the security for the hosted virtual machines from their own perspectives, causing a *loss-of-control* security problem.

In this research project, we address the *loss-of-control* security problem of the IaaS platform by introducing the concept of *virtualization-aware* security solutions. *Virtualization-aware* security solutions enable both cloud providers and consumers to fully utilize the virtualization advantages of the IaaS platform from both perspectives. Particularly, we introduce *CloudSec₊₊*, a security appliance that has the ability to provide active, transparent and real-time monitoring and protection for the running operating systems in the hosted virtual machines, from outside the virtual machines themselves. *CloudSec₊₊* enables provision of security from a cloud provider's perspective with no need to have any control over the hosted virtual machines, while passively incorporating consumers in maintaining the security of their hosted virtual machines. *CloudSec₊₊* has the ability to **systematically** protect multiple concurrent guest operating systems against zero-day threats that could target operating system's kernel. In particular, *CloudSec₊₊* has the ability to defend against different types of pointer manipulation attacks (whether known or unknown) that target runtime kernel dynamic data and memory.

Implementing such a *virtualization-aware* security solution is a multi-disciplinary research project, where a number of new mechanisms and techniques, in different research areas, were introduced to address different challenging problems. These mechanisms and techniques include: (i) a virtual machine introspection framework that enables active and transparent moni-

toring of the hosted virtual machines at a hypervisor level. (ii) A new points-to analysis algorithm and a kernel dynamic objects discovery tool to support the automation of overcoming the semantic gap problem for C-based operating systems, such as Windows, Linux and Solaris, without a prior knowledge of the operating system's runtime kernel data layout. (iii) A set of operating system runtime security tools that has the ability to defend against kernel dynamic data zero-day threats in a real-time fashion. (iv) A security virtual appliance that contains our developed *virtualization-aware* security solution that has the ability to monitor and protect multiple concurrent virtual machines hosted on an IaaS platform.

Acknowledgements

I would like to express my heartfelt gratitude and thanks to my supervisors, Prof. John Grundy and Dr. James Hamlyn-Harris, for their support, guidance, and mentorship over the past four years. I thank both of them for their steadfast encouragement and invaluable advice. They greatly supported me to hurdle all the obstacles in the completion of this research work. Honestly, I have always counted myself very lucky and blessed to have both of them as my supervisors. I also thank all the friends, colleagues, and staff at SUCCESS, especially Gillian Foster, for all their help and support during my PhD journey.

My highest and warmest gratitude, appreciation, and thanks to my parents, brother, sister and friends for their endless love, encouragement, and support to me to go through the ups and downs of my life including my PhD journey. A special feeling of gratitude to my mother who has been a source of encouragement and inspiration throughout my life. I also cannot forget to thank my kids Malak and Adam for being helpful and supportive in delaying me towards finishing this work ☺ but honestly I enjoyed the journey with them. I have done my best to be a good and friendly mum for them despite my responsibilities.

Last but not least, the love of my life, my husband, Mohamed. No words can express my love and sincere gratitude towards him. I could not achieve anything in my life and this thesis is not an exception without him. He has been a great supporter at work and at home, with endless love and encouragement. Very special thanks for your practical and emotional support.

Financial support for this research was provided by Swinburne University of Technology and VMsafe libraries research licence was provided by VMware.

This PhD thesis is dedicated to the memory of my dear father.

Declaration

Hereby I certify that, this thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of my knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Amani S. Ibrahim

Date

List of Publications

A large portion of this research project was published in a number of papers. Below is a list of my publications during my PhD journey relevant to this thesis topic:

1. Amani S. Ibrahim, John C. Grundy, James Hamlyn-Harris, Mohamed Almorsy, "DIGGER: Identifying Operating System Dynamic Kernel Objects for Run-time Security Analysis", International Journal on Internet and Distributed Computing Systems (IJIDCS), 2013.
2. Amani S. Ibrahim, John C. Grundy, James Hamlyn-Harris, Mohamed Almorsy, "Identifying OS Kernel Runtime Objects for Run-time Security Analysis", 6th International Conference on Network and System Security (NSS 2012), Nov 21-23 2012, Wu Yi Shan, Fujian, China, Springer.
3. Amani S. Ibrahim, John C. Grundy, James Hamlyn-Harris, Mohamed Almorsy, "Operating System Kernel Data Disambiguation to Support Security Analysis", 6th International Conference on Network and System Security (NSS 2012), Nov 21-23 2012, Wu Yi Shan, Fujian, China, Springer.
4. Amani S. Ibrahim, John C. Grundy, James Hamlyn-Harris, Mohamed Almorsy, "Supporting Operating System Kernel Data Disambiguation using Points-to Analysis", 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), Sept 3-7 2012, Essen, Germany, ACM Press.
5. Amani S. Ibrahim, James Hamlyn-Harris, John Grundy and Mohamed Almorsy, "Supporting Virtualization-Aware Security Solutions using a Systematic Approach to Overcome the Semantic Gap", 5th IEEE Conference on Cloud Computing (CLOUD 2012), IEEE CS Press, Waikiki, Hawaii, USA, June 24-29 2012.
6. Amani S. Ibrahim, James Hamlyn-Harris, John Grundy and Mohamed Almorsy, "CloudSec: A Security Monitoring Appliance for Virtual Machines in the IaaS Cloud Model", 5th International Conference on Network and System Security (NSS 2011), Milan, Italy on 6 – 8 September, 2011.
7. Amani S. Ibrahim, James Hamlyn-Harris and John Grundy, "Emerging Security Challenges of Cloud Virtual Infrastructure", In Proceedings of the 2010 Asia Pacific Cloud Workshop 2010 (co located with APSEC2010), Sydney, Nov 30 2010.
8. Mohamed Almorsy, John Grundy, and Amani S. Ibrahim, "Adaptive Security Management in SaaS Applications", Security, Privacy and Trust in Cloud Systems, Book Springer (in press).
9. Mohamed Almorsy, John Grundy, and Amani S. Ibrahim, "Adaptable, Model-driven Security Engineering for SaaS Cloud-based Applications", Automated Software Engineering Journal, to appear.

Other papers co-authored during my PhD:

1. Mohamed Almorsy, John C. Grundy, Amani S. Ibrahim, "MDSE@R: Model-Driven Security Engineering at Runtime", The 4th International Symposium on Cyberspace Safety and Security (CSS 2012), Dec 12-13 2012, Melbourne, Australia.
2. Mohamed Almorsy, John Grundy, and Amani S. Ibrahim, "Automated Software Architecture Security Risk Analysis Using Formalized Signatures", 2013 IEEE/ACM International Conference on Software Engineering (ICSE 2013), San Francisco, May 2013, IEEE CS Press.
3. Mohamed Almorsy, John C. Grundy, Amani S. Ibrahim, "VAM-aaS: Online Cloud Services Security Vulnerability Analysis and Mitigation-as-a-Service", The 13th International Conference on Web Information System Engineering (WISE 2012), Nov 28-30 2012, Paphos, Cyprus, Springer.
4. Mohamed Almorsy, John C. Grundy, Amani S. Ibrahim, "Supporting Automated Vulnerability Analysis using Formalized Vulnerability Signatures", 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), Sept 3-7 2012, Essen, Germany, ACM Press.
5. Mohamed Almorsy, John C. Grundy, Amani S. Ibrahim, "Supporting Automated Software Re-Engineering Using "Re-Aspects"", 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), Sept 3-7 2012, Essen, Germany, ACM Press.
6. Mohamed Almorsy, John Grundy and Amani S. Ibrahim, "TOSSMA: A Tenant-Oriented SaaS Security Management Architecture", 5th IEEE Conference on Cloud Computing (CLOUD 2012), IEEE CS Press, Waikiki, Hawaii, USA, June 24-29 2012.
7. Mohamed Almorsy, John Grundy and Amani S. Ibrahim, "SMURF: Supporting Multi-tenancy Using Re-Aspects Framework", 17th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2012), Paris, France, July 2012, IEEE CS Press.
8. Mohamed Almorsy, John Grundy and Amani S. Ibrahim, "Collaboration-Based Cloud Computing Security Management Framework", In Proceedings of 2011 IEEE International Conference on Cloud Computing (CLOUD 2011), Washington DC, USA on 4 – 9 July, 2011, IEEE.

Table of Contents

Abstract	II
Acknowledgements	IV
Declaration	V
List of Publications	VI
List of Figures	XI
List of Tables	XIV
Chapter 1 Introduction	1
1.1 Research Background and Motivation	1
1.1.1 Virtualization-Related Technical Problems	2
1.1.2 Operating System Security-Related Technical Problems	4
1.2 Key Research Contributions	6
1.3 Thesis Outline	8
Chapter 2 The IaaS Security Problem	11
2.1 Cloud Computing	11
2.2 The Virtualization Double-Edged Sword	13
2.2.1 Virtualization Benefits	14
2.2.2 The Downside: Virtualization Security Risks	17
2.2.3 Virtual Machine Security	18
2.3 Operating Systems and the “Pointer Problem”	24
2.4 Points-to Analysis	27
2.5 Summary	32
Chapter 3 Literature Review	33
3.1 Introduction	33
3.2 Cloud Computing Security	35
3.2.1 IaaS Security	36
3.3 Operating System Security	47
3.3.1 Kernel Data Rootkits	49
3.3.2 Operating System Verification and Unsafe C Memory	57
3.4 Related Work Limitations Summary	67
Chapter 4 Developing a Virtualization-Aware Security Solution for IaaS Platforms	71
4.1 Introduction	71
4.2 A New Virtualization-Aware Security Solution	73
4.2.1 Active Monitoring	75
4.2.2 Semantic Gap	76
4.2.3 Uncovering Kernel Dynamic Objects	77
4.2.4 Kernel Integrity Checks	78
4.2.5 Security Appliance Deployment	79
4.3 Summary	80

Chapter 5	CloudSec: A Virtual Machine Introspection Framework	81
5.1	Introduction.....	81
5.2	Threat Model.....	82
5.2.1	Design Requirements.....	84
5.3	CloudSec High-Level Architecture	85
5.3.1	CloudSec Operation.....	88
5.4	CloudSec Implementation Details.....	90
5.4.1	Memory Management.....	91
5.4.2	Implementation Details.....	94
5.4.3	CloudSec Deployment Model.....	102
5.5	CloudSec Evaluation Details	102
5.6	Discussion.....	107
5.7	Summary	110
Chapter 6	OS-KDD: Operating System Kernel Data Disambiguator	111
6.1	OS-KDD Overview.....	111
6.2	Type-Graph Description.....	113
6.2.1	OS-KDD Representation Language.....	115
6.3	Type-Graph Construction.....	116
6.3.1	Source Code Transformation	118
6.3.2	OS-KDD Points-to Analysis Overview	119
6.4	Points-to Analysis Algorithm	127
6.4.1	Intraprocedural Analysis.....	127
6.4.2	Interprocedural Analysis.....	133
6.4.3	Graph Unification	134
6.4.4	Context-Sensitive Points-To Analysis	135
6.5	Implementation and Evaluation	139
6.5.1	Soundness and Precision Experiment.....	140
6.5.2	Operating System Kernel Analysis Experiment	143
6.6	Discussion.....	147
6.7	Summary	149
Chapter 7	DIGGER: A Kernel Runtime Objects Discovery Tool	151
7.1	Overview.....	152
7.2	DIGGER High-Level Architecture	153
7.2.1	Signature Extraction Component.....	154
7.2.2	Dynamic Memory Analysis Component.....	161
7.3	Implementation and Evaluation	165
7.3.1	Uncovering Runtime Objects	166
7.4	Discussion.....	169
7.5	Summary	174
Chapter 8	CloudSec₊₊: A virtualization-Aware Security Solution	175
8.1	Introduction.....	175
8.2	CloudSec ₊₊	176
8.2.1	Systematic Kernel Version Inference.....	177

8.2.2	CloudSec++ High-Level Architecture	179
8.3	Kernel Data Integrity Checking Tools	184
8.3.1	Hidden Dynamic Objects Detection Tool.....	187
8.3.2	Function Pointer Hooking Detection Tool	194
8.3.3	Dangling Pointers Detection Tool	202
8.3.4	Memory Forensics.....	206
8.4	Summary	210
Chapter 9	Conclusions and Future Work	211
9.1	Key Conclusions.....	211
9.1.1	CloudSec	211
9.1.2	OS-KDD	212
9.1.3	DIGGER	214
9.1.4	CloudSec++	215
9.2	Future Research.....	216
References		219

List of Figures

Figure 2-1. Server virtualization.	14
Figure 2-2. A summary of virtualization benefits.....	15
Figure 2-3. Virtual machine introspection at the hypervisor level.	19
Figure 2-4. Semantic gap problem.	21
Figure 2-5. Doubly-linked list implementation in Windows and Linux.	26
Figure 2-6. Casting example in C.....	27
Figure 3-1. Related work analysis structure.	34
Figure 4-1. The shard-responsibility security problem in the IaaS cloud platform.	72
Figure 4-2. The high-level process of developing our virtualization-aware security solution.	74
Figure 4-3. A simple view of how CloudSec works at a hypervisor level.....	76
Figure 5-1. CloudSec high-level architecture.....	87
Figure 5-2. The linear address translation mechanism [210].....	93
Figure 5-3. The relation between guest virtual addresses and host physical addresses.	94
Figure 5-4. A type-graph reflecting relations data structures and their members.	94
Figure 5-5. Pointer relations between data structures.....	95
Figure 5-6. A snapshot of EPROCESS structure from Windbg.	97
Figure 5-7. EPROCESS doubly-linked list in Windows operating system.	97
Figure 5-8. The manually-developed EPROCESS kernel data definition.	100
Figure 5-9. EPROCESS semantic gap construction.	101
Figure 5-10. KeServiceDescriptorTable reconstruction algorithm.	102
Figure 5-11. The experimental setup lab.	103
Figure 5-12. The internal view of a virtual machine using Windbg.	104
Figure 5-13. The internal view of the loaded modules of the <i>csrss.exe</i> process.	104
Figure 5-14. CloudSec external view of the <i>csrss.exe</i> process.....	104
Figure 5-15. The internal view of the SSDT using Windbg.	104
Figure 5-16. CloudSec external view of the SSDT.	105
Figure 5-17. SSDT hooking.	106

Figure 5-18. Process hiding detection.....	106
Figure 5-19. DLL Injection.....	106
Figure 5-20. Function hooking	106
Figure 5-21. DLL module hiding.....	106
Figure 5-22. CloudSec performance overhead.	107
Figure 6-1. OS-KDD Representation Language	116
Figure 6-2. High-level view of OS-KDD framework.	117
Figure 6-3. Snapshots of C ode snippets and their ASTs	120
Figure 6-4. Direct Inclusion-Based Relations Analysis.	123
Figure 6-5. Different snapshots of the direct relations type-graph for Windows research kernel.....	124
Figure 6-6. A motivating example in C language reflecting generic pointers and casting problems.	126
Figure 6-7. Intraprocedural analysis graph.	132
Figure 6-8. Interprocedural analysis graph.	134
Figure 6-9. Graph unification results.	135
Figure 6-10. Transitive closure computations results	138
Figure 6-11. Strongly connected components in a directed graph cycle.....	138
Figure 6-12. Kernel Source Code Analysis Sequence.	139
Figure 6-13. A snapshot of OS-KDD while analysing one of the program benchmarks.	141
Figure 7-1. The high-level architecture of DIGGER approach.....	153
Figure 7-2. A snapshot of Poolmon tool depicts a set of pool tags of Windows operating system.....	156
Figure 7-3. ExAllocatePoolWithTag pool memory allocation routine.	156
Figure 7-4. Object dispatcher header data structure.....	157
Figure 7-5. A snapshot from Windbg for the dispatcher header structure.....	157
Figure 7-6. The location of the object dispatcher header within the object structure.	158
Figure 7-7. The memory layout of allocated objects in the pool memory.	159
Figure 7-8. A flowchart summarizing the scanning algorithm of DIGGER.	165
Figure 7-9. Object details extraction normalized time.	169
Figure 7-10. A Snapshot of the slab allocator info file of Linux kernel.....	171
Figure 7-11. A partial list of important fields within the kmem_cache_t structure.....	171

Figure 7-12. slab_s data structure.....	172
Figure 7-13. Proposed DIGGER Implementation for Linux operating systems.	173
Figure 8-1. Robust signatures of EPROCESS data structure according to Brendan 's results.	178
Figure 8-2. CloudSec++ High-level architecture.....	180
Figure 8-3. CloudSec++ deployment model.	183
Figure 8-4. Object details extraction normalized time in seconds.....	184
Figure 8-5. A snapshot from WinObj of the different object types supported by Windows OS.	185
Figure 8-6. Windows operating system processes structured in a doubly linked list.	188
Figure 8-7. DKOM mechanism for hiding runtime objects.	188
Figure 8-8. IAT and EAT function pointer hooking.	195
Figure 8-9. Hooking SSDT system call table of Windows operating system.....	196
Figure 8-10. A function pointer example in Windows operating system.....	196
Figure 8-11. Absolute and relative memory addresses	198
Figure 8-12. Function pointers age at runtime.	198
Figure 8-13. Average time to locate function pointers in different object types.....	198
Figure 8-14. False alarm rates of locating function' pointers.	199
Figure 8-15. Dangling pointers example.....	202
Figure 8-16. A snapshot of the object header structure.	205
Figure 8-17. A snapshot of Windbg reflecting our analysis.....	205
Figure 8-18. Dangling pointers detection time in seconds.	206
Figure 8-19. A sample infection-graph.....	208

List of Tables

Table 2-1. Example showing the difference between Steensgaard's and Andersen's algorithms.	30
Table 3-1. A comparison between key related work in operating system kernel data integrity.	53
Table 3-2. Key limitations in current related work.	68
Table 6-1. Transfer function description.	131
Table 6-2. Soundness and precision results of OS-KDD on a suite of benchmark programs.	142
Table 6-3. Kernel source code analysis.	143
Table 6-4. A comparison between OS-KDD type-graphs and some kernel data value-invariants.	145
Table 7-1. Description of some fields in pool header and object header structures.	160
Table 7-2. Paged and non-paged pools size in different Windows operating system versions.	162
Table 7-3. Experimental results of DIGGER and WD on Windows XP 32 bit and 64bit.	167
Table 7-4. slabinfo file columns description.	171
Table 7-5. The description of kmem_cache_t structure fields.	172
Table 8-1. D-Hide evaluation results.	192
Table 8-2. FP-Hook evaluation results of detecting function pointers.	201
Table 8-3. D-Pointer evaluation results.	206

Chapter 1

Introduction

This chapter provides an overview of this thesis. It starts with a summary of the research background and motivation from which our research derived its key goals and objectives. Subsequently, our research contributions are outlined, followed by a brief description of the approaches and techniques employed to fulfil the objectives of this research. Finally, the thesis outline is presented summarizing the thesis chapter structure.

1.1 Research Background and Motivation

Cloud computing is a new computing paradigm that enables delivering reliable and scalable internet-based services. Infrastructure-as-a-Service (IaaS) is one of the main services delivered by the cloud platform that allows regular users to increase their computational and storage resources on the fly. IaaS is characterized by the concept of resource virtualization that enables running multiple virtual machines on the same physical server. Although virtualization has a great role in supporting the IaaS model, it makes the virtual infrastructure of the IaaS platform a target for potent security threats because of the “loss-of-control” problem.

One of the significant risks in the Infrastructure-as-a-Service (IaaS) cloud computing model is outsourcing virtual machines control to cloud consumers. In particular, the hosted virtual machines are owned and controlled by a cloud consumer, and at the same time they are hosted by a cloud provider. Therefore, providers host and run virtual machines but are not aware of their running contents, as these virtual machines are controlled by the consumers, causing a *loss-of-control* security problem. This makes the consumers and the hosted virtual machines untrusted entities from the provider’s perspective to provide sufficient protection using traditional *in-guest* security software, as such security software can be subverted by advanced kernel rootkits. Moreover, the absolute and public accessibility and sharing of the IaaS platform with the regular consumers (virtual machines owners who might include malicious hackers) has

increased the security risks and the attack surface that facilitates exploitation of vulnerabilities, not just in the hosted virtual machines but also in the underlying virtual infrastructure of the IaaS platform. These security problems raise the need for new security solutions that have the ability to protect the hosted virtual machines from outside the virtual machine itself. Such a security system would be an important step towards securing the underlying virtual infrastructure of the IaaS platform, not just the hosted virtual machines, as the hosted virtual machines are the only source of access to the virtual infrastructure of the IaaS platform.

In this research project, we introduce the concept of *virtualization-aware* security solutions that enable both cloud providers and consumers to fully utilize the virtualization advantages, and provide protection of the hosted virtual machines, from both perspectives. A number of challenging technical problems need to be addressed proficiently to enable delivering a robust security application. Some of these problems are related to the IaaS platform complexity, in particular are related to the virtualization characteristic that complicates the security process, and the others are related to the complexity of operating system security software, as discussed in section 1.1.1 and 1.1.2, respectively.

1.1.1 Virtualization-Related Technical Problems

Virtualization technology provides important characteristics that make it a very practical platform to implement *virtualization-aware* security solutions, by utilizing *virtual machine introspection* techniques. *Virtual machine introspection* techniques enable monitoring the hosted virtual machines from outside the virtual machines, at the hypervisor level. A hypervisor is a thin layer of firmware code that runs on the bare hardware to perform dynamic resource sharing to enable hosting multiple operating system instances, in the form of virtual machines, on the same physical server. Developing security applications that work at the hypervisor level is a prominent security model that enables provisioning robust security against advanced operating system security threats. However, there is a number of challenging problems that hinder the implementation of such *virtualization-aware* security systems, discussed below.

First, a key problem of working at the hypervisor level is that security tools can only view hardware bytes such as physical memory pages and disk blocks, which do not reflect any information about the internal state of a running operating system. This is in contrast to the internal view of a virtual machine, where we can view information about high-level operating system entities based on the kernel memory and APIs, such as running processes, threads, and system calls. This difference in views causes a “semantic gap” problem. In order to make *virtual machine introspection* useful, it is necessary to translate the hardware bytes into a model of actual running high-level operating system information. The main difficulty in doing this lies in the complexity of the operating system’s kernel runtime data layout. The kernel of a C-based operating system, such as Linux, Windows, UNIX and Solaris, contains thousands of heterogeneous data structures with direct and indirect pointer-based relations among each other, with no explicit integrity constraints. Such operating systems utilize data structures heavily to model runtime objects, and also use pointers (especially generic pointers) extensively to guarantee high and efficient performance. Such complex implementation of the kernel data layout makes the process of overcoming the *semantic gap* very challenging. Current research efforts for overcoming the *semantic gap* problem are mainly based on manual approaches, to statically disambiguate the kernel data layout to overcome the *semantic gap*. Such manual approach is very limited for several reasons¹ to cope with such complex layout of operating systems’ kernels. The limitations of the manual approach result in an inability to get a complete and accurate interpretation of the low-level hardware bytes, making the security software unreliable. Thus, new mechanisms are required to enable **robust** and **systematic** overcoming of the *semantic gap* problem to support implementing *virtualization-aware* security solutions.

Second, active and transparent monitoring is a key feature of any robust security software in order to enable real-time prevention of system threats. Active monitoring relies on the capability of interrupting system and memory call events for inspection. Most current research efforts implement active monitoring by installing protected security hooks in the virtual ma-

¹ Limitations are discussed in chapter 2, in detail.

chines' running operating systems. However such an approach is not reliable and also not supported in the IaaS model, as the virtual machines are totally under the consumers' control with no management from the provider's side. Enabling active monitoring from outside the virtual machine without placing any security code is another mandatory design challenge in developing *virtualization-aware* security solutions.

Third, the cloud architecture is very dynamic and the workload changes dramatically over time, due to the creation, removal and suspension of virtual machines. Moreover, the mobile nature of the virtual machines that allows them to migrate from one server to another leads to a non-predefined network topology. Such a complex and huge amount of workload flowing inside each physical server increases the complexity of the protected environment, and requires the security software to be able to efficiently cope with an environment that is changing over time. On the other hand, the security software should be able to protect different operating system versions with the same running software instance, as the hosted virtual machines do not just run different kernels; they also run different operating systems such Linux and Windows. A major problem with current security software is their limitation to a specific operating system and sometimes a specific kernel build. Coping with such heterogeneous environment requires developing new security solutions that can abstract the underlying protected platform from the design of the security software itself, in order to support generic protection for physical servers in the IaaS model.

1.1.2 Operating System Security-Related Technical Problems

In addition to the technical problems that relate to the IaaS platform complexity, there exists another set of challenges that relate to protecting operating systems efficiently against known and unknown system threats and the design of security software itself, which include:

First, C-based operating systems have a very complex runtime layout due to the wide use of pointers and data structures, which assist hackers to exploit vulnerabilities to take control of the running operating system. Such complex layout is not only a problem in overcoming *semantic gap*; it is also a problem in checking the integrity of the running operating system's

kernel. Verifying pointer-dereferencing operations of generic pointers at runtime to guarantee runtime memory integrity is a serious challenge. This challenge comes from the fact that generic pointers only get their values and types at runtime according to the calling contexts of the running operating system. Thus runtime integrity constraints cannot be extracted from the operating system's source code or even at its compilation time, making the process of ensuring the reliability and integrity of the running operating system's kernel very challenging. Furthermore, kernel integrity checks are complicated because of kernel dynamic objects. Kernel dynamic objects change at runtime in location, value and the number of running instances. Unfortunately, they violate consistency constraints that cannot be extracted from the operating system's kernel source code directly, as their attributes depend on the calling contexts of the running system [1]. This makes the process of uncovering the running instances of kernel dynamic objects another challenge in order to monitor and protect these objects.

Second, another common technical problem in any security software is the performance overhead of the running security software. Ideally, performance is affected directly by the robustness of the protection. Lightweight software usually provides weak protection, and vice versa. Running security software dramatically impacts system performance as the security software typically needs to trap system calls and events before allowing execution. Trapping every system activity in the running operating system is not practical to support real-time monitoring and protection. The most important thing in developing robust security software with a reasonable performance overhead is the recognition of the critical system components. Consequently, the most important question is how we can identify the criticality of each system component and based on what criteria. A balance between the sufficient protection and low performance overhead cannot be easily implemented and new security countermeasures and techniques need to be developed to support this.

Third, malware writers usually try their best to exploit "zero-day" vulnerabilities, *i.e.* a never-seen-before exploit, in current operating systems. What makes it easier for them to do this is the complex operating system kernel implementation that does not have sufficient integrity checking controls. On the other hand, in order to rectify a new vulnerability, a security company

usually needs to release a software update or a patch to minimise the threat consequences. Such an approach leaves the operating system unprotected for a while, and consequently leaves the whole platform vulnerable to an attack risk. Thus, robust security systems should ideally be self-defending and threats self-detectable, and self-mitigatable. However, the ability to defend against something that is as yet unknown is very challenging and requires new automated security mechanisms that can monitor all operating system runtime entities to check for anomalies and unusual behaviours. Furthermore, robust security software needs to be able to cope with new operating system updates and new released kernel versions in a quick and an easy way, without the need to wait for manual analysis by security experts to release software updates.

1.2 Key Research Contributions

In this thesis, we introduce a solution for the *loss-of-control* security problem inherent in the IaaS cloud platform. Our proposed solution effectively addresses the *loss-of-control* problem over the hosted virtual machines and at the same time enables delivering robust protection for the running operating systems of the virtual machines, from a cloud provider's perspective. The proposed security solution also has the intelligence to systematically recognize operating system's layout to solve the *semantic gap* and perform systematic integrity checks, without a need to the manual experts' knowledge. Below we summarize the key contributions that we have achieved in this research project in order to develop and implement the proposed *virtualization-aware* security solution.

First, the main and first component in order to provision external security at a hypervisor level is building an introspection framework to solve the *semantic gap* and actively monitor the hosted virtual machines. To achieve this we introduce *CloudSec*. *CloudSec* is a new *virtual machine introspection* framework that provides fine-grained inspection of the hosted virtual machines' physical memory. A key feature of *CloudSec* is the ability to provide active monitoring for the hosted virtual machines without installing any security hooks inside these virtual machines. Active and transparent monitoring in *CloudSec* is efficiently achieved by moving the security hooks from the operating system level of the hosted virtual machines to the hypervisor

level. This guarantees more reliability and trustworthiness of the security software, as the hypervisor is considered part of our trusted platform.

Second, in order to enable systematic and proficient solution for the semantic gap problem and kernel integrity checks, we present OS-KDD. OS-KDD is a new *points-to* analysis tool that has the ability to systematically build accurate kernel data definitions that reflect the pointer-based relations of any C-based operating system, without a prior knowledge of the operating system's kernel data layout. OS-KDD disambiguates all indirect pointer relations, including generic pointers, by performing static *points-to* analysis on the operating systems' kernel source code. In OS-KDD, precision is an important factor, thus we designed and implemented a new *points-to* analysis algorithm that has the ability to provide field, flow and context-sensitive *points-to* analysis for large C programs that contain millions lines of code (LOC), such as large operating system kernels. Our *points-to* analysis algorithm has four analysis phases: *intraprocedural analysis*, *interprocedural analysis*, *graph unification* and *context-sensitive analysis*. In the intraprocedural analysis phase we create graph nodes and connect the basic edges by computing the transfer functions that describe the modification side effect of the program parts. In the interprocedural analysis phase we compute the internal *points-to* relations between procedure nodes and their call sites using the summary-based approach. In the graph unification analysis phase, we ensure that we have a balanced graph to achieve precise field-sensitivity. Finally, we achieve context-sensitivity that enables us disambiguate the generic pointer relations and casting operations. A key feature of OS-KDD is the usage of abstract syntax trees as the basis for implementing our *points-to* analysis algorithm to improve the time and memory usage efficiency of the analysis, and scale to large programs of million LOC.

Third, locating kernel runtime objects from a trusted source is a key to implement different operating system applications including *virtualization-aware* security solutions. To achieve this, we present DIGGER; DIGGER is a hybrid mechanism that combines a new *value-invariant* approach and an advanced *memory mapping* technique, in order to enable accurate discovery for kernel runtime objects with fast and nearly complete coverage of the kernel address space. DIGGER mainly has two key features that distinguish it from the current approaches. DIGGER

has the ability to systematically discover nearly all kernel runtime objects, with no prior knowledge of the runtime data layout of the operating system's kernel. DIGGER has fast coverage and a low performance overhead on a running operating system to uncover the whole kernel address space to locate the running instances of kernel dynamic objects.

Fourth, we introduce a set of runtime kernel data integrity checking tools that enable detecting various kinds of pointer manipulation malware (known and unknown) that target kernel data. These tools have the ability to check the integrity of kernel static and dynamic data at runtime against the following types of malware: dangling pointers, function pointer hooking and direct kernel object manipulation rootkits. We also introduce an offline memory forensics tool that analyses the physical memory for rootkit infection evidence for further investigations about the rootkit and its modifications effect in the kernel address space.

Fifth, the final task in this research project was integrating the previously developed components into a *virtualization-aware* security solution. We present *CloudSec₊₊* that provisions security from a cloud provider's perspective while incorporating consumers in a passive way in maintaining the security of their hosted virtual machines. *CloudSec₊₊* has the ability to efficiently and systematically provide protection for multiple concurrent hosted virtual machines running different operating system versions with a single running instance of *CloudSec₊₊*. *CloudSec₊₊* has a reasonable performance overhead that does not affect the operations of the protected virtual machines. *CloudSec₊₊* efficiently addresses all the technical problems discussed previously in section 1.1.1 and 1.1.2.

1.3 Thesis Outline

This thesis addresses the problem of developing a *virtualization-aware* security solution that has the ability to provide the pre-emptive protection for the running operating systems of the hosted virtual machines in the IaaS cloud platform. Our solution for this research project was based on developing different new components and techniques, in order to be able to achieve our research objectives and deliver the proposed security solution.

In chapter 2, we discuss the security problem in cloud computing technology, especially the Infrastructure-as-a-Service model. We mainly focus on the security implications of adapting virtualization technology for the cloud platform, such as the *loss-of-control* security problem over the hosted virtual machines and the complex and heterogeneous workload flowing in the platform servers. We also discuss how the security process of operating systems running in the hosted virtual machines have changed and why this has necessitated new security deployment models to cope with the new security requirements of the virtualization technology. In addition, we discuss the operating system's kernel data complexity problem and its role in assisting hackers to exploit new vulnerabilities in the running operating system.

In chapter 3, we discuss the related work of this research project along with the key limitations in each research area. In this research project a number of different research problems have been addressed with new solutions based on the limitations of the current approaches. In this chapter, we survey the literature in several research areas including cloud computing and virtualization security, operating system security, kernel rootkits and integrity checks, memory bugs and errors, and static program analysis techniques including *points-to* analysis. We then summarize key limitations of each research category and how these limitations can affect the robustness of the proposed solution of this research project.

In chapter 4, we summarize the whole approach used in this research project that will be explained in detail in the later chapters. In particular, we give an overview of the big picture of our research project to develop a *virtualization-aware* security solution and how the developed components will be integrated together to achieve the intended objective of the proposed security software of the IaaS platform.

In chapter 5, we introduce *CloudSec*, a *virtual machine introspection* framework that has the ability to actively monitor running virtual machines in a near real-time fashion. In this chapter, we explore the high-level architecture of *CloudSec*, the process of enabling active and transparent monitoring, and our approach to overcome the *semantic gap* of the hosted virtual machines. We also discuss the implementation and evaluation results of *CloudSec*. Finally, we discuss key limitations of *CloudSec* and how these limitations are addressed in the following

chapters to enable efficient protection of the running operating systems.

Chapter 6 discusses one of the key components of this research project, OS-KDD. OS-KDD is a new *points-to* analysis tool that enables analysing the source code of large-scale C programs, such as C-based operating systems, in order to solve the ambiguity of the indirect *points-to* relations of system data. OS-KDD enables systematically building accurate kernel definitions for C-based operating systems, without a prior knowledge of the operating system's kernel data layout. In this chapter, we discuss the high-level architecture of OS-KDD, the *points-to* analysis algorithm that enables performing the field, flow and context-sensitive analysis, and the implementation and evaluation details of OS-KDD.

Chapter 7 covers another key component in this research project, DIGGER. DIGGER is a new approach and tool that enables accurate, fast and nearly complete coverage of the operating system kernel runtime objects. In this chapter, we discuss the high-level architecture of DIGGER, the process of extracting efficient *value-invariants* of kernel objects, the *pool-memory tagging schema*, and the runtime memory scanner component that uncover the presence of kernel dynamic objects' running instances, along with the implementation and evaluation details, of each component of DIGGER.

In chapter 8, we discuss *CloudSec₊₊*, a security virtual appliance that mainly depends on *CloudSec*, OS-KDD and DIGGER in order to provide systematic security for operating system kernel data of the hosted virtual machines. In this chapter, we also discuss a set of security and memory forensics tools that are deployed in *CloudSec₊₊* in order to defend against zero-day threats that target operating system's kernel dynamic data, such as object hiding, dangling pointers and function pointer manipulation.

Finally in chapter 9, we summarize the key conclusions from this research project and then give an overview of promising future work that could be done in order to make our approaches and tools even more effective.

Chapter 2

The IaaS Security Problem

In this chapter, we discuss the major security challenge of the Infrastructure-as-a-Service cloud computing model, which is at the core of our research project. We mainly focus on the security implications that have been identified due to the adoption of virtualization technology on the cloud platform. We also discuss how the security process of operating systems running on the hosted virtual machines has changed, and how this requires new security deployment models to cope with the new security requirements of virtualization technology.

In this chapter, we focus on two key points of this research project: virtualization and operating systems security. This chapter is organized as follows: in section 2.1, we give a general overview of cloud computing technology, its importance, and the different service models of cloud technology. In section 2.2, we discuss the implications of adopting virtualization technology and how such adoption can become a double-edged sword that better supports security but at the same time adds new security risks to the cloud platform. We also discuss the traditional and new security distribution models of operating systems that can be used to protect the hosted virtual machines in a cloud platform. In section 2.3, we discuss the complexity of pointers and memory runtime errors of C-based operating systems, and show how pointers and their inefficient handling and analysis can lead to limited protection and security vulnerabilities in operating systems at system runtime. Finally, in section 2.4 we discuss the basics of *points-to* techniques and how *points-to* analysis could help in solving a lot of operating system runtime problems that might lead to memory bugs and vulnerabilities.

2.1 Cloud Computing

Cloud computing is a new computing paradigm, where IT resources and services are abstracted from the underlying infrastructure and provided on-demand at scale in a multi-tenant environment [2]. Cloud computing technology provides IT enterprises with a flexible, easy and

cost-effective way to operate, manage and maintain their own IT business assets *e.g.* servers, networks, data and applications. Recently, leading enterprise companies – *e.g.* Salesforce, Amazon and Microsoft – have had a great interest in incorporating cloud computing technology in their operational strategies to reduce costs and achieve higher capabilities and reachability of their applications. A recent survey from Gartner [3] about cloud computing market revenues showed that cloud computing market will be at least doubled by 2014; where the market value in 2010 was around USD 68 Billion and expected to reach in 2014 up to USD 148 Billion.

Cloud computing has three main service delivery models: *Software-as-a-Service (SaaS)*, *Platform-as-a-Service (PaaS)* and *Infrastructures-as-a-Service (IaaS)*. SaaS is a new software distribution model where software vendors or service providers centrally host their software and IT services on a cloud platform. This platform makes these software and services available to the regular consumers over the Internet with flexible payment methods based on the consumers' usage and business needs. SaaS has become a demanded prevalent service delivery model for many business applications such customer relationship management (CRM), enterprise resource planning (ERP), and human resource management (HRM). PaaS is a service delivery model that provides the required environment, platforms, development kits and tools for developing and provisioning cloud-based applications. The users of this model are not usually regular consumers of the cloud as in SaaS and IaaS; commonly they are the software developers that target to develop and run cloud applications. One of the key features in PaaS is that it provides a set of preconfigured features and packages where consumers can include in their developed applications. PaaS also allows easy and quick application development and hosting process with a few mouse clicks, where not much client-side experience is required to run and host the developed applications in a cloud platform. IaaS is the core of this research project. IaaS enables provisioning computational resources, data storage, and communication channels as internet-based services to the regular consumers. In other words, IaaS enables adding computational resources (such as CPUs and GPUs), associated storage and communication capacities on the fly, according to the dynamically changing business needs in a cost-effective way.

The foundation of cloud computing, especially the IaaS model, is virtualization. Virtualization enables abstracting the physical resources to integrate multiple servers into a single physical server to achieve better hardware utilization rates and boost operational efficiency. Despite the outstanding benefits of cloud computing, cloud computing has complicated the management and security process of its outsourced IT assets. Cloud computing erased many of the traditional physical boundaries and perimeters that were used to protect and manage organizations' servers and replaced them with virtual IT assets such as virtual machines, virtual networks and virtual firewalls. This results in higher risk rates and a more complex threat mitigation process. On the other hand, the absolute and public availability, accessibility and sharing of cloud services and resources with the regular consumers – that might include malicious hackers – has increased the security risks and the attack surface, assisting hackers in exploiting vulnerabilities in these services to take control over the cloud platform, hosted services or customers' IT assets (virtual machines).

2.2 The Virtualization Double-Edged Sword

In this research project, we focus on the security problem of the IaaS cloud computing model. IaaS is characterized by the concept of resource virtualization that enables running multiple operating system instances – called Virtual Machines (VMs) – on the same physical server, as shown in Figure 2-1. These virtual machines are independent operating environments that have access to virtual resources *via* a thin layer of firmware code – called a hypervisor. Hypervisors are mainly responsible for dynamic resource sharing and isolation between hosted virtual machines. Hypervisors are also known as virtual machine monitors (VMMs). Traditionally, virtual machine monitors were run on the bare hardware with basic functionalities that support virtualizing a limited number of guest operating systems with moderate degree of isolation. With the revolution of cloud computing and the advent of the IaaS model, virtual machine monitoring software has become an embedded special-purpose operating system that supports additional functionality and provides robust isolations between the virtual machines and the running workloads, and is called a hypervisor. Xen, ESX, and Hyper-V are examples of hypervisors.

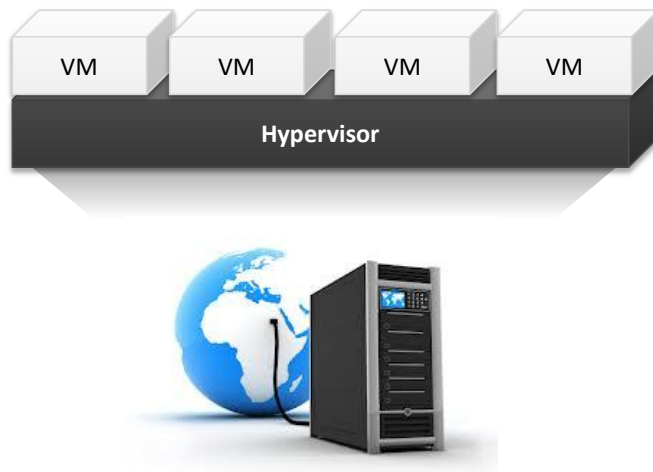


Figure 2-1. Server virtualization.

Adapting virtualization for cloud platforms has revealed a lot of important benefits that can enhance the security process of its hosted virtual assets, along with a lot of new security risks that did not exist in the traditional physical hosts that complicated the security process from another hand. In section 2.2.1, we discuss briefly the major benefit of virtualization in the IT world. In section 2.2.2, we summarize the new security risks that resulted from adapting virtualization in the enterprise IT companies.

2.2.1 Virtualization Benefits

Virtualization has simplified IT operations with easier management, maintenance, support and rapid response to the changing business needs. Virtualization is being adopted widely by a growing number of organizations to reduce costs and enable higher availability of their applications. Figure 2-2 summarizes key benefits gained from adopting the virtualization technology, discussed below.

Abstraction and Isolation. Server virtualization adds a layer of abstraction between the virtual resources (e.g. virtual machines and virtual networks) and the underlying physical infrastructure. This layering enables limiting direct access to the underlying hardware and thus limits the amount of damage that might affect the IaaS platform from regular system users. On the other hand, such an abstraction layer enables efficient isolation between the different operating system running instances along with their workloads that flow inside the server. New

hardware technologies – such as Intel® Virtual Technology hardware – greatly support better and robust isolation that guarantees safe sharing of system resources between the running virtual machines by utilizing hardware protection techniques to protect the running hypervisor.

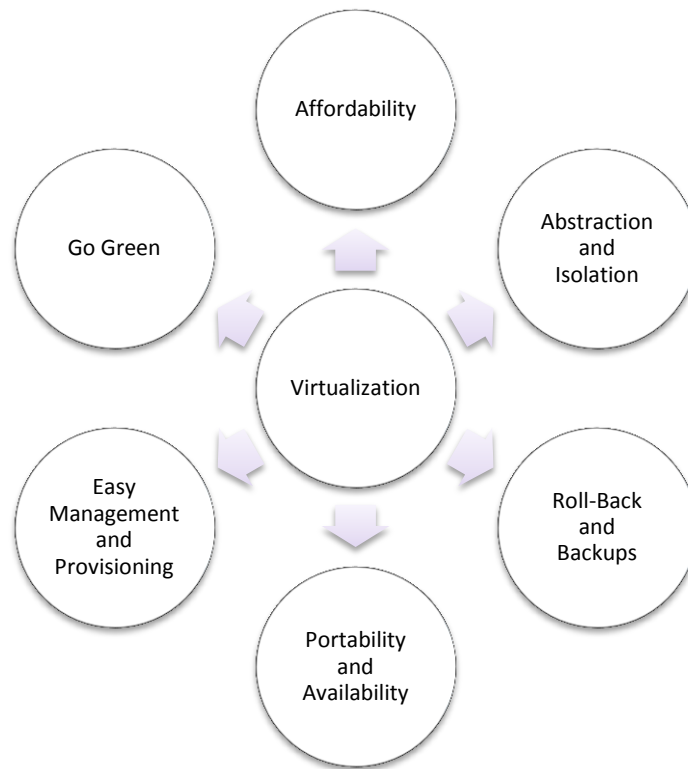


Figure 2-2. A summary of virtualization benefits.

Roll-Back and Backups. Virtualization enables restoring an infected virtual machine to a particular running state to get rid of a security violation that could affect the virtual machine. Rolling-back might not be the optimal solution to defend against system malware while keeping the business applications running without any interruption or losing for important data. However, it is still an effective solution when security patches and updates are not available for a specific security threat. Furthermore, virtualization makes backups and disaster recovery of the hosted virtual machines easy. For instance, if a virtual machine is infected, it can be shut down and replaced with another standby virtual machine to continue its functionality without noticeably losing business continuity. Disaster recovery strategies enable efficient troubleshooting of the malformed virtual machines while quickly restoring access to the running system. Re-

cently, disaster recovery process can be automated to quickly respond to system failures with pre-configured recovery plans for different failover scenarios.

Portability and Availability. Portability comes as a result of abstracting the software from the underlying platforms and hardware, and thus the need of developing specialised software for a specific platform or hardware is eliminated. Virtualization also allows high availability of the running services because of the redundant topology of the running services and applications. For instance, if a server fails, all its virtual machines and running services can be automatically migrated and restarted on another server with unnoticeable downtime and thus with no impact on the consumer. Such live migration supports avoiding many technical problems that might happen because of system maintenance or performance issues that could affect consumers' business continuity and availability.

Easy Management and Provision. Virtualization allows spreading software running platforms, such as middleware platforms and web servers, into separate virtual environments. This enables an easy and effective way to manage and provision hosted software or virtual machines, with fewer configurations and administration tasks. Cloud consumers can easily create and provision virtual machine images and software services with few mouse clicks without a need to spend hours installing operating systems and applications like what happens in the traditional physical servers.

Affordability. Cloud services are a cost-effective way to run businesses because consumers pay only for what they use. Rather than investing in expensive hardware upgrades, cloud computing enables automatic and easy upgrade without huge costs and IT support, where the cloud provider is indirectly responsible for such support and upgrade process.

Green Computing. Integrating servers into virtual machines and aggregating them into fewer physical servers means lowering the power and cooling costs that have a big negative impact on the environment.

2.2.2 The Downside: Virtualization Security Risks

Although virtualization has a key role in supporting the IaaS model with great benefits, it makes the virtual infrastructure of the IaaS platform a potential target for new potent security threats [4]. The problem mainly comes from the *loss-of-control* security problem over the hosted virtual machines. In particular, a hosted virtual machine is owned and controlled by a cloud consumer, and at the same time it is hosted by a cloud provider. Consequently, providers host and run virtual machines but are not aware of their running contents, as these virtual machines are controlled by the consumers, causing a *loss-of-control* security problem.

In IaaS model, many unsupported operating systems and applications can be deployed by any user. Therefore, a cloud consumer might be a malicious hacker running a malicious service on a virtual machine, in order to exploit a vulnerability that enables compromising the hosting platform or the other hosted virtual machines [5]. The hosted virtual machines could also be compromised – by a third party – using advanced kernel rootkits that assist in gaining control over the running operating system and the installed applications, thereby altering the behaviour of the installed security software. Thus, cloud consumers and the hosted virtual machines cannot be trusted from the provider’s perspective to provide security for the running operating system of a hosted virtual machine using the traditional *in-guest* security software. In section 2.2.3, we focus on the *loss-of-control* security problem over the hosted virtual machines and the new security design requirements that are needed in order to develop new security software that can robustly protect the hosted virtual machines in the IaaS and scale to the complex workload running in IaaS physical servers.

In addition to the *loss-of-control* security problem, the security of a host physical server becomes more complex when different and heterogeneous workloads are running on the system – *i.e.* different operating systems, platforms and applications. In addition to the huge amount of traffic and workload flowing inside each physical server that increases the complexity of the protected platform. Such complex environment requires a highly scalable security application that can cope with such huge workload to provide real-time protection for the different running operating systems and hosted applications.

2.2.3 Virtual Machine Security

The lack of verifiable trust between cloud consumers and providers is a basic security deficiency of the IaaS model. When considering security for the hosted virtual machines in the IaaS platform, we should keep in mind two important things: *first*, these virtual machines are exposed to hacking as they are under the consumer's control. *Second*, these virtual machines are sharing the same hardware resources (such as physical memory and storage) with the other hosted virtual machines. This means that an infected virtual machine could assist in exploiting vulnerabilities in the underlying platform (*i.e.* hypervisor) and thus affect that system and the other hosted virtual machines.

Traditional *in-guest* security solutions that are deployed in running operating systems are no longer an effective solution to secure the hosted virtual machines in the IaaS platform. Although traditional *in-guest* security solutions have the ability to get high-level semantic-rich information about an operating system running state, they make security software unreliable, opaque to the user, and can be subverted by advanced kernel rootkits – even if the security software is installed in ring 0 [6]. This is because *in-guest* security solutions rely on the operating system kernel trustworthiness and thus do not have the ability to efficiently protect the hosted virtual machines.

To address risks of the hosted virtual machines and the *loss-of-control* security problem, new *virtualization-aware* security solutions should be introduced. These solutions should have the ability to protect the hosted virtual machines from outside the virtual machine itself and without relying on operating system kernel trustworthiness. The good thing is that virtualization technology provides important characteristics that make it a very practical platform to implement *virtualization-aware* security solutions. These characteristics include:

First, virtualization allows utilizing *Virtual Machine Introspection* (VMI) techniques [7] that enable monitoring the hosted virtual machines externally, at the hypervisor level, as shown in Figure 2-3. Virtual machine introspection has a number of key features that make it robust and reliable: (i) *virtual machine introspection* enables isolating the security solution from other server workloads by deploying the security solution in a dedicated and isolated virtual machine

with a dedicated communication channel with the hypervisor, making it difficult for hackers to detect the installed security software. (ii) External monitoring *via virtual machine introspection* gives the security software complete control over the hosted virtual machines including the installed operating system, running software, and hardware. A major goal of malicious hackers is control, by which the hacker will have the ability to monitor, intercept and modify the state of running software on an operating system. Controlling a system allows malware to remain invisible by obviating or disabling the installed *in-guest* security software. Control of a system is determined by which side (attacker or defender) occupies the lower layers in system operational flow. Lower layers definitely have more control on upper layers because lower layers implement the abstractions upon which upper layers are built. Thus, deploying security software at a hypervisor level allows a complete control over the running virtual machines, as the security software is installed in a layer lower than the virtual machines' layer.

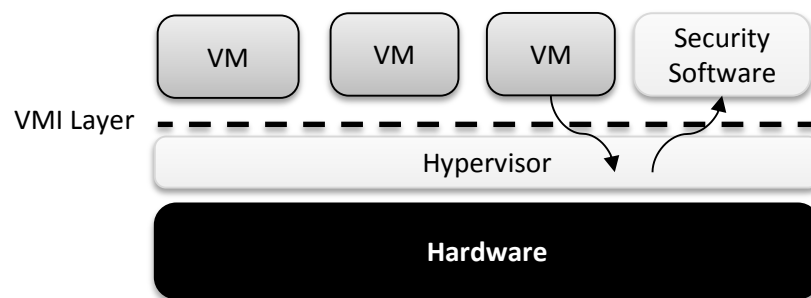


Figure 2-3. Virtual machine introspection at the hypervisor level.

Second, hypervisors have a high level of assurance because of their small footprint in the physical memory. This small footprint reduces the number of potential vulnerabilities that can target the hypervisor code. Moreover, exploiting a hypervisor requires direct physical access to its kernel code which is generally inaccessible. In addition, a lot of new hardware-based techniques have been developed to ensure the integrity of the hypervisor kernel code (details are discussed in chapter 3). Therefore, security solutions that are deployed at a hypervisor level are highly trusted, as the underlying platform is trusted and thus gives more robustness for the deployed security software.

The big challenge with implementing *virtual machine introspection* techniques is that only hardware bytes – *e.g.* physical memory pages and disk block – can be observed, at the hypervisor level. This is in contrast to the internal view of the virtual machine, where we can view high-level entities such as processes, I/O requests, and system calls – using the kernel memory and APIs. This difference in views is called a *semantic gap*. In order to make virtual machine introspection useful for monitoring the running state of the operating system, it is necessary to translate the hardware bytes from the observed virtual machine to actual running high-level operating system information. Current research efforts and practices for overcoming the semantic gap problem mainly fall into two categories²: *first*, stealthily injecting a piece of machine code into the kernel address space of a running operating system to read the internal state of the virtual machine. This approach has not been implemented widely [8, 9], as the injected code could be subverted by advanced kernel rootkits and thus the behaviour of the injected code could be altered to deliver false runtime information about the operating system running state. *Second*, the most common approach used to overcome the *semantic gap* problem is the *memory mapping* technique. *Memory mapping* means accurately mapping between the runtime kernel data layout of an operating system and the underlying hardware memory layout of the virtual machine [7, 10, 11], as shown in Figure 2-4. Such mapping is a challenging task because of the operating system’s kernel data layout complexity. Current research efforts of this technique [6, 12, 13] are limited, as: (i) most approaches depend on operating system’s expert knowledge of the runtime kernel data layout to manually solve the *semantic gap*. Thus, they only cover 28% of kernel data structures, as discussed by Carbone *et al.* [14], that relate to the well-known kernel objects such as processes, threads and device drivers. (ii) Manual kernel data definitions that are based on hard-offset codes allow opportunities for hackers to develop new rootkits that can evade such introspection tools as proved by Bahram *et al.* [15]. (iii) The time and effort spent divining the internals of a particular version of an operating system may not be feasible in other versions of the same operating system [11]. (iv) Most researches of memory mapping techniques depend on the traditional *memory traversal* techniques and *value-invariant* approaches

² Current research efforts and their limitations are discussed in chapter 3, in detail.

to uncover the runtime kernel dynamic objects. *Memory traversal* techniques and *value-invariant* approaches are vulnerable and not accurate. Details of these techniques are discussed in section 2.2.3.1.

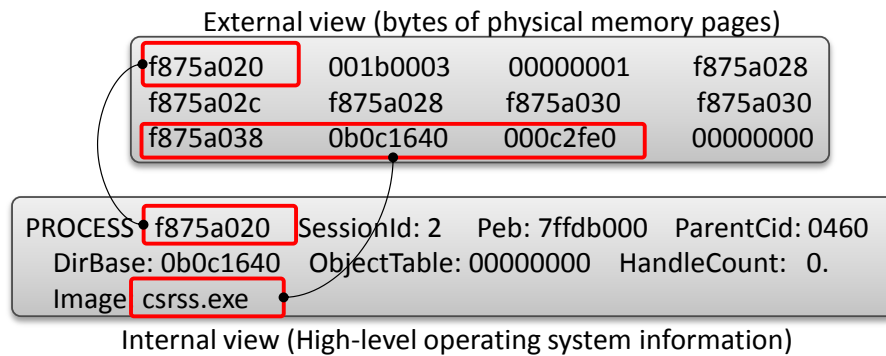


Figure 2-4. Semantic gap problem.

The limitations of current approaches to efficiently overcome the semantic gap result in an inability to get a complete and accurate interpretation of the low-level hardware bytes, making the manual approach inadequate. This made the problem of automatically computing accurate kernel data definitions that covers all kernel data one of our research project tasks.

2.2.3.1 Kernel Data structures and Objects

In operating systems, we usually refer to a running instance of a data structure (also called data type) as an *object*. Kernel data structures can be generally classified into: *control data structures* and *non-control data structures*.

Control data structures are the kernel static data that are used in control-transfer instructions. Kernel control data structures are generally classified into: (i) CPU-specific types, such as descriptor tables that are used to describe memory segments used by the kernel, such as GDT and IDT. (ii) Operating system-specific, such as the system call table that contain a list of native kernel routines and their addresses that are used by all operating system operations. The common thing about these tables is that they are marked as *read-only* in the physical memory and thus their values and locations do not change at runtime, so uncovering these data structures externally is straight-forward. Moreover, pointer manipulations of such data structures can be

easily detected, thus their protection is also straight-forward.

Non-control data structures are the kernel dynamic objects such as processes, threads and device drivers. An operating system kernel contains thousands of non-control data structures, and these data structures **do** change at runtime in location, value and number of running instances (the number of running instances could reach hundreds for a single object type and it depends on the calling contexts of the running operating system). This makes the process of externally uncovering these data structures challenging. Moreover, it is complicated by the fact that: pointer manipulations of non-control data structures are difficult to verify because of their dynamically changing nature at runtime. Modifications to non-control data structures violate integrity constraints that cannot be easily extracted from operating system's source code. This is because the data structure syntax is controlled by the operating system's code while their semantic meaning is controlled by the calling contexts of the running operating system [16]. Dynamic kernel objects are allocated at runtime and accessed through heap-directed pointers. Thus, exploiting dynamic data structures will not make the operating system treat the exploited structure as an invalid instance of a given type, or even detect hidden or maliciously modified objects. This definitely encourages hackers to exploit these data structures to develop new rootkits to gain control over the running operating system. For example, Windows and Linux operating systems keep track of runtime objects with the help of linked lists. A major problem with these lists is use of C null pointers³. Modifications to null pointers violate integrity constraints that cannot be extracted from source code as they depend on system calling contexts at runtime. This makes it easy to unlink an active object by manipulating its pointers and thus the object becomes invisible for the kernel and for monitoring tools that depend on kernel APIs⁴.

Locating kernel dynamic objects in the physical memory is an important step towards implementing different operating system security solutions such as kernel data integrity checkers [17], memory forensics [18], virtual machine introspection [6] and brute-force scanning [19].

³ Null pointers are discussed in section 2.3

⁴ Details of pointer manipulation and hidden objects are discussed in chapter 8, in detail.

Efficient security solutions should **not** rely on the operating system kernel memory or kernel APIs to extract runtime objects, as they may be compromised and thus give false information. On the other hand, the complex data layout of an operating system's kernel makes it very challenging to locate all system objects in real-time. Previous solutions limit themselves to kernel static data *e.g.* system call and descriptor tables [4], or can reach only a subset of the dynamic kernel data [17, 20], resulting in security vulnerabilities and limited protection. Most approaches of uncovering dynamic kernel objects fall into two main categories: *memory mapping techniques* and *value-invariant approaches*.

Memory mapping techniques identify kernel runtime objects by recursively traversing the kernel address space starting from operating system global variables, and then follow pointer dereferencing to discover the running instances of the different object types [1, 14, 21]. The traversal process is done according to a predefined kernel data definition that statically reflects the runtime kernel data layout in the memory. Memory mapping techniques have a number of limitations, as: (i) they are vulnerable to Direct Kernel Object Manipulation (DKOM) rootkits [22]. Such rootkits exploit the indirect *points-to* relations between kernel runtime objects thereby hiding running objects. DKOM allows kernel rootkits to modify the runtime layout of kernel data – *e.g.* processes and threads, just by manipulating object's pointers and without adding or modifying object's code or data. Such manipulation allows rootkits to hide system objects from system users, while they are operating behind the scenes. (ii) Memory mapping techniques cannot follow generic pointer dereferencing, because they only leverage type definitions and thus cannot know the target types or values of these generic pointers [15]. This means that in order to accurately traverse the kernel address space, all the generic pointers scattered around the kernel address space need to be resolved first. (iii) Memory traversal techniques have a significant performance overhead because of the poor spatial locality of the general-purpose operating systems [23]. The problem with general-purpose allocators that are used by C-based operating systems is that they primarily focus on reducing the allocation overhead and enhancing the memory space utilization. Thus, when the memory manager allocates an object, only allocation sequence and object size are considered [23]. Hence, kernel

runtime objects of the same type could be scattered around in the kernel address space – typically not the whole kernel address space however the address space designated for such allocations is still big in size and affects system performance. Thus, uncovering a single instance of a running object would require accessing and traversing several memory pages.

Value-invariant approaches could be considered as signature-based approaches. This is where certain fields of a data structure with their values are formalized as a signature that is used to scan the physical memory for matching instances, such as DeepScanner [24], DIMSUM [25] and SigGraph [19]. *Value-invariant* approaches are not an effective way of finding kernel dynamic objects for the following reasons: (i) a signature for a specific object type may not always exist for a data structure, as discussed by Lin et al. [19]. For example, it is difficult to generate *value-invariant* signatures for data structures that are part of linked-lists (single, doubly and trees) because the actual running contents of these structures depend on the calling contexts at runtime [26]. This means that the actual objects (type, number of running instances and memory locations) in these lists can be recognized only at runtime. (ii) *Value-invariant* approaches do not handle the rich generic pointers problem of kernel data structures and thus they are also vulnerable to object hiding attacks and pointer exploits, as happens in *memory traversal* techniques. (iii) The performance overhead of *value-invariant* approaches is extremely high, as they typically scan the whole kernel address space with large signatures that makes real-time monitoring of system runtime objects impractical.

2.3 Operating Systems and the “Pointer Problem”

Ensuring reliability of C-based operating systems against memory vulnerabilities is very challenging, because of the extensive use of pointers and data structures in their source code. C-based operating systems utilize C data structures heavily to model the dynamic runtime objects. They also use pointers extensively to simulate call-by-reference semantics, emulate object-oriented dispatch *via* function pointers, avoid expensive copying of large objects, implement lists, trees and other complex data structures, and also as references to objects allocated dynamically in kernel memory and user heaps [27]. Moreover, dynamically allocated objects

can be cast to multiple types during their lifetimes, further complicated by the fact that kernel data structures are implementation-dependent where a pointer deposited in a field under one object can be accessed from a different field under another object.

Pointers, dereferencing and dynamically allocated objects are the most critical points in C-based operating systems that assist in exploiting vulnerabilities in the running operating system. Despite the serious threat that could arise from maliciously manipulating these critical pointers, operating systems vendors continue to use C to implement operating systems for better efficiency and higher performance. Unlike safe high-level languages such as Java and C#, C pointers can point into anywhere in memory or in the middle of an object. Pointers are extremely powerful as they enable manipulating the contexts of memory addresses at runtime. Pointers are complex to handle and can cause many memory errors and bugs – such as memory leaks and buffer overflows – if not implemented correctly.

A pointer is usually associated with a type and a value that is specified at declaration time at the development stage or can be linked at compilation-time. However, pointers in C could be initialized to a value of `NULL` or initialized without a declared type, as a `void` pointer. Furthermore, pointers can be cast to different types at system runtime, making the initially declared type untrusted. To get a concrete idea of the pointers problem in C-based operating systems, we discuss the context of three well-known problems in C-operating systems, namely `void` pointers, `null` pointers and casting.

Void pointers. A `void` pointer defined at declaration time means that it is a pointer that can point to any data type, and the actual type of such `void` pointer can only be identified at runtime according to system calling contexts. Thus, `void` pointers support a form of polymorphism, where the target object type(s) can only be identified at runtime. The wide use of `void` pointers hinders performing systematic integrity checks on kernel dynamic data, where there are no type constraints for a `void` pointer. `Void` pointers are powerful vulnerabilities for hackers to exploit, in order to point to somewhere else in memory or execute vulnerable code such as return-oriented programming [5] and jump-oriented programming [28] rootkits.

Null pointers. A `null` pointer points nowhere – *i.e.* points to unallocated memory or a value of zero – and thus dereferencing a null pointer may cause address violations in the memory or attempt to execute an illegal operation. In operating systems, `null` pointers are used mainly to implement linked-lists, which are heavily used in operating systems to maintain and manage dynamically allocated kernel objects. The C definition of a linked-list only shows that a linked-list data member points to another linked-list data member, as shown in Figure 2-5. However, at system runtime members of a linked-list point to a specific object type according to the calling contexts of the system, not to another linked-list. This means that the actual objects (*i.e.* type, number of running instances and memory locations) structured in a linked-list can be recognized only at runtime, not at compilation time. Manipulating `null` pointers assists malicious hackers to hide or change runtime objects. This is because runtime manipulations of `null` pointers violate integrity constraints that cannot be extracted from source code as they depend mainly on calling contexts at runtime.

<pre>typedef struct _LIST_ENTRY { PLIST_ENTRY Flink; PLIST_ENTRY Blink; } LIST_ENTRY, *PLIST_ENTRY;</pre>	<pre>struct list_head { struct list_head *next; struct list_head *prev; };</pre>
(a)	(b)

Figure 2-5. Doubly-linked list implementation in Windows and Linux.

Figure (a) shows the implementation in Windows operating system source code, and (b) in Linux operating system source code.

Type casting. In C, type declarations are hints indicating how the variables are likely to be used at the program runtime; however casting operations are likely to happen at runtime to change the original declared type. C data types can be subverted by casting. A pointer of a given type can be cast to point to more than one type during its lifetime [29]. A major problem with casts is that they induce relationships between objects that appear to be unrelated [30]. These implicit *points-to* relations enable hackers to exploit the layout of kernel runtime objects in the physical memory and thus get a false view of the running state of operating system. Figure 2-6

shows a code snippet demonstrating how casting can implicitly develop *points-to* relations between runtime objects. Line 3 reflects a cast operation, between the `ExHandler` data type (line 1) and the `EPROCESS` data type (line 3).

```

...
typedef struct _ExHandle {
    int* handle;
} ExHandler; //--> (1)
...
PEPROCESS ActiveProcess; //--> (2)
...
PEPROCESS AllocatePrMemory() {
    return (PEPROCESS) malloc(sizeof(EPROCESS));
}

void CreateProcess(PEPROCESS p_ptr) {
    p_ptr = (PEPROCESS)AllocatePrMemory();
    ActiveProcess = p_ptr;
    p_ptr->UniqueProcessId = ExHandler(ActiveProcess); //--> (3)
    ...
}

```

Figure 2-6. Casting example in C.

In Windows and Linux operating systems, from our analysis, nearly 40% of the inter-data structure relations are pointer-based relations (indirect relations), and 35% of these indirect relations are generic pointers. In such a complex kernel data layout, the runtime memory layout of the data structures cannot be predicted during compilation time. This makes kernel data a rich target for rootkits that exploit the *points-to* relations between kernel data structure running instances using direct kernel object manipulating techniques or by overwriting function pointers located in dynamic kernel memory, allowing attacks such as object hiding.

2.4 Points-to Analysis

The goal of *points-to* analysis techniques is to statically compute a set of memory locations (*points-to* sets) to which a pointer may point to at runtime. Rather than a pointer is being declared as `null` or `void`, or cast at run-time, *points-to* analysis helps in statically determining the set of actual data type(s) and values (memory locations) that a particular pointer can dereference at runtime.

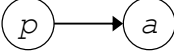
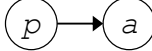
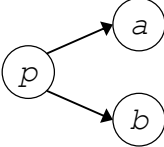

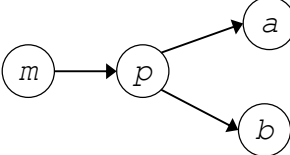
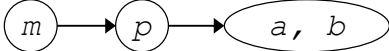
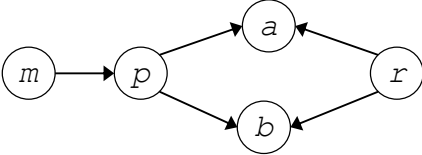

Points-to analysis has been used in a variety of static program analysis tools to check for memory runtime errors that could arise because of the inefficient use of pointers in the programs source code. Simply, basic *points-to* analysis is an advanced alias analysis approach that attempts to compute equivalence relations of program pointers [31]. Two variables are aliases for each other if they point to the same memory location, where changing the contents of one pointer will indirectly change the other [32]. *Points-to* analysis of C programs mainly differ in how alias information is grouped. There are two main algorithms to group alias information: Andersen's [33] and Steensgaard's [34]. Anderson's *points-to* analysis algorithm creates a node for each variable and the node may have different edges, while Steensgaard's *points-to* analysis algorithm groups alias sets in one node and each node just have one directed edge. Andersen's algorithm processes each statement in the program in arbitrary order to build a *points-to* graph. Andersen's approach is the slowest but the most precise and Steensgaard's approach is the fast but less precise.

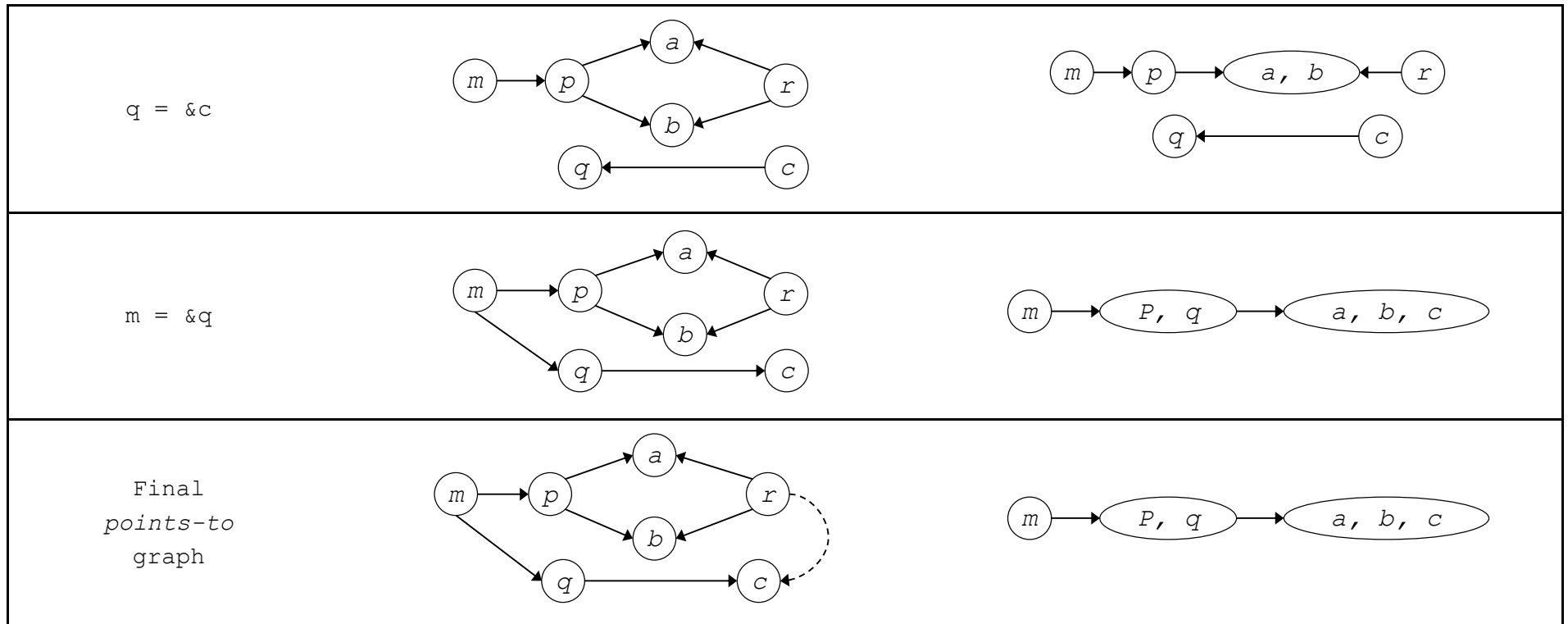
Table 2-1 shows a C code fragment and the *points-to* sets computed by those algorithms. Based on such general aliasing approaches, there are different types of analysis aspects that make the trade-off between performance and precision: *first*, achieving context-sensitivity, where aliasing information is based on distinguishing between heap objects created *via* different call-paths to the program procedures. The context of a procedure call is distinguished by its call-path which is the path from the procedure entry to its call site. Context-sensitive algorithms are highly precise, but very slow in performance and complicated to be implemented. Achieving context-sensitivity enables accurate type-inference of the declared pointers including `void` pointers and runtime casting operations. Type inference determines the actual type(s) of an object by analysing the usage of this object in the program. *Second*, achieving field-sensitivity, where aliasing information should distinguish the different fields inside a single data structure/object and each field must have its own *points-to* set. In other words, field-sensitivity enables identifying the internal connectivity pattern of data structures. Another analysis aspect that is usually considered when performing field-sensitive analysis is inclusion-based analysis. Inclusion-based analysis means that two pointers may point to overlapping but different sets of

objects [29], and this is likely to exist in operating system's kernel data. Thus, all objects that may be pointed to by a data structure's field are represented as a standalone node in the *points-to* analysis graph. *Third*, achieving flow-sensitivity, where aliasing information considers the effects of pointer dereferencing instructions with respect to the call-graph of the program. In other words, flow-sensitivity considers the flow of values in a program in order to compute the *points-to* sets. A program call-graph is a graph with a set of nodes and edges representing dependency relations between program statements to reflect the statements that control the execution of the other program statements.

Points-to analysis is traditionally performed in two main phases: *intraprocedural* analysis and *interprocedural* analysis. Intraprocedural analysis is local analysis phase of each procedure in the program to compute a local *points-to* set based on local information of each procedure, such as formal-in parameters and arguments. In interprocedural analysis the procedures are analysed with respect to the incoming parameters and global variables that are used in that procedure based on the program call-graph. These two phases could include field and flow-sensitivity information; however context-sensitivity is usually achieved with an extra phase of the analysis based on the used technique. Highly precise and scalable *points-to* analysis is usually an NP-hard problem [35], which is very expensive to implement. Over the last decade a large number of new algorithms have been introduced to perform the analysis with high precision rates while avoid the expensive analysis cost [36-38], as discussed in chapter 3.

Table 2-1. Example showing the difference between Steensgaard's and Andersen's algorithms.

Statement	Steensgaard's Algorithm	Andersen's Algorithm
p = &a;	 <pre> graph LR p((p)) --> a((a)) </pre>	 <pre> graph LR p((p)) --> a((a)) </pre>
p = &b;	 <pre> graph LR p((p)) --> a((a)) p --> b((b)) </pre>	 <pre> graph LR p((p)) --> ab([a, b]) </pre>
m = &p;	 <pre> graph LR m((m)) --> p((p)) p --> a((a)) p --> b((b)) </pre>	 <pre> graph LR m((m)) --> p((p)) p --> ab([a, b]) </pre>
r = *m	 <pre> graph LR m((m)) --> p((p)) p --> a((a)) p --> b((b)) r((r)) --> a r --> b </pre>	 <pre> graph LR m((m)) --> p((p)) p --> ab([a, b]) r((r)) --> ab </pre>



2.5 Summary

In this chapter, we discussed the security problem of the IaaS cloud computing model and a collection of different technologies, techniques and technical problems that have been tackled in this research project. We focused on the *loss-of-control* security problem of the hosted virtual machines in the IaaS cloud platform, and on the pointers problem in operating systems that lead to memory violations and bugs. These two areas are the main big problems that are addressed in this research project. We also discussed how *points-to* analysis techniques could be utilized to limit runtime memory bugs and errors in operating systems caused because of pointers. We explained the basics of *points-to* analysis techniques and the different analysis aspects that make the tradeoff between precision and scalability.

Chapter 3

Literature Review

In order to meet the research objectives discussed in chapter 2, a number of multi-disciplinary technical problems have been studied. In this chapter, we review and summarize key existing research efforts and the state-of-the-art in different research areas including cloud computing and virtualization security, operating system security, kernel rootkits and kernel integrity, memory bugs and errors, and static program analysis techniques including *points-to* analysis.

This chapter is organized as follows: in Section 3.1, we give an overview of the research areas cover in this literature review analysis. In Section 3.2, we review related work in the area of cloud computing security, with a focus on IaaS security. Then, we discuss key related work in the area of virtual machine security, virtual machine introspection, the semantic gap and active monitoring. In Section 3.3, we study operating system security and discuss different aspects in operating system security such as: (i) kernel rootkits and the key existing approaches to detect and defend against such rootkits. (ii) Kernel dynamic objects and key related work to uncover these objects from a trusted source. (iii) Operating system memory errors, bugs and generic pointers and key related approaches and techniques used to prevent such memory unsafe violations, such as *points-to* analysis, separation logic and shape analysis. In section 3.4, we summarize key limitations of the related work and discuss the need for new security approaches and how such approaches could meet our research objectives.

3.1 Introduction

The main research problem of this project is solving the *loss-of-control* security problem of the IaaS platform in order to enable pre-emptive protection of the hosted virtual machines externally. To address this problem efficiently, a number of sub-problems needed to be addressed first. Figure 3-1 summarizes key research problems that have been studied in this research project in order to facilitate the development of our *virtualization-aware* security solution.

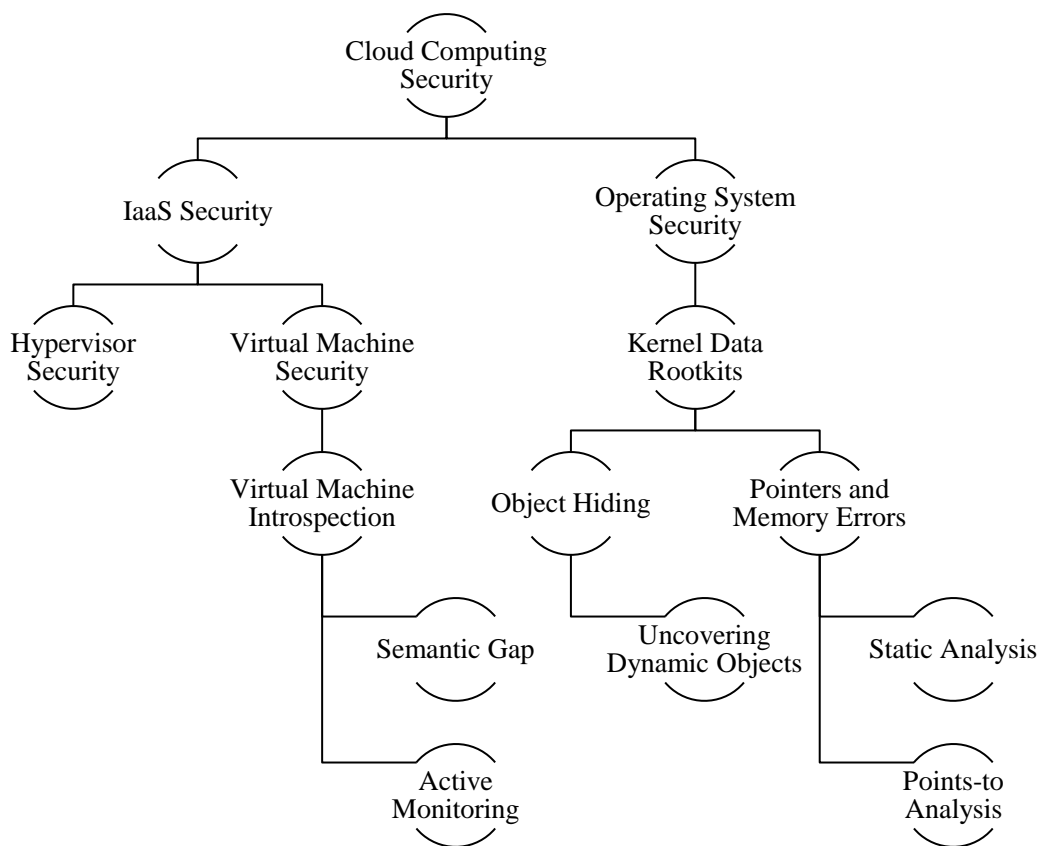


Figure 3-1. Related work analysis structure.

Based on the organization in Figure 3-1, we discuss key related work and research practices of each problem area shown in Figure 3-1. We have two main research problems and each problem has other sub-research problems. *First*, the security problem of the virtual infrastructure in IaaS cloud platforms. IaaS security can be broken down into component research areas, including: hypervisor security, virtual machines security, virtual machine introspection, active monitoring, semantic gap and virtual appliances. In section 3.2, we review key related research practices discussing generally the security problem of the different cloud computing models. We then give more attention to the security of the IaaS model. Finally, we discuss the current research efforts to protect the hosted virtual machines, at a hypervisor level – which is the core of our research project. Moreover, we discuss the current solutions for main technical problems that need to be addressed in order to enable protecting virtual machines externally. These problems include: virtual machine introspection, semantic gap problem, active monitoring and virtual appliance deployment models.

Second, another key research problem of our research project is operating system security, more specifically operating system kernel rootkits and integrity checks. This includes a number of sub-areas that needed to be analysed such as kernel data rootkits, kernel dynamic objects, operating system memory safety problems and weak typing, and the different analysis techniques used to overcome such memory problems. In Section 3.3, we discuss key related work of these areas with their main limitations. We focus mainly on kernel data and the key existing approaches to defend against such kernel dynamic data threats. We also discuss operating system memory violations caused by the pointers and weak typing of the C programming language. Related to the memory safety area, we discuss key related approaches that have been used in order to disambiguate generic pointers scattered in kernel memory, such as *points-to* analysis, separation logic and shape analysis.

3.2 Cloud Computing Security

Recently, cloud computing security has been a major concern for many researchers discussing the main challenges of adapting the different models of the cloud and their relevant security issues [39-50]. Each layer in the cloud model has its own vulnerabilities that can affect the security and trust of the cloud. Researchers in [51-53] discussed the cloud security problem from a cloud provider perspective, exploring the security implications of adapting the different technologies in cloud platforms and the different cloud service delivery models. Grobauer *et al.* [54] discussed different vulnerabilities of the basic cloud models as a key to understanding cloud risks, such as virtual machine escaping, session hacking and insecure cryptography. Yildiz *et al.* [55] explored a conceptual model to identify security requirements for different cloud levels, such as network, servers, and storage, applications and cloud management layers. Lenk *et al.* [56] discussed the cloud computing stack architecture and components, and the different cloud services. Brock *et al.* [57] proposed different security evaluations and countermeasures in the cloud security area, and introduced a conceptual security framework for the cloud to apply sign-on access token security. Qamar *et al.* [58] discussed various technical and non-technical open issues in cloud computing related to security and costs. Muttik *et al.* [59] reviewed key

security software that can fit in the cloud platform such as anti-spam and antivirus to protect the hosted services. Kandukuri *et al.* [41] explored some security issues in the cloud, such as how customers could trust providers through the Service Level Agreements (SLAs). Kandukuri *et al.* explored the contents of the SLA, and how these contents can guarantee for cloud consumers secure access to their services in the cloud.

From the above related work, we see that the security problem in cloud computing is a quite big and important problem with many research problems that relate to whether the adaption of different technologies in the different models of the cloud, or to the security risks of the cloud models themselves. In our research project, we selected the IaaS cloud computing model for detailed study and focused on its security problem in an attempt to provide a solution to one of its critical problems, which is the outsourced virtual machine security problem. Key related work of the IaaS model and its security problem are discussed in section 3.2.1.

3.2.1 IaaS Security

In this section, we focus, in particular, on the IaaS model as it is the main core of our research project. We review key related work of this service model and discuss its main security issues that relate to the hypervisors and hosted virtual machines.

Dawoud *et al.* [50] explored the different security challenges of the virtual infrastructure in the IaaS model that arise from the virtual machine mobility feature and the inter-communications between virtual machines that do not need to go through the physical network interface of the server and take place in the server memory via a virtual switch. In addition, Dawoud studied security threats of the main components of the IaaS delivery model, including: Service Level Agreements (SLAs), utility computing, cloud software, virtualization, internet connectivity, web browsers and hardware. Dawoud *et al.* also introduced a conceptual security model that covers the IaaS components in their entirety. This model is composed of: (i) a security configuration policy to guarantee secure configuration for each component in the IaaS layer. (ii) Secure resources policy management to control roles and privileges. (iii) Security policy monitoring and auditing entity to track system life cycle. Hwang *et al.* [60] explored the

basic security requirements to develop a security solution for the IaaS model, including the physical (hypervisor and hardware) and virtual infrastructure (virtual machines, virtual network and storage). Caron *et al.* [61] proposed a system of security metrics specific to the IaaS model, used to develop virtual machine placement algorithms. Their proposed metrics system is five-layered and covers hardware, hypervisor kernel, operating system security module, processes, and application. The metrics enable evaluating the security levels of the cloud IaaS infrastructures while considering the customer security requirements. Based on these metrics, they generate a set of virtual machine placement algorithms that accommodates both cloud provider and customer needs.

IBM [62] introduced the idea of Trusted Virtual Datacenter (TVDC). A TVDC is a security solution that addresses both infrastructure and management issues introduced by datacenter virtualization. The main idea behind TVDC is grouping virtual machines and resources that are collaborating towards a common purpose into workloads called Trusted Virtual Domains (TVDs). It provides strong isolation between workloads by enforcing a mandatory access control policy throughout a datacentre. This policy defines which virtual machines can access which resources and which virtual machines can communicate with each other. This access control policy is developed based on the virtual machine meta-information to store security labels that are used to determine if the virtual machine can access a specific resource.

Krautheim [63] proposed a security model that assumes that the cloud consumer should control the security of their hosted virtual machines including the running applications, while letting the cloud provider only control the security of the physical infrastructure. Such a model falls into a conflict with our assumptions about the IaaS platform. This is because the virtual machine that is managed by a cloud consumer still has access to the physical hardware and the shared memory even if this access is indirect *via* the hypervisor. Recently, Common Vulnerabilities and Exposures (CVE) has reported multiple resource sharing exploits in the Xen and ESX hypervisors [64, 65], caused by hosted virtual machines. Thus, the security of the hosted virtual machine cannot just be left for the cloud consumer to manage its security, as this consumer might simply be a malicious hacker.

3.2.1.1 Hypervisors Security

Because the hypervisor runs underneath the virtual machines, it is a desirable target for malicious hackers, who would use the hypervisor to take control of the host server and the hosted virtual machines. Such threat is called hyperjacking [66]. Hyperjacking requires the hacker to either have physical access to the server (to install the malicious code), or exploit a vulnerability in a hosted virtual machine in order to be able to inject that malicious code into the hypervisor kernel. Wojtczuk [67] discussed how the Xen 3.x hypervisor can be modified at runtime with Direct Memory Access (DMA) transfers to install backdoors in the hypervisor's kernel, by allowing loading compiled C code into Xen memory. To prevent such security threats, DMA access has been restricted in the current chipsets by IOMMU [68] or Intel VT technology [69], making this attack inapplicable.

In order to protect hypervisors, a number of approaches have been developed to guarantee their load-time integrity to ensure secure operational environment. SecVisor [70] is a small hypervisor that uses hardware memory protection techniques to ensure kernel code integrity, by enforcing $W \oplus X$ property to prevent execution of any injected malicious code. $W \oplus X$ is a protection policy enforced on memory pages to make them either writable or executable, but not both simultaneously. SecVisor uses page tables as the basis of its MMU-based memory protections. SecVisor virtualizes the MMU and IOMMU, and this enables SecVisor to intercept and check all modifications to the MMU and IOMMU state. NICKLE [71] is a similar system that checks the lifetime kernel code integrity of the hypervisor using memory shadowing. Any code to be executed for the first time is authenticated using a specific cryptographic hash value and then copied to the shadow memory transparently.

Steinberg *et al.* [72] proposed a micro-hypervisor design that contains the basic security and performance critical mechanisms such as: communication, resource delegation, interrupt control and exception handling. The rest of the hypervisor components – *e.g.* file system, network stacks and device drivers – are controlled and provided by a dedicated virtual machine monitor that runs in the user-level for each hosted virtual machine. Although, such design minimizes the hypervisor footprint in the memory and thus reduces its attack surface, it con-

sumes the server resources – *i.e.* RAM, CPU and storage – because of the virtual machine monitors that are allocated for each virtual machine. This design makes it impractical to be applied in real industrial environment.

Shinagawa *et al.* [73] introduced a small hypervisor architecture that is designed to enforce memory access and I/O device security through virtual machines and thus reduces the code size of the hypervisor. This is done by removing the device drivers and device models from the hypervisor and utilizing device drivers of the guest operating system to handle real devices of the hosting server. Thus, the hypervisor has a small set of drivers to mediate access to I/O devices. Despite the reduced size of the proposed hypervisor, this architecture limits the hypervisor functionalities because it is partially under the control of the virtual machines. Also, if the device drivers of a virtual machine are infected with a malicious code, it can bypass the hypervisor and access hardware directly. Murray and Milos [74] discussed a similar approach that enables strengthening the Trusted Computing Base (TCB) of a Xen-based system, by removing Dom0 (Domain 0) user-space from the TCB to improve Xen's TCB security against full control malware.

In summary, hypervisors are a critical point in the IaaS security, and effective protection mechanisms are usually hardware-based that are related to the underlying hardware architecture such as Intel TXT [75]. The hypervisor security problem is not considered in our research project, as we mainly focus on virtual machine security which is from our point of view the most likely vector of attack that could cause security breaches in the IaaS platform including hypervisor and virtual network.

3.2.1.2 *Virtual Machines Security*

Guest virtual machines are a significant source of security threats in the IaaS platform, and they are vulnerable to all threat types that can affect traditional operating systems – *e.g.* rootkits, spoofing, malwares, DoS, Buffer overflow. In addition to these threats, virtual machines are vulnerable to new types of security threats that are specific to the virtualization feature, as discussed in section 3.2.1.2.1.

Virtualization has made the process of creating and provision virtual machines very quick and easy with just few mouse clicks or simple commands. This ease can lead to virtual machine sprawls. Virtual machine sprawl happens when too many virtual machines are created without careful planning, management and service consolidation [76]. Virtual machine sprawl affects the physical resources of the server such as memory, storage and CPUs, and this could make the administration of the entire infrastructure very difficult. To prevent virtual machine sprawl, administrators depend on a set of virtual machine management applications to define and enforce a process for the deployment and provision of virtual machines. Elmroth and Larsson [77] explored the standards of virtual machine management (descriptors, placement, migration, monitoring) in federated cloud environments, and discussed three different scenarios for virtual machine migration (migration due to lack of system resources, primary site failure and need to remote site to run the VM, migration request from server to another). Li *et al.* [78] applied the feedback control theory to represent virtual machine-based architecture for adaptive management of virtualized resources in cloud computing. Feedback control offers a conceptual tool to address resource management changes and disturbance in workloads and configurations. MIRAGE [79] is an image management system that provides an access control framework that regulates virtual machine image sharing, and provide image filters to control image information and remove unwanted information. MIRAGE consists of four parts: access control framework to control images check-in and check-out, image filtration mechanism to filter the publisher data before the new user work on it, a provision mechanism to follow up the history of the images, and a repository maintenance module to make utility scanning to the existing images, to avoid malwares. Wang *et al.* [80] presented a policy controlled secure solution for virtual machine live migration in personal clouds. This work utilizes the Trusted Computing Base (TCB) concept in both hardware and software to enable safe virtual machine migration and management. Hao *et al.* [81] addressed the problem of virtual machine migration and management across multiple networks in an efficient manner without suspending their operations. This is done by logically combining multiple geographical distributed physical devices to form a single virtualized logical router, where each physical device mimics a virtual card with multiple virtual ports.

3.2.1.2.1 *Virtualization Rootkits*

Another threat that is related to the virtualization feature is dormant virtual machines. Dormant virtual machines are the virtual machines that are not running on the server; however they are still hosted as files in the physical server – *i.e.* offline virtual machines. When a virtual machine is offline, it is still available to any application that has access to the server storage, and it is therefore susceptible to malware infections [82]. Mirzoev and Yang [76] ran an experiment in their test lab environment, where a dormant virtual machine running Windows 2000 server in an earlier version of a virtualized product was infected by the Blaster Worm. The worm was able to duplicate itself and infiltrate other unprotected suspended virtual machines. Another observation was that a virtual machine running as a DHCP server continued to hand out IP addresses even when it was offline. To protect dormant virtual machines, datacentre administrators consider these virtual machines are regular files that should be checked with antivirus software. Schwarzkopf [83] *et al.* also proposed an approach that deals with the dormant virtual machines problem, by helping consumers and providers to keep virtual machines up to date. This is not just by updating the running operating system but also any running software that are out-dated.

A number of proof-of-concept rootkits have been developed to prove that virtualization can be detrimental to a running operating system, if the appropriate protection is not provided. Virtual-machine based rootkit (VMBR) [84] is a rootkit that converts a traditional server that runs an operating system into a virtual machine. This rootkit installs a virtual machine monitor underneath an existing operating system and hosts that operating system into a virtual machine. In order to insert a VMBR underneath an existing system, a VMBR manipulates the system boot sequence to ensure that it is loaded before the operating system itself. Then, the VMBR boots the existing operating system using the virtual machine monitor. As a result, the operating system runs normally, and the VMBR runs stealthy underneath the operating system. In order to do this, a hacker must first gain high privileged access to the operating system to modify its boot sequence. The authors of the VMBR also showed how to detect a VMBR using their proposed system, SubVirt. This is done by placing SubVirt below the VMBR to get complete control of it. GuardHype [66] was also introduced as a hypervisor that can defend against VMBRs.

GuardHype mediates the access of hypervisors to the hardware virtualization extensions, effectively acting as a hypervisor for a hypervisor. HVM [85] is a proof-of-concept of a hardware-based virtual machine rootkit for Mac OS X using Intel VT-x architecture. HVM also install itself by moving the operating system into a virtual machine while the operating system is running, similar to VMBRs.

Franklin *et al.* [86] discussed the problem of detecting remote virtual machine monitors by devising fuzzy benchmarking to detect their presence on a remote server. The basic idea behind their fuzzy benchmarking approach is measuring the execution time of particular code execution sequences, by developing a fuzzy benchmarking program whose execution differs from the perspective of an external verifier when a target host is virtual machine. Paleari *et al.* [87] proposed an automatic technique to generate red-pills to check if a program is executed through a CPU emulator or a real physical CPU. A red-pill is a sequence of bytes that corresponds to the assembly instructions of a specific architecture. The proposed technique has been used to discover new red-pills for detecting two IA-32 CPU emulators of virtual machine monitors.

Blue Pill [88] is a rootkit based on x86 virtualization technology that targets Microsoft's Windows Vista. It exploits AMD64 SVM extensions to host the operating system in a virtual machine and export a generic hardware interface inside the virtual machine. Blue Pill cannot install itself without hardware virtualization support of the underlying architecture.

3.2.1.3 *Virtual Machine Introspection*

External protection of the operating systems running in the hosted virtual machines is a well-known research problem in security, even before the cloud computing era. This is because of the robustness of external protection in detecting advanced kernel rootkits that cannot be detected using traditional *in-guest* security solutions. In this section, we explore current research efforts and practices used to protect virtual machines *via virtual machine introspection* against a variety of operating system security threats.

Monirul *et al.* [89] proposed a general-purpose framework to protect operating systems against kernel rootkits in the hosted virtual machines, by utilizing hardware memory protection.

Vieira *et al.* [90] introduced an intrusion detection technology to protect the hosted virtual machines based on behavioural-based techniques. Martignoni *et al.* [91] introduced a framework for dynamic malware analysis and profiling in virtual machines. This framework is based on creating a separate virtual machine to be the security lab for the IaaS platform and to monitor the running operating systems in the hosted virtual machine. Oliveira *et al.* [92] introduced a framework for protecting kernel code and data using an external monitoring software. X-Spy [93] is a hypervisor-based intrusion detection system designed to protect hosted virtual machines from kernel rootkits. X-Spy is deployed in R-Xen, an enhanced hypervisor architecture that focuses on enhancing Domain0 reliability of the Xen hypervisor.

All of the above approaches, whatever the objective of the security tool, require a robust introspection framework that enables reading the internal state of the hosted virtual machine, at a hypervisor level. *Virtual machine introspection* was firstly introduced by Garfinkel *et al.* [7] in 2003, to support developing robust hypervisor-based intrusion detection systems. More recently, virtual machine introspection has been implemented widely in order to develop security solutions to operating systems [6, 7, 10, 11, 15, 94]. Nance *et al.* [10] surveyed the different virtual machine introspection approaches and introduced a VMI tool, called VIX, that works for Xen hypervisor. Xenaccess [95] is another virtual machine introspection library for the Xen hypervisor which monitors the internal state of the virtual machines. Xenaccess was built based on `libxc` and `blk_tap` modules to enable reading memory and disk hardware bytes of guest operating systems. Neugschwandtner *et al.* [12] introduced dAnubis, a virtual machine introspection tool in order to profile malicious kernel code access. dAnubis provides real-time monitoring and analysis of the malicious Windows device drivers by monitoring all available interfaces through which the driver can interact with the kernel. KernelGuard [96] is a memory access active monitor to prevent malicious memory access to some protected kernel data. KernelGuard watches write-access to specific kernel code and data pages that are protected by KernelGuard. XenKimono [97] is an intrusion detection system aimed at discovering malicious intrusions by analysing the guest kernel data externally. The translation of the memory bytes to high-level operating system information is done by using kernel symbols from the DomU kernel

binaries of Linux operating system. Onoue *et al.* [98] proposed a security system that controls the behaviour of processes in a guest virtual machine by controlling system calls between the virtual machines and the virtual machine monitor. Onoue *et al.* utilize VMI techniques to monitor the low-level events of the hosted virtual machines. Kienzle *et al.* [99] utilizes virtual machine introspection for endpoint configuration compliance monitoring. These endpoint configurations include checking the authorized and installed software, registry keys, and software files by monitoring the hosted virtual machine externally from Domain0 in Xen. Gibraltar [17] adapts virtual machine introspection techniques, but in a physical environment rather than a virtualized platform. This is done by connecting the two hosts with a PCI network card to enable the security software to access the physical memory of the other host and then passively monitor this host. Nick *et al.* [20] also applies external monitoring using PCI cards to access to the physical memory.

A key problem with the above researches is their use of a manual approach to overcome the semantic gap to implement the virtual introspection frameworks. They depend mainly on their prior knowledge and hands-on experience of an operating system kernel to manually build a kernel data definition that reflects the underlying runtime data layout. The manual approach only covers around 28% of kernel data structures – as discussed by Carbone *et al.* [14] – that relate to well-known objects and structures of the operating system such as processes and threads. These approaches also do not consider the generic pointer relations between kernel data structures, making their approaches imprecise and vulnerable to wide range of attacks that can exploit these pointers in kernel dynamic data. These limitations result in limited protection, as an accurate and complete kernel data definition is an important step in implementing a robust *virtual machine introspection* framework. Another limitation of the above approaches is that they depend on memory traversal techniques to uncover system runtime objects and kernel data based on the developed kernel data definitions. Memory traversal techniques are time-consuming and thus present a high performance overhead on the security solution. Moreover, memory traversal does not enable efficient detection of kernel object manipulation rootkits, as they are based on dereferencing information without performing any integrity

checks on the dereferenced pointer value(s) or type(s). Moreover, mapping a kernel memory reflects kernel memory status at a specific time and that does not enable real-time monitoring or protection. On the other hand, the previously discussed researches can be subverted by new type of rootkits as demonstrated by DKSM [15].

DKSM (Direct Kernel Structure Manipulation) is a proof-of-concept attack to show that existing *virtual machine introspection* solutions could be subverted to provide false information about the operating system internal state. DKSM is based on the observation that kernel data structures are the main key to any introspection tool. DKSM is able to add or remove specific fields from particular kernel data structure. By doing so, the template-based approach of existing introspection tools will be using the wrong templates to infer guest states and thus derive inaccurate results. This is because of the wide spread of generic pointers that facilitate the malicious manipulation of kernel dynamic data. Generic pointers can only get their values/types at runtime based on the calling contexts of the running operating system. Thus, modifying these pointers, and adding or removing fields would not violate any integrity constraints of the running operating system and thus affect the accuracy and reliability of the introspected results.

Psyco-Virt [100] is an agent-based intrusion detection system that implement host and network intrusion detection systems via *virtual machine introspection* framework to protect Xen environments. Psyco-Virt places agents in the hosted virtual machines to overcome their semantic gap, and monitors the changes of specific memory pages of the virtual machines that store the instructions of critical components, such as the installed agents. These pages are marked as read-only, and any attempt to modify them implies that an attacker is trying to execute arbitrary instructions. IntroVert [101] is a virtual machine introspection framework that bridges the semantic gap using operating system's kernel APIs to acquire the internal state of the virtual machine. ReVirt [94] utilized virtual machine introspection to monitor the execution of the guest operating systems and the installed software. ReVirt also depends on the kernel APIs to bridge the semantic gap. SADE [8] depends on placing stealthy security code inside the guest operating systems to overcome the semantic gap and actively protect them. These researches mostly depend on kernel APIs and memory to overcome the semantic gap, instead of depending

on their experience and knowledge of the operating system kernel data layout. This approach is more generic where it can cover a wider range of kernel data, not like the manual approach is limited to a small subset of kernel dynamic data. However, this approach is highly untrusted. Kernel APIs can be easily modified either by kernel rootkits or by the virtual machine owners themselves. Thus kernel APIs can pass false information to the introspection framework about the internal state of the running virtual machines.

3.2.1.3.1 *Active Monitoring*

Another important issue in implementing *virtual machine introspection* frameworks is the active monitoring feature that enables preventing the threats instead of just passively detecting them. Payne *et al.* [102] explored how *virtual machine introspection* techniques can be utilized to apply secure active monitoring inside virtualized environments. Their approach to apply active monitoring depends on installing security hooks inside the guest operating systems. These hooks are protected at a hypervisor level to prevent any malicious access to them. This is done by applying page-write permissions on the memory pages that contain the security hooks. Kvmsec [103] is a system that can monitor and protect guest virtual machines in real-time to control unauthorized changes inside guest virtual machines. Kvmsec relies on the KVM hypervisor and is composed of two main components, one in the guest to enable active monitoring and the other in the KVM itself. The guest code has access rights to kernel memory and the virtual machine monitor. SIM [104] is a secure in-VM monitoring system that enables active monitoring by places a security code in a dedicated and protected guest address space inside the operating system of a hosted virtual machine.

The approaches enable active monitoring by placing security code inside the hosted virtual machines. According to our threat model – discussed in chapter 4 – the approach of placing security code in the hosted virtual machines is not secure enough to be implemented in our *virtualization-aware* security solution. This is because virtual machines, including operating system and virtual hardware, are under the management and control of the cloud consumer.

3.2.1.3.2 *Virtual Appliances*

Virtual appliances have been recently introduced as a robust solution to deploy security software of virtualized environments, such as virtual machine introspection frameworks and cloud security applications. Alexander *et al.* [105] introduced a composition-as-a-service cloud model (*i.e.* virtual appliance) that leverages virtualization and IaaS for software service composition and deployment. Alexander proposed a cloud-agnostic modelling of solution requirements that can be transformed into cloud-specific configurations and be used to automatically generate a deployment plan. Chiueh *et al* [8] also introduced the idea of virtual appliances to deploy security software designed to monitor virtual machines *via virtual machine introspection*. Virtual appliances are a promising deployment model that eliminates memory redundancy in the server physical memory through the memory de-duplication features that is supported in most hypervisors. This is done through kernel component refactoring that takes out common kernel components shared by virtual machines running on the same physical machine and runs them on a separate virtual appliance [8].

3.3 Operating System Security

Researchers have extensively studied operating system security including kernel code and data. There are two main techniques used to analyse and defeat operating system threats: static and dynamic analysis. Static analysis of operating system rootkits focuses mainly on studying and identifying rootkit behaviours in order to formulate a set of integrity constraints that can defend against these rootkits at system runtime [106-109]. For dynamic malware analysis, there exist two major approaches: internal analysis based on kernel memory and APIs [110, 111], and external analysis using *virtual machine introspection* techniques or PCI memory controllers [6, 9, 21, 112-114]. In this section, we give more attention to external analysis techniques as one key requirement of our research project is protecting operating systems running in the virtual machines externally. Moreover, even if the external monitoring is not a design requirement, external monitoring and security is much more robust than internal analysis approaches and guarantees highly accurate results, as discussed in chapter 2.

The vast majority of current external analysis and operating system security approaches are based on the Xen hypervisor. Dewan *et al.* [115] introduced a hypervisor-based security system to secure the runtime memory against malware with root privileged access to the memory by protecting the page tables of the target program in the physical memory. Their approach mainly depends on creating another set of page tables, named protected page tables, in the physical memory area that is assigned to the hypervisor. After verifying the target program that needs to be protected, their memory protection module copies the entries that correspond to the program virtual address range from the shadow page tables to the protected page tables. Next, the protection module marks all entries in the protected page tables as “not present” and the corresponding in the shadow page tables as “not present”. Any attempt by the operating system to access the virtual address space of a specific process, a hypervisor page fault happens, causing a virtual machine exit event to the hypervisor to intercept this page fault. Based on the EIP (Instruction Pointer register) and CR3 registers values, the protection module determines the type of the memory access to allow or not. If the instruction is valid, the protection module switches the pages tables by changing the value of the CR3 register. Despite the robustness of this approach, it incurs a very high performance overhead on the running operating system which might not be acceptable by the virtual machine owner. XenKIMONO [97] is an intrusion detection system based on Xen hypervisor for introspection and guest integrity enforcement. XenKIMONO is implemented in a form of a daemon process into Dom0 of the Xen hypervisor to detect kernel code violations and kernel hidden objects. Code violations are detected by calculating the hashes of specific memory areas, and then at runtime it periodically recalculates these hashes and compares it to the saved values. For kernel objects, XenKIMONO compares its computed external view against an internal view of the virtual machine that has been captured by an internal program.

Lares [102] leverages hardware-based page-level protection so that any write-access to the protected memory pages with the kernel hook can be monitored and verified. Sharif *et al.* [89] presented a general-purpose framework to monitor operating system applications against kernel-based rootkits in the hosted virtual machines, by utilizing hardware memory protection

techniques to place a security code inside the hosted virtual machines to protect the operating system kernel. HookSafe [116] is also a hardware-based protection system that relocates kernel hooks from their original locations to a page-aligned centralized location, and then use a thin hook indirection layer to regulate accesses with hardware-based page-level protection. In other words, they create a shadow copy of the kernel hooks in a centralized location, and any attempt to modify this copy will be intercepted and verified by the hypervisor. Regular read access will be redirected to the shadow copy directly with no inspection. HookSafe claims that it can defend against system hooks with 6% slowdown in performance benchmarks. Manitou [117] is a system that ensures that a virtual machine only executes authorized code by computing the hash of each page before executing the code it includes. The page is executable, if authorized hashes list contains the corresponding hash value. Laureano *et al.* [118] employed a behavioural-based detection tools of anomalous system call sequences after a learning phase in which “normal” system calls are identified. Processes with anomalous system call sequences are labelled suspicious. For these processes, certain dangerous system calls will in turn be blocked. Rkprofiler [114] is a security tool that monitors the behaviour of kernel malware in virtual machines. They capture the function calls made by the kernel malware and constructs call graphs from the trace files. They propose a method called Aggressive Memory Tagging by performing symbol resolution using the Microsoft Symbol Server.

3.3.1 Kernel Data Rootkits

A main focus of this research project is defending against kernel dynamic data threats. Defending against kernel data rootkits is one of the biggest research problems in the operating system security area. There are lots of approaches and techniques used to defend against different types of kernel data rootkits [20, 119-122]. In this section, we survey the most prolific techniques and approaches used to defend against pointer rootkits that affect kernel data such as memory errors and bugs, hooking rootkits, and object hiding attacks.

Gibraltar [17] is an external detection tool for kernel data structures rootkits, based on a PCI card. It is based on a set of *value-invariants* of well-known kernel data structures. The

invariants are created based on observing the running state of the kernel during a training time. The invariants are then formulated as specifications of the kernel data that are enforced at runtime. Petroni *et al.* [20] presented a specification-based approach to model the well-known control kernel data structures based on the experience of operating system security experts. Then these experts are formulated as a set of specifications for the target data structure to be enforced periodically at runtime. Part of these specifications is a set of constraints that must hold at runtime in order to keep the system correct. The provided specifications are a C-like constructs – as introduced before by Demsky and Rinard [123] – to describe the layout of the objects in runtime memory. One of the main limitations of these approaches is how the *value-invariants* are formulated, and based on what criteria the field of the data structures are being selected to be included in the proposed invariant. Those approaches depend mainly on knowledge of the operating system kernel data layout to manually formulate the invariants set.

Dolan-Gavitt *et al.* [16] proposed an automated mechanism to formally guarantee the robustness of the value-invariants of a data structure. They proposed an approach to generate robust signatures for kernel data structures, by employing a feature selection process that ensures that the features chosen are those that cannot be controlled by the attacker. This is done by profiling the target data structure to determine the commonly used fields, and then fuzz these fields to determine which are essential to the correct operation of the operating system. In other words, the proposed approach profiles operating system execution in order to determine the most frequently accessed fields of a target data structure and then actively tries to modify their contents to determine which are critical to the correct functioning of the operating system. Despite the robustness of this approach, it is time consuming and impractical to be implemented, where operating systems' kernels have thousands of data structures that needs to be trained, observed and fuzzed to formulate their signatures.

K-Tracer [124] is a system for extracting malware behaviour. This is done by performing automated analysis to the data manipulation behaviour of rootkits in a sandboxed environment. The analysis is a flow-sensitive dynamic slicing applied on the well-known kernel data structures – *e.g.* system calls and interrupts – through the executed code paths. They applied back-

ward and forward dynamic slicing on selected execution paths of the kernel to identify all sensitive data (pre-defined) and the events that manipulate these data manipulations. Such analysis reveals information about data access patterns, data modifications and triggers used by the rootkits. K-Tracer is only able to detect known attacks by detecting their malformed behaviour, and not able to detect zero-day threats.

OSck [1] is a system for specifying and enforcing the integrity of the operating system data structures by detecting violations in the integrity properties specified to the hypervisor. OSck defends the static and persistent kernel data, such as system call table, by write-protecting their kernel text, and protect the dynamic data by ensuring that dynamic control transfers targets functions that are safe for dereferencing at a particular call site. This is by verifying that all paths of the kernel could be traverse from a global root to a function pointer will result in calling a safe function. They also provide an API interface to enable writing the required integrity checks for in-memory kernel data structures, by running a privileged process inside the guest. This approach is efficient for control data structures, by enforcing write-protect on their kernel text, but not suitable for non-control data structures that change at runtime. This approach is also not trusted to be applied for IaaS security solutions. This is because providing an API interface to write the integrity checks for non-control data structures will violate one of the key technical requirements of *virtualization-aware* security solution, which is *in-guest* code cannot be a part of the security solution.

Wang *et al.* [125] proposed a different approach, HookMap, which systematically discover all kernel hooks that can used by persistent kernel rootkits to tamper with the kernel data and hide themselves. This is done by monitoring the kernel-side execution of the security software, to uncover all kernel hooks related to the execution path of the security software that can be hijacked by a rootkit. In other words, they perform dynamic analysis to track the call and jump (control flow transfer instructions) instructions to identify the possible kernel data hooks. They start from an identified control flow transfer instruction and examine in backwards manner any related instructions to identify the source that might affect the destination value. After identifying the hooks, they resolve the memory address to get the semantic definition by using the

symbol table of Linux operating system. HookMap then installs breakpoints inside the guest kernel in a specified kernel function to detect the context switching. Despite the effectiveness of these approaches in formulating a set of integrity constraints to specific kernel rootkits that can be applied on selected execution paths of the kernel; these approaches are quite limited in detecting zero-day threats that target to manipulate new kernel data and execution paths in the kernel code.

Table 3-1 shows a comparison between key aspects of the previously discussed approaches. This table summarizes their approaches to: (i) defend against which category of kernel data structures – *i.e.* control or non-control data structures; (ii) the techniques used to specify and enforce the integrity constraints; and (iii) the main security features such as active or passive monitoring, detecting zero-day threats, external or internal monitoring, and semantic gap construction approach – *i.e.* memory traversal or value-invariant. The control DS column denotes kernel control data structures. The non-Control DS column denotes non-control kernel data structures. The Inv column denotes value-invariant approach. The CFA column denotes data-flow analysis. CFI column is the control flow integrity. Type denotes validating the types of the pointers of the data structures. Ex denotes external monitoring either via hypervisors, VMM or PCIs. RT denotes real-time monitoring. SG is the approach used to overcome the semantic gap, either memory traversing or AMD (Allocation-Driven Mapping). OS denotes the running operating system in the virtual machines. H denotes the hosting platform.

Table 3-1. A comparison between key related work in operating system kernel data integrity.

	Kernel DS		Security Techniques						Security Characteristics					Platform	
	Control DS	Non-Control DS	Dynamic Slicing	Kernel Analysis	Inv	DFA CFI	Type Safety	Ex	Zero-Day Threats	In-Guest Code	Prevent (Active)	RT	SG	OS	H
[124]		✓	✓	Dynamic		✓		✓					Traversing	W	QEMU
[113]	✓	✓		Dynamic				✓	✓				ADM	L	QEMU
[1]	✓	✓				✓	✓	✓	✓	✓		✓		L	KVM
[125]	✓	✓	✓	Dynamic		✓		✓					Traversing	L	QEMU
[17]		✓		Static	✓			✓						L	PCI
[96]	✓			Dynamic							✓			L	QEMU
[16]	✓													L	Windows
[126]	✓							✓		✓	✓			L	QEMU
[127]								✓					Traversing	L	Xen
[128]		✓								✓				W	Windows
[20]		✓			✓			✓						L	PCI
[120]				Static		✓		✓					Traversing	L	Xen

3.3.1.1 Object hiding attacks

Different approaches and techniques have been studied to detect object hiding rootkits. Some research efforts used data invariants – mostly hard-coded operating system expert knowledge – such as matching running processes list with the thread scheduler list [20, 128]. Some other tools scan kernel memory using signatures to detect hidden objects [18, 129, 130]. Others use kernel memory mapping techniques to detect hidden processes evidence [14, 17, 96, 120]. Others depend on logging malware accesses to the memory [114, 131], or track the value of the CR3 register with the process directory table base value such as Antfarm [132]. All of these approaches limit themselves to a specific range of kernel objects such as processes and threads with no attention for the other dynamic kernel objects that are a valuable target for direct kernel manipulation rootkits in order to run stealthy malware.

Other approaches are based on the cross-view comparison approach, where a comparison between two views (untrusted view from inside the operating system and a trusted view from outside the operating system using VMI or PCI-based approaches) is performed. Any discrepancies between the views are considered as hidden objects [14, 133]. VMwatcher [127], GhostBuster [134] and X-Spy [93] are hypervisor-based hidden object detection tools that are based on the cross-view approach to DKOM rootkits. The internal view in these tools is captured by running security code inside the virtual machine to read the internal state of the operating system. Xuxian Jiang *et al.* [127] also proposed a similar approach to detect stealthy malware through virtual machine introspection and cross-view comparison. Lycosid [135] is a virtual machine introspection-based hidden process detection tool. Lycosid is based on the cross-view comparison approach to detect hidden processes by comparing the length of the processes list captured from the raw hardware bytes and from inside the operating system high level internal view. If the trusted list is longer than the untrusted one, then a hidden process exists in the virtual machine.

Cross-view comparison approaches have limited effectiveness as the internal view can be subverted to serve the hackers needs and trick the internal tool with fake information. In addi-

tion, all of the above approaches are time-consuming and require a deep knowledge with the operating system kernel, in addition to the high performance overhead.

3.3.1.2 Locating Kernel Runtime Objects

Locating kernel runtime objects is an important task in many operating system security solutions such as kernel data integrity checking [17], memory forensics [18], brute-force scanning [19], virtualization-aware security solutions [21], and anti-malware tools [106]. However, current research efforts focus on some specific, well-known objects such as processes, threads and network connections [136].

PoolFinder [137] is a memory forensics tool that scans pool memory for object tags. PTfinder [18] scans the pool memory to list the running processes and threads using hardcoded offsets and addresses. Schuster *et al.* [18, 138] analysed memory dumps to search for processes and threads by analysing pool memory. The Memory Forensics Toolkit [139] lists the running processes and the loaded modules for Windows operating system memory dumps. MemParser [140] and Kntlist [141] are other tools that enumerate processes and dump their memory.

Other research efforts uncover system runtime objects using memory mapping techniques [1, 14, 21]. Such approaches are limited and not accurate, as: *first*, they are vulnerable to direct kernel object manipulation rootkits that exploit the *points-to* relations between kernel runtime objects to hide running objects. *Second*, they cannot follow generic pointer dereferencing, because they only leverage type definitions and thus cannot know the target types or values of these generic pointers. *Third*, memory traversal has a very high performance overhead because of the poor spatial locality of the operating systems.

Royal [88] introduced a hidden processes detection tool called “Azure”. Azure utilizes *virtual machine introspection* to locate the running processes for Windows XP in physical memory. Azure depends on a manually-developed kernel data definition that reflects the runtime data layout of the EPROCESS structure to detect their execution evidence in the memory. Azure then compare the computed processes to the current execution of the CR3 (A register contains the page directory pointer of the current process) to confirm its computed

results and detect hidden processes.

DeepScanner [24], DIMSUM [25] and SigGraph [19], use the value invariants of certain fields of a data structure as a signature to scan the memory for matching running instances. However, many kernel data structures cannot be covered by such value-invariant schemes, as discussed before in chapter 2. For example, it is difficult to generate value-invariants for data structures that are part of linked-lists, because the actual running contents of these structures depend on the calling contexts at runtime [26]. In addition, value-invariants approaches do not fully exploit the rich generic pointers of data structures' fields, and are not able to uncover the *points-to* relations between the different data structures.

Rhee *et al.* [113] introduced a runtime kernel memory mapping schema, called allocation-driven mapping. This schema enables systematically identifying dynamic kernel objects including their types and lifetimes. This is done by capturing objects' allocations and deallocations instructions, and without relying on the runtime content of the kernel memory. To detect the type of an allocated object, they systematically capture code positions for memory allocation calls and the call site and analyse the call site offline to determine the type of the call object being allocated. This approach has a very high performance overhead and thus cannot be used in traditional operating system security tools where it can only be used in implementing the advanced debugging tools. Also, despite the feature of detecting allocations and deallocations in near real-time, they cannot even identify the object type. They need to analyse executed instructions offline to identify object types and details.

To the best of our knowledge all existing approaches, whether value-invariant or memory traversal – with the exception of KOP [14], and SigGraph [19] – depend on operating system experts knowledge to provide kernel data layout definition that resolves the *points-to* relations between structures. SigGraph follows a systematic approach to define the kernel data layout, in order to perform brute force scanning using the value-invariant approach. However, it only resolves the direct *points-to* relations between data structures without the ability to solve generic pointers ambiguities, making their approach unable to generate complete and robust signatures for the kernel. KOP is the first tool that employs a systematic approach to solve the indirect

points-to relations of the kernel data. However, KOP is limited in that: the *points-to* sets of the `void` typed objects are not precise and thus they use a set of operating system-specific constraints at runtime to find out the appropriate candidate for the objects. KOP assumes the ability to detect hidden objects based on the traditional memory traversal techniques that are vulnerable to object hiding. Moreover, both KOP and SigGraph have high performance overhead to uncover kernel runtime objects in a memory snapshot.

3.3.2 Operating System Verification and Unsafe C Memory

One of the main major weak points in operating systems' kernels is the high usage of C pointers in kernel code and data, as discussed before in chapter 2. These pointers are a major source of kernel rootkits that target kernel dynamic objects [20, 22, 126, 131] and cause memory errors and bugs [142-146]. Many techniques have been presented to check for memory bugs and errors whether by verifying user pointer dereferencing statically using static memory checking tools or software model checkers, or by analysing the source code of the operating systems or the intended program using *points-to* analysis techniques, separation logic or shape analysis. In this section, we review these techniques and focus more on *points-to* analysis techniques with its different analysis aspects, because *points-to* analysis was a selected solution in our research project.

3.3.2.1 Static Memory Analysis Tools

As discussed in chapter 2, many of the memory errors in operating system occur because of the weak typing of the C language. Foster *et al.* [147] introduced a framework for adding type qualifiers to C programs, by extending the standard type rules of C language to model the flow of qualifiers through a program. The framework checks for unchecked user pointer dereferencing in the Linux kernel. While this approach is effective, there is a high performance overhead for the running operating system.

Many static memory checking tools have been introduced over the last decade to check for memory errors and bugs. The main objective of these tools is to automatically determine

runtime properties (*e.g.* pointer dereferencing and type information) at compilation time. CRED (C Range Error Detector) [148] is a dynamic bounds checker which checks for buffer over-flow exploits in user input data that may cause runtime memory errors. CRED is based on the idea of referent objects [148]. A pointer's referent object is the intended object to be referenced by that pointer. Thus, when a pointer is dereferenced, it must point within the bounds of its referent object. MECA [149] is a system and annotation language for memory error checking. MECA works by performing a flow-sensitive static analysis on the programs to locate unchecked user pointer dereferencing. Sparse [150] is another static memory error finding tool developed by the Linux community to bad pointer dereferencing in Linux kernels. ESP [151] is a flow-sensitive static analysis program verification tool that verifies unchecked user pointer dereferencing in Windows operating systems. Blanchet *et al.* [152] also introduced an abstract interpretation static program analyser to verify user pointer dereferencing of the critical embedded real-time software.

Software model checking algorithms are also used to check for bugs and errors in large program. Software model checking is an algorithmic analysis of programs designed to prove properties of their executions [153], such as SLAM [154] and BLAST [155]. The software model checking approach is hard to apply on operating system kernels where the program size is extremely large. This makes the cost of developing the specification and the model of the program too expensive. Thus, software model checking approaches are not as yet recommended for large operating systems.

Separation logic [156] is an extension of Hoare logic [157] to deal with pointers. Separation logic has been used widely to formally verify and check for operating systems' memory errors and bugs. In particular, separation logic is used to formally represent and verify heap data structures. Separation logic is based on the assumption; if the precondition includes all memory locations and all possible stores, variables and values that can be accessed during the program runtime, this will never lead to runtime memory error. Many research practices apply separation logic in different ways to perform formal verification of operating system memory especially heap memory [158-160]. Kolanski *et al.* [161] introduced an approach based on separation logic

for reasoning about virtual memory in operating systems, *e.g.* page tables, physical and virtual memory accesses, and shared memory. Marti *et al.* [160] also proposed a formal verification method of the heap manager of Topsy embedded operating system using separation Logic. Metha *et al.* [162] proposed a modelling and reasoning approach for C programs, by representing head data structures as a mapping from a location to value using separation logic. Filliâtre *et al.* [163] introduced a verification tool that takes a C program annotated with assertions and generates verification conditions. Separation logic is a highly effective mechanism to formally verify pointers and memory locations of heap memory. However, it is quite complicated to apply, especially if the verification targets to cover all kernel memory, not just heap memory of the user-mode where `void` pointers and `null` pointers are unlikely to exist.

Another approach used to check for memory errors is shape analysis. Shape analysis is a program static analysis approach that discovers accurately the shapes of data structures allocated in heap memory at a program point of the program's execution path [164, 165]. Shape analysis goes a step beyond pointer aliasing information to infer properties such as whether a variable points to a cyclic or acyclic linked-list [164]. Shape analysis can be considered a simple form of pointer analysis that focuses on data structures, such as linked-lists, rather than all the regular pointer variables. Lee *et al.* [166] presented a shape analysis approach to automatically discover the shape of complex data structures in large programs such as linked-lists and trees. The discovered shape is then used to check for memory errors in the corresponding program at system runtime. Anders *et al.* and Češka *et al.* [164, 167] introduced a shape analysis approach based on the use of generic higher-order inductive predicates. The generic higher-order inductive predicates describe the spatial relationships with a method of synthesizing new parameterized spatial predicates that can be used in combination with the higher-order predicates. Further work [168-170] also introduces a shape analysis approach to express invariants of data structures allocated in heap memory using a grammar-based language. Shape analysis is not our best solution to check for memory errors. In our research project, we do not only focus on data structures. We include every pointer and pointer-compatible variable in our analysis including operating system global and local variables, generic pointers and function pointers, as discussed

in chapter 6. Moreover, our approach flattens data structures to a scalar field, where each data member of a data structure should have its own and separate points-to sets and invariants. Based on that, *points-to* analysis was the best approach to be used in our research project to solve the problem of generic pointers ambiguity and weak typing in C-based operating systems. In the following section, we discuss key related work in *points-to* analysis techniques including the main analysis aspects of *points-to* analysis algorithms which as: such as context-sensitivity, flow-sensitivity and field-sensitivity.

3.3.2.2 *Points-to Analysis*

Points-to analysis has been a rich point of research to solve a lot of technical problems in programs, developed in C\C++ and Java such as memory error detection, program understating problems, and compiler optimization [27, 34, 171-178]. C\C++ programs have the greatest attention because of pointers, casting and weak typing problems discussed previously in chapter 2. Basically, *points-to* analysis mainly differ in how we group aliasing information. The first two approaches to achieve *points-to* analysis were introduced by Andersen [33] and Steensgaard [34]. Anderson's approach creates a node for each variable and the node may have different edges connected to other nodes in the same type-graph, while Steensgaard's approach groups alias sets that point to the same memory locations and dereference the same objects in one node and each node have only one directed edge. Both approaches are flow-insensitive and context-insensitive points-to analysis algorithms. Andersen's is the slowest but the most precise and Steensgaard's is the fast but less precise. Based on these two main algorithms, many other algorithms have been developed with impressive improvements in algorithmic scalability [179-181] and performance overhead [35, 182].

Most *points-to* analysis research efforts focus on achieving scalable and efficient context-sensitive and field-sensitive on large programs [35, 36, 173, 175, 177, 179, 183, 184]. Achieving field-sensitivity is relatively straight-forward and the difficulty always remains in achieving fast, scalable and precise context-sensitivity. Roughly speaking, context-sensitivity is usually achieved by sacrificing performance or scalability [185-187]. Context-sensitivity is

achieved via different approaches such as computing transitive closure, computing SSAs (single static assignments), heap cloning and CFL-reachability (Context-free Language) formulation. Almost all of these different approaches are applied through the standard *points-to* analysis phases; intraprocedural, interprocedural and context-sensitive analysis phases.

Avots *et al.* [29] presented a context-sensitive and field-sensitive *points-to* analysis algorithm in order to enable detecting pointer dereferencing vulnerabilities in C programs. Their analysis are based on the assumption that; each object is allocated in a separate memory space and; a pointer to an object can only be derived from a pointer to the same object. Context-sensitivity is handled by applying a cloning-based approach. Cloning-based approaches conceptually means generating multiple instances of a procedure such that each single calling context for a procedure invokes a different instance of it. Contexts in their algorithm are distinguished using the entire call-paths between callers and callees. Avots *et al.* also introduced a type inference approach based on their *points-to* analysis algorithm, to determine the actual target types of the fields that have declared `void` typed pointers or objects. This helps in computing a set of types that can be safely dereferenced through a pointer or an object at system runtime. Avots approach is scalable to analyse medium-size programs of around 30K LOC (Lines of Code) in a reasonable time with precise results. Liang *et al.* [188] also introduced a fast modular context-sensitive analysis algorithm that is based on heap cloning approach. Liang's algorithm mainly identifies and analyse runtime memory locations that could be passed to a procedure. Lattner *et al.* [173] introduced a new fast full-heap cloning *points-to* analysis algorithm, named DSA, to achieve context-sensitivity for large programs of 200K LOC in a few seconds. The basic idea behind the fast scalable analysis of Lattner's approach is incrementally building the call-graph of the program during the *points-to* analysis, not in a separate analysis step as happens in many algorithms. DSA is also capable of analysing linked-lists and data structures to identify their lifetime and actual runtime types. Recursions in DSA are handled by building the Strongly Connected Components (SSCs) of the program – based on Tarjan's algorithm [189] – of the call-graph in the bottom-up analysis phase. Then, for each strongly connected component, context-sensitivity analysis is applied instead of applying the con-

text-sensitivity analysis for each function in a strongly connected component. The DSA algorithm is implemented within the LLVM compiler and the analysis is performed at the link-time of a compiled program. Despite the effectiveness and fast performance of the DSA algorithm, it cannot be used in our research project to achieve highly precise and scalable *points-to* analysis of an operating system's kernel source code. The DSA is mainly used in compilers to provide a fast aliasing approach that do not cover the generic pointers, casting and typing problems accurately. These problems cannot be resolved at compilation time and requires a static deep analysis of all pointers and pointer-compatible variables in the program.

Summary-based algorithms have been also used to enhance the analysis scalability and performance in achieving context-sensitivity. Summary-based *points-to* algorithms are more precise and less expensive than heap cloning in order to achieve context-sensitivity [36, 184]. Summary-based *points-to* algorithms are mainly based on computing procedure summaries that reflect all of their side effects as a caller beyond its context at system runtime. Nystrom *et al.* [184] presented also a summary-based context-sensitive *points-to* analysis to address the scalability and precision problems by in-lining function calls in the interprocedural analysis phase based on the computed procedure summarizes.

The context-free language-reachability formulation approach is used also to model heap objects and function calls [177, 178] to compute precise *points-to* sets, where a context-free language is defined in the analysis algorithm. CFL-reachability *points-to* analysis algorithms proved high precision but with less scalability for large programs. Xu *et al.* [178] introduced a context-free language-based algorithm that scales to analyse large Java programs with acceptable precision rate. Dereferencing operations in heap memory is formulated as an all-pairs context-free language-reachability problem over a simplified balanced-parentheses language. Instead of building the program call-graph to perform the *points-to* analysis, a symbolic *points-to* graph that reflects store and load operations is used instead, to enable faster *points-to* analysis. Context-sensitivity is then achieved by computing the functions summaries and then propagates the reachability information from callees to callers. Whaley *et al.* [181] proposed a fast and scalable context-sensitive *points-to* analysis for large Java programs of around 700K

LOC. The high performance and scalability of this approach comes on the cost of ignoring heap variables from the context-sensitively analysis phase. Generally speaking, performing *points-to* analysis on C programs is more difficult than Java programs, simply because Java is a more type-safe language where it handles the generic pointers problem in its software development platform. Heintze *et al.* [182] achieved context-sensitivity by computing the dynamic transitive closure of the program, where indirect *points-to* relations are resolved *via* reachability queries on the transitive closure graph. This algorithm assumes that an edge between two variables must be a non-null path. This means that paths between generic pointers cannot be queried on the transitive closure graph, which is a mandatory part of our research problem. In [181, 190] context-sensitivity is handled using the full call-paths of the call sites, and they sacrifice scalability for precision.

Buss *et al.* [191] designed a *points-to* analysis algorithm to support implementing source-to-source transformation tools. The proposed *points-to* analysis algorithm operates on program's abstract syntax tree instead of the program's source code directly. Their proposed *points-to* analysis algorithm is an iterative flow-insensitive and context-insensitive. Buss *et al.* use the must and may *points-to* relations in their algorithm to achieve precision. A must relation is assigned if all the possible execution paths to a program point include a direct dereferencing operation, and a may relation is assigned if an indirect dereferencing operation is encountered during program execution. They handle return pointers by computing the possible return values to be bound to the left-hand side of the assignment statement at the call site. Function pointers are also handled in a traditional way by building an iterative function call graph. Buss *et al.* *points-to* analysis algorithm can analyse small C programs up to 28K LOC within reasonable time and acceptable precision.

Yu *et al.* [36] proposed a context-sensitive and field-sensitive *points-to* analysis with a full-sparse flow-sensitive analysis on a static single assignment flow-insensitive graphs. This enabled achieving higher scalability for medium-sized programs. The basic idea behind the Yu *et al.* algorithm is introducing *points-to* levels that allow computing the *points-to* relations of a pointer at a particular level based on the *points-to* relations of the pointers at the higher levels in

the system graph. This is based on the assumption that whenever a variable is analysed, all the other variables that may have an impact on its value or dereferencing operations either have been analysed earlier or are being analysed at the same time. The main problem of this algorithm is the double analysis of the control flow graph of the target program. The control-flow graph is built once to construct the static single assignment form of the pointers of a current level, and the second time to propagate the *points-to* sets of each level to their use sites. Moreover, static single assignment is not highly precise when analysing linked-lists and data structures.

One of the key problems with achieving context-sensitivity while adding an acceptable performance overhead is cycle detection. Most *points-to* algorithms mainly depend on detecting cycles in the call graph and then collapsing the cycle components into a single node. Thus, all nodes in the same cycle are guaranteed to have similar *points-to* sets and can safely be collapsed together [35]. However, this method has a significant performance overhead on the algorithm's performance as their cycle detection approach works on the whole call graph, whether a cycle exists or not. Pearce *et al.* [192] presented an efficient mechanism for online cycle detection. In order to avoid cycle detection at every edge insertion in the program type graph, the algorithm dynamically maintains a topological ordering of the call graph of the program. Pearce *et al.* later proposed a more efficient algorithm [193]. Rather than detecting cycles at every edge insertion, the entire call graph is periodically swept to detect any cycles that have been formed since the last sweep of the program [35]. Ben and Lin [35] discussed an approach for efficient cycle detection online, called lazy cycle detection. This technique is lazy because rather than trying to detect cycles when they are created in the call graph, it waits until the effect of the cycle becomes evident. Such approach has some pros and cons; however the advantage of decreasing the performance overhead without highly affecting the accuracy of the points-to analysis algorithm outweighs the disadvantages which mainly lie in the false positive rate of detecting cycles.

Flow sensitive analysis is no less important than field and context-sensitive analysis and it also has been a rich research area. Hind and Pioli [194, 195] discussed the benefits of combining flow sensitivity with context sensitivity to improve analysis performance and precision. Hind and Pioli presented a flow-sensitive *points-to* analysis algorithm using a classical iterative

analysis approach where pointer information are passed only to callees that are reachable from a global variable or from one of the procedure parameters in the call graph. Wilson *et al.* [196] presented a context and partially flow sensitive *points-to* analysis for small program based on partial transfer functions that summarize the effects of procedures at the intraprocedural analysis phase. Hasti *et al.* [197] proposed a technique to iteratively build single static assignments form for program variables with known aliasing to perform partial flow sensitive *points-to* analysis. Chase *et al.* [31] achieved flow sensitivity also by computing single static assignments form dynamically during the *points-to* analysis phase instead of computing it in a separate phase to enhance the analysis performance. Zhu *et al.* [198] took initial steps towards using Binary Decision Diagrams (BDDs) to implement a precise flow sensitive *points-to*. Tok *et al.* [199] presented a technique to perform fast flow sensitive analysis using def-use chains to reorder the instructions of the program based on the computed def-use chains, in order to enhance performance. A def-use chain contains a definition of a variable and all the variable uses that are reachable only from that variable definition. Hardekopf and Lin [38] presented a *semi-sparse* flow sensitive pointer analysis that combines binary decision diagrams and single static assignment to perform a sparse analysis on program typed variables (not `void` typed pointers) and iterative flow sensitive analysis is performed on pointer variables, including `void` pointers, to improve scalability.

In our research project, we used *points-to* analysis in a new and a different way from the previous discussed approaches. The main objective of most of the previous approaches is to achieve scalability without losing the fast performance and precision of the algorithm. However in our project scalability and precision are the only factors that we consider in our analysis. Thus, none of these approaches meet our requirements in analysing the operating system's kernel as they do not scale to the enormous size and complexity typical of an operating system kernel. Moreover, most algorithms are used during program compilation to name objects by allocation site, not by the full access path, which do not solve null pointers ambiguity. Thus, they do not enable solving the ambiguity of `null` pointers.

To the best of our knowledge, KOP [14] – a Microsoft tool – is the first and only static analysis tool that employed *points-to* analysis in order to analyse kernel source code to solve generic pointer ambiguities. However, KOP has a number of limitations: KOP uses a medium level intermediate representation (MIR) of the kernel source code. This medium level intermediate representation complicates the analysis and results in improper *points-to* sets. This is because medium level intermediate representation is extremely big in size, omits very important information such as declarations, data types and type casting, and creates a lot of temporary variables that are allocated identically to source code variables and thus are not easily distinguishable from source code variables [200]. Also in KOP, the *points-to* sets of the `void` pointer typed variables are not highly precise and thus they use a set of operating system specific constraint criteria at runtime to find out the appropriate candidate type for the objects, and this consequently the runtime performance of the operating system. KOP also does not handle casting and `null` pointer problems. Moreover, KOP assumes that it has the ability to detect kernel dynamic hidden objects based on the traditional memory traversal techniques, however the traditional memory traversal techniques are vulnerable to direct kernel object manipulation rootkits that maliciously modify object pointers as discussed by Bahram *et al.* [15].

3.4 Related Work Limitations Summary

Developing a *virtualization-aware* security solution becomes an urgent necessity for IaaS cloud computing model. To the best of our knowledge, no current related work introduced such a solution to robustly and externally protect the hosted *virtual machines* in the IaaS platform. To develop such a *virtualization-aware* security solution, multi-disciplinary technical problems needed to be addressed in the following areas: virtual machine introspection, semantic gap and operating system kernel data security, and kernel memory, pointers and kernel dynamically allocated objects.

In this chapter, we reviewed key related work in these areas and we can conclude that each of these areas still has a number of limitations that makes it inappropriate to develop systematic and robust security software for IaaS platforms, as shown in Table 3-2. In summary, the main limitations of the previously discussed research efforts that relates to our research project are: the manual approach used by most virtual machine introspection techniques to overcome the *semantic gap* problem, the insecure implementation of active monitoring at the virtual machine level, the inability to defend against zero-day threats that could affect kernel dynamic data and kernel memory. Based on the limitations discussed in this chapter, we developed a set of new techniques and methodologies to address these limitations and meet our design requirements – that are discussed in chapter 4, 5, 6, 7 and 8.

Table 3-2. Key limitations in current related work.

Research Area	Technical Problems	Limitations of current approaches
IaaS Security	Virtualization aware security solutions	<ul style="list-style-type: none"> – The lack of a systematic security solution that can provision security from a cloud provider’s perspective to protect the hosted virtual machines in the IaaS platforms. – Current security solutions deployment models do not enable a central management for a cloud instance and thus does not support virtual machine migration between the protected servers and generic protection for the different running operating systems in a cloud server.
Virtual Machine Security	Systematic solutions for semantic gap	<ul style="list-style-type: none"> – Current research practices depend on a manual approach to overcome the semantic gap problem. Such manual implementation for semantic gap solutions has many critical limitations that make it unreliable to enable implementing robust security software. – The inefficient memory traversal techniques and value-invariants approaches used to uncover kernel dynamic objects at system runtime opens the door for a big number of system rootkits that depend on malicious pointer manipulation to take control over the running operating system.
	Active and transparent monitoring	<ul style="list-style-type: none"> – Most active monitoring implementations do not enable safe active and transparent monitoring of the hosted virtual machine that could be implemented in the IaaS platform. Also most of the current research efforts have a conflict with the design requirements of the proposed <i>virtualization-aware</i> security solution of this research project.

Research Area	Technical Problems	Limitations of current approaches
Virtual Machine Security	Performance overhead	<ul style="list-style-type: none"> – Performance overhead of current security solutions to provide real-time protection is high, as they mainly depend on duplicating the virtual machines' memory. The original copy works as normal however it does not pass instructions to the processor, instead, if the inspection was successful, the other memory space copy take control to pass the instructions to the processor. – Memory traversal and value invariant mechanisms also have high performance overhead that makes it difficult to support the implementation of near real-time security software.
Operating Systems Security	Generic pointers	<ul style="list-style-type: none"> – Almost few researchers who tackled the problem of disambiguating generic pointers located in operating system's kernel dynamic data, in order to enable systematic recognition of the runtime data layout of an operating system's kernel. However these approaches do not solve the generic pointers problem efficiently and pointers such as <code>void</code> and <code>null</code> pointers are still a pain in operating systems with no sufficient solution.
	Uncovering kernel runtime objects	<ul style="list-style-type: none"> – Current implementations of memory traversal techniques and value invariant approaches that are used to locate kernel runtime objects in kernel memory have critical security limitations and cannot efficiently and robustly uncover the runtime kernel dynamic objects.

Research Area	Technical Problems	Limitations of current approaches
<p>Operating Systems Security</p>	<p>Detecting zero-day threats</p>	<ul style="list-style-type: none"> – Most of the current research efforts focus on detecting a specific type of security threats that is already known. Zero-day threats have had little attention in most of the current research efforts. – Also systematic security solutions that have the intelligence to be rootkit self-detectable and self-mitigatable have been tackled in research to support efficient and fast detection of zero day threats that could target the different operating systems.
<p>Points-to Analysis</p>	<p>Scalability and Precision</p>	<ul style="list-style-type: none"> – Most of the existing <i>points-to</i> analysis algorithms do not scale to the enormous size and complexity typical of an operating system kernel. Moreover, they commonly are used during program compilation to name objects by allocation site, not by the full access path, which do not solve <code>null</code> pointers ambiguity. – In summary, most of the current points-to algorithms are not capable of providing highly scalable and precise points-to analysis that main solve the casting, weak typing, and <code>null</code> and <code>void</code> pointers problems.

Chapter 4

Developing a Virtualization-Aware Security Solution for IaaS Platforms

In chapter 2, we have discussed key security challenges and problems in the IaaS cloud platform, and in chapter 3 we reviewed the current research efforts and their limitations in meeting our research objectives. Based on that, we introduce in this chapter our solution to the *loss-of-control* security problem over the hosted virtual machines in the IaaS platform. In this chapter, we give an overview on the big picture of our research project to develop a *virtualization-aware* security solution that has the ability to provide pre-emptive protection externally for hosted virtual machines, without placing any security codes in their guest operating systems. In Section 4.1, we briefly present the main technical problems and design requirements that our security solution consider in its design and implementation phases, and in section 4.2, we overview the high-level architecture of our *virtualization-aware* security solution and our new approaches and mechanisms used to develop the different components of the proposed security software.

4.1 Introduction

As previously discussed in chapter 2, a key problem of the IaaS platform is the *loss-of-control* security problem over the hosted virtual machines. This makes the security process of these virtual machines a shared responsibility between the cloud provider and consumers. As shown in Figure 4-1, cloud providers are responsible for the security of all the cloud platform layers, while at the same time consumers are holding the security responsibility of their virtual IT assets. Thus, the greatest challenge in the IaaS cloud platform is applying a security mechanism that enables both cloud providers and consumers to fully utilize the virtualization advantages from both perspectives. In other words, these security mechanisms should: (i) enable providers to safely host virtual machines beyond the security level provided by the consumer using the

traditional *in-guest* security applications. (ii) Enable providers to safely host virtual machines without restricting consumers to a specific set of security mechanisms or security software provided and supported by the cloud provider.

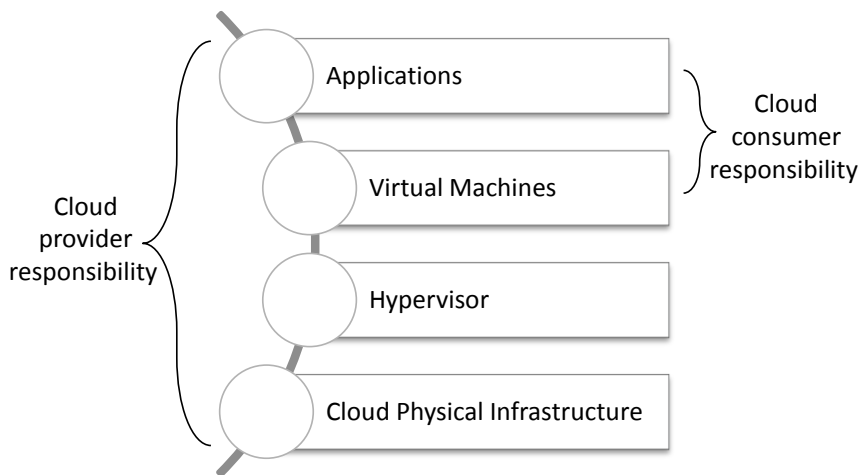


Figure 4-1. The shard-responsibility security problem in the IaaS cloud platform.

In such a threat model, protecting the IaaS platform using traditional *in-guest* security systems is not an effective solution that guarantees robust protection for the hosted virtual machines and the underlying virtual infrastructure. Thus, to efficiently address IaaS security requirements, new *virtualization-aware* security solutions need to be introduced. Such solutions should have the ability to actively protect the hosted virtual machines from outside the virtual machine itself and without relying on operating system kernel trustworthiness, by moving protection below the guest operating system level.

In this research project, we developed a *virtualization-aware* security solution that delivers advanced security to protect the running guest operating system instances of the hosted virtual machines externally. This security software ensures compliance (isolating critical workloads) and maximizes performance and operational flexibility for both cloud consumers and providers. In section 4.2, we briefly overview our general approaches and mechanisms to develop the main components of the proposed security solution.

4.2 A New Virtualization-Aware Security Solution

A *virtualization-aware* security solution should have and support specific characteristics and design requirements in order to achieve its goals. These include:

- *High Flexibility.* Allowing enterprises to outsource hardware while maintaining complete control and protection over the cloud platform without adding any security restrictions on the cloud consumers and their own virtual machines.
- *Mobility.* Virtualization enables live migration of virtual machines from one server to another, for the purpose of workload balancing and management, and performance and maintenance issues. Thus, a *virtualization-aware* security solution should be able to send and receive the security status of a virtual machine to and from the corresponding security software between the different servers of an IaaS platform to support safe virtual machine migration. The security software should also automatically – after the virtual machine migrates to another server – resume protection and update its computed memory addresses of the corresponding virtual machines in a near real-time. In short, the security software should be able to protect virtual machines at the virtual machine level, regardless of the location of the virtual machine within the cloud platform.
- *Scalability and Performance.* Server aggregation in cloud platforms duplicates the amount of workloads that run inside cloud physical servers. Thus it increases the complexity of managing and providing security for the cloud workload. Consequently, the security software should be highly scalable to monitor and protect multiple concurrent virtual machines at the same time with a single instance of the running security software, and with the lowest possible performance overhead. Running security software ordinarily adds a significant amount of overhead to the underlying operating system, and this overhead definitely increases in virtualized environment because of the hypervisor layer that traps all access requests to physical hardware from the hosted virtual machines. In addition, hypervisors introduce an additional layer of memory address translation because of the

shadow page tables that map a guest linear memory address to a host physical memory address which also affects the software performance.

- *Reliability and Robustness.* A *virtualization-aware* security solution should be reliable and resistant to the increasing level of rootkit efficiency that might compromise the security of operating systems and hypervisors. Moreover, hypervisor vulnerabilities will typically be exploited by a rootkit that has already compromised a virtual machine. So, the optimal way to protect against hypervisor vulnerabilities is to prevent rootkits from getting access to the hosted virtual machines in the first place.

Implementing the aforementioned security solution is a multi-disciplinary research project, where various research problems needed to be tackled in order to enable efficient and systematic protection for the hosted virtual machines in IaaS platforms. Figure 4-2 shows the high-level process of developing our *virtualization-aware* security solution. Five main research problems needed to be resolved in order to deliver this security solution, discussed in tions 4.2.1, 4.2.2, 4.2.3, 4.2.4 and 4.2.5.

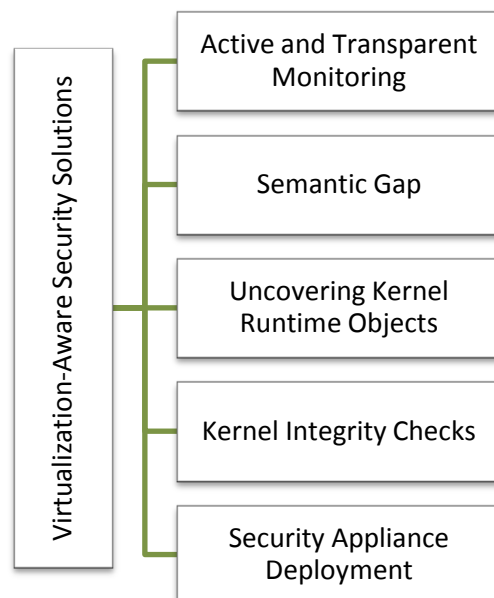


Figure 4-2. The high-level process of developing our virtualization-aware security solution.

4.2.1 Active Monitoring

Transparent and active monitoring is a paramount feature in any security software in order to enable the sufficient protection against system threats. A first and crucial step to implementing a *virtualization-aware* security solution is enabling efficient monitoring for the hosted virtual machines without installing any security code inside the guest operating systems or without relying on the operating system kernel trustworthiness. This is because guest operating systems are untrusted and thus security related information about a virtual machine internal state cannot be reliably obtained from the operating system kernel memory of APIs.

As discussed previously in chapter 2, *virtual machine introspection* techniques enable monitoring the hosted virtual machines externally. However, in order to implement a powerful introspection framework and solve the semantic gap problem properly, two main challenges needed to be addressed: (i) the introspection framework should be transparent to the running virtual machines and able to provide near real-time monitoring and protection. (ii) The introspection framework should have the ability to provide active monitoring instead of the common passive monitoring mechanisms to enable threat prevention not just detection. This also should be done without installing any security hooks in the guest operating systems.

To solve the monitoring problem, we developed *CloudSec*, a *virtual machine introspection* framework that transparently and actively monitors the running virtual machines in a virtualized computing environment. *CloudSec* utilizes *virtual machine introspection* techniques to externally monitor the running virtual machines in order to provide fine-grained inspection of the virtual machines' physical memory, at a hypervisor level. The main idea behind *CloudSec* is installing the security hooks at the hypervisor level instead of the guest operating system level. Particularly, *CloudSec* works as a watchdog at the hypervisor level with access rights to hosted virtual machines' physical memory *via* the hypervisor, as shown in Figure 4-3. Such design allows *CloudSec* to read the internal state of the running virtual machines without taxing the virtual machine performance or mobility. Details of *CloudSec* architecture and approach are discussed in chapter 5.

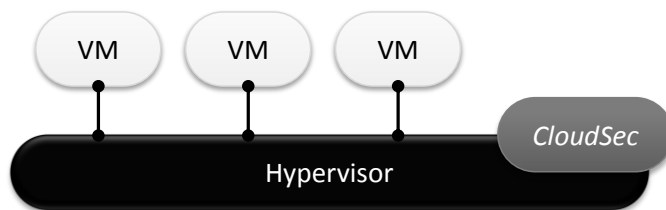


Figure 4-3. A simple view of how CloudSec works at a hypervisor level.

4.2.2 Semantic Gap

The second and most important step in order to provide systematic security for the hosted virtual machines is accurately interpreting the internal state of the hosted virtual machines from the introspected hardware memory bytes *via* the introspection framework, in order to overcome the *semantic gap* problem. The most robust solution for the *semantic gap* problem is accurately mapping between the guest operating system runtime kernel data layout and the virtual machine's underlying memory layout. However, as discussed before, one of the key problems of the C-based operating systems is the existence of thousands of data structures that have direct and indirect *points-to* relations between each other with no explicit integrity constraints. In such a complex data layout, the runtime memory layout of the data structures cannot be predicted during compilation time, making the process of mapping between the data layout and the memory layout very complicated. Moreover, as discussed in chapter 3, the problem with the current state-of-the-art solutions that follow this approach is that; most depend on their prior knowledge of the operating system kernel data layout to manually solve the *semantic gap*. Acquiring the runtime kernel data layout manually is not an easy task; as this requires massive hands-on experience with the kernel runtime data layouts in different operating system versions and kernel build. Also, such a huge effort in analysing a specific kernel version may not be applicable to another version of the same operating system. Thus, applying such limited approach represents a significant barrier to developing *virtualization-aware* security solutions. Another key problem with the current *virtual machine introspection* tools is that they are not operating system flexible tools. A single tool may have the ability to solve the semantic gap for a specific operating system such Linux or Windows, but not for both.

Based on the above limitations, we introduce, in chapter 6, OS-KDD. OS-KDD is a static analysis tool that has the ability to systematically build an accurate kernel data definition that precisely and statically reflects the runtime data layout of an operating system's kernel. OS-KDD is operating system flexible; it has the ability to analyse a wide range of platforms used to run mission-critical applications in the cloud platforms, including Microsoft Windows, Linux and Solaris. OS-KDD efficiently handles the generic pointers challenge of the C-based operating systems. OS-KDD takes the source code of an operating system kernel as input and outputs a directed *type-graph* that accurately represents the runtime kernel data layout, statically. The generated *type-graph* is not only used to efficiently overcome the *semantic gap* problem and get an external and accurate view of the virtual machine's internal running state, but also used to enable accurate uncovering for the dynamic kernel runtime objects and to generate a set of constraints on kernel dynamic data to be used in performing kernel data integrity checks.

4.2.3 Uncovering Kernel Dynamic Objects

A following step after overcoming the *semantic gap* problem and defining the runtime kernel data layout is uncovering – in near real-time – the running instances of operating system's kernel runtime objects in order to monitor their behaviours and check their integrity. As discussed before in chapter 2, accurately uncovering the running instances of the operating system kernel objects is an important task in many security solutions not just limited to *virtualization-aware* security solutions. However, current research efforts (whether those who depend on memory mapping techniques or value-invariant approaches) are limited for several seasons – discussed previously in chapter 2. Two key limitations of those approaches are the high performance overhead and the inability to handle the indirect *points-to* relations and their embedded objects properly.

Motivated by the limitations of these approaches and the need to accurately identify the running instances of runtime kernel dynamic objects from a trusted source that does not depend on the operating system kernel trustworthiness, we have developed a new approach called DIGGER. DIGGER is capable of systematically uncovering all system runtime objects without

a prior knowledge of the operating system kernel data layout in memory. DIGGER employs a hybrid mechanism that combines a new *value-invariant* approach and an advanced *memory mapping* technique, in order to get accurate results with nearly complete coverage for the kernel address space and with a low performance overhead. The value-invariant approach is used to discover the kernel objects with no need of memory mapping information, while the memory mapping technique is used to retrieve the object's details in depth including *points-to* relations with the other running data structures, without any prior knowledge of the runtime data layout in memory of the running operating system's kernel.

DIGGER uses the four byte pool memory tagging schema as a new value-invariant signature to uncover kernel runtime objects from the kernel address space. DIGGER then uses the generated *type-graph* from OS-KDD – that summarizes the different data types located in the kernel along with their connectivity patterns – to enable systematic mapping of objects' details after locating these objects in the runtime memory. In chapter 7 we discuss DIGGER, in detail.

4.2.4 Kernel Integrity Checks

The ability of a malicious hacker to remotely exploit a runtime vulnerability in a guest operating system is a significant threat that might give the hacker a complete control for the victim guest operating system. This would subsequently enable the hacker to break into the underlying cloud platform or the co-located virtual machines in the same physical server.

As the operating system kernel is the core of trust (where every other running application relies on it), we mainly focus on protecting the operating system kernel – especially kernel data. Kernel data rootkits are challenging to defend against because they assist in creating memory vulnerabilities without injecting stealthy code into the running guest operating system. Checking the integrity of an operating system's kernel data has been a major concern as reported in many operating system security research [1, 14, 20]. However, current research efforts and practices [16, 17, 110] are limited as they depend on their prior knowledge of the kernel runtime data layout to develop a set of integrity constraints to be enforced at system runtime. Current approaches also do not consider the widespread use of generic pointers across kernel data,

making their approach imprecise and vulnerable to a wide range of attacks that can exploit these pointers. Such approaches significantly reduce the likelihood of detecting zero-day threats.

In this thesis, we focus mainly on kernel data rootkits that target or modify the dynamically allocated kernel runtime objects. In chapter 8, we present a set of high-performance integrity checking tools that provides protection for the running guest operating systems in a near real-time fashion. Our integrity checking approach mainly depends on DIGGER and OS-KDD to systematically detect behaviour violations of the runtime objects due to malicious pointer modifications in memory. The proposed integrity checking tools mainly works on the kernel memory bugs caused by the memory runtime errors caused by pointers and unsafe types of C/C++. These tools mainly check for dangling pointers, hidden objects and function pointers' modifications within the dynamic kernel objects. In chapter 8, we discuss the approaches, implementation and evaluation details of these integrity checking tools.

4.2.5 Security Appliance Deployment

The final step in implementing our *virtualization-aware* security solution is deploying and integrating the previously developed components in a single solution that has the ability to monitor the virtual machines' behaviour, and detect and prevent threats. To provide this reliable and high performance security solution, virtual appliances are a good solution for deploying security software. Virtual appliances offer a new paradigm for software delivery by packaging pre-configured, virtualization-ready solutions in a single software package that is secure and easy to distribute, deploy and manage [201]. Virtual appliance deployment processes typically require installing the security solution artefacts, installing and configuring middleware containers *e.g.* web servers and databases, and setting up the communication, isolation, and security of the security software with the underlying cloud infrastructure [105]. In chapter 8, we discuss the details of developing and deploying our security appliance in the IaaS platform to meet our design requirements.

4.3 Summary

In this chapter, we gave an overview on the big picture of our research project which is developing a *virtualization-aware* security solution that has the ability to systematically protect the hosted virtual machines in IaaS platforms. In order to develop such a security solution, different new components and mechanisms need to be developed and integrated together in order to enable delivering systematic, accurate and high-performance security for the guest operating systems.

Chapter 5

CloudSec: A Virtual Machine Introspection Framework

A prerequisite to implementing *virtualization-aware* security solutions is to develop a transparent and active security monitoring framework that works at a hypervisor level. In this chapter, we addressed and discussed the challenges of implementing such framework and introduced a monitoring framework, named *CloudSec*. *CloudSec* is an introspection framework designed to externally and transparently monitor the hosted virtual machines in the IaaS platform. *CloudSec* utilizes *virtual machine introspection* techniques to provide fine-grained inspection of virtual machines' physical memory, in order to enable near-real time and active monitoring for the running operating system kernel and memory events.

5.1 Introduction

Virtualization has a great role in supporting the development of such security solutions by utilizing *virtual machine introspection* techniques. *Virtual machine introspection* enables observation of a running virtual machine's state and events from outside the virtual machine, at a hypervisor level. In particular, these outside observations are supposed to have a similar semantic view of system states as from inside the virtual machine [15]. Developing a trusted *virtual machine introspection* framework is the first and most important step towards developing a robust and effective security tool. Key problems behind utilizing *virtual machine introspection* techniques are the *semantic gap* and *active monitoring*.

Notwithstanding the drawbacks of the manual approaches used in memory mapping technique (discussed before in chapter 2 and 3) in overcoming the semantic gap, this technique was used in implementing *CloudSec*. This is because the main focus of *CloudSec*, in this chapter, is the introduction of an introspection framework that has the ability to provide active and transparent monitoring, rather than systematically overcoming the semantic gap problem. However,

we tackle the systematic overcoming of this semantic gap problem in *CloudSec₊₊* using OS-KDD, described in subsequent chapters. We developed *CloudSec*, a proof-of-concept prototype using the VMsafe libraries on a VMware ESX cloud platform.

CloudSec has the ability to provide feasible monitoring for the hosted virtual machines, without placing any security code inside the running operating systems of these virtual machines. *CloudSec* achieves active and transparent monitoring by placing security hooks into the hypervisor level rather than into the virtual machines' running operating system level. These hooks trap system events of the virtual machines, by interrupting memory call events at the hypervisor level, and then transfer control to the security solution rather than to the virtual machine itself. For instance, consider if a guest virtual machine issues a request to allocate memory or write to the memory *via* the hypervisor. The hypervisor will receive the request *via* a specific communication channel that exists between the guest virtual machines and a specific interface in the hypervisor, and then the hypervisor will transfer the control to the security solution *via* a separate secure communication channel in order to inspect the system event call request. After that a command is transferred back to the hypervisor which indicates whether to proceed with the request execution or suspend it based on the security solution's decision.

The rest of this chapter is organized as follows: The rest of section 5.1 discusses the main design requirements to implement *CloudSec*, in addition to the security threat model of *CloudSec*. Section 5.3 describes the high-level architecture of *CloudSec*, and in section 5.4, we explain in detail the steps to implement and deploy *CloudSec* in an IaaS platform. Section 0 describes an evaluation of the prototype, and finally section 5.5 discusses key strengths and limitations of *CloudSec*.

5.2 Threat Model

Basically, the threat model of *CloudSec* does not add new assumptions to the standard assumptions used in most *virtual machine introspection* tools [6, 7, 94]. We assume a trustworthy hypervisor, based on the assumption that the source code of the hypervisor is much smaller and more reliable than general-purpose operating systems. In particular, hypervisors are ideally

designed to be a thin software layer that is installed directly on the hardware, and their code is robustly verifiable and secure. Also, existing hardware based protection technologies including Trusted Platform Module [202] and Intel trusted Execution Technology [203] are capable of effectively establishing a root of trust by guaranteeing the loading of a hypervisor in a trustworthy manner. In other words, these technologies guarantee the load-time integrity of the hypervisor [116]. Based on these assumptions the hypervisor – including the virtual switch and all its kernel code – is part of the Trusted Computing Base (TCB) of *CloudSec*. The trusted computing base of a system is the set of all hardware, firmware, and software components that are critical to its security, in the sense that vulnerabilities occurring inside the TCB might jeopardize the security properties of the entire system. By contrast, parts of a system outside the TCB must not be able to misbehave in a way that would leak any more privileges than are granted to them in accordance to the security policy [204]. Based on this concept, the hosted virtual machines are not part of the TCB, and cannot be trusted to execute any monitoring/security code inside them. This is because a hacker might gain a root privilege access to the virtual machine. The malicious hacker might even be the virtual machine owner, and thus the hacker can modify any code or data in the operating system’s kernel. However, we assume that a malicious hacker does not have access rights to the real physical memory of the virtual machine. In virtualization, hardware events must go through the hypervisor first before getting back to the virtual machine and these events are monitored by *CloudSec*.

The design of our security architecture mainly depends on deploying a dedicated virtual machine, named the *security virtual machine*, to host the developed monitoring software. This security virtual machine has a privileged access to the hypervisor and is isolated from the other server workloads. Therefore, the security virtual machine is also part of the trusted computing base of *CloudSec* architecture.

Beyond these general assumptions, we have made an assumption specific to *CloudSec* architecture to allow more flexibility in the IaaS platform and to not conflict with the cloud consumer design requirements. We do not enforce the hosted virtual machines to undergo a secure boot cycle [205], as *CloudSec* works directly on the hardware bytes which are considered

a trusted source of information to detect hidden and polymorphism threats⁵. If true, this assumption makes *CloudSec* more flexible than other similar approaches. For example, Bryan *et al.* [206] assume a trusted execution environment for the hosted virtual machines where these virtual machines are isolated from malicious software running in the user environment.

5.2.1 Design Requirements

We designed *CloudSec* using a set of predefined security requirements in order to meet the security goals of the IaaS environment. These security requirements, described below, are the basic foundation for implementing *CloudSec*.

Active and Transparent Monitoring. Many current approaches implement a passive monitoring technique where an action cannot be taken to prevent a threat [120, 127, 132]. This makes the security software limited to being just a monitoring tool rather than a security tool that can prevent a threat and limit a threat consequences. Active monitoring means the ability to suspend system events execution and to pass control to the security solution for an action to be taken. There are many research efforts that provide an active monitoring feature in their security tools. Most depend on installing security hooks in the hosted virtual machines in order to trap system events [9, 102]. However, this type of active monitoring is not particularly trustworthy because part of the security software (which is considered part of the trusted computing base) is placed inside the untrusted virtual machine (which is not part of trusted computing base). Even if the security code is protected, it is still placed in the consumer's virtual machine where the cloud provider has no control over it. Moreover, one of the key goals in IaaS security applications is removing any security code from the hosted virtual machines.

Isolation. Security solutions should be isolated from cloud consumers to avoid tampering with the security software behaviour. Virtualization enables isolating multiple co-located virtual

⁵ Later in chapters 8, we discuss how our systematic approach has the ability to efficiently detect any modifications to the operating system kernel data, without the requirement of load-time integrity of the virtual machines' running operating systems.

machines running on the same physical host by using dedicated virtual switches and secure communication channels. Isolation makes it difficult for hackers to detect the existence of a security tool in their victim virtual machines. Thus, in *CloudSec*, all of its components are isolated and protected in a dedicated virtual machine with a privileged access to the underlying hypervisor. For better security, the security virtual machine is deployed as a virtual appliance. Recently, *virtual appliances* [201] have been introduced as a new solution for deploying security virtual machines. A virtual appliance, like a virtual machine, incorporates application, operating system and virtual hardware. However, virtual appliances differ from virtual machines in that they are delivered as preconfigured solutions running a “Just enough Operating System” (JeOS). JeOS is a purpose-built operating system that supports only the functions of the installed software [201]. JeOS occupies a much smaller footprint than a general-purpose operating systems and thus a JeOS is more stable and secure than a conventional operating system, reducing the number of vulnerabilities and exploits that could occur [8].

5.3 CloudSec High-Level Architecture

CloudSec utilizes *virtual machine introspection* techniques to provide fine-grained inspection of the hosted virtual machines’ physical memory, without installing any security code inside the virtual machines. Monitoring volatile memory enables effective detection of user and kernel rootkits, as volatile memory should have imprints for such malware, even self-hiding rootkits. *CloudSec* actively monitors the dynamically changing kernel data to enable effective detection and prevention for kernel data rootkits.

A key feature in *CloudSec* is the ability to provide active monitoring, which is a key requirement in order to provide effective security mechanisms. Active and transparent monitoring is efficiently achieved by moving the security hooks from the operating system level of the hosted virtual machines to the hypervisor level. This improves the reliability and trustworthiness of the security software. The security virtual machine contains the core of the security and monitoring tools, and it works directly with the hypervisor, not with the virtual machines, to detect system and memory call events of the running virtual machines. In Type I virtualization

the hardware interrupts of system calls go directly to the hypervisor and then they are either multiplexed within the hypervisor or passed to a special virtual machine that multiplexes the events [207]. This makes implementations of security hooks much easier at the hypervisor level.

CloudSec architecture is illustrated in Figure 5-1. The main idea behind designing *CloudSec* is having a private communication interface between the *security virtual machine* and the hypervisor. This interface enables the *security virtual machine* to get access to the other hosted virtual machines – *via* the virtual machine introspection APIs that are placed in the hypervisor – in a secure way and without a direct communication channel between the *security virtual machine* and the other hosted virtual machines. The communication channel is conducted over a separate virtual network using a separate virtual switch (vSwitch). The communication interface between the hypervisor and the security virtual machine is composed of two main components: *frontend component* and *backend component*.

Frontend Component. The *frontend* component is a set of C libraries that is part of the security software and is placed in the *security virtual machine*. This *frontend* component enables the security virtual machine to communicate with the hypervisor *via* a secure communication channel. It uses this to obtain information about the monitored virtual machines' running operating systems from the hypervisor and also to control access to physical memory and CPU registers. Particularly, the *frontend* component makes *CloudSec* an external extension of the hypervisor to enable transparent access to physical memory of the hosted virtual machines.

Backend Component. The *backend* component is part of the hypervisor code that enables the hypervisor to gain control over the hosted virtual machine to suspend any access to the physical memory and CPU according to the access triggers installed by *CloudSec* using the frontend. As the hypervisor mediates interactions between virtual machines and the host physical hardware, the guest physical memory is controlled by the hypervisor through the shadow page tables, and the hypervisor does not provide any direct control over the host physical memory even for *CloudSec* unless through the *backend* component.

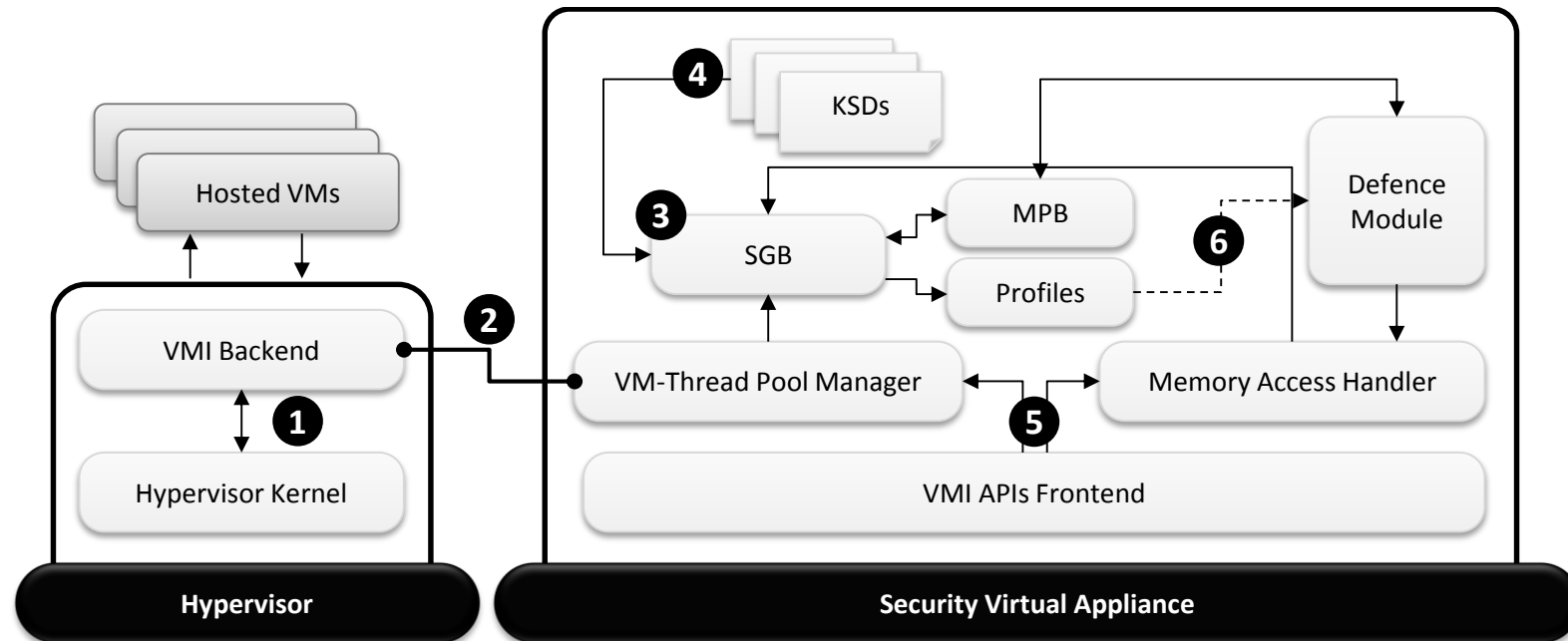


Figure 5-1. CloudSec high-level architecture.

5.3.1 CloudSec Operation

The goal of the security virtual machine is to scan the physical memory of a hosted virtual machine to locate system runtime objects. In order to enable active monitoring, the security virtual machine is loaded and started before any other hosted virtual machine. In particular, to be able to actively monitor a memory page/address of a hosted virtual machine, the memory pages/address need to be recognized by *CloudSec* first, in order to: (i) map the physical address of this page/address correctly in the address space of the physical server, and (ii) to be able to suspend its execution. Thus, after the physical server is booted up, the security virtual machine is started before any other hosted virtual machine, and the following steps take place in order:

- 1) Whenever a hosted virtual machine is started up, the *security virtual machine* is notified *via* the *frontend* component (1). *CloudSec* then creates a separate thread for each newly activated virtual machine using the VM-Thread Pool Manager, to analyse and monitor its memory (2).
- 2) *CloudSec* overcomes the semantic gap using the traditional memory mapping technique that maps between the hardware layout and kernel data layout. Thus, *CloudSec* first checks the control registers of the virtual machine's CPUs to identify the memory layout of the virtual machine's underlying hardware, and then checks the kernel version of the running operating system to get the appropriate kernel data definition⁶. A prerequisite to implement memory mapping techniques is having a kernel data definition that accurately reflects the runtime data layout of a specific range of the kernel data structures. In *CloudSec*, this definition was developed manually based on our knowledge and experience with the operating systems' kernels. A kernel data definition mainly reflects the direct and indirect

⁶ At this stage *CloudSec* cannot systematically recognize the kernel version of the running operating system in a hosted virtual machine, and it depends on the provider's knowledge with the running operating system version. Later in chapter 8, we discuss in *CloudSec++*, an improvement that has the ability to identify the kernel version of a running operating system from analysing the physical memory, and without the provider's support.

pointer relations among kernel data. The components and layout of the kernel data definition differ from one kernel version to another. Based on the kernel version, the *security virtual machine* loads the corresponding kernel data definition. Developing manual kernel data definitions are illustrated in the implementation section of this chapter.

- 3) After loading the appropriate kernel data definition, *CloudSec* starts solving the semantic gap through our Semantic Gap Builder (SGB) component **(3)** **(4)**. The main idea behind memory mapping techniques is traversing kernel memory starting from the operating system global variables and then following pointer dereferencing until all memory objects are discovered. Consequently, the semantic gap builder component reads specific physical memory pages based on the addresses of the operating system global variables that are computed in the corresponding kernel data definition. *CloudSec* does not have direct access to the virtual machines' physical memory. *CloudSec* gains such access *via* the communication channel with the hypervisor. The *backend* component reads these physical memory pages into the Memory Pages Buffer (MPB), and then the semantic gap builder component maps these physical memory bytes to the corresponding kernel data definition. This mapping builds an external view of the running virtual machine that accurately reflects the high-level semantic view of the running operating system. The constructed view includes a set of kernel runtime objects *e.g.* running processes and threads, loaded modules, system table, and interrupt, local and general descriptor tables.
- 4) After overcoming the semantic gap, *CloudSec* creates a profile for each virtual machine containing its reconstructed high-level semantic view, to be used by the Defence Module.
- 5) *CloudSec* then installs memory access triggers or timer-based triggers on the page(s) that contain the kernel data structures that needs to be monitored and protected according to the applied defence mechanisms. Whenever memory access to such pages occurs, the *backend* notifies the Memory Access Handlers

(MAHs) (5), and the hypervisor suspends execution. Memory access handlers load the requested memory page(s) into the Defence Module or the semantic gap builder component to extract kernel data structure updates (if required by the defence module). Then the defence modules analyse the current running kernel data structures for security threats, according to the applied defence mechanism, before control is given back to the virtual machine executed instructions (6).

- 6) Normally, upon completion of step 5, the security virtual machine signals to the hypervisor, via the *frontend* component, that the memory page/address has passed inspection and could be: (i) executed (execution of the interrupted virtual CPU in the monitored VM resumes), or (ii) discarded.

The previous process enables *CloudSec* to perform active monitoring, without installing any hooks inside the virtual machine to suspend instructions execution. Although the technique used in *CloudSec* to enable active monitoring presents a heavy page fault in the monitored virtual machine, we consider it the safest approach to implement active and transparent monitoring. On the other hand, one of the main reasons to adopt virtualization technology in developing security solutions for IaaS platforms is the isolation feature of the security software from the other running virtual machines. Thus, placing a security code in the virtual machine's running operating system violates this feature. However, to minimize the page fault performance overhead in *CloudSec*, the security virtual machine invokes only a limited number of pages that contain the important kernel data, as discussed in chapter 8⁷.

5.4 CloudSec Implementation Details

VMware® ESX 4.1 hypervisor was our chosen implementation platform. We chose this because of availability of the VMSafe libraries [208] that enables implementing *virtual machine introspection* frameworks. The ESX hypervisor is a type I hypervisor that is installed directly on the hardware. Thus, ESX hypervisor architecture supports the design requirements including active

⁷ Details of performance enhancements are discussed in chapter 8.

monitoring with no hypervisor modifications. VMsafe APIs enables reading the hardware bytes of the hosted virtual machines.

The monitoring software deployed in the security virtual machine of *CloudSec* is a C program written using Posix threads⁸ [209] and VMsafe APIs. The C program includes hard-coded offsets of the manually developed kernel data definitions, the semantic gap component module and a string matching module that is called whenever a signature matches the currently executing page of memory in the security virtual machine. As a proof-of-concept, we developed a manual kernel data definition for Windows XP SP3 and used this definition to accurately map the physical memory of the hosted virtual machines to uncover the running instances of kernel objects.

5.4.1 Memory Management

One of the key requirements needed to overcome the semantic gap problem using memory mapping techniques is to understand the underlying memory layout that is used by the operating system to map between physical memory pages and the kernel data definition of the running operating system. Thus, before exploring the implementation details of *CloudSec*, we discuss briefly in this section explain the memory hardware layout in operating systems, in addition to memory management techniques used in Type I hypervisors, as they are different from the traditional memory management model of conventional operating systems.

5.4.1.1 Hardware Memory Paging Modes

In this section we overview the basic paging modes of the hardware with reference to Intel® 64 and IA-32 Architectures Software Developer’s Manual Guide [210]. There exist four main paging modes supported by the hardware to manage the physical memory layout. These paging modes are controlled by the control registers CR0 and CR4 of the CPU whether for a physical host or a virtual machine. CR0 has different control flags that control the processor’s operating

⁸ POSIX is a set of APIs for implementing multi-threading.

and thus manage the hardware memory layout. CR4 is used in protected mode operations such as page size extension and machine check exceptions. For example, if the CR0.PG bit is set and CR4.PAE bit is clear, then 32-bit paging is used. If the CR0.PG bit is set and CR4.PAE bit is set then Physical Address Extension (PAE) paging is used.

In our proof-of-concept prototype, we used the 32-bit paging mode - physical address extension disabled. In this mode, the memory is divided into pages or frames of 0×1000 (4096) bytes each. Each process has a page directory table and each page directory contains 1024 page directory entries. Each page table entry contains an address for a page table and each page table contains 1024 page table entries, and each page table entry points to the base address of a page. Figure 5-2 summarizes this process. This gives $1024 * 1024 * 4096$ or 4GB total virtual address space for each process. In Windows operating system, the 4GB of virtual address space is by default divided into two halves: the first 2GB (from 0×80000000 to $0 \times FFFFFFFF$) is for the kernel address space, and the second 2GB (from 0×00000000 to $0 \times 7FFFFFFF$) is for the user address space.

When an application requests access (allocate, read or write) permission for a specific virtual address *via* a the corresponding system function call, the operating system returns a pointer to the requested memory page, along with an offset into that page for the requested address. This low-level access leaves the application to interpret the data within each page of memory. To translate a virtual address in a certain process user address space to the corresponding physical address space, we use the linear address translation mechanism which requires the physical address of the process directory table for the given process, as shown in Figure 5-2. CR3 control register enables the CPU to translate linear memory addresses into physical memory addresses. This is done by locating the corresponding page directory and page tables entries for the current executed task. The first 20 bits of CR3 are the page directory base register, which stores the physical address of the first page directory entry [210]. To calculate a kernel physical address from a given virtual address, we subtract 0×80000000 .

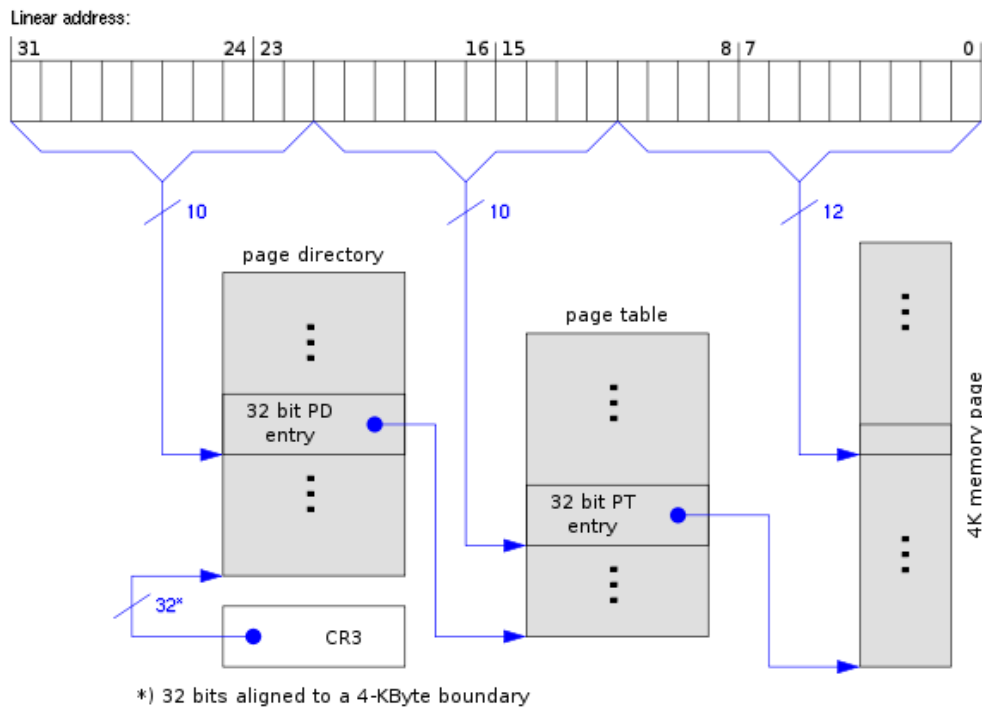


Figure 5-2. The linear address translation mechanism [210].

5.4.1.2 Shadow Page Tables

The hypervisor introduces another layer of address translation, which is the mapping of the guest physical pages to the host physical pages. With virtualization, the hypervisor constructs shadow page tables that combine both the guest page tables and the additional layer of page tables to directly map a guest linear page to a host physical page on the hardware page table. As a result, there are three memory layers: guest virtual memory, guest physical memory, and host physical memory, as shown Figure 5-3. A shadow page table consists of two types of memory address translations [211]: *first*, Guest virtual address to guest physical address. This mapping is performed directly from the virtual machine's page table using the linear address translation mechanism depicted above in Figure 5-2. *Second*, Guest physical address to host physical address. This type of address mapping is managed by hypervisor. Every time a program in a guest virtual machine requests access to a virtual address, the memory management unit of the hypervisor looks directly into the shadow page table and find the corresponding host physical address. If there is a change in the virtual machine's page table, the hypervisor has to intercept this change and update the corresponding part of shadow page table.

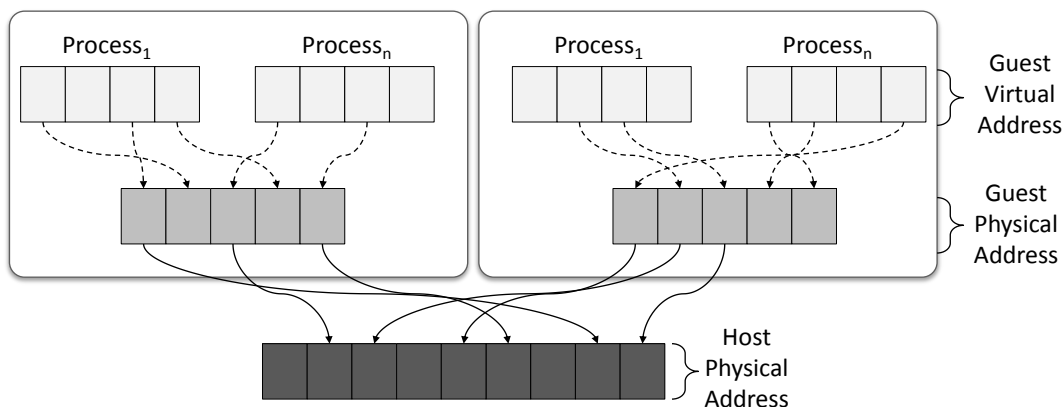


Figure 5-3. The relation between guest virtual addresses and host physical addresses.

5.4.2 Implementation Details

Memory mapping techniques require accurate kernel data definitions in order to overcome the semantic gap. A kernel data definition reflects statically the relations between kernel data at runtime. A kernel data definition is simply a directed *type-graph* that has: (i) nodes representing data structures, and (ii) edges representing data members of the data structures, as shown in Figure 5-4. The problem with operating systems’ kernel is the wide-spread use of pointer relations between kernel data. Pointer relations fall into two main categories: (i) direct pointer relations, where a data member *points-to* somewhere else in the memory (address) and this address holds the address of another data member, denoted by the blue solid rectangles in Figure 5-5. *Second*, Indirect pointer relations, where a data member of a `void` or `null` pointer type is pointing to an address and the runtime contents of this address cannot be identified statically at compilation time and can only be identified at runtime, denoted by the red dashed rectangles in in Figure 5-5.

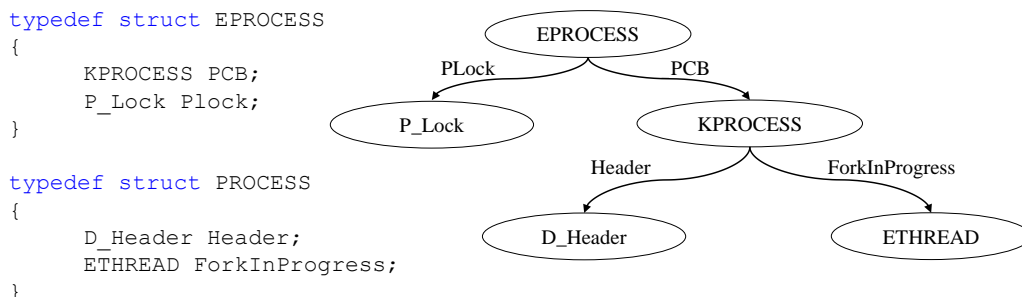


Figure 5-4. A type-graph reflecting relations data structures and their members.

In *CloudSec*, kernel data definitions were developed manually to reflect pointer-based relations between data structures using hard-coded offsets, and the indirect *points-to* relations are computed based on our hands-on experiments with the corresponding operating system’s kernel.

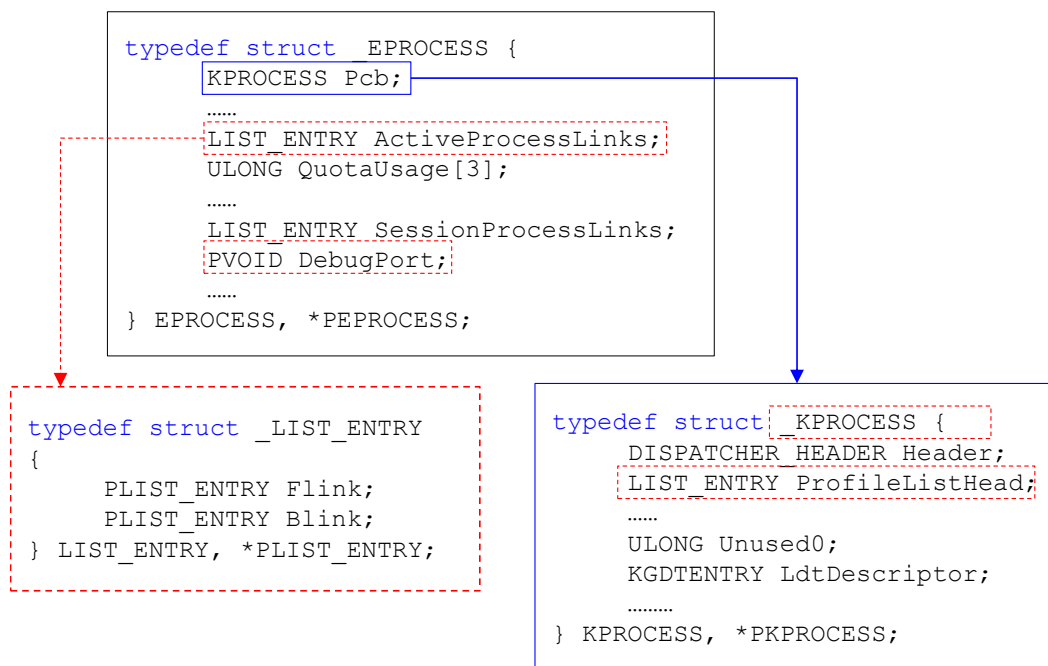


Figure 5-5. Pointer relations between data structures.

Operating systems like Microsoft Windows and Linux embody basic object-oriented design principles [212], where they structure their kernel data into set of predefined data structures and objects. This structure of the kernel enables reconstructing kernel data structures through recursive traversing for the physical memory using the operating system global variables, without relying on the operating system kernel itself that could be exploited. The problem is more difficult in commodity operating systems like Microsoft Windows, because unlike Linux, the Windows operating system is not open source. Also the Windows kernel data structures are opaque – in that the addresses of operating system global variables change from one Windows build to another. Fortunately, Microsoft provides Microsoft Symbols [213] which provides symbols for each Windows kernel build. These symbols can be used as a reference for reconstructing the kernel data definitions. For Linux, kernel data definitions can be obtained from the kernel symbol table (*i.e.* `System.map`).

In our implementation, we selected two important Windows kernel structures to be constructed externally as a proof-of-concept of our prototype: the `EPROCESS` structure and the `KeServiceDescriptorTable` structure. These two data structures are often a target for hackers to install hooks, inject malicious code and to hide malicious processes – their details are discussed below.

5.4.2.1 *EPROCESS Blocks*

A process object type in Windows operating system is represented in kernel address space as executive process blocks called `EPROCESS` structure. An `EPROCESS` structure contains detailed information about a running process, as well as pointer relations to other data structures (*e.g.* threads, tokens and drivers) and attributes related to the running process. Rootkits often target the `EPROCESS` data structure to run stealthy code in the physical memory. Figure 5-6 depicts a snapshot of the `EPROCESS` structure of the `system` process for Windows XP SP3, using the *Windbg* tool. The figure reflects the offsets of the data members and their corresponding addresses in the kernel address space. These offsets were hardcoded in the corresponding data definition. By locating the running instances of such structure in physical memory pages, *CloudSec* can externally list all the running processes including their details (*e.g.* name, ID, threads, loaded modules, Export and Import Address Tables, Virtual Address Descriptors).

For each running process, there is a dedicated `EPROCESS` structure and all `EPROCESS` blocks are structured in a doubly-linked list called `ActiveProcessLinks`, as shown in Figure 5-7. Doubly-linked list, represented in Windows operating system as `_LIST_ENTRY` structure, means that each block has a `Flink` entry that refers to the next node in the doubly linked list and a `Blink` entry that refers to the previous node. Thus, getting the address of one `EPROCESS` structure enables obtaining the rest of the running processes' `EPROCESS` blocks through traversing the corresponding doubly-linked list.


```

kd> dt _EPROCESS poi(PsInitialSystemProcess)
nt!_EPROCESS
+0x000 Pcb : _KPROCESS
+0x06c ProcessLock : _EX_PUSH_LOCK
+0x070 CreateTime : _LARGE_INTEGER 0x0
+0x078 ExitTime : _LARGE_INTEGER 0x0
+0x080 RundownProtect : _EX_RUNDOWN_REF
+0x084 UniqueProcessId : 0x00000004
+0x088 ActiveProcessLinks : _LIST_ENTRY [ 0x82077638 - 0x8055a1d8 ]
+0x090 QuotaUsage : [3] 0
+0x09c QuotaPeak : [3] 0
+0x0a8 CommitCharge : 7
+0x0ac PeakVirtualSize : 0x2a1000
+0x0b0 VirtualSize : 0x1d9000
+0x0b4 SessionProcessLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x0bc DebugPort : (null)
+0x0c0 ExceptionPort : (null)
+0x0c4 ObjectTable : 0xe1001c50 _HANDLE_TABLE
+0x0c8 Token : _EX_FAST_REF
+0x0cc WorkingSetLock : _FAST_MUTEX
+0x0ec WorkingSetPage : 0
+0x0f0 AddressCreationLock : _FAST_MUTEX
+0x110 HyperSpaceLock : 0
+0x114 ForkInProgress : (null)
+0x118 HardwareTrigger : 0
+0x11c VadRoot : 0x825c3220
+0x120 VadHint : 0x823b86c8
+0x124 CloneRoot : (null)
+0x128 NumberOfPrivatePages : 3
+0x12c NumberOfLockedPages : 0
+0x130 Win32Process : (null)
+0x134 Job : (null)

```

Figure 5-6. A snapshot of EPROCESS structure from Windbg.

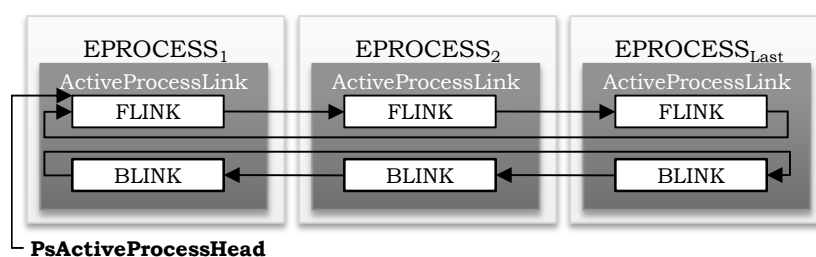


Figure 5-7. EPROCESS doubly-linked list in Windows operating system.

EPROCESS structures are dynamic data objects; their virtual addresses are assigned at run-time. The key challenge here is how to get the address of one block taking into consideration that we can only read the virtual machines' physical memory bytes. From our experience with the Windows operating system, we found that the first loaded process is always the *system*

process, so it is the first node in the `ActiveProcessLinks` doubly-linked list. Also, from our analysis of live memory dumps from running operating system kernels, we found that the global variable `PsActiveProcessHead` points to the `ActiveProcessLinks` address of the *system* process, as shown in Figure 5-7. Similar to this and drawing on our experience with the Windows operating system, we discovered most of the important data structures and manually built up the corresponding kernel data definition. The physical addresses of operating system global variables are static and do not change at runtime. In this way, *CloudSec* can identify the address of `PsActiveProcessHead` from Windows symbols. In *CloudSec* prototype, the address is `0x805638b8`. Subtracting `0x88` from the value pointed to by this address gets the *System* process's `EPROCESS` virtual address. After translating the virtual address to physical address and reading the corresponding physical memory page, we can extract process details *e.g.* process name, ID, threads from the `EPROCESS` according to the corresponding kernel data definition.

To build up the process loaded modules (DLLs) list, we first get the `PEB` structure virtual address, and then translate it to a physical address to read its physical memory page. From the process `PEB` structure we extract the address of the `_PEB_LDR_DATA` data structure that contains a pointer to doubly-linked list for the process loaded modules in different orders: load order, memory order and initialization order. Each loaded module is represented with a `_LDR_DATA_TABLE_ENTRY` structure that contains module name, base address size, `Flink` and `Blink`. To determine the end of the modules list, we check the `Flink` of current node against the address of the first module structure in the list. If the two are equal, then we have traversed all of the loaded process modules.

At this point, we have located the details of the first *system* process. To extract the rest of the running processes, we traverse the doubly-linked list of the `ActiveProcessLinks` structure for the next node, and repeat the previous steps to build the process details. To determine the end of the process list, we check the `Flink` of each `EPROCESS` against the address of `PsActiveProcessHead`. If they are equal, then the current `EPROCESS` structure is the last process in the list. Figure 5-8 reflects a part of the manually developed kernel data definition

of key structures in `EPROCESS` structure that we have used in *CloudSec* to traverse the hosted virtual machines' physical memory to build up running guest operating system processes' details. Figure 5-9 summarizes the algorithm for such traversal process to overcome the semantic gap problem of the running processes.

A drawback of listing the processes through the doubly linked list is that this method becomes vulnerable to process hiding rootkits that modify the `Flink` and `Blink` addresses to hide a process. Mihir *et al.* [128] explores different techniques for detecting hidden processes with their limitations and drawbacks, and introduce their solution that is based on monitoring the scheduled threads of the processes. The authors also mentioned that their technique can be overcome by recent rootkits that build their own scheduler. To solve this problem, we update *CloudSec* process list when any creation or termination for a process occurs. We achieve this by installing memory access triggers on the physical memory page that contains the System Service Dispatcher Table (SSDT), to monitor `NtCreateProcess` and `NtTerminateProcess` functions that are exported by the SSDT. Each of the SSDT functions has a unique system call number. This number is loaded into the `EAX` register while the function is being called. We check the `EAX` register value during each memory access to the SSDT physical page. If the `EAX` register holds the system call number of one of those functions, then we build our processes list again. If we find that a process does not exist in the current process list but exists in the initial process list, then we check the process ID, process directory table and `ThreadListHead` members of this process's `EPROCESS`. Those members can't be changed during the process runtime [16]. If these members are still available with the same values, this indicates that the process has been hidden, but not terminated.

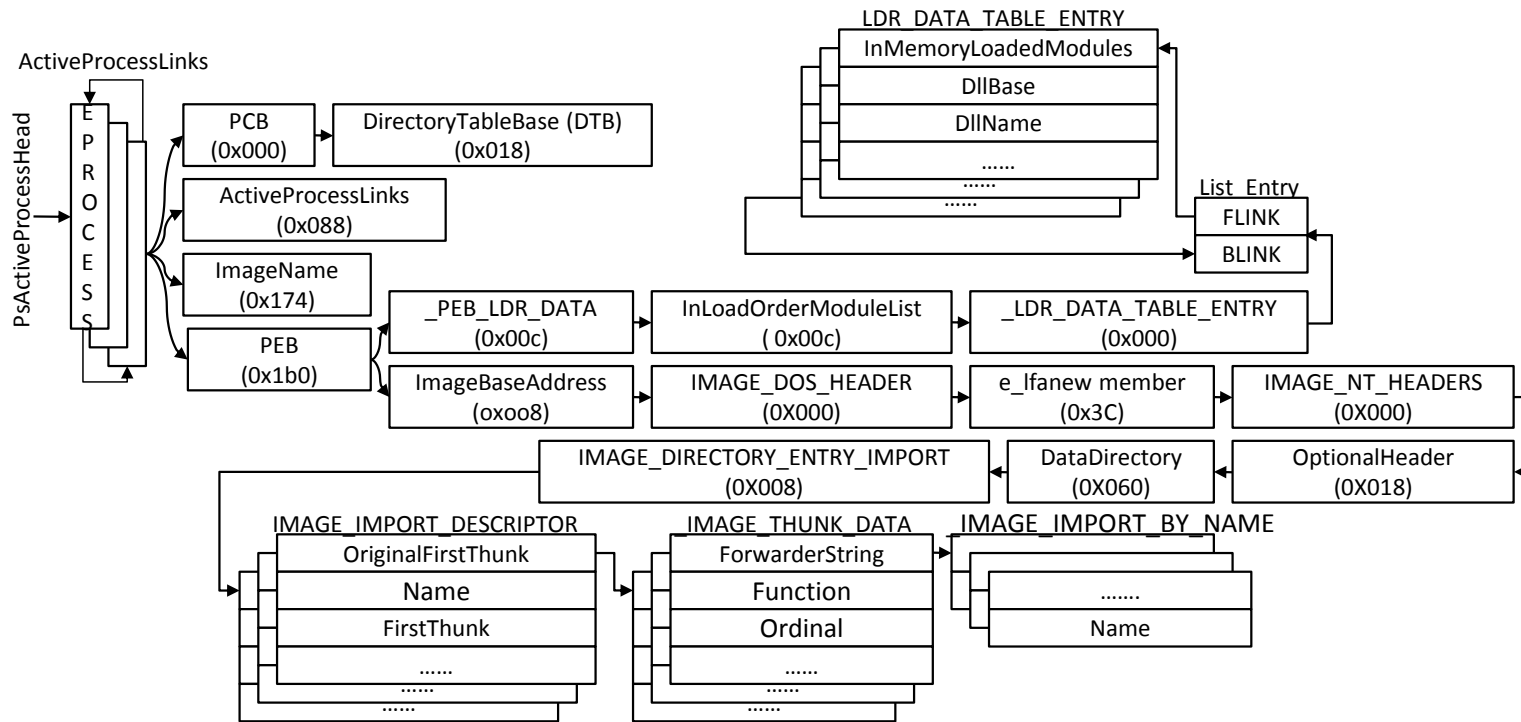


Figure 5-8. The manually-developed EPROCESS kernel data definition.

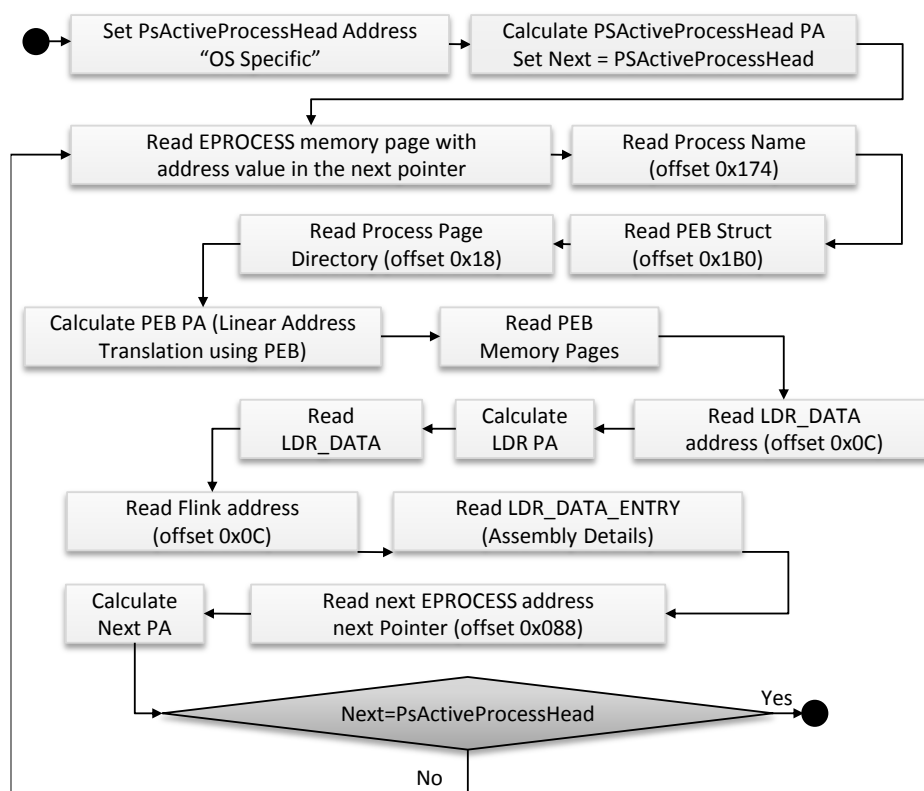


Figure 5-9. EPROCESS semantic gap construction.

5.4.2.2 KeServiceDescriptorTable Structure

The `KeServiceDescriptorTable` structure references the System Service Dispatcher Table (SSDT) that contains a list of the native kernel APIs and their addresses, making it one of the central points of execution flow control in the kernel. Reconstructing this structure externally without relying on the kernel enables detecting any system-wide hook. Hackers often attack and try to compromise this structure as it allows redirecting operating system calls to other code. The address of the SSDT kernel data structure can be found within the Service Descriptor Table (SDT). The SDT is referenced by the `KeServiceDescriptorTable` global structure and its address is static in the Windows operating system. The first member of the SDT structure is the `KiServiceTable` (SSDT address), and the third entry at offset `0x0c` is the number of SSDT entries. To enumerate the SSDT entries, we traverse the physical memory page that contains the SSDT address, until the number of read entries is equal to the value at the offset `0x0c`, as shown in Figure 5-10. Reconstructing the SSDT, allows *CloudSec* to install write-memory-access triggers on the SSDT page, to detect SSDT hooking rootkits that

target to install system-wide hooks.

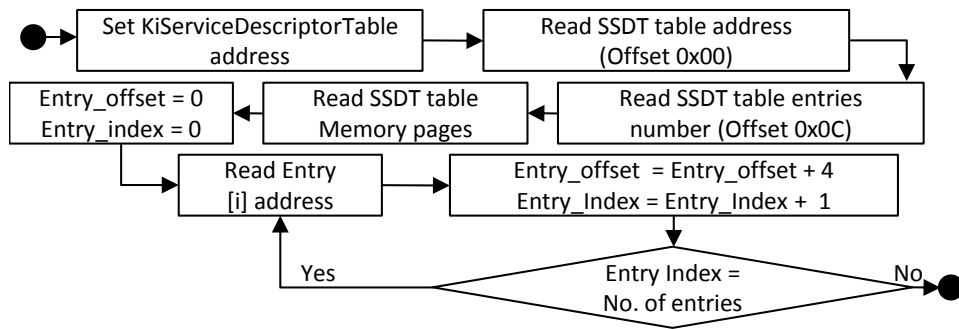


Figure 5-10. KeServiceDescriptorTable reconstruction algorithm.

5.4.3 CloudSec Deployment Model

Our evaluation platform for *CloudSec* was the Intel x86 family of microprocessors. The cloud platform hardware consists of a HP Z400 – 2.8 GHz Intel® Xeon® CPU (VT-x) with 6126 MB of RAM. This workstation runs VMware ESX 4.1. The ESX server hosts three virtual machines, as shown in Figure 5-11. These machines were configured as follows: (i) *Security virtual machine*. The *security virtual machine* is configured with 2GB of RAM and two virtual CPUs and is deployed as a virtual appliance running Ubuntu Linux 8.04 Server JeOS. The security virtual machine hosts the vCompute APIs OF VMSafe libraries and our monitoring software. The *security virtual machine* is isolated from other server network workload in a separate virtual network by creating a dedicated vSwitch and communication channel in the ESX hypervisor. (ii) *Hosted virtual machines*. *CloudSec* prototype contained two hosted virtual machines for validation purposes. Both virtual machines are each allocated with 1.5GB of RAM and two virtual CPUs running Windows XP SP3 32-bit operating system. *CloudSec* Evaluation Details We used Windbg from the Debugging Tools for Windows [214] to validate that *CloudSec* bridges the semantic gap successfully. Windbg runs within the guest operating system and hence has direct access to and detailed knowledge of running guest operating system kernel data structures. We compared the external *CloudSec* view of mapping the introspected virtual machines' physical memory to Windows operating system's kernel data structures, with the internal view of the virtual machine using Windbg. This showed that *CloudSec* produces identical

results as Windbg, thus accurately overcoming the semantic gap for the hosted virtual machines, without installing any monitoring code inside the target VM.

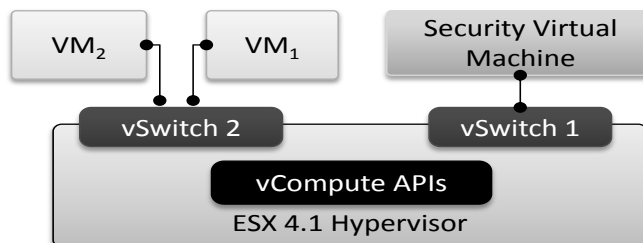


Figure 5-11. The experimental setup lab.

CloudSec provides detailed information about the running processes inside a hosted virtual machine, from outside the virtual machine. Figure 5-12 shows screenshots for the internal view for the running processes using Windbg. We selected the *csrss.exe* process for the comparison. Figure 5-12 shows the *csrss.exe* basic information such as the virtual address of the `EPROCESS`, `PEB`, and the `DirectoryBase (DirBase)`. Figure 5-13 shows the loaded modules of this process. On the other hand, Figure 5-14 shows a snapshot of the *CloudSec* external view of an `EPROCESS` running instances, focusing on the details of a process at the address `0x896a9020`. After constructing the semantic information we extracted the process name *csrss.exe*, DTB and loaded modules and other information. The comparison shows that the results are identical in both the internal and external view. The addresses of the `EPROCESS`, `PEB` and `DTB` are the similar in both views. The memory loaded modules are also the same with the same base addresses. Figure 5-15 and Figure 5-16 show screenshots of the internal and external views of the SSDT using Windbg and *CloudSec*, respectively. As mentioned before, each entry has a unique number so entries are in order. By comparing the addresses in the internal and external views, we found that our interpretation of the physical memory pages to operating system semantic information matched accurately. The SSDT address is the same in both views, which is `0x80504480`, and for example, the first SSDT entry address in the two views is `0x805145f6`, which is the address of the `NtConnectPort`, because this function system call index is 0. For the other entries, the addresses of entries in both views are the same.

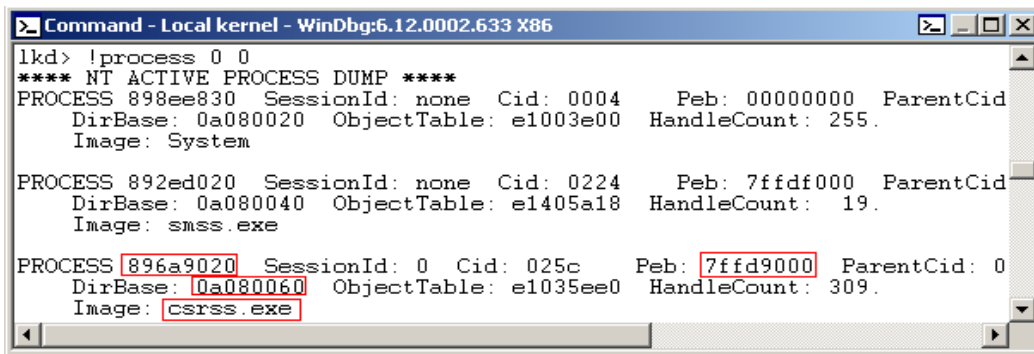


Figure 5-12. The internal view of a virtual machine using Windbg.

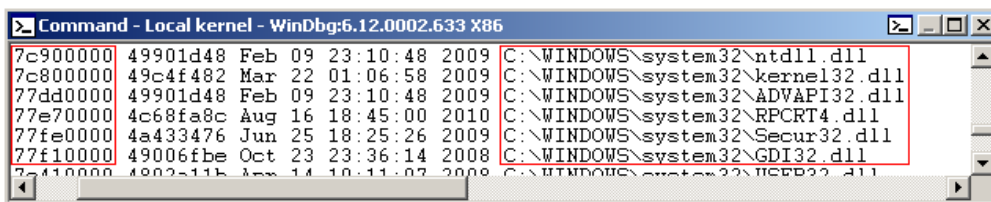


Figure 5-13. The internal view of the loaded modules of the csrss.exe process.

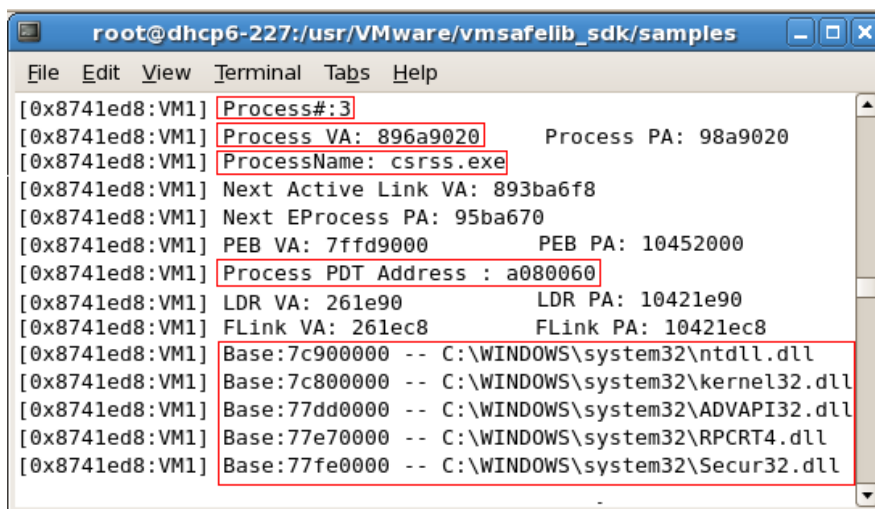


Figure 5-14. CloudSec external view of the csrss.exe process.

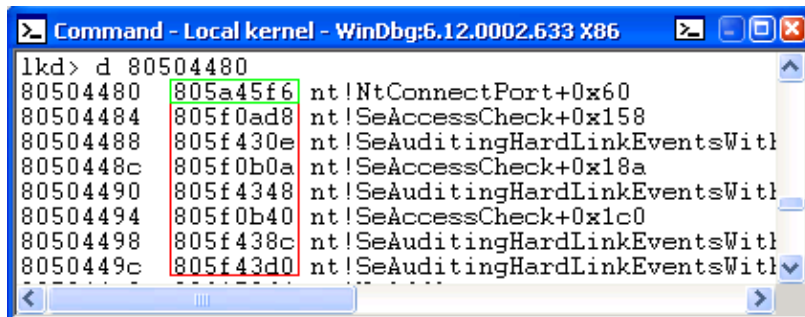


Figure 5-15. The internal view of the SSDT using Windbg.

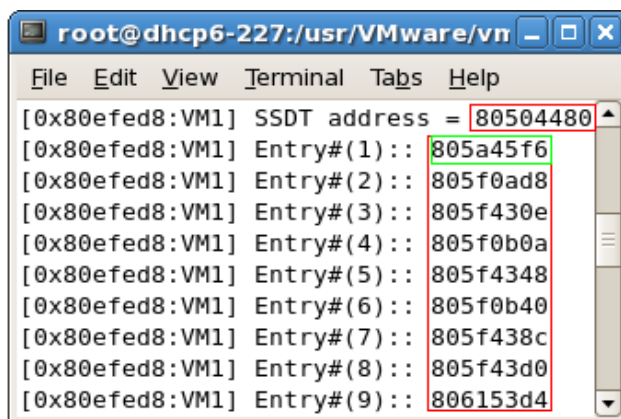


Figure 5-16. CloudSec external view of the SSDT.

To evaluate the active monitoring feature of *CloudSec*, we used 9 real rootkits that perform data hooks and object manipulation attack⁹s. These rootkits perform IAT hooking, DLL injection and DLL and process hiding [215-220] and perform SSDT and GDT hooking [221-223], named FU and FuTo Rootkits, Remote DLL rootkit, NtIllusion rootkit, Injective Code inside Import Table rootkit, NtOpenProcess hook, SSDT hooking and *REAL* NT rootkits.

CloudSec was able to detect and take appropriate action to prevent the malicious rootkits' behaviours successfully. Figure 5-17 shows a detection and prevention process of an SSDT function pointer hooking rootkit. Figure 5-18 reflects detection of a process hiding rootkit and the prevention action in this case is making the hiding process visible to the virtual machine's running operating system by modifying the corresponding pointers. *CloudSec* has no more actions to do with such rootkits, as protecting running processes is the consumer's property. Figure 5-19 reflects a DLL injection rootkit and Figure 5-20 reflects function hooking of a DLL routine. Figure 5-21 a DLL hiding rootkit. The prevention action of most detected threats was by overwriting the memory bytes to the original memory addresses and values based on the hard-coded offsets in the kernel data definition.

⁹ The main objective of the experiment is validating our active monitoring feature rather than detecting rootkits (known and unknown) using efficient techniques.

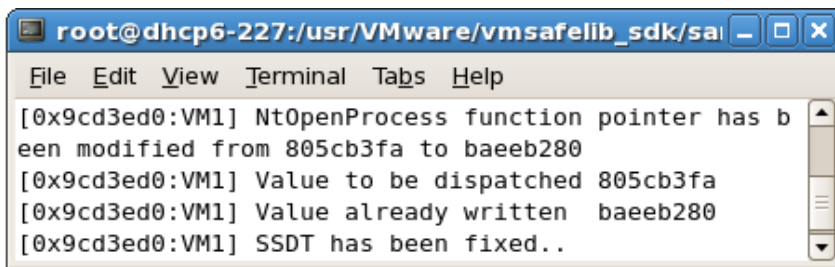


Figure 5-17. SSDT hooking.

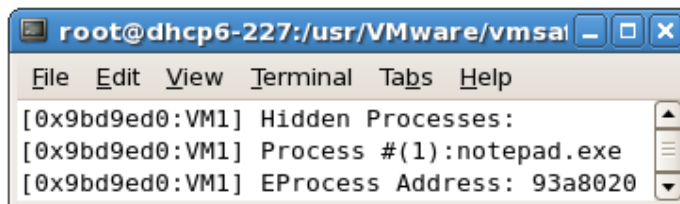


Figure 5-18. Process hiding detection.

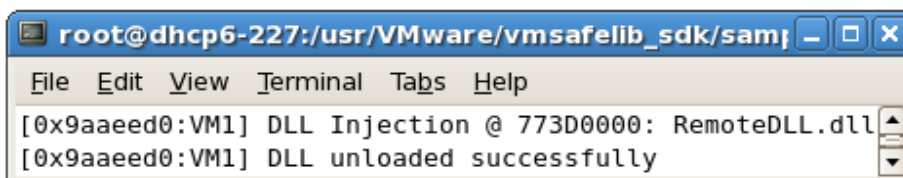


Figure 5-19. DLL Injection.

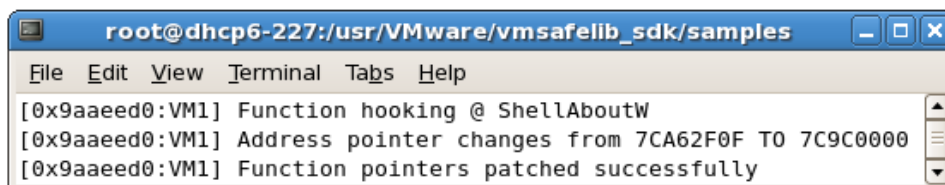


Figure 5-20. Function hooking .

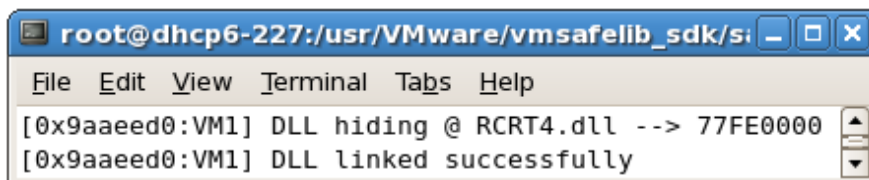
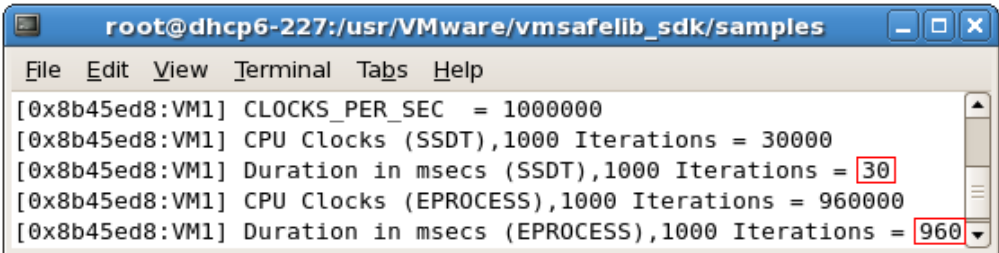


Figure 5-21. DLL module hiding.

5.4.3.1 Performance Overhead

CloudSec builds all the necessary information to bridge the semantic gap and install monitoring triggers once a virtual machine is booted-up. The performance overhead of CloudSec depends

on the interception overheads of the vCompute APIs and the amount of processing performed by our monitoring code. VMsafe APIs (including vCompute) reduce performance overhead because security inspections are processed in the hypervisor kernel [224]. To calculate the elapsed time for running our code, we called *CloudSec* code functions 1000 times and calculated the average time taken. Figure 5-22 shows the elapsed CPU clocks and time in milliseconds (msecs) for 1000 iterations for both the EPROCESS and SSDT construction and listing functions. Time is calculated by dividing the CPU clocks for each function calls by the CLOCK_PER_SEC (processor dependant variable). Listing the processes (including processes details and loaded modules) consumes about 0.96 msecs, and locating and reading the SSDT table consumes 0.03 msecs. These results indicate that the performance overhead to enable near real-time, active and transparent monitoring to uncover the specific kernel runtime objects monitored by *CloudSec* is very low.



```

root@dhcp6-227:/usr/VMware/vmsafelib_sdk/samples
File Edit View Terminal Tabs Help
[0x8b45ed8:VM1] CLOCKS_PER_SEC = 1000000
[0x8b45ed8:VM1] CPU Clocks (SSDT),1000 Iterations = 30000
[0x8b45ed8:VM1] Duration in msecs (SSDT),1000 Iterations = 30
[0x8b45ed8:VM1] CPU Clocks (EPROCESS),1000 Iterations = 960000
[0x8b45ed8:VM1] Duration in msecs (EPROCESS),1000 Iterations = 960

```

Figure 5-22. CloudSec performance overhead.

The normal boot-up time for virtual machines hosted in our platform is 12 seconds, this being the time from the time we press power on until the operating system is loaded. *CloudSec* reads the SSDT entries before the operating system is completely loaded by 7 seconds. As the virtual machine is powered on, *CloudSec* queries the hypervisor to find out the version of the running operating system, and starts locating and reading the SSDT. For listing the running processes, *CloudSec* waits until processes are loaded into memory. Once a process is loaded, *CloudSec* reads the process immediately.

5.5 Discussion

CloudSec is an introspection framework designed to facilitate the monitoring process of an

operating system of a running virtual machine, hosted in an IaaS cloud platform. *CloudSec* enables transparent and near real-time monitoring for the operating system running state, at a hypervisor level. *CloudSec* also enables active monitoring, which is needed to enforce security decisions on system events. Our experiments with *CloudSec* have shown that it can provide rich, high-level operating system information about a limited part of kernel data, from outside the virtual machine and without placing any code in the virtual machine.

To date, nearly all virtual machine introspection research efforts had been based on the Xen hypervisor. The Xen hypervisor itself is small (approximately 100 KLOC) relative to other hypervisors. However, Xen relies mainly on a Domain0 virtual machine which is part of its trusted computing base. This results in a large footprint hypervisor, increasing the possibility that vulnerabilities and exploits that can occur. VMware ESX has a smaller footprint than Xen because the ESX hypervisor does not depend on any external components. Virtual machine introspection research that is based on the VMware ESX environment is not seen much in research.

CloudSec has a number of limitations that make it ineffective approach to develop robust and systematic security solutions, for the following reasons: *first*, we must depend on the security expert experience with the operating system kernel data layout to manually solve the semantic gap. This is limited to 28% of kernel data structures – as discussed by Carbone et al. [14]. These data structures relate to the well-known kernel objects such as processes, threads and device drivers. *Second*, the process of uncovering kernel runtime objects in memory difficult and time consuming process. Moreover, not all data structures are easy to be uncovered using the traditional memory traversal technique, as discussed in chapters 2 and 6. *Third*, the traditional memory traversal techniques used to uncover the runtime dynamic kernel objects are vulnerable and not accurate, as discussed in chapter 2. These limitations result in an inability to get a complete and accurate interpretation of the low-level hardware bytes, making the manual analysis approach inadequate. This causes security holes, limited protection and an inability to detect zero-day threats. Based on that, a systematic solution is required to efficiently solve the semantic gap problem. However, implementing such a solution is challenging as it should be

generic enough to cover all kernel data, yet accurate enough to correctly uncover the running instances. The challenge arises from the complexity of the operating system kernel data. In chapter 6, we will describe a new static analysis tool that enables analysing operating system's source code to systematically build a kernel data definition that reflects the runtime kernel data layout, without a prior experience of the operating system. In chapter 7, we introduce a solution that efficiently overcomes the limitations of the memory traversal techniques and enables fast and nearly complete coverage of the kernel address space to uncover the running instances of kernel dynamic objects.

5.6 Summary

Virtualization has complicated the security process of the IaaS platform, but it has also provided ways of enabling monitoring the virtual machines, at the hypervisor level. In this chapter, we described *CloudSec*, a security appliance that provides active, transparent and near real-time security monitoring for multiple concurrent virtual machines hosted on an IaaS platform. *CloudSec* provides fine-grained inspection for physical memory of the hosted virtual machines in the IaaS platform. One of the key features of *CloudSec* is its ability to provide active and transparent monitoring by placing the security hooks at the hypervisor level to intercept hardware events. This allows the security virtual machine to set access triggers on selected memory pages of the hosted virtual machines' address space to suspend hardware event execution until the event is analysed. Our experiments showed that the constructed external memory view by *CloudSec* is identical to the internal view of the hosted virtual machine's running kernel data instances. We have also shown that the performance overhead of traversing the physical memory to build the external view is relatively low enough to support *real-time* external virtual machine monitoring.

Chapter 6

OS-KDD: Operating System Kernel Data

Disambiguator

In chapter 5, we discussed our *virtual machine introspection* framework *CloudSec* that has the ability to actively overcome the semantic *gap problem* and monitor hosted virtual machines in the infrastructure-as-a-Service cloud platform. One of the main limitations of *CloudSec* is the manual approach used to overcome the semantic gap. In this chapter, we introduce a new approach, called Operating System Kernel Data Disambiguator (OS-KDD), to systematically build an accurate kernel definition for a C-based operating system, without a prior knowledge of the operating system's kernel runtime data layout. OS-KDD builds a kernel data definition by performing static *points-to* analysis on the operating system's kernel source code in order to compute a directed *type-graph* that statically reflects the runtime kernel data layout. Section 6.1 gives an overview on OS-KDD's approach including the main key features of it. In section 6.2, we discuss the structure of the generated *type-graph* and the representation language used to represent OS-KDD analysis steps. In section 6.3, we describe the high-level process of OS-KDD, and section 6.4 we discuss in detail the *points-to* analysis algorithm. The implementation and evaluation details are presented in section 6.5 and finally we discuss the key features and limitations of our approach.

6.1 OS-KDD Overview

One of the main limitations of *CloudSec*, and many other virtual machine introspection tools [6, 7, 9-11, 102, 103, 225, 226], is their use of a manual approach to build a kernel data definition that statically reflects the runtime kernel data layout. In this chapter, we present OS-KDD (Operating System Kernel Data Disambiguator), a static analysis tool that has the ability to systematically build an accurate kernel data definition of any C-based operating system, without prior knowledge of the operating system's kernel runtime data layout.

The generated definition precisely models kernel data structures, reflects both direct and indirect relations among data structures, and generates constraint-sets on the pointer-based relations of the data structures in order to enable systematic integrity checks on kernel data. OS-KDD is not just limited to operating systems. It is also a powerful tool for analysing C program source code to: (i) discover data structures and generic pointers' bugs that could cause memory violations, and (ii) to perform accurate type inference to determine the actual target types of program's runtime objects to formulate robust integrity constraint-sets that can be enforced at system runtime.

OS-KDD disambiguates the pointer-based relations, including generic pointers, by performing static *points-to* analysis on operating systems' kernel source code. In OS-KDD, precision is an important factor, as the main goal of OS-KDD is computing the most precise *points-to* sets for each generic pointer dereferencing operation. To meet our analysis requirements, we designed and implemented a new *points-to* analysis algorithm that has the ability to provide field, flow and context-sensitive *points-to* analysis for large C programs that contain millions lines of code such as operating system kernels. To facilitate and speed up the analysis process, we use Abstract Syntax Trees (AST) as a high-level representation for the kernel source code. Abstract syntax trees capture the essential structure of the code that reflects the code semantic, while omitting the unnecessary syntactic details of the programming language used to write up the code.

OS-KDD takes the source code of an operating system's kernel as input and outputs a directed *type-graph* that represents the kernel data definition. This *type-graph* summarizes the different data types located in the kernel address space along with their connectivity patterns. It also reflects the inclusion-based relations between data structures for both direct and indirect pointer-based relations. These inclusion-based relations enable formulating a constraint-set between data structures, in order to enable performing systematic integrity checks on kernel dynamic data at runtime.

6.2 Type-Graph Description

For a target C program source code, OS-KDD computes a directed *type-graph* $G(N, E, R)$ that summarizes memory objects accessible within procedures along with their *points-to* relations with the other kernel data structures, where:

- N is a set of nodes representing the program objects such as global and local variables, fields, array elements, procedure arguments\parameters and returns.
- E is a set of directed edges across nodes representing values, assignments, returns and function calls between graph nodes.
- R indicates, for each procedure node in the *type-graph*, whether the node is part of a cycle or not. This bit enables efficient cycle detection during the *points-to* analysis phases. Handling cycles efficiently helps improving the performance of the *points-to* analysis algorithm. The R bit is initially false for all nodes, and its value is updated based on our cycle detection mechanism, discussed later in this chapter.

In OS-KDD, there are four types of graph nodes and four types of graph edges. This collection of nodes and edges greatly reduces the number of nodes visited during context-sensitivity analysis, without loss of precision. All graph nodes must be disjoint, as we developed our analysis algorithm based on Anderson’s approach that states that a node is created for each variable and a single node may have different edges pointing to other nodes. Each variable is represented by a unique node and each edge represents a constraint between the source and destination nodes. A Graph node N is one of the following:

- *Variable Node*. A variable node represents a set of memory locations such as local and global variables including procedure’s formal-in parameters. These variables are possibly pointers to objects and have an initial declared type.
- *Field Reference Node*. A field reference node represents data structure fields, where each field reference node has an associated parent node that represents the field’s data structure. A field reference node is also used to represent elements of

an array. Each field reference node has a declared type as stated in the source code.

- *Procedure Call Node*. A procedure call node represents procedure calls within the program to represent unresolved call sites in the context of the current function. A function call node is represented as a function name plus an index; index = -1 if the node represents a function return. Otherwise index = i , where i is the index of formal-in argument. For example, given a function call $G(\text{arg}_x, \text{arg}_y)$; in this case OS-KDD creates two nodes $G:1$ and $G:2$ representing passed formal-in arguments arg_x and arg_y , respectively.
- *Cast Node*. A cast node represents explicit casting where the type of the node is the typecast and the name is the casted variable or function.
- *Allocation Node*. An allocation node represents a heap object created using `malloc`. Each runtime object is represented by one allocation node. An allocation node has by default a `void*` type, unless a type cast statement is added to the `malloc` statement in order to be dereferenceable.

The source and destination of a directed edge E are both fields of a graph node N . A graph edge E is a function of type $\langle N_{\text{source}}, F_{\text{source}} \rangle \rightarrow \langle N_{\text{dest}}, F_{\text{dest}} \rangle$, where N_{source} and $N_{\text{dest}} \in N$, $F_{\text{source}} \in \text{fields}(N_{\text{source}})$ and $F_d \in \text{fields}(N_{\text{dest}})$. Directed edges between graph nodes are one of the following:

- *Points-to Edge*. A *points-to* edge represents a *points-to* relation between two nodes according to the edge direction, and denoted as \rightsquigarrow .
- *Inlist Edge*. represents a *points-to* relation between two nodes but on a local scope based on the results of the escape analysis¹⁰ of the pointer, thus if \exists node A has *inlist* edge to node B , then $B \in \text{pts}(A)$ where $\text{pts}(A)$ means the *points-to* set of A . An *inlist* edge is denoted as \rightarrow .

¹⁰ Escape analysis is a method for determining the dynamic scope of pointers.

- *Outlist Edge*. An outlist edge is not a relation edge, but represents a directed path between two nodes that are used to perform the interprocedural and the context-sensitive analysis in an efficient manner, and denoted as \leftarrow .
- *Parent-Child Edge*. A parent-child edge represents a relation between a parent node and a child node – *i.e.* relation between structure and fields, or array and its elements.

We also state some definitions that clarify some notations used in the graph representation, for easier understanding for the *points-to* analysis algorithm. For a graph $G = (N, E)$, we assume:

- **Definition 1.** We say that node p *points-to* node q , written $p \rightsquigarrow q$, if $p = q$ or $p \rightarrow q \in E$ or $\exists x : [p \rightarrow x \in E \wedge x \rightsquigarrow q]$. We also say that q is reachable from p .
- **Definition 2.** The set of *inlist* edges for a node, $n \subseteq N$, is defined as $E(n) = [p \rightarrow q \in E \mid p \in n]$. The set of *outlist* edges for a node $n \subseteq N$, is defined as $E(n) = [p \leftarrow q \in E \mid p, q \in n \mid E \not\subseteq p \rightsquigarrow q]$.

6.2.1 OS-KDD Representation Language

To formalize the representation of OS-KDD analysis, we define a very simple programming paradigm that describes the C programming language – in a naïve way – as shown in Figure 6-1. This facilitates a simple formal representation of the OS-KDD approach. OS-KDD covers most of C programming language features. A C program basically has the following form: pre-processor commands including compiler directives, procedures, variables, statements and expressions, and comments. OS-KDD only works on the procedures, variables and the program statements and expressions. Pre-processor commands are being processed in the source code transformation process of OS-KDD; details are discussed in section 6.3.1.

Any programming language has a type system that is used to reduce memory bugs and overflows by defining interfaces between the different program parts. The C language has a naïve type system. Based on that naïve type system, in our representation language, we presume

that there are basic integer, character and floating point number pointers, called pointer-compatible variables. These types will be included in the *points-to* analysis phase even if they are not declared as pointers. There are also complex types using `typedef` to form complex types from the basic language type system. We can assign a type to variables, expressions, and procedures.

$$\begin{aligned}
 \text{type} &:= \text{void} \mid \text{char} \mid \text{int} \mid \text{int} * \mid \text{int} ** \mid \text{typedef} \\
 \text{Complex type} &:= \text{typedef struct } \{(\overrightarrow{\text{type var}})\} \\
 \text{procedure} &:= \text{type proc } (\overrightarrow{\text{type var}})\{\text{statement}\} \\
 \text{statement} &:= \text{type var} \mid \text{var} = \mathbb{Z} \mid \text{return var} \\
 &\quad \mid \text{var1} = * \text{var2} \mid \text{var1} = \&\text{var2} \mid \text{var} = \text{null} \\
 &\quad \mid * \text{var1} = \text{var2} \mid \text{var1} = \text{proc } (\overrightarrow{\text{var2}}) \\
 &\quad \mid \text{proc } (\overrightarrow{\text{var}}) \mid (* \text{proc})(\overrightarrow{\text{var}}) \\
 &\quad \mid \text{if } (\text{cond}) \text{ statement else statement} \\
 &\quad \mid \text{while } (\text{cond}) \text{ statemenet}
 \end{aligned}$$

Figure 6-1. OS-KDD Representation Language

Program statements include simple assignment, pointer load (read) and store (write) operations, conditions, loops, procedure calls including return statements and indirect function calls. Here, we consider all forms of assignments: $x=y$, $x=\&y$, $*x=y$, $x=*y$, and function calls.

6.3 Type-Graph Construction

The main objective of OS-KDD is to systematically compute an accurate *type-graph* for an operating system's kernel source code. This *type-graph* statically and precisely reflects the runtime kernel data layout of a specific operating system's kernel version. A high-level representation of OS-KDD is shown in Figure 6-2. A *type-graph* is created and refined in a two main analysis phases: *first*, source code transformation. The objective of this phase is transforming the program source code into abstract syntax trees (1). Code transformation into abstract syntax

trees enables more efficient and scalable *points-to* analysis for large-scale applications. Section 6.3.1 discusses in details this phase of the analysis. *Second, points-to* analysis phase. This analysis phase is the key to automate the generation of the kernel data definitions, without a need for a prior knowledge of the runtime kernel data layout. This is done by performing *points-to* analysis on the operating system’s source code, in order to get an accurate estimation for each pointer dereferencing operation, for both pointers and pointer-compatible variables. This analysis phase has two main rounds; one for computing the direct *points-to* relations between kernel dynamic data (2), and the second round to compute the indirect *points-to* relations such as `void` and `null` pointers, and casting operations (3). An overview of our *points-to* analysis algorithms is explained in section 6.3.2 and the details of the *points-to* analysis algorithm is covered in in section 6.4.

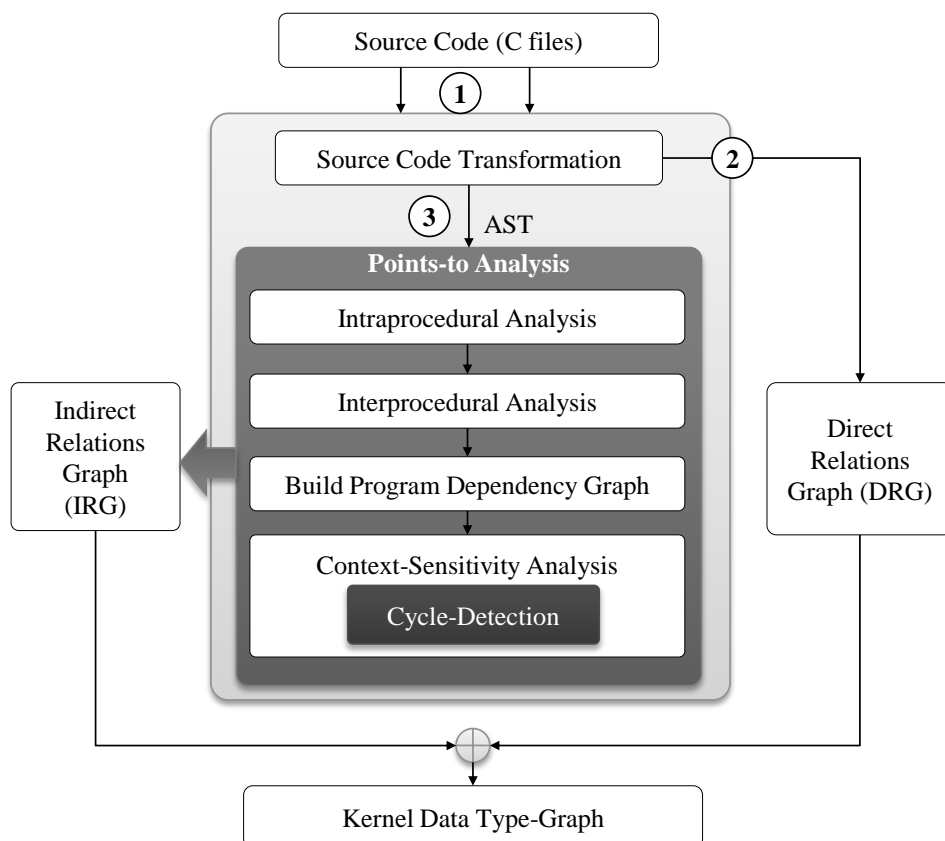


Figure 6-2. High-level view of OS-KDD framework.

6.3.1 Source Code Transformation

The first phase of OS-KDD framework is the: the code transformation process from a C source code into a high-level representation of the source code that has a context-free grammar (CFG) feature in order to enable efficient *points-to* analysis.

Most modern programming languages are context-free grammar languages. A context-free grammar is a formal grammar in which every production rule is of the form $V \rightarrow w$, where V is a single nonterminal symbol, and w is a string of terminals and/or non-terminals. A formal grammar is considered context-free when its production rules can be applied regardless of the context of a nonterminal [227]. However, C and C++ programming languages are not context-free grammar languages. They are context-sensitive languages; for example, in C and C++ users are allowed to declare new types that are not defined in the type system of these languages, and in this case the context-free grammar approach cannot predict the way in which those variables should be used at program runtime. One solution to this problem is transforming the code into another representation language whether low-level or high level representation language. *Points-to* analysis algorithms have usually been implemented in compiler optimization problems, which typically require transforming the source code into a low-level representational language. Low-level representational languages discard most of the original structure of the source code, and thus omit very important information such as declarations, data types and type casting. They also are extremely big in size as they create lots of temporary variables that are allocated identically to source code variables and thus are not easily distinguishable from source code variables [200]. Thus, a low-level representational language is not an efficient way to represent the program source code.

In the OS-KDD framework, we transform the C program source code into a high-level representation by parsing the code into abstract syntax trees. Abstract syntax trees are used widely in program analysis tools, as they capture the essential structure of the program source code that reflects the program semantics, while omitting the unnecessary syntactic details. Such transformation eliminates the irrelevant information such as literal symbols and layout while keeping everything about the original program source code structure. An abstract syntax tree of

a C/C++ function over alphabets of attributes Σ_L , Σ_E can be viewed as an attributed tree (N, E, μ_L, μ_E) with nodes N , edges E , and labelling functions $\mu_L: N \rightarrow \Sigma_N$ and $\mu_E: E \rightarrow \Sigma_E$. The labelling functions assign attributes to nodes and edges, respectively. Nodes are labelled with program statements and expressions, while edges are attributed with the role of a branch [228].

C parsers usually work like C compilers; they must receive preprocessed C code in order to function correctly. In order to parse C/C++ source code into abstract syntax trees, a preprocessor is required to transform the program into a preprocessed code before the parsing process. Preprocessed C code is generated using a C preprocessor. The C/C++ preprocessor is a macro processor that is usually used by compilers to transform a program before compilation. A C preprocessor provides four main facilities: (i) handling compiler directives like `#include` and `#define`, (ii) macros, (iii) conditional compilation, and (iv) line control. To parse a source code program into abstract syntax trees, we just use the header file inclusion feature of the preprocessor. Simply, a C preprocessor replaces header file declarations such as `#include <xx.h>` with the entire text of the header file itself. From the preprocessed C code, we generate the abstract syntax trees using a C parser that fully supports the C89 and C99 language. Figure 6-3 shows two C code snippets and their corresponding abstract syntax trees with a textual representation for the trees instead of using the DOT language. Figure 6-3(a) depicts how a C type definition is parsed into AST and Figure 6-3(b) show the same process for a function call.

6.3.2 OS-KDD Points-to Analysis Overview

In section 6.3.1 we discussed the source code transformation process that builds the abstract syntax trees of a target source code program to be ready for our *points-to* analysis algorithm. In this section, we give a high-level view of how OS-KDD computes a *type-graph* from the abstract syntax trees using our *points-to* analysis algorithm and in section 6.4 we discuss the *points-to* analysis algorithm in details.

```
typedef struct basket
{
    MCF_arc_p a;
    MCF_cost_t cost;
    MCF_cost_t abs_cost;
}BASKET;
```

} C Code

```
Typedef: BASKET, [], ['typedef']
TypeDecl: BASKET, []
Struct: basket
Decl: a, [], [], []
TypeDecl: a, []
IdentifierType: ['MCF_arc_p']
Decl: cost, [], [], []
TypeDecl: cost, []
IdentifierType: ['MCF_cost_t']
Decl: abs_cost, [], [], []
TypeDecl: abs_cost, []
IdentifierType: ['MCF_cost_t']
```

} AST

(a)

```
sort_basket( 1, basket_size ); → C Code
```

```
FuncCall:
  ID: sort_basket
  ExprList:
    Constant: int, 1
    ID: basket_size
  Assignment: =
  UnaryOp: *
  ID: red_cost_of_bea
  StructRef: ->
  ArrayRef:
    ID: perm
    Constant: int, 1
  ID: cost
Return:
  StructRef: ->
  ArrayRef:
    ID: perm
    Constant: int, 1
  ID: a
```

} AST

(b)

Figure 6-3. Snapshots of C ode snippets and their ASTs

In chapter 2, we discussed in detail the complexity problem of an operating system’s kernel data, and the wide usage of data structures and generic pointers in an operating system’s kernel

source code. Based on that, we needed to develop a *points-to* analysis algorithm that satisfies the following features in order to meet our analysis requirements: *first*, one of the key features of our *points-to* analysis algorithm is achieving true context-sensitivity to distinguish heap objects in an efficient and scalable manner. In OS-KDD, we achieve context-sensitivity by detecting the entire acyclic call paths of the different calling contexts, not by allocation site. This approach is also known as a heap cloning approach [229] that allows distinguishing the different instances of a data structure created in different calling contexts. *Second*, one of the main objectives of performing *points-to* analysis on the kernel source code is generating a constraint-set between data structures to be used at runtime, in order to perform kernel data integrity checks and detect memory bugs and overflows. This is done by performing an inclusion inclusion-based analysis on the relations between data structures during the *points-to* analysis. *Third*, as our *points-to* analysis mainly works on data structures and the complex data types defined in the kernel source code, we should guarantee precise field-sensitivity analysis to distinguish the *points-to* sets of the different data structure fields. Identifying accurately the internal connectivity patterns of a data structure with the other data structures guarantees accurate results that greatly affect the trustiness of the proposed security solution.

To meet these requirements, we have developed a new *points-to* analysis algorithm to statically analyse the kernel's source code to get an approximation for every generic pointer dereferencing based on Anderson's approach. Our *points-to* analysis algorithm is field, flow and context-sensitive *points-to* analysis for large C programs that contain millions lines of code such as operating system kernels. The field-sensitivity feature is applied on data structures and arrays. Context-sensitivity is achieved by detecting the entire acyclic call paths of the different calling contexts. Type casting is handled by analysing the usage of the cast variables and objects in the code base. We consider all forms of assignments including load and store operations, and function calls including indirect function calls, as described before in Figure 6-1. Our *points-to* algorithm works only on pointers and pointer-compatible variables that are defined in section 6.2.1. Algorithm 6-1 summarizes our *points-to* algorithm. The type-graph is created and refined in a two-step process:

First, Direct Points-to Relations Analysis. The direct *points-to* relations analysis step is straight-forward and its target is building the direct *points-to* relations graph (*DRG*) that reflects the direct inclusion-based relations between kernel data structures that have clear type definitions. From the generated abstract syntax trees, OS-KDD performs a simple compiler-pass approach to extract the data structure type definitions by looking for `typedef` aliases, and extract their fields with the corresponding type definitions. Nodes are data structures and edges are data members (inclusion relations) of the data structures. Figure 6-4 depicts an example for a C type definition code, and its corresponding direct relations graph. Figure 6-5 shows different snapshots of the direct relations *type-graph* for analysing Windows research kernel showing the different direct relations among kernel data.

Algorithm 6-1 OS-KDD high-level analysis algorithm

1:	Parse C program source code into an AST;
2:	Compute the direct <i>points-to</i> relations;
3:	Build direct points-to relations <i>type-graph</i> , denoted <i>DRG</i> ;
4:	Intraprocedural_analysis (AST);
5:	Build local points-to relations <i>type-graph</i> , denoted <i>LTG</i> ;
6:	Interprocedural_analysis (AST);
7:	Update_graph (<i>LTG</i>);
8:	Build the program dependency-graph, denoted <i>PDG</i> ;
9:	repeat
10:	for <i>lev</i> from lowest to highest do
11:	points_to_analysis(<i>LTG</i> , <i>lev</i>);
12:	Build indirect points-to relations <i>type-graph</i> , denoted <i>IRG</i> ;
13:	end for
14:	until reach the highest level of the dependency-graph
15:	Graph_unification(<i>IRG</i>);
16:	Context_sensitivity_analysis(<i>IRG</i>);
17:	merge_graphs(<i>IRG</i> , <i>DRG</i>);
18:	Build final type-graph, denoted <i>TG</i> ;

Second, Indirect Points-to Relations Analysis. The indirect *points-to* relations analysis is the most important and complicated analysis phase. This analysis phase computes the indirect *points-to* relations between data structures to build the indirect *points-to* relations graph (*IRG*) using our *points-to* analysis algorithm. The *type-graph* of the indirect relations (*IRG*) is com-

puted by our *points-to* analysis algorithm in a four-step process: Intraprocedural Analysis, Interprocedural Analysis, Graph Unification and Context-Sensitive *Points-To* Analysis. The details of the points-to analysis algorithm are discussed thoroughly in section 6.4.

```

typedef struct EPROCESS
{
    KPROCESS PCB;
    P_Lock P_Lock;
}
typedef struct PROCESS
{
    D_Header Header;
    ETHREAD ForkInProgress;
}
    
```

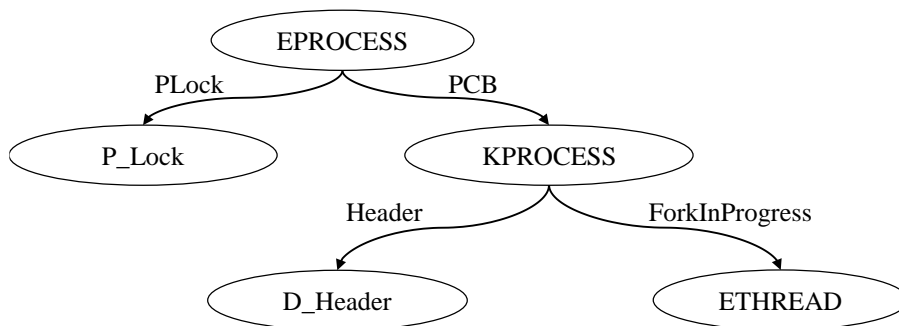
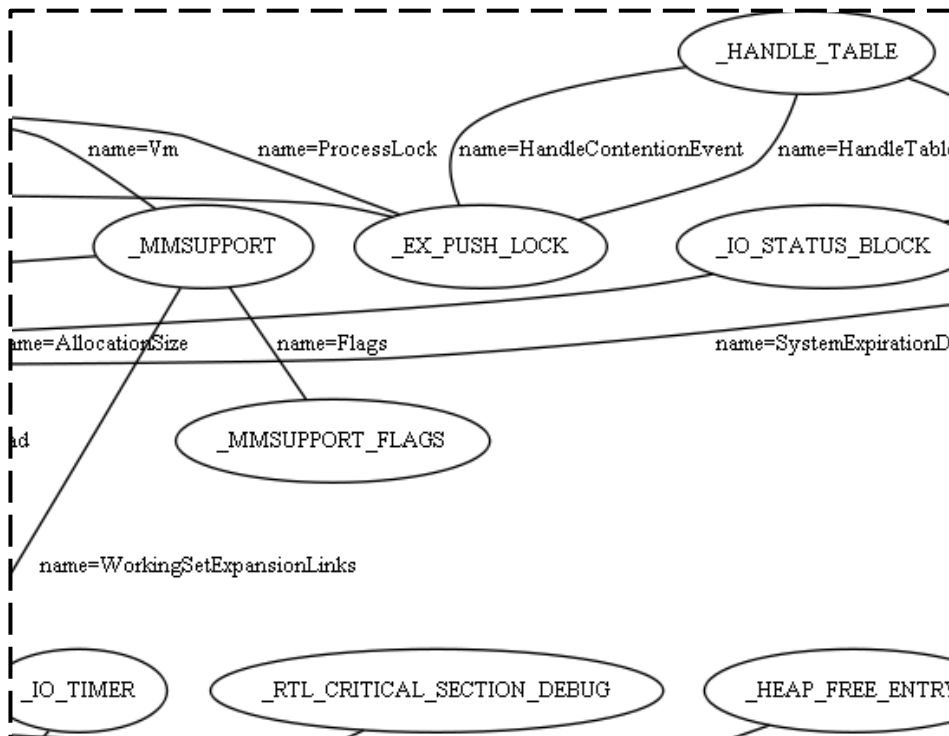


Figure 6-4. Direct Inclusion-Based Relations Analysis.



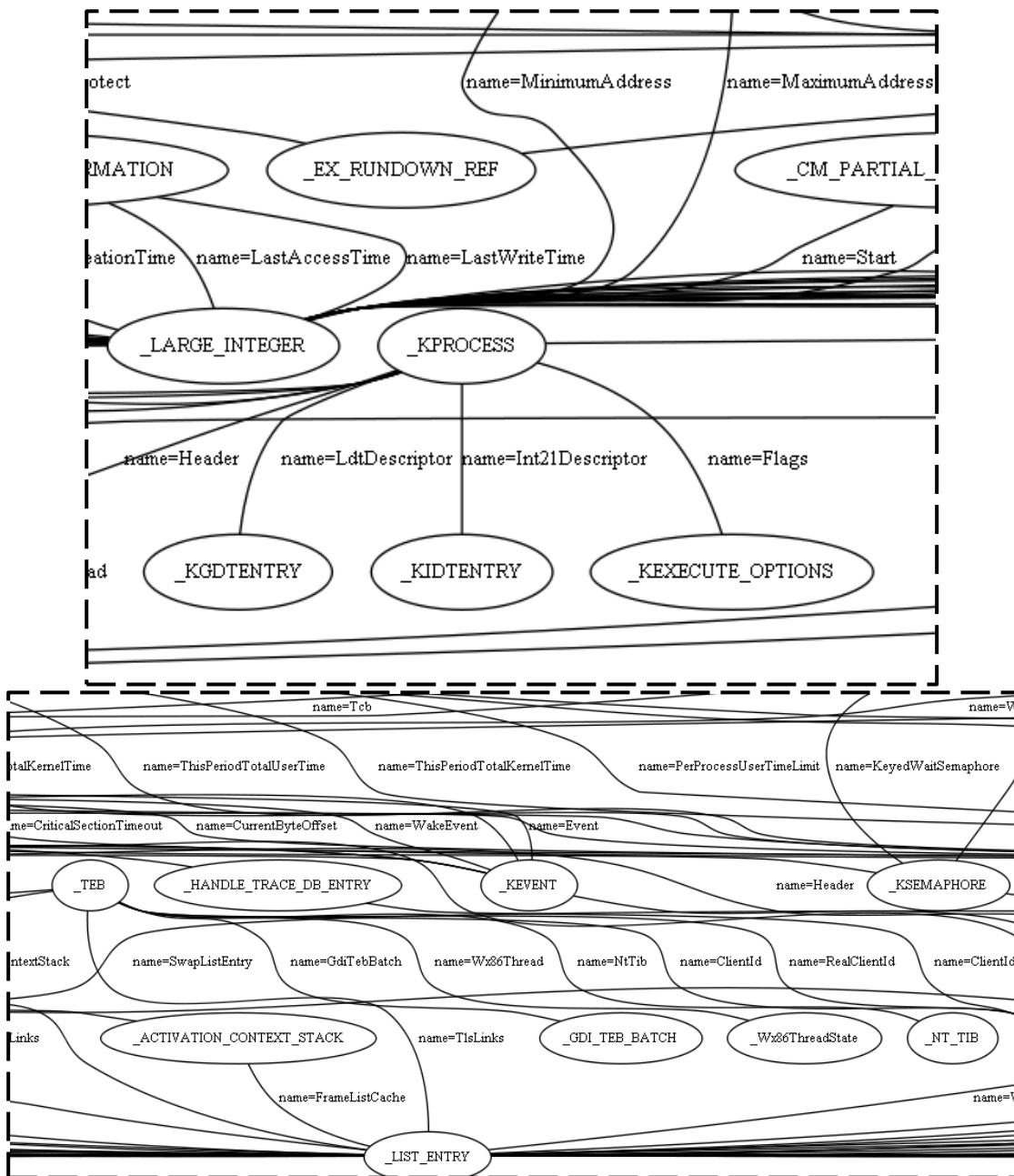


Figure 6-5. Different snapshots of the direct relations type-graph for Windows research kernel.

6.3.2.1 Usage Example

For a better illustration of how OS-KDD works to disambiguate generic pointers problem using our *points-to* analysis algorithm, we will use the code snippet in Figure 6-6 as a running example to explain our analysis phases and objectives.

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
};
```

```

} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER;

typedef struct _KProcess {
    LIST_ENTRY ThreadHeadList;
} KProcess, *PKProcess;

typedef struct _EPROCESS {
    void* UniqueProcessId;
    LIST_ENTRY ActiveProcessLinks;
    KProcess kpc;
    int DebugPort;
} EPROCESS, *PEPROCESS;

typedef struct _EThread {
    void* UniqueThreadId;
    PEPROCESS parentProcess;
    LIST_ENTRY ActiveThreadLinks;
} EThread, *PEThread;

typedef struct _ExHandle {
    int* handle;
} ExHandle;

LIST_ENTRY PsActiveProcessHead;
PEPROCESS MyProcesses[2];
PEPROCESS AllocatePrMemory()
{
    PEPROCESS p;
    .....
    p = (PEPROCESS) malloc(sizeof(EPROCESS));
    return p;
}
PEThread AllocateThMemory()
{
    PEThread p;
    .....
    p = (PEThread) malloc(sizeof(PEThread));
    return p;
}

void CreateProcess(PEPROCESS p_ptr)
{
    p_ptr = (PEPROCESS)AllocatePrMemory();

```

```

MyProcesses[0] = p_ptr;
.....
p_ptr->UniqueProcessId = ExHandler(MyProcesses[0]);
p_ptr->DebugPort = ExHandler(MyProcesses[0]);
updatelinks(&p_ptr->ActiveProcessLinks, &PsActiveProcessHead);
CreateThread(p_ptr);
}

void DeleteProcess(PEPROCESS p)
{
    LIST_ENTRY NextEntry = p->kpc->ThreadListHead.Flink;
    while (NextEntry != &p->kpc->ThreadListHead)
    {
        WaitThread = GetThread(NextEntry);
        NextEntry = NextEntry->Flink;
    }
    EThread CreateThread(PEPROCESS p)
    PETHread th = (PETHread)AllocateThMemory();
    int handle = 12345; th->UniqueThreadId = ExHandler();
    updatelinks(&th->ActiveThreadLinks, p->kpc->ThreadHeadList);
}

void* ExHandler()
{
    ExHandle tempHandle;
    .....
    return tempHandle.handle;
}

void updatelinks(PLIST_ENTRY src, PLIST_ENTRY tgt)
{
    src->Flink = tgt->Flink;
    tgt->Blink = src->Blink;
}

void main()
{
    PEPROCESS ptr[2];
    CreateProcess(ptr[0]);
    CreateProcess(ptr[1]);
}

```

Figure 6-6. A motivating example in C language reflecting generic pointers and casting problems.

6.4 Points-to Analysis Algorithm

Indirect inclusion-based relations (generic pointers dereferencing) cannot be computed from the generated abstract syntax trees directly. Thus, we developed a *points-to* analysis algorithm that has the ability to perform field, flow and context sensitive analysis on large-scale C programs. Our *points-to* analysis algorithm has four sub-steps in order to compute the indirect relations *type-graph*: Intraprocedural Analysis, Interprocedural Analysis, Graph Unification and Context-sensitive Analysis – discussed in details in section 6.4.1, 6.4.2, 6.4.3 and 6.4.4, respectively.

6.4.1 Intraprocedural Analysis

The goal of the intraprocedural analysis step is to compute a local graph – denoted as LTG (Local Type Graph) – for each procedure in the program graph but without information about caller or callees. Algorithm 6-2 summarizes the process of performing an intraprocedural analysis for a procedure P . The intraprocedural analysis module of OS-KDD takes the abstract syntax trees of the target program as input and outputs a local *type-graph* with nodes for all pointers, pointer-compatible and global variables, `malloc` operations, assignments and return instructions. Analysis is performed on each procedure in the program, and OS-KDD assumes unknown initial values for parameters, local and global variables, and all memory locations reachable from the analysed procedure.

Algorithm 6-2. Intraprocedural Analysis Algorithm

1:	Procedure IntraproceduralAnalysis (ASTFile F)
2:	\forall ASTLine $L \in F$
3:	if $L \ni$ variable var (<i>type</i>) then EscapeAnalysis (var);
4:	if ($scope == null$) then $var \subseteq$ global variable
5:	elseif $L \ni$ function parameters then $var \subseteq$ Local function parameter
6:	else $V \subseteq$ Local variable
7:	CreateNode (var , $scope$);
8:	endif
9:	if $L \ni$ assignment function call return statement then ComputeTransFun();
10:	end
11:	Procedure ComputeTransferFunction (L)
12:	end

Section 6.4.1.1 explains the nodes creation part and section 6.4.1.2 explains how edges are computed and connected between graph nodes to generate constraints sets between graph nodes.

6.4.1.1 Graph Nodes Creation

The intraprocedural analysis module performs a linear scan over the abstract syntax trees of the program procedures and creates the graph nodes, as follows:

- *Pointers and Pointer-Compatible Variables.* For each pointer declaration or pointer-compatible variable declaration, the intraprocedural analysis module creates a new node for it and performs escape analysis to check the function scope of the variable. Variables generally are local or global variables.
- *Procedures.* For each procedure definition `proc`, the intraprocedural analysis module creates a node for each formal-in parameter `param`. For procedure call; the analysis module creates two nodes for each formal-in argument `arg`; one is for the argument itself and the node holds the name of that argument, and the second node is an auxiliary node contains the argument index (relative position) in the procedure. These auxiliary nodes are used in the interprocedural analysis phase to compute the implicit assignment relations between the formal-in arguments and parameters. For example, given a procedure call `proc(x, y)`, $x, y \in \text{arg}$; the analysis module creates two nodes for the arguments `x` and `y`, in addition to other two auxiliary nodes `proc:1` and `proc:2`.
- *Assignment Statements.* An assignment statement expresses a store, write (include `malloc` statements) or load operation in the program execution. The analysis module creates a node for the variables of the left and right hand sides, if not already created in one of the previous steps. A `malloc` statement is always the right-hand side of an assignment statement. We create a node for allocated object of `void` type unless the `malloc` statement is cast.
- *Return Statements.* The analysis module creates two nodes; one node for the return statement itself and the other one for the returned value.

- *Casting Statements.* The analysis module creates a node of `void` type, as the actual target type of their variables will be estimated during the analysis.

Initially, any newly created node of a variable `var` has the type \mathbb{T}_{var} of its declared variable, and \mathbb{T}_{var} is updated at dereference operations and when `var` is indexed into a structure or array pointed to by another variable `varn` and `varn` has a different type \mathbb{T}_n .

6.4.1.2 Edges Computations

To obtain high scalability in performing the *points-to* analysis, we compute a transfer function for each procedure, procedure call, and return and assignment statements in order to summarize the relations between graph nodes. This transfer function describes the modification side effect of these entities independently of the procedure input that summarizes its *points-to* relations. A transfer function is basically a formal way of describing a relationship between an input and output. At this stage of the analysis OS-KDD builds the initial edges based on the computed *transfer function* as described in Table 6-1, as follow.

First, procedures; A procedure's transfer function is a relation between the formal-in parameters and the auxiliary nodes that hold the indexes of these parameters. The directed edges between these nodes are expressed as: (i) an *inlist* edge between each formal-in parameter node and its relevant auxiliary node, and (ii) an *outlist* edge from the auxiliary node to its relevant formal-in parameter node. *Second, procedure calls;* a transfer function for a procedure call reflects the relation between the nodes of formal-in arguments and auxiliary nodes. The directed edges between these nodes are expressed as an *inlist* edge between each argument node and its relevant auxiliary node. A procedure call includes computing edges for the program global variables, escaped local variables and dynamically allocated objects whose values may be accessed by `proc` or the procedure that *proc* invokes. *Third, return statements;* a transfer function for a return statement is a relation among left hand side, the procedure return node and the returned value node. The directed edges between these nodes are expressed as: (i) an *inlist* edge between the left hand side and the return node. (ii) An *inlist* edge between the return node and returned value node. (iii) An *outlist* edge between the return node and the left hand side.

Similarly, the formal-out parameters of a procedure `proc` include not only its return value but also the global variables, escaped local variables (from this procedure) and dynamic allocated objects whose values may be modified by `proc` or the procedures that `proc` invokes. *Fourth, assignment* statements; a transfer function for an assignment statement is a relation between the nodes of left and right hand sides of the assignment statement. The directed edges between these nodes are expressed as: (i) an *inlist* edge from left hand-side to right hand-side, and (ii) an *outlist* edge from the right hand-side to left hand-side.

Table 6-1. Transfer function description.

Local points-to sets $pts()$, constraints between nodes, and edges (\rightarrow a directed *inlist* edge between two nodes, \leftarrow a directed *outlist* edge).

	Code	Local $pts()$	Constraints	Edges
Procedure	$proc(\overrightarrow{param})$	$pts(proc:index_{param}) \supseteq pts(param_{index})$	$proc:index_{param} \supseteq p$	$index_{param} \rightarrow p$ $index_{param} \leftarrow p$
Call	$proc(\overrightarrow{arg})$	$pts(arg_{index}) \supseteq proc(index_{param})$	$arg_{index} \supseteq index_{param}$	$arg_{index} \rightarrow index_{param}$
Return	$p = proc();$ $type\ proc(\overrightarrow{type\ var})$ {return x ;}	$pts(proc_{-1}) \supseteq pts(x)$	$proc_{-1} \supseteq x$	$proc_{-1} \rightarrow x$
Assignment	$p = \&q$	$loc(q) \in pts(p)$	$p \supseteq \{q\}$	$p \rightarrow q$ $p \leftarrow q$
	$p = q$	$pts(p) \supseteq pts(q)$	$p \supseteq q$	$p \rightsquigarrow q$ $p \leftarrow q$
	$p = *q$	$\forall v \in pts(q) : pts(p) \supseteq pts(v)$	$p \supseteq *q$	$p \rightarrow *q \rightarrow v : p \rightsquigarrow v$ $p \leftarrow *q \leftarrow v$
	$*p = q$	$\forall v \in pts(p) : pts(v) \supseteq pts(q)$	$*p \supseteq q$	$v \rightarrow *p \rightarrow q : v \rightsquigarrow p$ $v \leftarrow *p \leftarrow q$

For better understanding, we show the analysis results for some code snippets from the motivating example depicted in Figure 6-1 – section 6.3.2.1. Consider a call to procedure `Updatelinks`, where the formal-in parameters are `(src, tgt)`, and the actual passed arguments are `(&ActiveProcessLinks, &ActiveProcessHead)`, and consider these explicit assignment statements `src→Flink = tgt→Flink` and `tgt→Blink = src→Blink`; OS-KDD computes the transfer function for those statements as shown in Figure 6-7(a) and Figure 6-7(b), respectively. For return statements, given this fragment of code `UniqueThreadId = ExHandler()`, the computed transfer function is as is shown in Figure 6-7 (c).

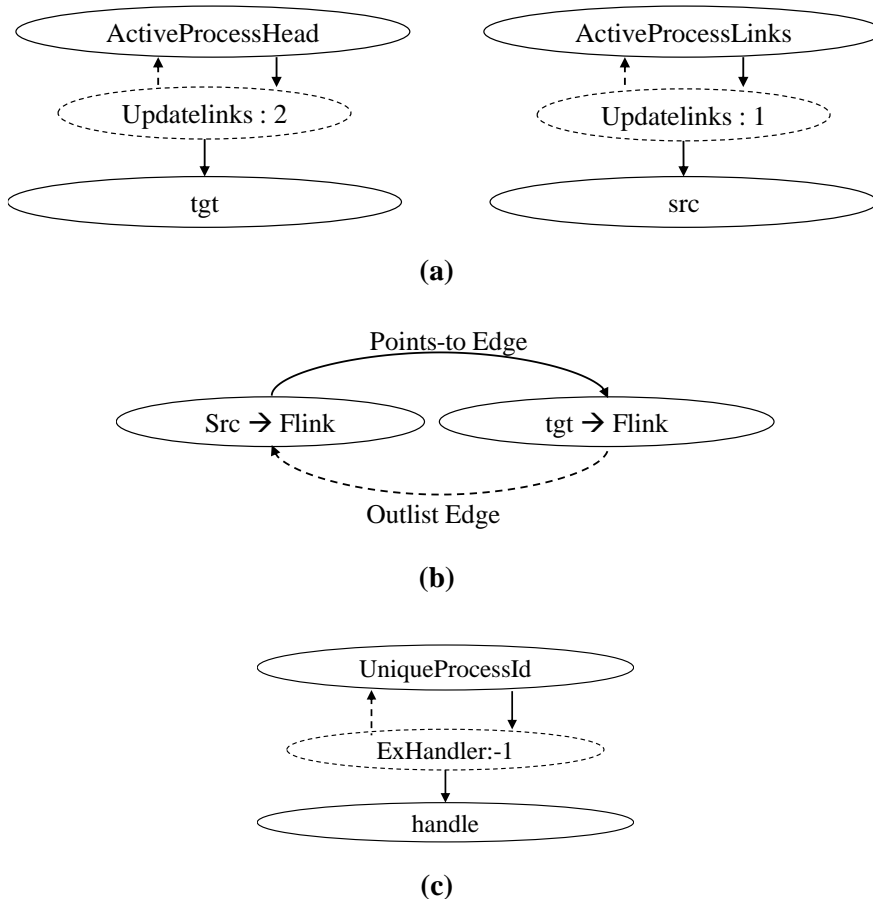


Figure 6-7. Intraprocedural analysis graph.

From the computed transfer function and after connecting the graph edges we formulate a set-constraint – shown in Table 6-1 – that determines the nature of the source and destination

nodes of an edge. For example, an edge from node p to node q constraints the *points-to* set of p to be a subset of the *points-to* set of q . The constraint-set built at this stage of the analysis is an initial set that will be updated based on the results of the later analysis steps.

6.4.2 Interprocedural Analysis

In this phase, we perform an interprocedural analysis that enables performing *points-to* analysis across the different program files to perform a whole-program analysis. This enables adding new edges between graph nodes in order to eliminate incomplete information between callers and callees of procedures. The result of the interprocedural analysis phase is updating the LTG graph by adding new edges and removing some of the existing edges, in order to compute the calling effects (returns, arguments and parameters), but without any information about the calling contexts of the procedures. Algorithm 6-3 summarizes this interprocedural analysis step.

Based on the computed transfer function in the intraprocedural analysis phase, OS-KDD starts to map between callers and callees. This is done by propagating the local *points-to* sets computed at the intraprocedural step to their use sites consistently with arguments' indexes in the corresponding call sites, and then removing the internal *inlist* edges. Thus we can map between the procedure arguments and parameters, and procedure returns. Figure 6-8 shows the analysis results of this step for the examples discussed in the intraprocedural analysis step. Figure 6-8(a) and Figure 6-8(b) show the analysis results for a procedure declaration and its call, procedure returns, receptively.

Algorithm 6-3. Intraprocedural Analysis Algorithm

1:	Procedure Interprocedural Analysis (Graph G)
2:	\forall Node $N \in G$
3:	if $\exists N$ has the form $N(\text{Procedure Name} : \text{index})$ then
4:	Create inlist edge ($N.outlist$, $N.inlist$); Create outlist edge ($N.intlist$, $N.outlist$);
6:	Delete dummy nodes ();
7:	end if
8:	end

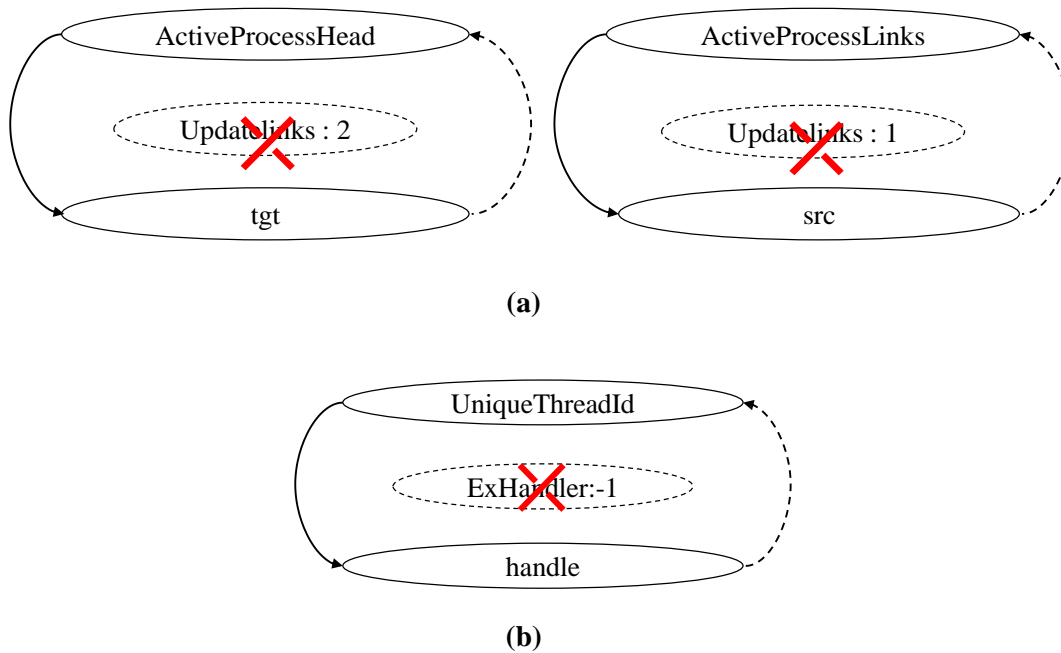


Figure 6-8. Interprocedural analysis graph.

6.4.3 Graph Unification

Before proceeding with the context-sensitivity analysis step, we need to ensure that field-sensitivity property is applied accurately and the *points-to* sets of reference nodes are relatively complete – but without calling contexts yet. For a better understating for the problem, consider the following piece of code from our motivating example:

```
void updatelinks (PList_Entry src, PList_Entry tgt)
updatelinks (&ptr->ActiveProcessLinks, &ActiveProcessHead)
```

In this piece of code, we pass an object type to the procedure `updatelinks`, however, the procedure manipulates the fields of the passed object `Flink` and `Blink`, as the definition of the `_PList_Entry` data type is:

```
typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER;
```

To solve this problem, we apply a unification algorithm to the *type-graph*, as follows: given node A with *points-to* set S and $s \in S$, if s has child-relation edge with a field reference node f ; we create a points-to edge between f and A . Figure 6-9 shows the analysis results of this step of the aforementioned example. Graph unification is an important step to generate an accurate and balanced graph to ensure true field-sensitivity analysis results.

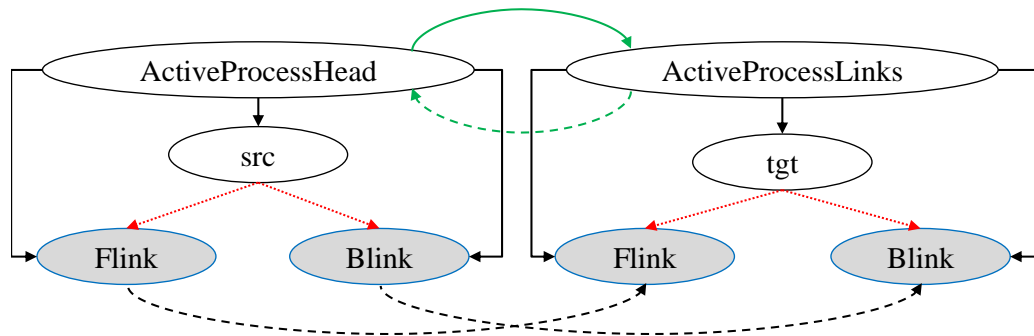


Figure 6-9. Graph unification results.

6.4.4 Context-Sensitive Points-To Analysis

Without context-sensitivity, the analysis of procedures that have different calling contexts would result in imprecise *points-to* sets. The key in achieving true context-sensitivity in OS-KDD is: (i) using the computed transfer function for each procedure call and apply its calling contexts, to bind the output of the procedure call according to the calling site. (ii) A graph node must not be analysed until all of its callers and callees – affecting that value of that node – have been analysed and their *points-to* sets are updated into the graph. (iii) Compute the transitive closure of some graph nodes to build complete strongly connected components, in order to fully resolve recursion cycles.

In OS-KDD, context-sensitivity is achieved by detecting the entire call paths between callers and callees. During a program execution, we have: (i) *cyclic call paths*; a cyclic call path means the existence of function recursions in the program's source code. Details of applying context-sensitivity in the presence of recursion cycles are discussed in section 6.4.4.1. (ii) *Acyclic call paths*; an acyclic call path means there is no recursive calls in the program call-graph. Context-sensitivity in this case is applied straight forward using our algorithm.

Algorithm 6-4 summarizes the context-sensitive *points-to* analysis module of OS-KDD.

Algorithm 6-4. Context-Sensitivity Points-to Analysis

1:	Procedure PointsToAnalysis (PDG PDG, Graph G, TransferFunction TF)
2:	\forall Node N \in G
3:	\forall InListNode in \in N.InList
4:	Compute points-to set (in);
5:	N. PointstoSet. Add (in. PointstoSet);
6:	N. PointstoSet. Add (in);
7:	\forall PointedToNode toN \in N. PointstoSet
8:	\forall Child ch \in N. Children
9:	CopyNode (ch);
10:	Connect edges ();
11:	UpdateNodePointsTo (N, toN);
12:	Write the Graph();
13:	end
14:	Procedure UpdateNodePointsTo (Node N, PointedToNode toN)
15:	if N.fnScope \neq toN.fnscope) then \forall SubPointedToNode StoN \in toN. PointstoSet
16:	if StoN.fnScope == N.fnScope then N. PointstoSet. Add(StoN);
17:	else UpdateNodePointsTo (N, toN);
18:	end

For acyclic call paths, OS-KDD starts the analysis by building a call graph of the target program's source code. A call graph is a directed graph that represents calling relations between procedures in a program. Each node N_c represents a procedure and an edge E_c from p to q , where $(p, q) \in N_c$, indicates that procedure p calls procedure q . The call graph can be computed easily from the interprocedural graph, as the main objective of interprocedural graph is mapping between callers and callees. Then, the graph is traversed in a topological order [193], starting with the top node – according to the call graph – that does not have any node dependency, and thus we guarantee that each node has its *inlist* nodes already analysed before proceeding with the node itself. For a directed acyclic call-graph, $G(N_c, E_c)$, a topological ordering means assigning a priority value to each node in the graph such that, for all edges $p \rightsquigarrow q \in E$, $pts(p) < pts(q)$. Thereafter pointers are analysed iteratively until their *points-to* sets are fully traversed, and then we propagate the *points-to* set of each node into its successors accumulating to the bottom node. Context-sensitivity is achieved by distinguish the calling contexts for a procedure using the

points-to sets of the higher-levels nodes. A *points-to* edge here is a tuple $\langle n, v, c \rangle$ represents a pointer n points to variable v at context c , where the context is defined by a sequence of functions and their call-sites to compute valid call paths between the graph nodes.

A hidden problem in *points-to* analysis is the implicit relations between the program objects that have not been explicitly declared in the source code, but created due to indirect pointer dereferencing *via* function pointers. Most of these relations could be detected during the context-sensitive analysis phase. However, variables that are in the same function scope might have implicit relations between each other as one of these variables might escape to other thread of execution that modifies indirectly other pointer-compatible variables within the same original scope. In order to detect such indirect relations we compute the transitive closure of the some specific graph nodes, in order to detect indirect *points-to* relations between nodes within a local function scope that have not been detected in the intraprocedural analysis phase.

Transitive closure can be thought of a graph nodes reachability problem. Consider a graph $G(N, E)$ with nodes $(p, q) \in N$, a transitive closure \equiv a directed path from p to q in G and the result of node q depends on node p . When the value of a given node is modified or dereferenced, the values of all reachable nodes must also be updated. In our algorithm, we compute the transitive closure only for a set of nodes that has the same function scope, but are not but not included in one *points-to* set. Such that, \forall two nodes v and n where $v \in pts(n)$ and v and n has different function scope, check the function scope of n and x where $x \in pts(v)$, if the function scope is the same then create a transitive closure relation expressed as a *points-to* edge between n and x . Figure 6-10 shows the transitive closure analysis results for the `UpdateLinks` example. We have discovered that there is an indirect *points-to* relation from `ActiveProcessHead` to `ActiveProcessLinks`.

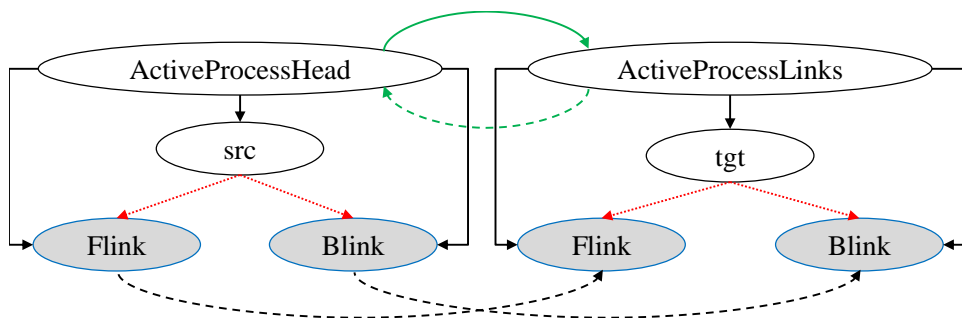


Figure 6-10. Transitive closure computations results

6.4.4.1 Function Recursion

To handle recursive calls, the call graph is divided into strongly connected components (SCCs). A directed graph is called strongly connected if there is a path from each node in the graph to every other node. In order to compute accurate strongly connected components, indirect function call information should be updated accurately into the *type-graph*, to guarantee precise results. A strongly connected component of graph $G (N, E)$ is a maximal set of nodes n , where $n \subseteq N$ such that $\forall C \forall D \in N, C \rightsquigarrow D$ and $D \rightsquigarrow C$, as shown in Figure 6-11.

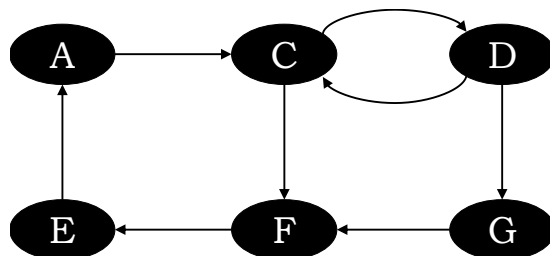


Figure 6-11. Strongly connected components in a directed graph cycle.

Computing strongly connected components can greatly affect performance; therefore we follow an inline approach that only detects cycles that are visited during the context-sensitive analysis step not when a cycle is created in the program’s source code. From the call-graph, we detect strongly connected components using Tarjan’s algorithm [189] – also called depth-first search – and back-propagate component information along edges traversed by the depth-first search. The power of this algorithm is the ability to identify all cycles in linear time. If a new edge $C \rightsquigarrow D$ introduces a cycle then C must be visited during a forward depth-first search from D . Then, for each node that forms a strongly connected component with other node, we mark the

R bit of the node as true. Once a cycle has been detected we collapse its nodes into a new node that represents a cycle, as all nodes in the same SCC are guaranteed to have identical *points-to* sets and can safely be collapsed together. Then, we reapply the context-sensitive analysis algorithm described in section 6.4.3 on these new nodes that represents SCCs. Node collapsing enables getting a finite set of reduced call paths, as all nodes in a SCC have the same call paths between callers and callees.

6.5 Implementation and Evaluation

We have implemented a prototype of OS-KDD using C#. OS-KDD uses *pycparser* [230] to generate AST files of the C program's source code. Figure 6-12 shows the analysis sequence of OS-KDD in the implementation phase. OS-KDD then uses abstract syntax trees files to apply the *points-to* analysis algorithm to generate the *type-graph*. we have used Microsoft's Parallel Extensions [231] to leverage multicore processors in an efficient and scalable manner to implement OS-KDD. Threading has also been used to improve parallelization of computations (.NET supports up to 32768 threads on a 64bit platform). In the intraprocedural analysis, OS-KDD analyses each AST file using a separate thread. For interprocedural analysis, OS-KDD allocates a thread for each procedure to parse the AST files to map between the procedure parameters and arguments. However, for the context-sensitive, the analysis is done on sequential-basis as each node depends on its predecessors. When the analysis is done, OS-KDD writes the *type-graph*. It replaces each variable node with its data type and for fields and array elements we add the declared parent type. Then, OS-KDD reformats the results of our analysis to the DOT language [232], as a simple visualization for the analysis results.

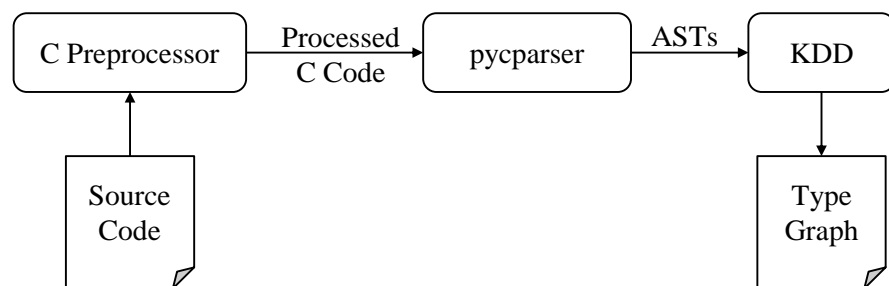


Figure 6-12. Kernel Source Code Analysis Sequence.

We performed two different types of experiments with OS-KDD to demonstrate its scalability and effectiveness. We measured the soundness and precision of OS-KDD using different sets of benchmark programs. We analysed the Linux kernel v3.0.22 and WRK (Windows Research Kernel) using OS-KDD, and performed a comparison between the computed pointer relations using OS-KDD and the manual efforts to solve these relations in both kernels. Our implementation and evaluation platform is a 2.5GHz core i5 processor with 12 GB RAM.

6.5.1 Soundness and Precision Experiment

A *points-to* analysis algorithm is *sound* if the *points-to* set for each variable contains all its actual runtime targets, and is *imprecise* if the inferred set is larger than necessary thus imprecise results could be sound [193]. For instance, if the inferred *points-to* set of variable p , $pts(p) = \{ a, c, b \}$ while the actual runtime targets are only a and b , then the algorithm is sound but not precise and thus there exist a false positive rate. If $pts(p) = \{ a, c \}$ and the actual runtime targets are a and b then the algorithm is neither sound nor precise, and thus there exist both false positives and negatives.

OS-KDD achieves a high rate of precision and 100% soundness of the inferred *points-to* sets. This exemplary rate of soundness achieved is because OS-KDD performs *points-to* analysis not only on the declared pointers, but also on the pointer compatible-variables. Thus, we guarantee that all pointer dereferencing operations whether direct or indirect is included in the analysis algorithm. However, OS-KDD cannot guarantee the same rate in precision, where precision depends on the *points-to* analysis algorithm and false alarms are likely to exist. We used a selection of C programs from the SPEC2000 and SPEC2006 benchmark suites, and other open source C programs, to measure the soundness and precision of OS-KDD. Figure 6-13 shows a snapshot of OS-KDD tool while analysing one of the program benchmarks.

```

file:///C:/Personal/PhD/PointoToAnalysisModified/PointoToAnalysis/bin/Debug/Point...
Finished pointer analysis of C:\PointerAnalysis\AST\pbeampp2.txt
Finished pointer analysis of C:\PointerAnalysis\AST\pbea.txt
Finished pointer analysis of C:\PointerAnalysis\AST\dblampp1.txt
Finished pointer analysis of C:\PointerAnalysis\AST\dsimplex.txt
Finished pointer analysis of C:\PointerAnalysis\AST\pbeampp.txt
Finished pointer analysis of C:\PointerAnalysis\AST\output.txt
Finished pointer analysis of C:\PointerAnalysis\AST\_dblampp.txt
Finished pointer analysis of C:\PointerAnalysis\AST\pbeampp1.txt
Finished pointer analysis of C:\PointerAnalysis\AST\pstart.txt
Finished pointer analysis of C:\PointerAnalysis\AST\psimplex.txt
Finished pointer analysis of C:\PointerAnalysis\AST\mcfutil.txt
Finished pointer analysis of C:\PointerAnalysis\AST\mcf light.txt
Finished pointer analysis of C:\PointerAnalysis\AST\parmanag.txt
Finished pointer analysis of C:\PointerAnalysis\AST\mcfutil1.txt
Finished pointer analysis of C:\PointerAnalysis\AST\dstart.txt
Finished pointer analysis of C:\PointerAnalysis\AST\dbea.txt
Finished pointer analysis of C:\PointerAnalysis\AST\main.txt

Number of files processed = 25
Time to analyze given files (in sec) = 25.1364377
Time to generate Dot file (in sec) = 0.0830048

```

Figure 6-13. A snapshot of OS-KDD while analysing one of the program benchmarks.

Table 6-2 shows the characteristics of these benchmark C programs, in addition to the experimental results of OS-KDD analysis of each of these benchmark programs. We also show indications of memory, time and processor usage of running OS-KDD on these benchmark programs. We used SLOCCount [233] to count Lines of Code (LOC) for the benchmark programs used in my experiments. “LOC” denotes the lines of code of the program. “Pointer Inst” denotes the number of pointer instructions. “Proc” is number of Procedures. “Struct” is number of the type definitions. “AST T” is the time consumed, “AST M” is memory usage, and “AST C” is CPU usage to generate the ASTs. “TG T” is the time consumed, “TG M” is memory usage, and “TG C” is CPU usage to build the *type-graph* of the target benchmark program. We show the peak value for both time and processor usage.

Table 6-2. Soundness and precision results of OS-KDD on a suite of benchmark programs.

Benchmark	LOC	Pointer Inst	Proc	Struct	AST T (sec)	AST M (MB)	AST C (%)	TG T (sec)	TG M (MB)	TG C (%)	P (%)	S (%)
art	1272	286	43	19	22.7	21.5	19.9	73.3	12.3	17.6	100	100
equake	1515	485	40	15	27.5	25.4	20.4	87.5	14.1	21.1	98.9	100
mcf	2414	453	42	22	43.2	41	28.5	14	23	27	98.2	100
gzip	8618	991	90	340	154.2	144.6	70.5	503.3	81.4	68.3	97.0	100
parser	11394	3872	356	145	305.2	191.2	76.7	661.4	107.8	74.3	94.5	100
vpr	17731	4592	228	398	316.1	298.7	80.2	1031.5	163.2	79	NA	100
gcc	222185	98384	1829	2806	3960.5	3756.5	93.5	12962	2200	94	NA	100
sendmail	113264	9424	1005	901	2017.2	1915.1	91.6	6609	1075.0	91.5	97.7	100
bzip2	4650	759	90	14	82.3	78.1	45.5	271.6	44.2	42.9	98.9	100
Wave5	7764	6523	84	245	142.6	142.2	70.5	504	81.9	69.8	97.3	100
crafty	20650	5422	236	402	325	315.2	82.1	1101.2	173.2	82.6	98.3	100

We manually verified each program to get an accurate estimation of the points-to set. For programs that are less than 4 KLOC, we instrumented pointers manually. For larger programs we picked a random set of generic pointers based on my understanding of the program. However, we could not measure precision for some programs because of their big size. We also ran each program and monitored allocations in physical memory to get the actual runtime targets (i.e. relevant points-to set). Then we used the equation below to calculate precision. Our results show that for the benchmark C programs analysed by OS-KDD, a high level of precision and 100% of soundness was achieved. The results also show that for significantly sized C programs, OS-KDD is able to process the application code with very acceptable CPU time and memory usage.

$$Precision = \frac{Relevant\ Points\ to\ Set \cap Retrieved\ Points\ to\ Set}{Retrieved\ Points\ to\ Set}$$

6.5.2 Operating System Kernel Analysis Experiment

To illustrate the scale of the problem presented by C-based operating systems, we performed a simple statistical analysis on the WRK (~ 3.5 million LOC) and Linux kernel v3.0.22 (~ 6 million LOC) to compute the amount of type definitions (data structures), global variables and generic pointers used in their source code. Table 6-3 summarizes this analysis. “TD” column shows the number of type definitions, “GV” is the number of global variables, “DL” shows the number of doubly linked lists, “Uint” is the amount of the declared unsigned integers, and “AST” shows AST files size in gigabyte.

Table 6-3. Kernel source code analysis.

	TD	GV	void*	Null*	DL	Uint	AST
Linux	11249	24857	5424	6157	8327	4571	1.6
WRK	4747	1858	1691	2345	1316	2587	0.9

OS-KDD scales to the very large size of such operating system’s kernel. OS-KDD needed 12.7 hours to analyse the WRK and 21.6 hours to analysis the Linux kernel. Comparing OS-KDD to KOP, KOP has to be run on a machine with 32GB RAM. As our *points-to* analysis

is performed offline and just once for each kernel version, the performance overhead of analysing kernels is acceptable, and does not present a problem for any security application that makes use of OS-KDD's generated *type-graphs*. Re-generation of the graph is only necessary for different versions of a kernel where data structure layout changes may have occurred. As the analysis is done offline and just run once for each kernel version, performance is not such an important factor in our analysis. However, OS-KDD can be extended to be more efficient with regard to service pack and patch updates. OS-KDD can be extended to locate code change locations and just analyse the corresponding data structures, variables and system code that relate to the code updates – details are discussed in chapter 9, in the future work section.

To evaluate the accuracy of OS-KDD's generated *type-graphs*, we performed a comparison between the pointer relations inferred by OS-KDD and the manual efforts of operating systems' experts to solve these indirect relations in both kernels. We manually compared around 130 generic pointers from WRK and 150 from the Linux kernel. These comparisons show that OS-KDD successfully deduced the candidate target type/value of these members with 100% soundness. Because of the huge size of the operating system's kernel code base, we could not measure its precision for nearly 60% of the members we used in our experiment, as there is no clear description for these members from any existing manual analysis. We were only able to measure precision for well-known objects that have been analysed manually by security experts and whose purpose and function is well-known and documented. The resulting precision was around 96% in both kernel versions, and Table 6-4 shows the results of our comparison.

Table 6-4. A comparison between OS-KDD type-graphs and some kernel data value-invariants.

OS	Structure / GV	T	KDD	S(%)	P(%)
Linux	thread_group	S	task_struct.thread_group: [task_struct.group_leader.thread_group thread_group.next: [list_head.next, task_struct.thread_group.next, task_struct.group_leader.thread_group] thread_group.next: [list_head.next, task_struct.thread_group.next, task_struct.group_leader.thread_group] Context: Thread	100	100
	journal_info	v*	journal_info: [btrfs_trans_handle, gfs2_trans, nilfs_transaction_info]	100	100
	cg_list	S	cg_list: [list_head, css_set.tasks, css_set __rcu.task] context: task_struct	100	100
	btrace_seq	v*	blktrace_seq, unsigned int	100	NA
Windows	ActiveProcessLinks	S	<i>ActiveProcessLinks</i> : [List_Entry, PsActiveProcessHead] <i>ActiveProcessLinks.Flink</i> : [List_Entry.Flink, PsActiveProcessHead.Flink] <i>ActiveProcessLinks.Blink</i> : [List_Entry.Blink, PsActiveProcessHead.Blink] Context: EPROCESS	100	100
	PsActiveProcess-Head	G	<i>PsActiveProcessHead</i> : [List_Entry, ActiveProcessLinks] <i>PsActiveProcessHead.Flink</i> : [ActiveProcessLinks.Flink, ActiveProcess-Links.Flink] <i>PsActiveProcessHead.Blink</i> : [ActiveProcessLinks.Blink, ActiveProcess-Links.Blink] Context: EPROCESS	100	100
	VadRoot	S	VarRoot: [MM_AVL_TABLE]	100	100

OS	Structure / GV	T	KDD	S(%)	P(%)
Windows	ThreadListHead	S	ThreadListHead: [List_Entry] ThreadListHead.Flink: [List_Entry.Flink]ThreadListHead.Blink: [List_Entry.Blink] Context: ETHREAD	100	100
	PsLoadedModuleList	G	PsLoadedModuleList: [List_Entry] <i>PsLoadedModuleList.Flink:</i> [KLDLDR_DATA_TABLE_ENTRY.InLoadOrderLinks.Flink, List_Entry.Flink] <i>PsLoadedModuleList.Blink:</i> [KLDLDR_DATA_TABLE_ENTRY. InLoadOrder- Links.Blink] Context: LDR_DATA	100	100
	LdtInformation	v*	<i>LdtInformation:</i> [PVOID, PROCESS_LDT_INFORMATION]	100	100
	DirectoryTableBase	U	<i>DirectoryTableBase:</i> [MmCreateProcessAddressSpace:-1] <i>DirectoryTableBase[0]:</i> [PageDirectoryIndex, ULONG64] <i>DirectoryTableBase[1]:</i> [HyperSpaceIndex, ULONG64]	100	100

6.6 Discussion

Our experiments with OS-KDD have shown that a generated *type-graph* is highly accurate and solves the `null` and `void` pointers, and casting problems with a high percentage of soundness and precision. OS-KDD is able to scale to the enormous size of operating system's kernel code, unlike most other *points-to* analysis tools that do not scale to such large-scale analysis. This scalability and high performance was achieved *via* a number of factors: *first*, using abstract syntax trees as a high-level representation for C programs. The compact and syntax-free abstract syntax trees improve time and memory usage efficiency of the analysis. This is because instrumenting abstract syntax trees is more efficient than instrumenting the low-level representation languages because many intermediate computations are saved from hashing, as discussed before in section 6.3.1. *Second*, using the transfer function approach enables precise description for the modification side effects of program procedures while keeping their computations costs to a minimum. Moreover, the compositional manner used in our implementation to developing OS-KDD. Each procedure *proc* is analysed independently and only information about *proc*'s summary, is communicated to other procedures that call *proc*. This enables OS-KDD to scale the analysis to millions of lines of code.

OS-KDD has the ability to accurately disambiguate generic pointers statically to get a precise estimation of their pointer dereferencing operations and extract robust type definitions for the kernel data structures. Such static analysis approach on kernel source code has several advantages that can support developing many other applications, not just *CloudSec++*, as follow: *first*, OS-KDD capabilities enable the implementation of different systematic security solutions that requires formal checking of memory operations such as pointer dereferencing in dynamic data. By this we mean that we have the ability to systematically protect kernel data without the need to understand deep details about kernel data layout in memory, as is done to date. *Second*, OS-KDD minimizes the performance overhead in security applications as a major part of the analysis process is done offline. If no static analysis were done, every pointer dereferencing operation would have to be instrumented, which increases performance overhead and makes the

security software inadequate. *Third*, OS-KDD is capable of systemically analysing the whole address space of an operating system's kernel with nearly a complete coverage of all pointer dereferencing operations. This maximizes the likelihood of detecting zero-day threats that target generic pointers (*via* bad pointer dereferencing) located in kernel dynamic data that do not have explicit integrity constraints that can be extracted from the operating system's kernel source code. *Fourth*, declared types of C variables are unreliable indications of how the variables are likely to be used at system runtime. OS-KDD enables accurate type inference of pointers and pointer-compatible variables of a C program. This enables determining the actual runtime type of a dynamic object by analysing the usage of this object in the code base. *Fifth*, OS-KDD enables checking the integrity of kernel code function pointers that reside in dynamic kernel objects, by inferring the target candidate type for each function pointer. This decreases the need to instrument every function pointer at system runtime, as the addresses of objects that hold these pointers change during runtime. In chapter 8, we discuss different tools that benefit from OS-KDD to enable integrity checks on kernel pointers and function pointers.

6.7 Summary

In this chapter, we described our *points-to* analysis tool (OS-KDD). OS-KDD is a static analysis tool that operates offline on a C-based operating system's kernel source code to generate a robust *type-graph* for the kernel data that reflects both the direct and indirect relations between structures, models data structures and generates constraint sets on the relations between them. Our experiments with OS-KDD prototype have shown that the generated *type-graphs* are accurate and solve the generic pointer problem with a high rate of soundness and precision.

Chapter 7

DIGGER: A Kernel Runtime Objects Discovery Tool

Dynamic kernel runtime objects may be a significant source of security and reliability problems in operating systems. Having a complete and accurate understanding of the runtime kernel dynamic data layout is thus necessary to implement an operating system security application. In this chapter, we discuss a new systematic approach, named DIGGER¹¹. DIGGER uncovers the presence of any running instances of operating system's kernel dynamic objects and data. DIGGER uses a hybrid approach to identify the running instances of kernel objects and data, and their type information. DIGGER provides necessary information to enable a security application to guard against generic pointers exploits and stealthy malware. DIGGER's approach makes use of memory pools used by operating systems, in order to achieve accurate results and high performance. DIGGER uses kernel data definitions generated by OS-KDD in order to automate the process of uncovering kernel dynamic objects and thus DIGGER does not require any prior knowledge of the runtime kernel data layout.

We have implemented a prototype of the DIGGER approach and conducted an evaluation of its efficiency and effectiveness. In section 7.1, we give an overview of the problem and DIGGER's approach to solving it, and in section 7.2, we discuss the high-level architecture of DIGGER and its main components in detail. The implementation and evaluation details are discussed in section 7.3. Section 7.3 covers the evaluation and implementation details of DIGGER and we discuss the key features and limitations of DIGGER in section 7.4.

¹¹ The DIGGER name is derived from the word dig; *i.e.* digging for information in the physical memory of the operating system.

7.1 Overview

In general-purpose operating systems, we usually refer to a running instance of a kernel data structure as a kernel object. Identifying accurately the running instances of operating system's kernel data is an important task in many operating system security solutions, not just for virtualization-aware security solutions applications, as discussed before in chapter 2 and 3. There are mainly two mechanisms used to discover kernel runtime objects: *memory mapping* techniques and *value-invariant* approaches. Both mechanisms have a number of critical limitations that make them vulnerable and inapplicable in real-time operating system security applications, as discussed previously in chapter 2.

Motivated by the limitations of these mechanisms and the need to accurately uncover dynamic kernel runtime objects reliably, we have developed a new approach to discover kernel runtime objects, called DIGGER. DIGGER is a new hybrid mechanism that combines a new *value-invariant* approach and an advanced *memory mapping* technique. This combination enables accurate discovery with fast and nearly complete coverage of the kernel address space. The *value-invariant* approach is used to uncover the presence of the running instances in the physical memory. DIGGER uses the pool memory tagging schema¹² in order to implement this *value-invariant* approach. On the other hand, the *memory mapping* technique is used to reconstruct the object type information in depth including the *points-to* relations with the other running instances – based on the generated *type-graphs* of OS-KDD. The level of the required object type information is selected by the security software user based on the required hierarchical-information depth. This controls the trade-off between details and performance overhead, as some object types have hundreds of fields with hierarchically-organised *points-to* relations with other system data. Details of in-depth analysis of runtime objects are covered in chapter 8. DIGGER has two key features that distinguish it from the other approaches: *first*, DIGGER has

¹² DIGGER is designed mainly to Windows operating system kernels, as the pool memory tagging schema is only related to Windows operating system kernel. However, in section 7.4.1 we show how DIGGER can be implemented in Linux operating systems using the slab allocation mechanism instead of the pool memory tagging schema.

the ability to systematically discover nearly all kernel objects, with no prior knowledge with the runtime kernel data layout. The key to achieve this is the usage of the *type-graphs* generated by OS-KDD, to enable systematic coverage of the pointer-based relations. A *type-graph* statically reflects the runtime kernel data layout of a specific kernel version, with accurate computations for the *points-to* relations. *Second*, the fast coverage and the low performance overhead of the runtime component of DIGGER by using the pool memory tagging schema that make it one of the fastest approaches to locate dynamic objects in memory.

7.2 DIGGER High-Level Architecture

The high-level process of DIGGER is shown in Figure 7-1. DIGGER has three main phases of the analysis; two of them are offline and occur once for each operating system kernel version, and the last phase is the dynamic analysis phase that uses the results of the first two phases in order to run the dynamic analysis and accurately locate the system objects. The DIGGER analysis phases are: *static analysis phase*, *signature extracting phase* and *dynamic analysis phase*. The main objective of the static analysis phase is parsing the operating system kernel source code in order to generate an accurate *type-graph* that reflects statically the runtime kernel data layout. The static analysis phase was discussed in detail in chapter 6, and in this chapter we focus only on the signature extraction and dynamic analysis phases, discussed in section 7.2.1 and 7.2.2, respectively.

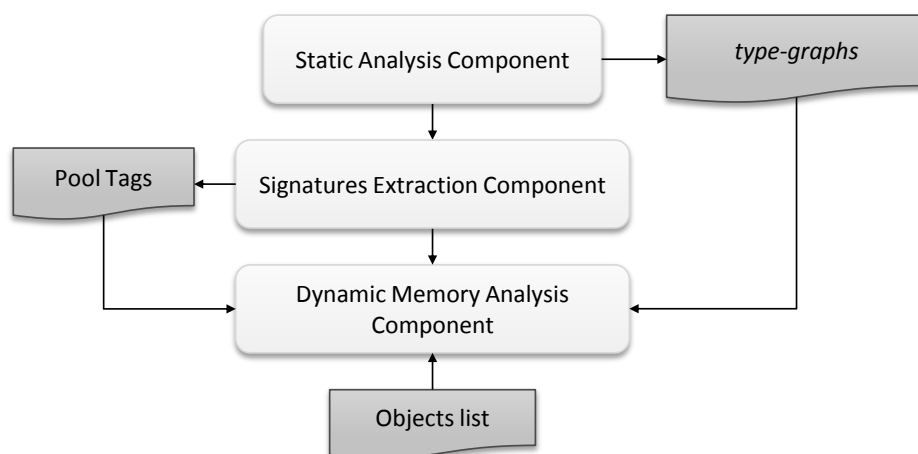


Figure 7-1. The high-level architecture of DIGGER approach.

7.2.1 Signature Extraction Component

One of the main challenges of implementing a *value-invariant* approach or performing signature-based scanning is obtaining robust, unique and small signatures for the operating system kernel data. The complexity mainly comes from: *first*, data structures' sizes are not small. From my analysis of Windows and Linux operating systems kernels, we found that a single data structure could be several hundreds of bytes. Thus, considering the whole data structure as a scanning signature would typically increase the discovery cost and the system performance overhead. *Second*, assuming that not all the fields of a data structure will be included in its signature, the question is “what are the most important fields that cannot be modified during runtime to be included in the scanning signature, to effectively detect stealthy malware and be difficult to be evaded?”. Dolan *et al.* [16] introduced an automated mechanism for generating robust signatures for kernel data structures. Dolan *et al.* profiled data structures and then fuzzed them to determine which were essential to the correct operation of the operating system. This mechanism enabled the identification of the most important fields in a data structure that cannot be modified during the object lifetime. Despite the effectiveness of this approach, it is time consuming as each single data structure requires massive analysis and runs in a sandboxed environment to determine its critical fields. *Third*, an operating system kernel contains thousands of heterogeneous data structures, and this makes the process of generating **unique** signatures for this huge number of the data structures is a challenging task.

In order to overcome the above challenges in DIGGER, we used the pool memory tagging schema of the Windows operating system memory manager to overcome the size and efficiency problems of the signatures (first two problems), and we were motivated by the below paragraph, from Windows internals book [234] – denoted WI-note – to overcome the uniqueness problem (third problem). Details are discussed in section 7.2.1.1 and 7.2.1.2, respectively.

“Not all data structures in the Windows operating system are objects. Only data that needs to be shared, protected, named, or made visible to user-mode programs (via system services) are placed in objects. Structures used by only one component of the operating system to implement internal functions are not objects.”

7.2.1.1 Signature Size and Efficiency Problem

When the Windows operating system object manager allocates a memory pool block to allocate an object, it associates the allocation with a pool tag. A *pool tag* is a unique four-byte tag for each object type that is associated with any dynamically allocated pool block, and it is stored in reverse order in memory. Figure 7-2 shows a snapshot of the *pools tags* used for pool allocations by kernel mode components and drivers in Windows operating system, using the Poolmon¹³ tool [235]. The full pool tag list for the Windows operating system can be extracted from the symbol information *i.e.* – Microsoft Symbols or from the kernel source code (Pooltag.txt file). Pool memory blocks are allocated *via* the routine `ExAllocatePoolWithTag`, shown in Figure 7-3. The formal-in parameters of this routine are: (i) *PoolType*; specifies the type of the used pool memory for the allocation and it is whether paged or non-paged pool memory. (ii) *NumberOfBytes*; specifies the number of bytes to allocate and it is different from one object type to another – *i.e.* the allocated object size. (iii) *Tag*; specifies the *pool tag* and it depends on the type of the allocated object. For instance, to allocate memory for a process where the pool tag is “Proc”, it will appear in the memory as “corP” and the ASCII value would be 0xe36f7250. `ExAllocatePoolWithTag` returns a pointer to the allocated memory pool block.

¹³ PoolMon is a memory pool monitoring tool that displays accurate data about memory allocations and deallocations from the paged and nonpaged memory pools.

Tag	Type	Allocs	Frees	Diff	Bytes
Leak	Nonp	601 < 0 >	0 < 0 >	601	615424000 < 0 >
CM31	Paged	19796 < 0 >	0 < 0 >	19796	85020672 < 0 >
C1cr	Paged	124660 < 0 >	121511 < 0 >	3149	10245216 < 0 >
CM25	Paged	1809 < 0 >	0 < 0 >	1809	7761920 < 0 >
CMAl	Paged	7309 < 1754 >	6438 < 1799 >	871	3567616 < -184320 >
MmRe	Paged	1255 < 8 >	732 < 5 >	523	3275136 < 4640 >
TSwd	Paged	20 < 0 >	9 < 0 >	11	2890672 < 0 >
MmSt	Paged	4543 < 248 >	3647 < 238 >	896	1717984 < 5744 >
Pool	Nonp	13 < 0 >	7 < 0 >	6	1717840 < 0 >
nUsC	Nonp	664 < 0 >	0 < 0 >	664	1531552 < 0 >
ClfI	Paged	19 < 0 >	0 < 0 >	19	1521296 < 0 >
Toke	Paged	22102 < 70 >	21400 < 62 >	702	1242448 < 11840 >
netv	Nonp	4371 < 0 >	2 < 0 >	4369	1172224 < 0 >
Ntff	Paged	2911 < 8 >	1971 < 0 >	940	1158000 < 9856 >
Umbk	Paged	9 < 0 >	0 < 0 >	9	901120 < 0 >
Thre	Nonp	1275 < 12 >	667 < 1 >	608	775328 < 14080 >
EtwB	Nonp	39 < 0 >	5 < 0 >	34	759856 < 0 >
CM29	Paged	45 < 0 >	0 < 0 >	45	737280 < 0 >
File	Nonp	45300 < 1096 >	43684 < 1088 >	1616	535776 < 2688 >
FMFn	Paged	25336 < 233 >	23905 < 220 >	1431	530240 < 4208 >

Figure 7-2. A snapshot of Poolmon tool depicts a set of pool tags of Windows operating system.

```

PVOID ExAllocatePoolWithTag(
    _In_ POOL_TYPE PoolType,
    _In_ SIZE_T NumberOfBytes,
    _In_ ULONG Tag
);

```

Figure 7-3. ExAllocatePoolWithTag pool memory allocation routine.

We use this pool tag as a *value-invariant* signature to uncover the presence of the running instances of the different kernel objects. The *pool tag* field is included in the pool header structure, named `_POOL_HEADER`. The pool header is a data structure used by the Windows object manager to keep track of memory allocations of each allocated pool block. The pool header holds information related to the allocation and free algorithms of allocated pool block.

However, Pool tags alone are not sufficient signatures. This is because the false positive rate in this case will be very high. For instance, if DIGGER scans the memory for the running processes with the process pool tag “Proc”, any memory bytes that parse the same ASCII string will be detected as an object instance from that process object type. Thus, an additional checking signature is required to ensure accurate results. To overcome this problem, we include the first two bytes of the object dispatcher header to be part of DIGGER *value-invariant* formula.

All kernel objects that can be waited to start (created in a wait state) – e.g. processes,

threads, ports, mutex – have an embedded dispatcher header. An object’s dispatcher header is a data structure named `_DISPATCHER_HEADER` that holds lots of information in a fixed-size data structure, as shown in Figure 7-4. A dispatcher header structure describes the type, size and state of a specific object, in addition to object synchronization information [236]. From our analysis we found that the first two bytes of the dispatcher header are unique for each object type, as they describe an object’s type and size, as shown in Figure 7-5. The value of these two bytes can be computed from the generated *type-graph*.

```
typedef struct _DISPATCHER_HEADER {
    union {
        struct {
            UCHAR Type;
            union {
                UCHAR Absolute;
                UCHAR NpxIrql;
            };
            union {
                UCHAR Size;
                UCHAR Hand;
            };
            union {
                UCHAR Inserted;
                BOOLEAN DebugActive;
            };
        };
        volatile LONG Lock;
    };

    LONG SignalState;
    LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;
```

Figure 7-4. Object dispatcher header data structure.

```
kd> dt _DISPATCHER_HEADER
+0x000 Type           : UChar
+0x001 Absolute      : UChar
+0x001 NpxIrql       : UChar
+0x002 Size          : UChar
```

Figure 7-5. A snapshot from Windbg for the dispatcher header structure. The figure shows the offsets of the type and size fields.

In order to apply the additional checking signature efficiently, a formal relation between the pool header and the object should be computed in order to validate the computed addresses and values. As shown in Figure 7-7, each object is prefixed with an object header structure, named `_OBJECT_HEADER`, and the whole object including the object header is prefixed by a pool header structure, named `_POOL_HEADER`. The dispatcher header is located at offset `0x000` of the start address of object itself. For instance, in the `EPROCESS` structure that is the data structure of process object type, the first field at offset `0x000` is the `KPROCESS` data structure and the `_DISPATCHER_HEADER` is the first field of this `KPROCESS` structure, as shown in Figure 7-6. Table 7-1 summarizes some of important fields in the above mentioned structures that will be used later in our analysis algorithm.

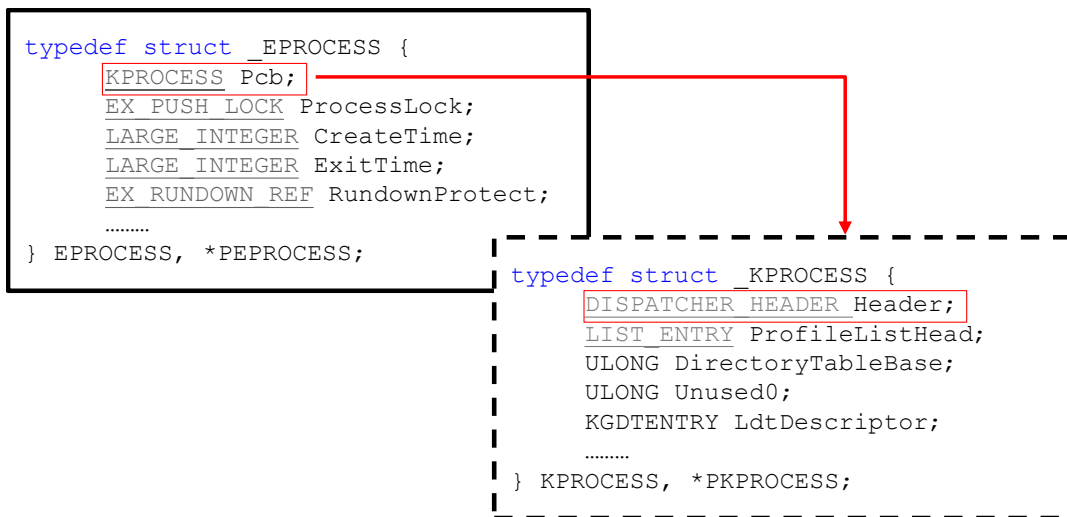


Figure 7-6. The location of the object dispatcher header within the object structure.

Key advantages of using pool memory tagging schema as scanning signatures for the kernel runtime objects include: (i) these pool tags are not tied to a specific kernel data layout and are thus effective in the different Windows operating system kernel versions, where some data layout change may occur. (ii) The compact signature size that does not exceed a few bytes decreases the performance overhead significantly.

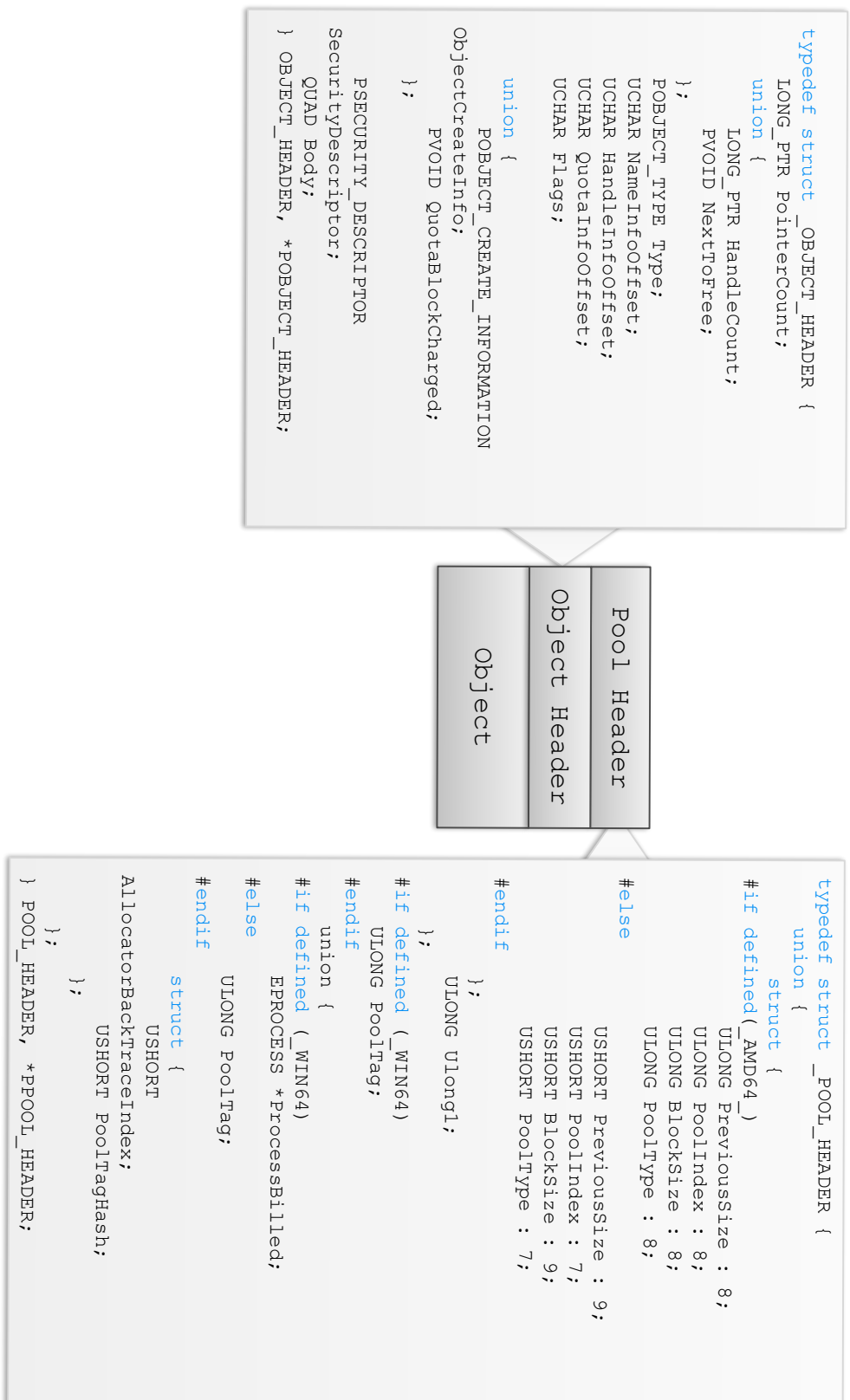


Figure 7-7. The memory layout of allocated objects in the pool memory.

Table 7-1. Description of some fields in pool header and object header structures.

Structure	Field	Description
_POOL_HEADER	PreviousSize	An eight byte offset to previous pool block
	BlockSize	Describes the size of the allocation including the size of the pool header.
	PoolType	The pool from which this allocation takes place, whether paged or nonpaged pool.
	PoolTag	The four character tag used when allocating the buffer.
_OBJECT_HEADER	PointerCount	A count of the number of references to the object
	HandleCount	The number of times a handle (an identifier to the object) has been opened.
	ObjectType	A pointer to the type of the allocated object.

7.2.1.2 Signature Uniqueness Problem

To solve the problem of generating unique signatures for kernel data structures, we considered the WI-note, described in section 7.2. To the best of our knowledge, all current operating system security research for Windows and Linux treats all kernel data structures as objects and does not consider the WI-note. Considering the WI-note enables filtering the list of data structures extracted at the static analysis phase using OS-KDD, and obtaining a precise list of the actual runtime object types. In DIGGER, we consider each kernel data structure that has an associated *pool tag* used by the memory manager as an object and the others are not considered as objects. These structures are treated as normal data structures and have less significance than those that are considered as objects. This massively reduces the number of object types from hundreds to dozens. Data structures that are classified as objects are discovered using the *pool memory tagging schema* – to locate its presence – and then the details of these objects are traversed using the *type-graph*. Other data structures that are not classified as objects are located using the *points-to* relations – that are computed in the *type-graph* – with the kernel objects to locate their presence and also traverse their details.

7.2.2 Dynamic Memory Analysis Component

DIGGER's dynamic memory analysis component is a simple memory scanner that locates system objects using our formulated signatures which are composed of the pool memory tagging schema and the additional checking signature. After that, the dynamic analysis component constructs the details of the uncovered objects using the corresponding *type-graph* of the running kernel version. The dynamic memory analysis component scans the kernel address space with the pool memory allocation granularity, and at every computed address the scanner assumes a valid structure of a running instance. Based on that computed address, the scanner reads the memory bytes and parses them based on the corresponding *type-graph*. The output of the dynamic memory analysis component is an *object graph* that has: (i) nodes that denote the running instances of data structures and the different object types of the kernel, and (ii) directed edges that express the dependency and *points-to* relations (direct and indirect) between these running instances and their fields.

7.2.2.1 DIGGER and Pool Memory

The kernel address space of a running operating system or even a memory image is big in size. The kernel address space ranges from 1GB to 2GB in 32bit operating systems and is up to 8TB in 64bit operating systems according to the available hardware memory and its memory layout. Thus, scanning a whole memory image or a live kernel memory would definitely affect the scanner performance. In DIGGER, to reduce performance overhead and scanning time of the kernel address space, the scanner only scans the memory pages of memory pools instead of the whole kernel address space.

Kernel address space has different pools, and each pool is separated from other pools and is used to allocate specific object types. The memory manager mainly creates two memory pools, and both are used by the kernel address space to store kernel and executive objects: first, *non-paged pool*; the non-paged pool consists of virtual memory addresses that always reside in physical memory and never paged out. Non-paged pool memory is used by the operating system kernel and system device drivers to allocate dynamic data that would be accessed when page

faults are not allowed. Non-paged pool memory is used to store objects such as *e.g.* processes, threads and tokens. This means that such memory pages are valuable targets for rootkits to be able to achieve its objective, as they remain resident in the system memory. *Second, paged pool;* the paged pool memory consists of memory addresses that can be paged in and out, and thus they are used to allocate less important objects that do not require persistent residence in the physical memory.

The amount of memory pages allocated to pools varies and the size is determined based on the operating system running kernel version, processor architecture, and installed RAM. For example, a 32-bit Windows operating system has a total of 2^{32} bytes or 4GB of address space. By default, Windows uses 2GB for system user-mode and 2GB for kernel-mode. The pool memory is part of the 2GB of kernel address space and it is around 460MB per single processor. Table 7-2 summarizes the amount of paged and non-paged pools in different versions of Windows operating systems.

In DIGGER, the scanner only scans the non-paged pool memory. This is because it is considered to be a trusted source of information where we can get all the running objects instances that are potential targets for hackers, as they always reside in physical memory. On uniprocessor or multiprocessor systems there exists only one non-paged pool and this number can be confirmed using the global variable `nt!ExpNumberOfNonPagedPools`. The operating system also maintains a number of global variables that define the start and end addresses of the paged and nonpaged pool memory: `MmPagedPoolStart`, `MmPagedPoolEnd`, `MmNonPagedPoolStart` and `MmNonPagedPoolEnd`. These global variables enable computing exactly the memory pages associated to the non-paged pool and this speeds up the scanning process by limiting the scanned memory area.

Table 7-2. Paged and non-paged pools size in different Windows operating system versions.

Windows Version	Kernel Address Space	Non-paged Pool		Paged Pool	
		Size in MB	Percentage of Kernel Memory	Size in MB	Percentage of Kernel Memory
32-bit Windows XP	2GB	4.03	0.19%	17.96	0.87%
32-bit Windows Server 2008	2GB	15.43	0.75%	24.52	1.19%
64-bit Windows XP	2GB	15.30	0.74%	28.11	1.37%
32-bit Windows Server 2008	2GB	39.20	1.91%	100.06	4.91%
64-bit Windows 7	8GB	174.48	2.13%	372.15	4.54%

7.2.2.2 The Scanning Algorithm

Using our developed pool memory tagging schema signatures as discussed in section 7.2.1, the DIGGER dynamic memory component scans the kernel address space with the granularity of the default size of the pool header data structure, which is eight-byte granularity. The most important fields, from my scanner perspective, in the `_POOL_HEADE`, is the `PoolTag` field. For example, to scan the kernel address space for the running instances of the *process* object type, where the pool tag of this object type is *proc*, the memory scanner continues reading the memory bytes until it reads the hexadecimal value (`0xe36f7250`) of the corresponding pool tag. The next step is confirming whether the located hexadecimal string at address `ADDRx` is a running object instance or not, by applying the additional checking signature on `ADDRx`. Then, the scanner confirms the existence of a running object instance by using the two bytes of the additional checking signature, discussed in section 7.2.1.

At this stage the dynamic memory analysis component assumes a valid structure of a process object running instance. However, until this step the scanner can only identify that there

is a running object instance of type T located at address X ¹⁴, but cannot uncover any details about the object itself – even the object name. To reconstruct the details of the target object located at address X , the graph of the corresponding data structure is then used to traverse the memory pages. This graph includes the dependency relations and offsets of the data structure and its fields and can be retrieved from the *type-graph* generated by OS-KDD. By adding the size of the pool header and the object header to X , the object's start address is computed, denoted T_x . Then we retrieve the object's details based to the *type-graph*. The size of the pool and object headers are calculated from the kernel *type-graph*.

After locating the object and traversing its details, the scanner checks the `BlockSize` value of the `_POOL_HEADER` structure, to find out the size of the pool block that has been allocated for an object instance O , denoted SZ_o . The scanner then skips these bytes to move to next adjacent pool block in the list, without the need to continue scanning the rest of the previous pool block. Within a page the allocated blocks are chained. Each block stores its size and the size of the previous block, which allows faster scanning for the allocated objects. Figure 7-8 summarizes my scanning algorithm in order to locate the running instance of kernel dynamic objects.

For un-mappable memory pages the size of pages set is relatively small. We perform a scan on the whole set of the memory pages using the pool tag and the additional checking signature. However, as the memory mapping information may not be available in such un-mappable memory pages, not all of the details for the discovered objects can be retrieved as we depend on the memory traversal technique according to the generated *type-graph*.

¹⁴ X is calculated by subtracting the offset of the *PoolTag* field (`offset == 0x004`) from the address `ADDRx`. The pool tag field enables computing the start address of the allocated object's pool block.

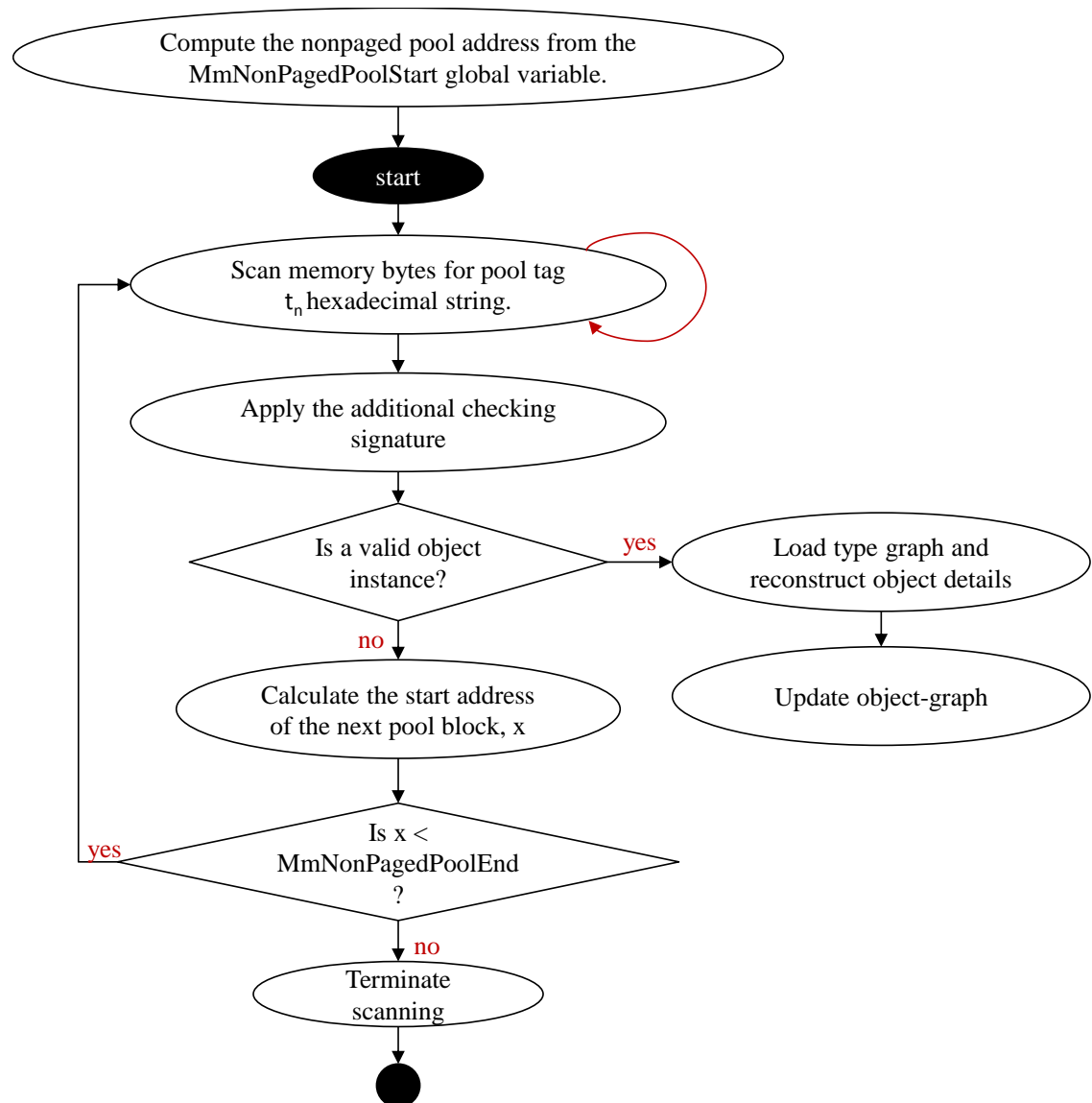


Figure 7-8. A flowchart summarizing the scanning algorithm of DIGGER.

7.3 Implementation and Evaluation

We have developed a prototype of DIGGER. The static analysis component was built using OS-KDD, as described in Chapter 6 [26, 237]. The signatures and runtime components are standalone programs and both components are implemented in C. The runtime component could perform the analysis whether online or offline. Offline analysis is performed on a memory snapshot raw dumps (*e.g.* dumps in the Memory Analysis Challenge and Windows crash dumps), and VMware suspended sessions. Online analysis is performed in a near-real time fashion in a virtualized environment by scanning virtual machines' physical memory at the

hypervisor level. Details of implementing and evaluating DIGGER at a hypervisor level are discussed in chapter 8. In this chapter we only discuss the implementation and evaluation details of DIGGER on memory snapshots (offline analysis). We have evaluated the basic functionality of DIGGER with respect to the identification of kernel runtime objects and the performance overhead of uncovering these objects¹⁵.

7.3.1 Uncovering Runtime Objects

To enable the efficient evaluation of DIGGER's approach, especially its runtime component, we needed a robust ground truth that reflects the exact objects layout in kernel memory so that we could compare it with DIGGER's results, in order to accurately measure the rate of false alarms. To build the ground truth, we extracted all the running instances of the different object types and data structures of the running Windows operating system memory image *via* program instrumentation using the Windows Debugger (WD). In other words, we instrumented the kernel to log every pool allocation and deallocation, along with the allocation/deallocation address using the Windows Debugger. In particular, we modified the GFlags (Global Flags Editor) feature of the memory manager to enable advanced debugging and troubleshooting of the pool memory.

We measured DIGGER efficiency as the fraction of the total allocated objects for which DIGGER was able to identify the correct object type. We performed experiments on different versions of the Windows operating system on a 2.8 GHz CPU with 2GB RAM. Each memory snapshot size was 4GB. Table 7-3 shows the results of DIGGER and WD in discovering the allocated instances for specific object types in two different Windows operating system kernel versions. Memory, paged and non-paged columns represent the size in pages (0x1000 granularity) of the kernel address space, paged pool and non-paged pool, respectively. DIG and WD refer to windows debugger's and DIGGER's scanning results respectively. FN, FP and FP* denote the false negative, reported false positive and the actual false positive rates, respectively.

¹⁵ In this chapter, we only evaluate the basic functionality of DIGGER which is locating system runtime objects. However in chapter 8 more evaluation experiments of DIGGER are discussed.

From Table 7-3 we can see that DIGGER achieves zero false negative rates, and a low false positive rate. However, from our manual analysis of the results, we found that this reported false positive rate is not an actual false positive. This difference represents the deallocated objects that still persist in the physical memory after object termination. We call such deallocated objects “dead memory-pages allocated objects (DMAOs)”. Dead memory pages objects are present because the Windows operating system does not usually clear the contents of memory pages after deallocation to avoid the overhead of writing zeroes to the physical memory. However, we noticed from our analysis that the handle count of the DMAOs is always zero (HandleCount field in the `_OBJECT_HEADER` structure). This enables differentiating the active objects from the DMAOs, and thus our actual false positive rate becomes zero, denoted FP* in Table 7-3.

Table 7-3. Experimental results of DIGGER and WD on Windows XP 32 bit and 64bit.

Object	Windows XP 32bit					Windows XP 64bit				
	Memory		Paged		Non-paged	Memory		Paged		Non-paged
	915255		27493		11741	1830000		35093		17231
	WD	DIG	FN %	FP %	FP* %	WD	DIG	FN %	FP %	FP* %
Process	119	121	0.00	1.65	0.00	125	125	0.00	0.00	0.00
Thread	2032	2041	0.00	0.44	0.00	2120	2121	0.00	0.04	0.00
Driver	243	243	0.00	0.0	0.00	211	211	0.00	0.00	0.00
Mutant	1582	1582	0.00	0.0	0.00	1609	1609	0.00	0.00	0.00
Port	500	501	0.00	0.19	0.00	542	542	0.00	0.00	0.00

We argue this finding that whenever the kernel has to allocate a new object it will return the pool block address from the pool free list head. When an object is deallocated, the kernel will free its memory. But that does not necessarily mean the memory gets overwritten. For instance, the `EPROCESS` structure of a newly created process will overwrite the object data of a process that has been terminated previously [138]. This because when a block is freed using the `free`

function call, the allocator just adds the block to the doubly linked list of free blocks without overwriting memory. The DMAOs can provide forensic information about an attacker's activity. We discuss in chapter 8, how DMAOs how can be used in developing memory forensics tools to check for rootkits evidence.

7.3.1.1 *Performance Overhead*

We have evaluated DIGGER's runtime performance to demonstrate that it can perform its memory analysis in a reasonable amount of time. We measured DIGGER's running time when analysing the memory snapshots used in my experiments. The median running time was around 0.8 minutes to uncover 12 different object types from the nonpaged pool, and 1.6 minutes to uncover another 15 object type from the paged pool. This time included the time of loading the memory snapshot from the disk to the runtime analysis component. We consider this running time to be acceptable for offline analysis and even for online analysis in virtualized environments. This is because DIGGER is able to detect the DMAOs that could be created and terminated between the scan time intervals. However, we cannot argue that DMAOs results would be 100% accurate. Comparing DIGGER with SigGraph [19], DIMSUM [25] and KOP [14], DIGGER is the fastest with highest coverage and lowest performance overhead. The performance overhead of extracting object details based on our generated *type-graph* differs according to the required details-depth. Figure 7-9 shows the time consumed (in seconds) to extract object details with different depths for all of the running instances from a specific object type. "I" denotes the number of the running objects from the object, and "D" denotes the depth of the extracted details. Depth means the number of dereferencing operations that could be reachable from a pointer or data structure. The depths declared in Figure 7-9 were selected randomly.

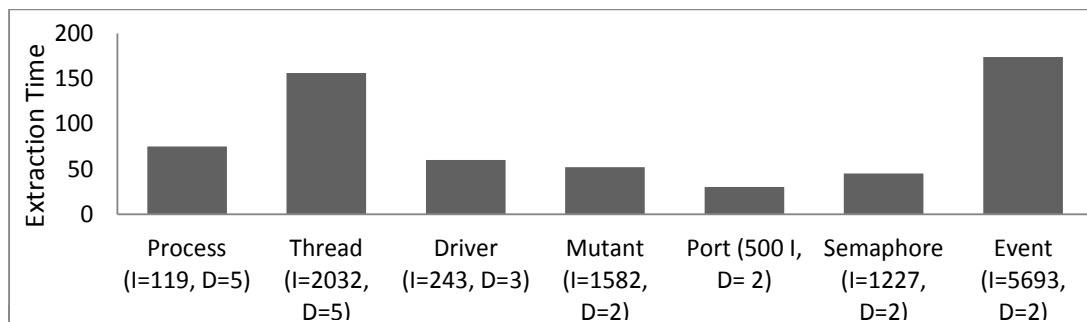


Figure 7-9. Object details extraction normalized time.

7.4 Discussion

DIGGER's approach enables obtaining a robust view of operating system kernel runtime objects. This view is not affected by the manipulation of actual kernel memory contents and thus enables developing different robust operating system kernel data security applications as discussed in chapter 8 – *e.g.* systematic kernel data integrity checks, detecting stealthy malware and brute-force scanning. Key features of DIGGER include: (i) using OS-KDD to statically recognize the runtime kernel data layout, in order to enable systematic coverage for the running instances of kernel dynamic data, without a need for having extensive hands-on experience with operating systems' kernel implementations. (ii) The robust and small signature size used to uncover runtime objects that has significantly reduced the performance overhead of the dynamic memory analysis component. This definitely helps developing lightweight security applications that does not affect system performance. (iii) DIGGER's runtime component is not related to a specific version of Windows operating system kernels. DIGGER can work on any version and either 32-bit or 64-bit layouts, as it does not depend on any hard-coded offsets for a specific kernel build. The pool memory tagging schema is the basic of any Windows NT kernel and thus DIGGER is applicable to a wide range of Windows operating systems such as Windows XP and Windows 7.

As the pool memory tagging schema is only related to the kernel of Windows operating systems, the current approach used in DIGGER is not applicable to Linux and UNIX operating systems. However, the DIGGER approach can be implemented in a similar way on Linux/UNIX operating systems, as these operating systems implement a similar mechanism to pool memory,

named slab allocation [238]. Slab allocation is a memory management mechanism for Linux and UNIX operating systems for allocating kernel runtime objects efficiently by providing better object layout in the kernel address space at system runtime. The basic idea behind the slab allocator is having memory chunks, known as slab caches, similar to memory pools in Windows operating systems, of commonly used objects kept in an initialized state – *i.e.* memory chunks are preallocated. Whenever the memory manager receives a memory allocation request for a specific object type, the memory manager instantly responds to the request with a pointer to a pre-allocated slot from the free slots list. With slab allocators, deallocation of an object does not free its memory, but only links this memory slot to the list of free slots [238]. This is similar to happen in Windows operating systems, where deallocated blocks are added to the free `ListHeads` list, to be used later by a newly allocated object.

In the Windows system kernel, there are only two memory pools and each pool contains different types of allocated objects and structures. However, Linux operating system's kernel has a number of caches that are structured in a doubly-linked list called a cache chain. A cache chain is similar to the `_LIST_HEAD` of the pool memory blocks used in the Windows kernel. Caches are created by the `kmem_cache_create()` routine. Each cache is designated for a specific object type, and maintains blocks of contiguous pages in memory called slabs (`struct slab_t`) [239]. Each slab is further divided into equal size segments of the object type that the cache is maintaining, similar to pool block in Windows operating system. A full list of these caches can be retrieved from the `/proc/slabinfo` file of the kernel source code. A snapshot of the `slabinfo` file is shown in Figure 7-10, and the columns details are described in Table 7-4, in order. All cache information is stored in a very big data structures, named `kmem_cache_s`, a snapshot of its important fields¹⁶ is shown in Figure 7-11 and `slab_t` is shown in Figure 7-12. Table 7-5 describes some important fields that are used in my analysis.

¹⁶ The important fields are represented from DIGGER's perspective.

kmem_cache	59	78	100	2	2	1
ip_fib_hash	10	113	32	1	1	1
ip_contrack	0	0	384	0	0	1
urb_priv	0	0	64	0	0	1
clip_arp_cache	0	0	128	0	0	1
ip_mrt_cache	0	0	96	0	0	1
tcp_tw_bucket	0	30	128	0	1	1
tcp_bind_bucket	5	113	32	1	1	1
tcp_open_request	0	0	96	0	0	1
inet_peer_cache	0	0	64	0	0	1

Figure 7-10. A Snapshot of the slab allocator info file of Linux kernel.

Table 7-4. slabinfo file columns description.

Column	Description
<i>cache-name</i>	Holds a human readable name for the cache.
<i>num-active-objs</i>	Expresses the number of objects that are in use.
<i>total-objs</i>	Reflects the number of objects that are available in total.
<i>obj-size</i>	The size of object.
<i>num-active-slabs</i>	The number of slabs containing active objects.
<i>total-slabs</i>	The number of slabs in total.
<i>num-pages-per-slab</i>	The pages required to create one slab.

```

struct kmem_cache_s {
    struct list_head    slabs_full;
    struct list_head    slabs_partial;
    struct list_head    slabs_free;
    unsigned int        objsize;
    unsigned int        flags; /* constant flags */
    unsigned int        num; /* # of objs per slab */
    spinlock_t          spinlock;
    ...
    unsigned int        gfporder;
    unsigned int        gfpflags;
    size_t              colour; /* cache colouring range */
    ...
    void (*ctor)(void *, kmem_cache_t *, unsigned long);
    void (*dtor)(void *, kmem_cache_t *, unsigned long);
    char                name[CACHE_NAMELEN];
    struct list_head    next;
    ...
};

```

Figure 7-11. A partial list of important fields within the kmem_cache_t structure.

```

typedef struct slab_s {
    struct list_head    list;
    unsigned long      colouroff;
    void                *s_mem;
    unsigned int        inuse;
    kmem_bufctl_t      free;
} slab_t;

```

Figure 7-12. slab_s data structure.

Table 7-5. The description of kmem_cache_t structure fields.

Structure	Fields	Description
Kmem_cache_s	slabs_full slabs_partial slabs_free	These data structures are the slabs associated with a specific cache to speed up the allocation and freeing of objects, and they denote: the doubly linked lists of the in-use objects, the list that has prime candidate for next object allocation, and the list that has no allocated objects, respectively.
	objsize	The size of the object contained within the cache.
	num	The number of cache objects per a slab.
	slabp_cache	A pointer to the kernel cache that is used for the slab_t
	name	A string describing the cache that holds a specific object type.
	next	A list_struct pointer to the next cache in the kernel cache chain.
Slab_t	list	The linked-list that holds the slab, and it is one of slab_full, slab_partial or slab_free.
	s_mem	The start address of the first object within the slab.
	inuse	The number of active objects in the slab.

To implement a customized version of DIGGER for the Linux operating system, we consider the `name` field of the `kmem_cache_s` structure as an equivalent to *pool tags* used in the original DIGGER version. Then using `objsize`, `s_mem`, `list` and `next` fields, DIGGER can scan the memory and traverse the linked lists: `slabs_full` and `slabs_partial`, in a similar way like what happened for Windows operating system. After locating a running instance, the *type-graph* is then used to retrieve the related object type information. Figure 7-13 summarizes the main algorithm that can be applied in DIGGER to enable objects discovery in Linux operating systems.

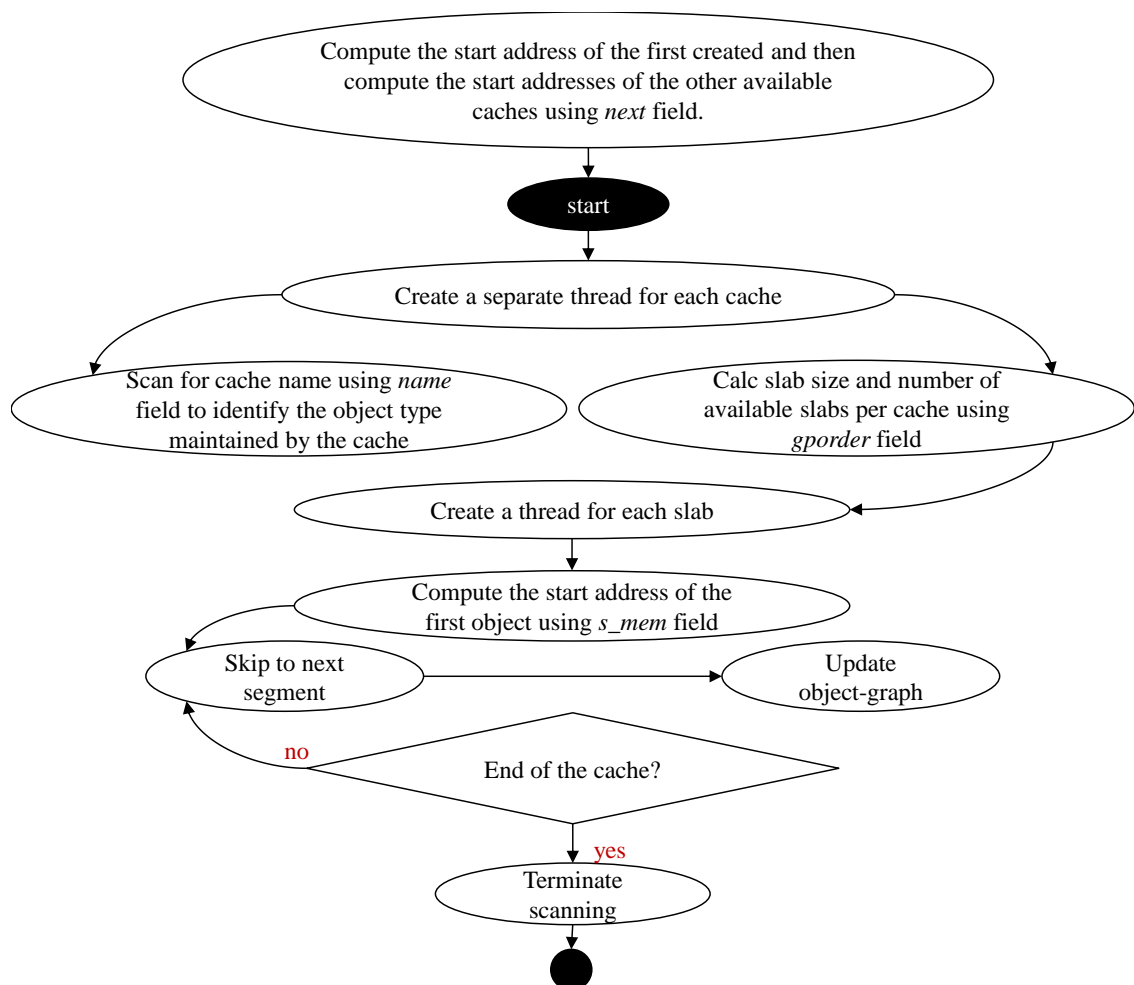


Figure 7-13. Proposed DIGGER Implementation for Linux operating systems.

7.5 Summary

Current efforts in uncovering the running instances of kernel dynamic objects from a robust view have a number of limitations that make them impractical to be used in operating system security solutions that supports real-time and robust protection. In this chapter, we introduced a new approach called DIGGER, which systemically, speedily and accurately uncover kernel runtime objects from a robust view that cannot be tampered with. DIGGER is a hybrid mechanism that combines a new *value-invariant* approach and an advanced *memory mapping* technique. Our evaluation of DIGGER has shown its effectiveness in uncovering system objects for Windows operating systems. DIGGER's approach is limited to Windows operating systems however it is generic enough to be used in the different versions of the Windows operating system kernels. Moreover, we discussed how the DIGGER approach could be customized and implemented for analysis of Linux and UNIX operating systems, by utilizing the slab allocation mechanism of the Linux operating system's kernel instead of the pool memory concept that related of Windows operating systems.

Chapter 8

CloudSec₊₊: A virtualization-Aware Security

Solution

In this chapter, we introduce *CloudSec₊₊*, an enhanced version of *CloudSec* that has the ability to provide systematic and pre-emptive protection for the hosted virtual machines in IaaS platform. *CloudSec₊₊* depends mainly on OS-KDD and DIGGER, in order to provide proficient security for the guest operating systems of hosted virtual machines. Moreover, *CloudSec₊₊* is designed to provide security from a cloud provider's perspective and at the same time incorporates the consumers in the security process of their virtual machines, without direct interaction with *CloudSec₊₊*. In this chapter, we also discuss a set of security and memory forensics tools that are deployed in *CloudSec₊₊* in order to defend against zero-day threats that target operating system kernels. These security tools work mainly on operating system kernel data to check the integrity of generic pointers located in runtime kernel dynamic objects. Checking the integrity of generic points enables detection of many types of kernel data threats such as object hiding, dangling pointers and function pointer manipulation.

8.1 Introduction

CloudSec₊₊ is a *virtualization-aware* security solution that has the ability to protect guest operating systems of the hosted virtual machines actively and systematically. It is designed to utilize all the components described in previous chapters: *CloudSec*, OS-KDD and DIGGER to deliver a robust security framework that works in the IaaS platform to systematically and externally protect the hosted virtual machines.

Our research contribution in this chapter is twofold: *first*, introducing an enhanced version of *CloudSec* – named *CloudSec₊₊* – that overcomes many of *CloudSec*'s limitations by using OS-KDD and DIGGER in its design and implementation phases. Moreover, *CloudSec₊₊* has a new design architecture that supports passive involvement of cloud consumers in the security

process of their hosted virtual machines, as discussed in section 8.2. *Second*, Introducing a set of operating systems security tools that are used to defend against zero-day threats that target operating system kernel dynamic data. Our developed tools focus on checking the integrity of the generic pointers located in the running instances of kernel dynamic objects, to enable defending against kernel rootkits that modify the system control flow by hooking kernel dynamic data, as discussed in section 8.3.

8.2 CloudSec++

The design of *CloudSec++* provides additional functionalities, compared to *CloudSec*, in order to achieve its intended tasks. These additional functionalities include: (i) *CloudSec++* enables systematic solution of the *semantic gap* problem for the guest operating systems without the very limiting prerequisite of operating system kernel data layout knowledge. *CloudSec++* mainly depends on OS-KDD to enable systematic generation of kernel data definitions in order to enable efficient overcoming of the semantic gap problem. (ii) *CloudSec++* systematically infers the kernel version of the running operating systems in the hosted virtual machines, to ensure complete automation of the security process. The exact kernel version of the running operating system is a key to enable accurate interpretation of the underlying hardware bytes. However, the exact running version of a hosted virtual machine is not always available to the cloud provider, as service pack updates and operating system upgrades are likely to happen. (iii) *CloudSec++* enables systematic and fast discovery of kernel runtime objects from a trusted source that cannot be tampered with, with a low performance overhead that enables near real-time protection by utilizing DIGGER in the external monitoring phase. *Fourth*, the new design of *CloudSec++* allows the inclusion of cloud consumers in maintaining the security process of their hosted virtual machines, without direct interaction with *CloudSec++* that could violate our design requirements discussed before in chapter 4 and 5.

The threat model in *CloudSec++* is identical to the one used in *CloudSec*, and the basic high-level architecture is nearly the same. Before discussing the architecture details of *CloudSec++* in section 8.2.2, we discuss in section 8.2.1 a new mechanism that enables sys-

tematic inference of the kernel version of running operating system in a hosted virtual machine. Such a process is required to enable pure automation for the security process in *CloudSec₊₊*.

8.2.1 Systematic Kernel Version Inference

In order to make *CloudSec₊₊* completely systematic, we need to provide *CloudSec₊₊* with the exact kernel version of the running guest operating system in order to start its job to overcome the semantic gap problem and provision the proper security. The kernel version of a guest operating system is not always known by the cloud provider, as it is the property of the cloud consumer. Moreover, even if the cloud provider is aware of the original kernel version of a virtual machine, the consumer could perform service pack updates or even operating system upgrades without a need to report that to the provider.

CloudSec₊₊ finds out the exact kernel version of the running operating system in a hosted virtual machine by systemically inferring the kernel version from the virtual machine's physical memory. To enable systematic inference of the running operating system kernel version, we benefit from the OS-KDD generated *type-graphs* to create a unique signature for each kernel version, as the kernel data layout may change from one kernel build to another.

To generate the kernel inference signatures, we tried to identify a common kernel data structure that has different signatures in each kernel build. From our analysis of Windows¹⁷ XP (SP2 and SP3) and Linux (v3.0.22 and v 3.1.10 – source code), we found that there is not a distinctive structure across the different kernel builds for each operating system. However, we noticed that data members' offsets of some structures do change in each build. Based on that, we can identify such a structure with its data members to be the kernel version inference signature. However to guarantee accurate results this structure must present all the time during kernel execution. In order to find out such trusted signature, we followed the approach used by Brendan *et al.* [16] in order to identify the best data members that could be included in the kernel

¹⁷ We used Windows symbol information to find the data members' offsets in the different kernel versions, as there is no source code for those versions except WRK.

version inference signature. Brendan *et al.* introduced a dynamic analysis approach that can profile a target data structure to determine its commonly used fields at runtime. They then apply specific fuzzing techniques on those fields to determine which are essential to the correct operation of the operating system, and the runtime modifications to these fields that cause runtime errors. Based on their computations for the trusted fields, we use these fields to form the kernel version inference signature.

Based on this approach, we found that the `EPROCESS` data structure, in Windows operating system, with some of its fields are robust to exist during system runtime and modifying their values causes system crash. Brendan's results stated that out of 221 fields in the `EPROCESS` structure, 72 fields were identified as always accessed during the profiling stage. Then at the runtime modification fuzzing stage, 29 fields out of the 72 fields failed the test – *i.e.* their modification did not result in any loss of the operating system functionality – and the rest passed the test however there was a false positive rate of some of fields that did not passed the test every time in the training phase. As a result, the robust fields on the `EPROCESS` structure are, as shown in Figure 8-1. Based on that, we generate an inference signature for each kernel build using `EPROCESS` data structure in Windows with the robust its members. `CloudSec++` then at runtime locates the first loaded process (which is always the `system` process that initializes the kernel subsystems and must exist during system runtime) and checks its signature to infer the running kernel version.

```
Pcb.ReadyListHead.Flink
Pcb.ThreadListHead.Flink
WorkingSetLock.Count
Vm.VmWorkingSetList
VadRoot
Token.Value
AddressCreationLock.Count
VadHint
Token.Object
QuotaBlock
ObjectTable
GrantedAccess
ActiveProcessLinks.Flink
Peb
Pcb.DirectoryTableBase.0
```

Figure 8-1. Robust signatures of `EPROCESS` data structure according to Brendan's results.

Brendan's approach can be also implemented to compute a set of robust kernel version inference signatures for other operating systems that uses data structures to represent objects such as Linux, UNIX and Solaris.

8.2.2 CloudSec++ High-Level Architecture

Figure 8-2 shows the high-level architecture of *CloudSec++*. *CloudSec++* has a similar architecture to *CloudSec*, with a few modifications, as follow: *first*, OS-KDD is a stand-alone tool that is not deployed in *CloudSec++*. The generated kernel data definitions are stored in the same location as in *CloudSec*. However, the definitions in the current architecture are generated by OS-KDD rather than the manual approach, and are represented as directed *type-graphs*. *Second*, DIGGER component is an online memory analysis tool that enables uncovering of kernel dynamic runtime objects. DIGGER is part of the *CloudSec++* architecture and is deployed in the *security virtual machine* of *CloudSec++*. In order for DIGGER to function properly, it has access to the *type-graphs* database and the memory access handler that enables reading the hardware bytes of the hosted virtual machines *via* the hypervisor. *Third*, the web portal virtual machine is a virtual machine used to allow cloud consumers to maintain and track the security status of their hosted virtual machines without direct access to the *security virtual machine* of *CloudSec++*. This virtual machine is running a web service that allows consumers to configure and enforce the security policies, upon consumer request, of their own virtual machines through *CloudSec++*. Moreover, this web service allows consumers to track their virtual machines' security status that is maintained by *CloudSec++*. The web portal virtual machine has a policies and reports database that stores the consumers' security policies and the security reports of the virtual machines that have been sent from *CloudSec++* to the consumer. Communications between *CloudSec++* and the web portal virtual machine is conducted *via* the hypervisor over a secure communication channel, and communications between the web portal virtual machines and the other hosted virtual machines are not allowed. Communications security with the web service is beyond our research scope, however communications security can be enhanced by applying various secure web services mechanisms [240-242].

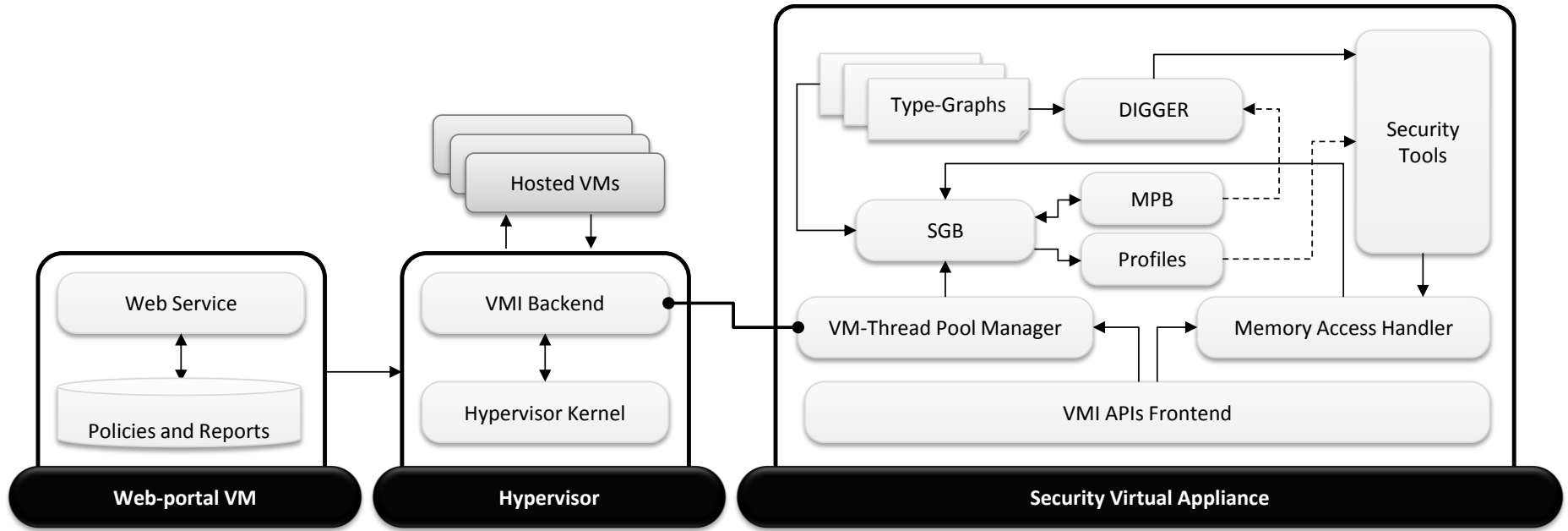


Figure 8-2. CloudSec++ High-level architecture.

The inclusion of the consumers in the security process has two scenarios from which a consumer can choose which to participate. Whichever the chosen scenario is, the consumer does not need to build perfect security policies to detect all system threats, as such feature is already enabled by default in *CloudSec₊₊* to protect the hosted virtual machines against advanced kernel rootkits that affects the operating system behaviour not the running applications. Consumers' security policies enable the consumer to protect the user-mode of the operating system of their virtual machines. The two security scenarios are:

First, certified virtual machines scenario. In this the scenario consumers agree on the conducted Service-Level Agreements (SLAs¹⁸) to get their virtual machines protected by *CloudSec₊₊* – the security software of the cloud provider. A service-level agreement is a service contract where the service details are formally defined, such as service performance, delivery methods and operational costs. SLAs for IaaS focus on characteristics of the hosting cloud platform, the shared responsibility security model, virtual machine specifications the billing information. For example, Amazon, one of the biggest IaaS providers in the market, defines in their SLAs the relation between providers and consumers. A provider is responsible to provide all the necessarily building blocks that enable consumers to create and manage virtual machines.

In certified virtual machines scenario, the provider becomes aware of the virtual machine behaviour (according to the security polices selected by the consumer) and this enables more accurate provision of the security process in the user-mode of the running operating system. In this scenario, consumers also get accurate security reports *via* the web portal about the security status of their virtual machines. These reports contain detailed information about kernel violations that has affected the running operating system. As *CloudSec₊₊* focuses on dynamic kernel data security, the policies include only how to control and protect kernel runtime objects that are

¹⁸ In this research project, we add to such SLAs that the provider has the right to protect its virtual infrastructure against memory leakage attacks that might be caused of the hosted virtual machines. Based on that, the running kernels of the hosted virtual machines are being monitored and protected to enable a secure operational environment for the other hosted virtual machines.

affected by the user calling contexts such as processes, threads and drivers. For example, a consumer can enforce policies *e.g.* “allow all processes/services except”, “block all processes/services except”. In this case *CloudSec₊₊* can protect the virtual machines from running malicious processes and hiding processes or services.

Second, uncertified virtual machines scenario. In this scenario consumers do not agree to be protected by *CloudSec₊₊*. In this case, the situation is more oblivious with regard to the user-mode protection. The consumers then should rely on their supported traditional *in-guest* security software to protect the user-mode of the running operating system. Only operating system kernels will be protected against advanced kernel rootkits that could lead to memory leakage or sharing exploits.

8.2.2.1 *CloudSec₊₊ Implementation and Evaluation*

The evaluation and implementation details of *CloudSec₊₊* are similar to the evaluation and implementation details of *CloudSec*. The difference between the two architectures is the automation process of overcoming the semantic gap by incorporating DIGGER and OS-KDD in *CloudSec₊₊* architecture, and the new design feature that enables consumers to maintain the security status of their virtual machines.

8.2.2.1.1 *Deployment Model*

Our platform for evaluating *CloudSec₊₊* is a 2.8 GHz Intel Xeon with 8GB of RAM, running ESX 4.1. Figure 8-3 depicts the deployment model of *CloudSec₊₊*. *CloudSec₊₊* is running ESX 4.1 hypervisor and hosts the *security virtual machine*, two hosted virtual machines and the web portal virtual machine. The *security virtual machine* is configured with 2GB RAM and deployed as a virtual appliance, running Ubuntu Linux 8.04 Server JeOS, and hosts the vCompute APIs and our monitoring and security code and tools. The monitoring code running in the security virtual machine and DIGGER’s code are normal Linux C programs written using vCompute and Posix Threads APIs. Both of the security virtual machines and the web portal virtual machines are isolated from other hosted virtual machines in separate virtual networks using two dedicated virtual switches. The hosted virtual machines are running Windows XP

64-bit and 32-bit, each on a 2.8 GHz CPU with 2GB RAM. The hosted virtual machines were executed under normal workload with an average of 50 processes and 910 threads.

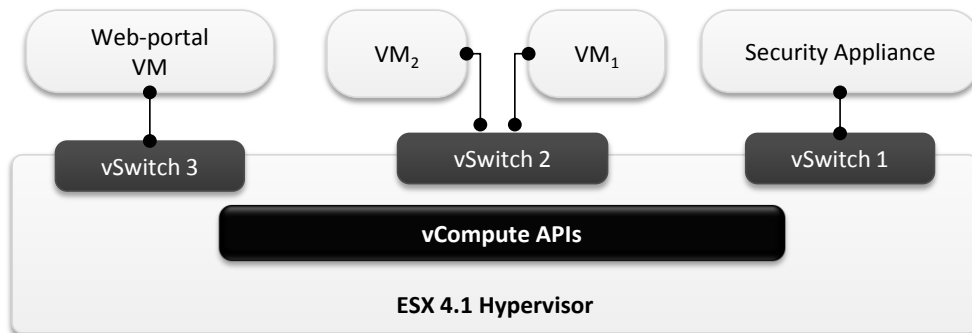


Figure 8-3. CloudSec++ deployment model.

The proposed deployment model of *CloudSec₊₊* has key advantages that enable easy and high performance protection for the IaaS platforms: *first*, the deployment model of *CloudSec₊₊* makes the external security provisioning approach much easier to deploy across a wide variety of user operating systems. This is because *CloudSec₊₊* has the ability to protect different operating systems such as UNIX, Linux or Windows, by just adjusting the online monitoring code and DIGGER approach to monitor and protect the running operating system. *Second*, a major goal of malicious hackers is control, by which the hacker would have the ability to monitor, intercept, and modify the state of other software on a system. In our deployment model, *CloudSec₊₊* works at a hypervisor level, which means that the security virtual machine is deployed at a layer lower than the hosted virtual machine layers. This enables complete control over the hosted virtual machines including their virtual hardware, while making it difficult for an attacker to even detect the security software.

8.2.2.1.2 Experimental Results

To validate that *CloudSec₊₊* effectively overcomes the semantic gap and monitors kernel runtime objects, we compared the external view of mapping the physical memory to runtime objects using *CloudSec₊₊* with the internal view of the virtual machines using Windows debugging tools. Using DIGGER and OS-KDD, *CloudSec₊₊* was able to successfully uncover the correctly identified kernel runtime objects with a zero rate of false alarms and with a low per-

formance overhead. We compared 124 random data structures and 13 different object types with a 2-level information depth (based on the computed *type-graph*), and the results were identical in both views. Figure 8-4 shows the time consumed (in seconds) to extract object and data structures details with a 2-level information depth for some data structures.

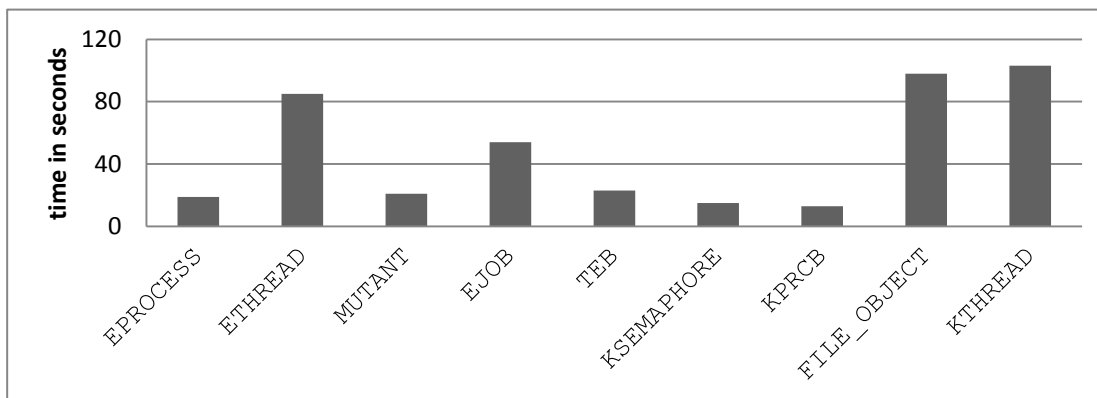


Figure 8-4. Object details extraction normalized time in seconds.

8.3 Kernel Data Integrity Checking Tools

Operating systems implement a hierarchical object model to provide consistent and secure access to the various internal services implemented in the operating system’s kernel [236]. Despite the benefit of easier management of runtime objects using this model, this model makes kernel runtime objects attractive targets for malicious hackers who want to take control of the kernel, as discussed previously in chapter 2. The general definition of an object is a running instance of a data structure encapsulated as an object. However, as discussed in chapter 7 – based on WI-note – we do not consider any running instance of a data structure to be an object. For instance, in the Windows operating system only few dozens of kernel data structures are encapsulated as kernel objects such as processes, threads and tokens, as shown in Figure 8-5.

Kernel objects are typically created either by the system user using the kernel native APIs or by the kernel itself to support a kernel-specific tasks. For example, to create a process a user application calls the `CreateProcess` routine, implemented in `Kernelbase.dll`. Then, after some validation and initialization by the kernel mode, `CreateProcessW` calls the native Windows service `NtCreateProcess` to create a process object [236]. A traditional approach

to subvert the kernel runtime objects is by modifying the text/code of the kernel native routines such as `CreateProcess`, `CreateProcessW` and `NtCreateProcess`. However, such approach can be easily detected as kernel code is always static and does not change at runtime.

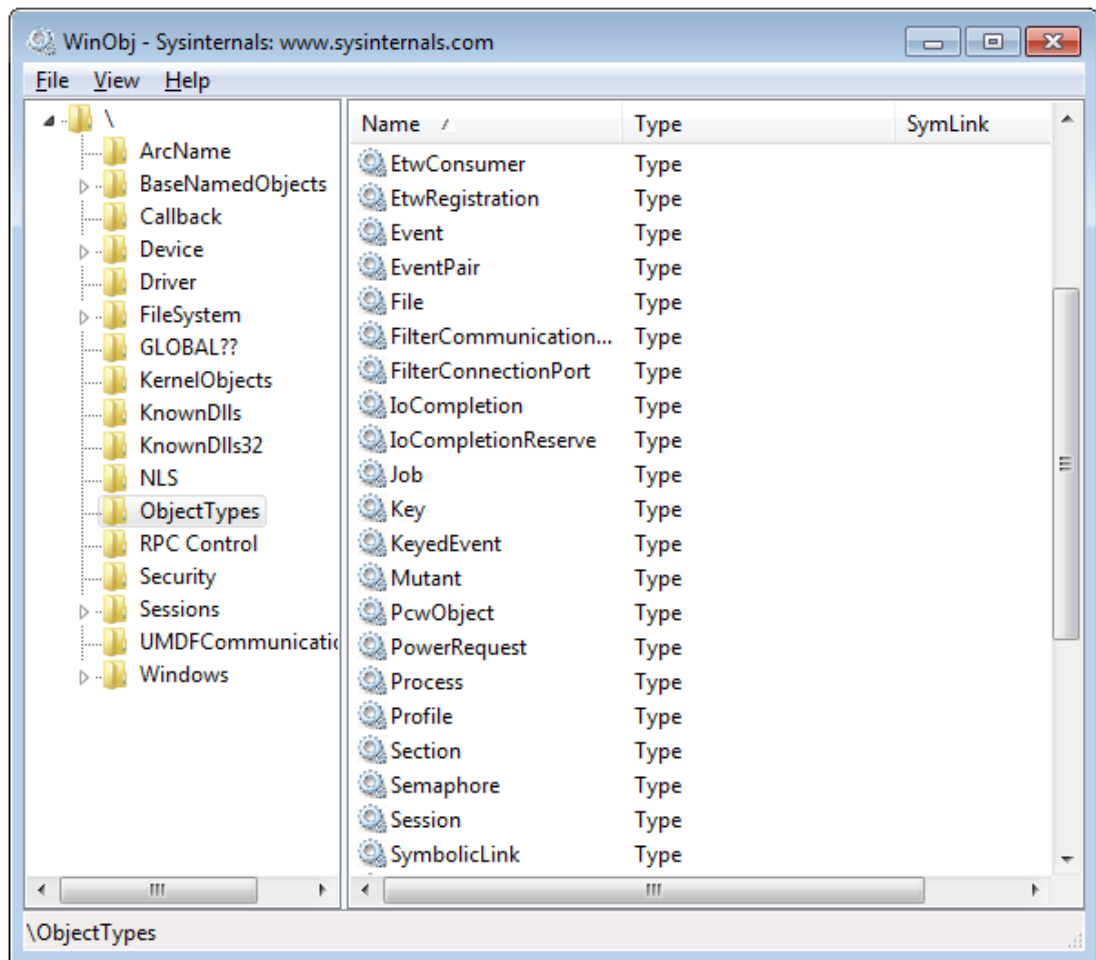


Figure 8-5. A snapshot from WinObj of the different object types supported by Windows OS.

Instead, kernel rootkits stealthily modify the non-control data structures (that change in location, value and number of running instances) by manipulating the pointers of these data structures at system runtime. Protecting the running instances of the non-control data structures is challenging because of: (i) the changeable nature of non-control data structures that makes the process of formulating a set of integrity constraints to check their integrity at runtime impractical. In other words, the contents of kernel dynamic data is not predictable and depends on the calling contexts at runtime, and thus detecting malformed objects and pointers is not an easy task. (ii) The huge number of non-control data structures that exist in the kernel address space

makes kernel dynamic data a rich target for hackers to subvert the kernel, especially with the significant existence of generic pointers. This makes the process of protecting them in near real-time impractical. Kernel data integrity checking likely have a large performance overhead, thus a kernel data integrity checker will have to operate periodically not in a near real-time fashion. However, periodically-based monitoring might miss catching short-lived objects such as token and keys. (iii) Finally, kernel runtime objects can be made invisible with no explicit integrity constraints that can be extracted from the source code of the operating system or even the running contexts that could make the running operating system detect that there is a hidden object or a malicious behaviour.

Based on the above discussion, we introduce in this section a set of kernel data integrity checking runtime tools that enable detecting various kinds of pointer manipulation malware that target kernel dynamic and static data. These tools have the ability to check the integrity of kernel dynamic data at runtime for the following types of malware: hiding runtime kernel objects, function pointer hooking and dangling pointers, details are discussed below in section 8.3.1, 8.3.2 and 8.3.3, respectively. In addition, we introduce an offline memory forensics tool that analyses the physical memory for evidence of rootkit infections for further forensics investigations.

CloudSec++ is not just limited to support our developed security tools. Many other security tools could be implemented in *CloudSec++* using DIGGER and OS-KDD. Details of this will be discussed in the future work section in chapter 9. In summary, with the help of a static analysis tool that can analyse kernel code with respect to its control and data flow, calling contexts and also able to disambiguate `void` and `null` pointers, and casting operations that take place at runtime; many operating system security tools can be developed to defend against zero-day threats that could target kernel code or data.

8.3.1 Hidden Dynamic Objects Detection Tool

8.3.1.1 Hidden Objects Problem Overview

Hiding kernel runtime objects – known also as Direct Kernel Object Manipulation (DKOM). DKOM is a well-known technique used to place a stealthy kernel rootkit in a victim operating system by hiding its presence from the system user and traditional security tools. Kernel hidden objects are a problem related to the number of running instances of a kernel object type (*i.e.* a specific data structure). As discussed before, this number changes at system runtime according to the different calling contexts of the system user which means that no straight-forward integrity constraints can be enforced on such changing number of running instances.

The main idea behind DKOM rootkits is manipulating the generic pointers located in kernel dynamic data, in order to point to somewhere else other than the memory addresses that these pointers are supposed to dereference. For better understanding of the object hiding problem in operating systems, we discuss a practical example of hiding a system process and how this process can remain invisible to system users and security software. Operating systems such as Windows and Linux keep track of runtime objects with the help of linked-lists that are circularly linked. A major problem with these lists is the use of null pointers. This makes it easy to unlink a running object by manipulating pointers and thus the object becomes invisible to the kernel and to security tools that depend on kernel APIs or memory *e.g.* HookFinder [106] or memory traversal *e.g.* and OSck [1]. Figure 8-6 depicts an example of 3 running processes structured in the `ActiveProcessLinks` doubly-linked list, in a Windows operating system. The `FLINK` of the doubly-linked list points to the next entry in the list, while `BLINK` points to the previous entry in the list. In addition, the `FLINK` of the last entry in the list points to the list head, as each doubly-linked list starts with a list head. A DKOM rootkit can simply change the `FLINK` and `BLINK` Fields of process 1 to point to process 3 instead of process 2, by manipulating the pointers as shown in Figure 8-7. Such manipulation detaches process 2 from the `ActiveProcessLinks` structure. However, process 2 is still running in the operating system memory and scheduled by the processor, because scheduling in the Windows kernel is

thread-based rather than process-based. Such modifications to kernel dynamic data violate integrity constraints that cannot be built from operating system’s source code directly. This is because data structure syntax is controlled by the operating system code while their semantic meaning is controlled by the runtime calling contexts – *i.e.* number of running instances and memory locations. Consequently, exploiting dynamic kernel dynamic data structures will not make the operating system treat the exploited structure as an invalid instance of a given type, or even detect hidden or malicious objects.

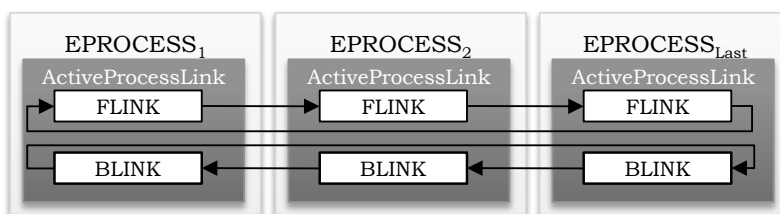


Figure 8-6. Windows operating system processes structured in a doubly linked list.

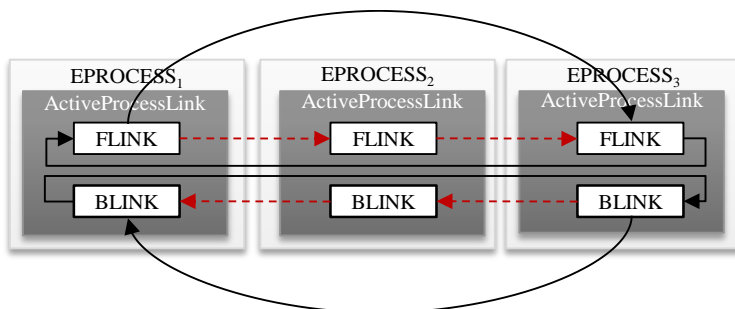


Figure 8-7. DKOM mechanism for hiding runtime objects.

8.3.1.2 D-Hide: A Hidden Objects Detection Tool

Based on the above problem we have developed a security tool called D-Hide that has the ability to systematically uncover all kinds of stealthy malware – not just limited to a specific object type, as done by most research efforts to date [18, 243]¹⁹. D-Hide detects the presence of hidden objects by scanning the physical memory directly and without relying on the operating system kernel trustworthiness. D-Hide is implemented using the C programming language and is deployed as an online security analysis tool in the security module of the *security virtual machine*.

¹⁹ Details are discussed in chapter 3.

D-Hide starts working after overcoming the semantic gap and building the security profiles of each virtual machine, as discussed in section 8.2. At this stage, D-Hide has an external view of the virtual machines' kernel data that was built based on DIGGER's approach and OS-KDD generated *type-graphs*. In D-Hide, we employ the cross-view comparison approach to detect runtime objects anomalies [93]. However, in order to detect the hidden objects using the cross-view comparison approach, we need to get the internal view of the virtual machines to compare it with the external view and extract discrepancies to be marked as malicious hidden objects. Such an internal view is not applicable in our threat model, as the security virtual machine does not have any access rights to the operating systems running in the hosted virtual machines. Our solution for this problem is getting such internal view "externally".

Windows internal tools depend on the kernel APIs to extract system runtime objects. These APIs depend on a similar approach to the traditional memory traversal techniques. Kernel APIs simply traverse the memory by following pointer dereferencing as what happens in the traditional memory traversal techniques. Based on that, we use the *type-graph* of the corresponding kernel version to traverse the memory externally from the *security virtual machine* of *CloudSec₊₊*. This allows us to get the other internal view of the comparison "externally". This is considered to be the internal view and we call it "ex view", and the view of DIGGER is the external view against which it will be compared. Discrepancies in this comparison reveal hidden kernel objects, and the prevention actions are taken as follow: *first*, if the target virtual machine is not a certified virtual machine, the pointers' values will be modified to make the hidden object visible again by linking the object to the corresponding doubly-linked list. This will make the hidden object visible to the security software installed inside the operating system and system users. No further action will be taken if the hidden object is a user-mode object like processes and threads, and does not target to modify the kernel memory or the pointers' values that reside in the kernel memory. Otherwise, an urgent alert will be issued to the consumer. *Second*, if the target virtual machine is a certified one then the detected hidden process will be destroyed from the memory by zeroing its memory address space and unlinking all of its pointers to the other running objects. A security alert will then be issued to the virtual machine owner with the threat

details and the prevention action taken. The security alert contains the threat detection time, threat type, the infected memory regions, protection action and the current security status of the virtual machine.

We implemented a memory traversal add-on for the *CloudSec++* security module that takes a *type-graph* as input and based on that graph, it traverses the kernel address space. We evaluated D-Hide's ability to identify hidden objects with four real-world kernel rootkit samples: FURootkit, FuToRootkit, AFX Rootkit and HideToolz, and three simple self-developed proof-of-concept rootkits that hide threads, files, mutants and events based on the DKOM technique. We infected a hosted virtual machine running Windows XP SP3 operating system with these rootkits to hide a number of processes, threads, files, mutants and events. To evaluate DIGGER's efficiency in detecting hidden objects, we used Windbg and Poolmon tools to check the validity of "ex view" and DIGGER's view, as follows:

- *First*, Windbg depends on the operating system's kernel APIs and thus its view should be similar to our "ex view". We compared the two results for a small sample of 7 different object types and 20 running instances of each object. The results were identical meaning that our add-on is correctly implemented.
- *Second*, Poolmon displays allocation-driven data about memory allocations and deallocations from the paged and nonpaged pools. Thus Poolmon's results should be similar to DIGGER's view. We confirmed that again by comparing 7 different object types and 20 running instance of each object. The results were identical meaning that DIGGER is correctly implemented.
- *Finally*, we performed a cross-view comparison between DIGGER's view and "ex view" and we found that all the hidden objects were correctly detected.

Table 8-1 summarizes our evaluation results. The DIGGER view, Poolmon, ex view and WinDbg columns show the number of running instances of each object type revealed from these views and tools. The results conforms that DIGGER's view and Poolmon are identical, the same for "ex view" and Windbg. The hidden objects column reflects the cross-view comparison results that show the number of hidden runtime objects for each object type. Name and address

columns shows the details of the detected hidden objects. The “Man. fields” column shows the fields that were manipulated to hide the object. The “Next Obj Address” and “Previous Obj Address” fields show the objects that their fields were manipulated to hide the objects – *i.e.* if the hidden object was obj_n , so these two columns reflects the addresses of obj_{n-1} and obj_{n+1} , respectively. Finally the false positive column shows the false positive rate of D-Hide.

D-Hide correctly identified all hidden objects with zero false alarms. D-Hide has two key characteristics that distinguish it from other hidden objects detection tools: (i) its ability to apply the cross-view comparison approach without the need for any internal tools *e.g.* task manager or Windbg that gets the internal view, as done in the current cross-view research [93, 134]. This feature enables deploying D-Hide to monitor and protect hosted virtual machines in the IaaS platform, where the cloud providers do not have any access rights to the hosted virtual machine. (ii) D-Hide is unlike previous tools [128, 135] that rely on the tool authors’ knowledge of operating system’s kernel runtime data layout and thus focus on detecting specific types of hidden objects by hardcoding security expert knowledge of the kernel data layout [20]. D-Hide is not limited to specific objects and can detect stealthy hidden runtime objects of any type. This feature is greatly supported by the generated type-graphs of OS-KDD that provides accurate and systematic mapping of the complex kernel data layout.

Table 8-1. D-Hide evaluation results.

Object Type	Pool Tag	DIGGER View	PoolMon	Ex view	Windbg	Hidden Objects	Hidden Objects		Man. Fields	Next Obj Address	Previous Obj Address	False Positive
							Name/ID	Address				
Process	Proc	144	144	140	140	4	winRAR.exe	0x9AA4590	FLink Blink	0x9ABDDA0	0x9A89788	0
							ALsvc.exe	0xa4ea720	FLink Blink	0xA6CF5B0	0xA4ea720	
							ipmtlg.exe	0x9A2F808	FLink Blink	0x9A49CF8	0x9A2E030	
							IEMonitor.exe	0x989EBC0	FLink Blink	0x97E6030	0x98B3030	
Thread	Thre	2576	2576	2571	2571	5	520	0x95F25E0	FLink Blink	0x95F8D40	0x95F5030	0
							821	0x9909030	FLink Blink	0x9911748	0x9905c98	
							191	0x90B9030	FLink Blink	0x90BA030	0x90B8970	
							837	0x9948030	FLink Blink	0x994B3C8	0x9947030	
							312	0x932CDA8	FLink Blink	0x9332030	0x9315030	

Object Type	Pool Tag	DIGGER View	PoolMon	Ex view	WinDbg	Hidden Objects	Hidden Object		Man. Fields	Next Obj Address	Previous Obj Address	False Positive
							Name/ID	Address				
File	File	9992	9992	9988	9988	4	6510	0x15EA8978	FLink Blink	0x15FF85C8	0x14D80038	0
							1722	0x9235B68	FLink Blink	0x923D538	0x92323D8	
							3212	0x97DF320	FLink Blink	0x97E4038	0x97D2108	
							6472	0x1578EDD0	FLink Blink	0x14B24038	0x156C4C90	
Mutant	Muta	470	470	467	467	2	209	0x9BF1510	FLink Blink	0x9BCFB40	0xA429F2F8	0
							121	0x9885580	FLink Blink	0x9888B00	0x9885CE0	
Event	Even	6708	6708	6705	6705	3	6341	0x53CCDDAC	FLink Blink	0x4519D824	0x53D8FCA4	0
							6333	0x53BEE4AC	FLink Blink	0x5383E1CC	0x53BEE7E4	
							219	0x9141264	FLink Blink	0x7BB3E244	0x8F85BD4	

8.3.2 Function Pointer Hooking Detection Tool

8.3.2.1 Function Pointers Problem Overview

The presence of NX-bit protection technologies and similar kernel code protection technologies make it is hard for rootkits to inject stealthy code into the kernel address space. NX-bit technology enforces a $W \oplus X$ property on the kernel memory pages that contains kernel code. The $W \oplus X$ property states that a given memory page can be either writable or executable, but not both at the same time [244]. As kernel code does not change at runtime and remains all the time similar to the desk copy, so the $W \oplus X$ property works well to prevent direct kernel code modification. Consequently, recent rootkits tends to execute their backdoors in the kernel address space without modifying the kernel text. This is done by modifying function pointers, instead of the function text itself, that are located in the kernel dynamic runtime objects such as loaded modules, import and export address tables [245]. A function pointer is a type of pointers, where instead of pointing to a data type, the pointer points to the entry point of a routine. Thus, by modifying a function pointer to point to malicious code located in another memory address, a hacker can execute this arbitrary code [109] instead of the intended benign code. In particular, when a function pointer is dereferenced, this pointer is be used to invoke the target function it points to and pass it arguments just like a normal function call, as demonstrated by Bush [246]. Function pointers exist frequently in operating systems' kernels and this provides a wide attack surface for hackers to exploit kernels indirectly. In order to defend against function pointer hooking, we need to know where the malicious code could be placed. Generally there are only two locations where a backdoor can be placed: *user address space* and *kernel address space*.

User Address Space. In case of placing the malicious code into the user mode address space; the code will affect only the address space of a specific running instance of an object type. Examples of such rootkits are IAT and EAT hooking of Windows operating systems, as shown in Figure 8-8. Defending against such local rootkits is beyond our virtualization-aware security solution tasks, as it is the responsibility of the cloud consumer.

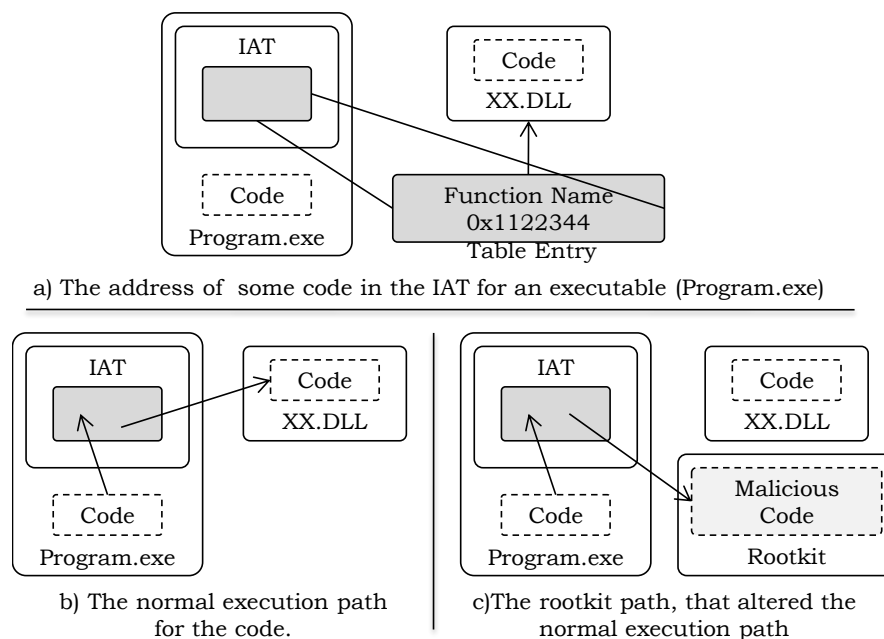


Figure 8-8. IAT and EAT function pointer hooking.

Kernel Address Space. In the case of placing the malicious code in the kernel-mode, there are two main locations: (i) *Kernel text pages.* Placing code in kernel text pages is a naïve approach that can be detected easily. Kernel text pages do not change at runtime and thus a $W \oplus X$ property can sufficiently protect kernel text pages. Another approach for changing kernel text pages is modifying the function pointers of the system call table and replaces a function entry with the address of a malicious code, as shown in Figure 8-9. Such approach is also easy to be detectable as memory addresses of the system call table do not change at runtime. (ii) *Memory pools.* Memory pools are used to allocate kernel dynamic objects. These pages are readable and writable at the same time and thus there are no integrity constraints that enable differentiating the memory addresses that point to functions entries or kernel objects, in order to detect function pointer hooks. Therefore, we focus on pool memory pages to check for function pointers hooking. Figure 8-11 shows a function pointer example in one of the non-control data structures that are usually allocated in pool memory in Windows operating system.

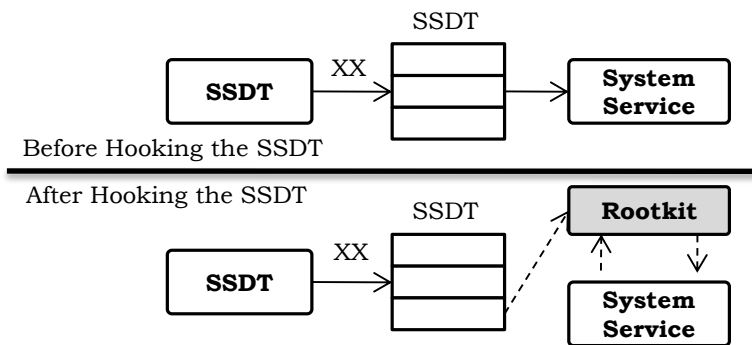


Figure 8-9. Hooking SSDT system call table of Windows operating system.

```

typedef struct BOOT_DRIVER_LIST_ENTRY {
    LIST_ENTRY Link;
    UNICODE_STRING FilePath;
    UNICODE_STRING RegistryPath;
    PKLDR_DATA_TABLE_ENTRY LdrEntry;
} BOOT_DRIVER_LIST_ENTRY, *PBOOT_DRIVER_LIST_ENTRY;

PBOOT_DRIVER_LIST_ENTRY DriverEntry

NTSTATUS DriverEntry(IN PDRIVER_OBJECT DriverObject, ..... )
{
    .....
    PEPROCESS Process = NULL;
    HANDLE ProcessHandle = NULL;
    .....
}
    
```

Figure 8-10. A function pointer example in Windows operating system.

8.3.2.2 FP-Protect: A Function Pointer Hooking Detection Tool

In order to detect and defend against function pointers hooking, we developed a runtime security analysis tool named FP-Protect. FP-Protect is written in C and deployed in the security virtual machine of *CloudSec++*. FP-protect works directly on the introspected hardware bytes after the semantic gap module finishes constructing the internal view, externally. FP-Protect first extracts all function pointers that are located in kernel runtime objects using the generated *type-graphs* of OS-KDD. A *type-graph* contains all the pointer details of the kernel address space (code and data), as discussed previously in chapter 6. FP-Protect focuses on functions’ pointers located in the non-paged pool memory only, in other words we check for function pointers that are located

in the dynamically allocated kernel runtime objects.

Protecting a function pointer located in the address space of the static kernel objects – *e.g.* system call table, IDT and GDT – is straightforward, as the dereferencing information of function pointers are fixed all the time during system runtime. The challenge is check the integrity of function pointers that are located in dynamic kernel objects. To check the integrity of these function pointers, *FP-Protect* checks the dereferencing details of the predefined function pointers of these objects. Any function pointer that dereferences a memory address that resides outside kernel code text pages is considered to be a function pointer hook. For each uncovered object, *FP-Protect* uses the corresponding *type-graph* to compute the memory addresses of the function pointers based on their relative addresses in the *type-graph*, as shown in Figure 8-11. The offsets of a field indicate the distance from the base address (object start address) to form the relative addresses. From the relative addresses (base address + offset), we can compute the absolute addresses (the actual address of a memory location) based on the located object physical address.

We built a small test environment to evaluate *FP-Hook*. We performed experiments on a virtual machine running Windows XP SP3 on a 2.8 GHz CPU with 2GB RAM. Figure 8-12 shows the detected function pointers for different object types at runtime in different time slots, where each time slot is 120 seconds. At a random runtime time t , we first scanned the memory of the virtual machine to locate function pointers in the running instances of process, driver, thread, file, mutant and port object types. Then after every 120 seconds we repeated the process twice. From our scan analysis, we found that: (i) most of function pointers are located in the running instances of device drivers and file object types. (ii) We did not find any function pointers in mutants. (iii) The number of detected function pointers changes over time based on the use of the system user (calling contexts).

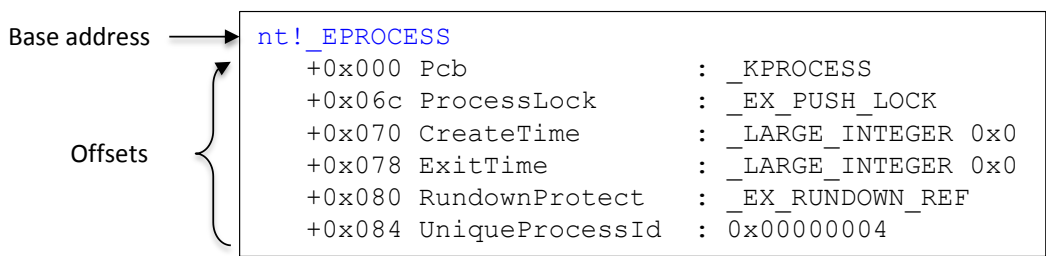


Figure 8-11. Absolute and relative memory addresses

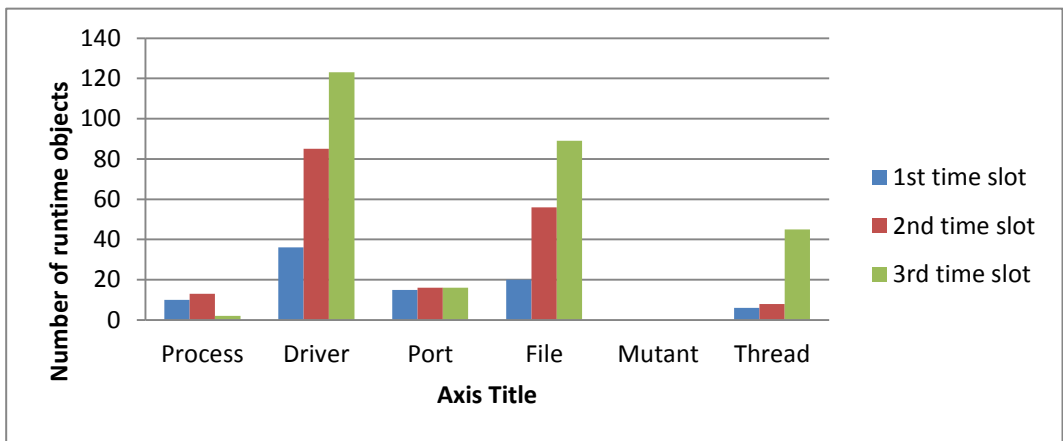


Figure 8-12. Function pointers age at runtime.

The average runtime in seconds to detect function pointers in each object type is shown in Figure 8-13. FP-Hook has a relatively low performance. To evaluate the accuracy of FP-Hook’s results, we manually checked the detected pointers at runtime and the false positive rate is as shown in Figure 8-14. The false alarm rate was in the drivers of around %0.04. This rate indicates that pointers were wrongly identified as function pointers. We think that false positives exist because of the un-deallocated memory objects that were discussed in chapter 7. Further investigations into the reasons of false alarms will allow us to end or lower the rate and this is a part of our future work.

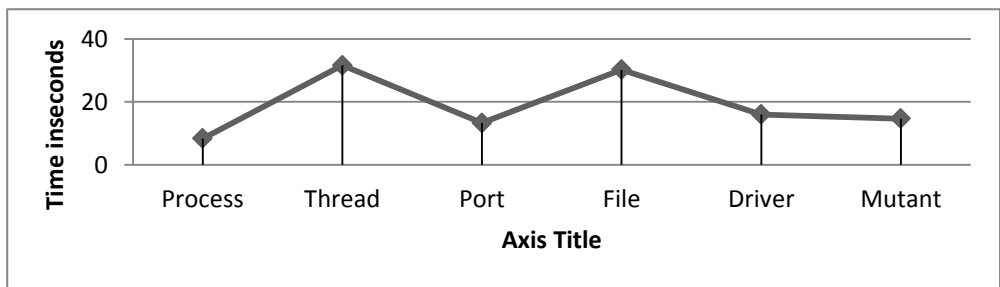


Figure 8-13. Average time to locate function pointers in different object types.

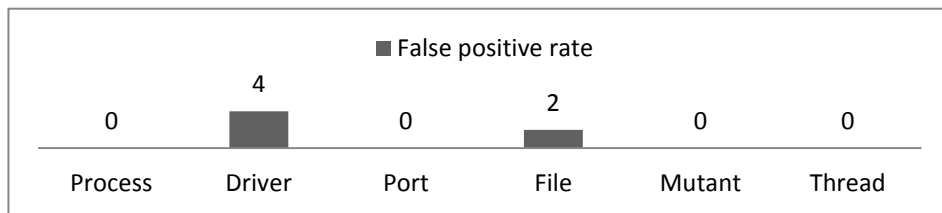


Figure 8-14. False alarm rates of locating function' pointers.

After memory locations of these function pointers are computed, we then detect function pointers' manipulations. In order to detect malicious function pointers' manipulations, we treat the problem as an access control problem, where the dereferenced address should point to a relative address within the object address space in the non-paged pool memory. Most of function pointer's hooking rootkits were not available for download, so we had to develop a few rootkit samples that modify function pointers in system device drivers of the Windows operating system. The developed rootkits are based on previously reported techniques [247, 248] that enables function hooking in operating system kernels. We focused on device drivers because they are usually allocated to enable a user-mode applications getting root access to the kernel memory. We developed two rootkits:

- *SimpleHook1*. SimpleHook1 is a simple proof-of-concept rootkit that overwrite the function pointer of the `IOCTL_dispatch_routine` to point to a malicious code that allocates a free memory pool block to allocate a new created process. The malicious code address is located in the paged pool memory.
- *SimpleHook2*. SimpleHook2 is also a simple proof-of-concept rootkit that overwrite the function pointer of `DriverEntry` to hide a running process by using kernel object manipulation techniques to unlink a process from its doubly linked list. The malicious code address is located in the paged pool memory.

The developed rootkits had some runtime errors that caused a system crash in nearly half of their runs. However as the rootkit development is out-of-scope we accepted the successful runs rate to infect the kernel and then use FP-Protect to detect the manipulated function pointers. Further investigations in the technical development phases of the rootkits in order to get an-error

free environment is part of our future work. FP-Protect successfully detected the hooked function pointers and then dispatched the manipulated pointers to their original dereferencing address based on the corresponding *type-graph*. Table 8-2 shows our evaluation results of FP-Hook to detect function pointer manipulations in `IOCTL_dispatch_routine` and `DriverEntry` routines.

Table 8-2. FP-Hook evaluation results of detecting function pointers.

Object Address	Driver Name	Function Pointer Offset	Function Pointer	Pointer type	Dereferenced Original Function	Dereferenced Malicious Function	Hook detected ?	Prevention action
0x821A4500	Fips	0x038	MajorFunction	*PDRIVER_DISPATCH	IOCTL_dispatch_routine	SimpleHook1	YES	Overwrite function pointer
0x820C2550	HTTP	0x038	MajorFunction	*PDRIVER_DISPATCH	IOCTL_dispatch_routine	SimpleHook1	YES	Overwrite function pointer
0x82201B10	RasAcad	0x02c	DriverInit	*PDRIVER_INITIALIZE	DriverEntry	SimpleHook2	YES	Overwrite function pointer
0x81ED3270	Cdfs	0x02c	DriverInit	*PDRIVER_INITIALIZE	DriverEntry	SimpleHook2	YES	Overwrite function pointer

8.3.3 Dangling Pointers Detection Tool

8.3.3.1 Dangling Pointers Problem Overview

Dangling pointers are pointers that do not point to a valid object type, and thus they can lead to memory bugs such as use-after-free and double-free vulnerabilities [249, 250]. Dangling pointers mainly arise when an object is deleted or deallocated, without modifying its pointers, values to null and linking it to the pool or heap free memory slots, so that it still has a pointer still points to the memory location of the deallocated memory [251], as shown in Figure 8-15. In particular, a dangling pointer is created when the object is deallocated and there is still a *points-to* relation from another object to that deallocated memory. This object may later reallocate or overwrite the deallocated memory causing a dangling pointer.

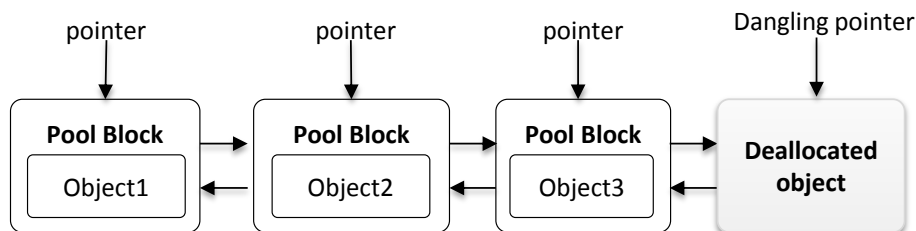


Figure 8-15. Dangling pointers example.

Dangling pointers allows a malicious code to run in kernel's memory in a stealthy way, by working under the umbrella of another legally allocated object. Identifying dangling pointers is time-consuming task as this requires two main analysis phases [143, 252-254]: detecting the creation of the dangling pointers and detecting the memory bugs caused a the dangling pointer.

First, analysing the program to detect the creation of dangling pointers. Dangling pointers are usually created by the operating system itself and its running applications. Operating systems like Windows and Linux sometimes do not clear the contents of a memory slot after its deallocation to avoid the overhead of writing zeroes to the physical memory and thus does not affect system performance. Such performance-related issue is a main reason for the creation of dangling pointers. In chapter 7, we discussed the problem of the dead DAMOs that are likely to

exist because of operating systems runtime bugs. Hackers can re-use DAMOs to reallocate an undeallocated memory block to another stealthy object or stealthy code. This makes the original program (that has allocated the memory block) to dereference the new object/code (that has been stealthy allocated) causing a dangling pointer. Thus, a memory violation could occur and the new object will be running in the kernel memory to execute its objectives. Another reason for dangling pointers creation is the shared objects. As discussed in chapter 2, kernel data structures are implementation-dependent where a pointer deposited in a field under one object can be accessed from a different field under another object. Thus multiple objects could have different pointer relations to a single running object. If this running object is deallocated from one of those pointing objects without updating the pointer count value of the other object because of a runtime error, the other pointing objects (that share pointers to that deallocated object) still consider it as a live object and perform dereferencing operations. This is unlikely to occur however it could happen because of an implementation bug in a program. Moreover, dangling pointers could be created by a malicious hacker that targets to manipulate the pointers of a running object by unlinking it from its parent object (simulate deallocation) while its memory block is still allocated in the memory.

Second, analysing the runtime memory to detect any memory bugs caused by the created dangling pointers. In particular, analyse the program to identify the dereferencing instructions of the created dangling pointer at system runtime to detect any malicious behaviour such as executing code and modifying runtime kernel data. Such a step requires an accurate context-sensitive *points-to* analysis to be performed, not just on the running operating system kernel but also on all the running applications that might allocate objects such as device drivers. Such process should be done online at system runtime, not statically, as dangling pointers cannot be defined at the static analysis phase in OS-KDD.

8.3.3.2 *D-Pointer: Dangling Pointers Detection Tool*

In other to address dangling pointers problem, we propose a new mechanism based on DIGGER's approach that has the ability to locate dangling pointers whatever was the reason of

their creations. We developed a tool based on DIGGER's approach called D-Pointer. D-Pointer is a fast mechanism that has the ability to prevent dangling pointers from occurrence in operating systems' kernels. D-Pointer does not have two analysis tasks like the current approaches, as discussed above. D-Pointer has a single analysis phase at runtime that enables detecting any deallocated object that still exist in the kernel memory and still has some live pointers dereferencing it. D-Pointer shifts the focus from the creation of the dangling pointer to the prevention of these dangling pointers and its consequences bugs from taking place before even they are utilized by any hacker. D-Pointer protects the kernel address space from such memory safety violations by ensuring that: *first*, each deallocated runtime kernel object has no live pointer relations with any other live object. To achieve this, we made use of the object header data structure, discussed before in chapter 7. As shown in Figure 8-16, `OBJECT_HEADER` data structure has a `PointerCount` and `HandleCount` fields. `HandleCount` is used as discussed before in chapter 7 to detect the un-deallocated memory objects. `PointerCount` is then used to detect if an un-deallocated memory object still has pointers from other running objects that can dereference it. In particular, `PointerCount` field maintains the number of references to a running object. An object gets deallocated when its `PointerCount` becomes zero. From our analysis to different memory images using WinDbg, we found out that it is likely to find an object with `HandleCount` equals to zero and `PointerCount` equals to 1, as shown in Figure 8-17. *Second*, the detected deallocated objects that still have a `PointerCount` greater than zero are then efficiently freed from the memory and linked to the pool list that holds the free pool blocks. We do this by following the approach discussed by Tarjei [245]. Tarjei discussed a mechanism used by malicious hackers to unlink and link memory pool blocks from the free pool blocks list. We use the same mechanism to fix dangling pointers by linking its deallocated object to the free pool blocks linked list.

At runtime, whenever an object is uncovered, D-Pointer checks for the dangling pointers and deallocate then using the technique described above. To evaluate our approach we ran D-Pointer to monitor a running virtual machine running Windows XP SP3 on a 2.8 GHz CPU with 2GB RAM. We did not use any rootkits; D-Pointer was looking only for the internally

caused dangling pointers by the operating system runtime errors. Table 8-3 shows the detected dangling pointers in a running operating system for different object types. Table 8-3 summarized the number of the running instances of each object type, the detected dangling pointers, the addresses of these dangling pointers and the `PointerCount` field value of the infected objects. By manually analysing the infected objects that contain dangling pointers, we found that they are DMAOs for previously allocated objects and their `PointerCount` fields are still active and pointing to another object (the field value is greater than zero).

```
typedef struct _OBJECT_HEADER {
    LONG_PTR PointerCount;
    union {
        LONG_PTR HandleCount;
        PVOID NextToFree;
    };
    POBJECT_TYPE Type;
    .....
} OBJECT_HEADER, *POBJECT_HEADER;
```

Figure 8-16. A snapshot of the object header structure.

```
kd> dt _OBJECT_HEADER 8966BB8
nt!_OBJECT_HEADER
+0x000 PointerCount      : 1
+0x004 HandleCount      : 0
+0x004 NextToFree       : 0x00000001
+0x008 Type              : 0x812b5730 _OBJECT_TYPE
+0x00c NameInfoOffset   : 0
+0x00d HandleInfoOffset : 0
+0x00e QuotaInfoOffset  : 0
```

Figure 8-17. A snapshot of Windbg reflecting our analysis.

Figure 8-18 reflects the time consumed in seconds to analyse the different types of kernel runtime objects to locate dangling pointers in them. D-Pointer has a low performance overhead and thus it supports near real-time protection against dangling pointers that could be created in kernel dynamic data.

Table 8-3. D-Pointer evaluation results

Object Type	Running Instances	Dangling Pointers	Percentage	Object Address	PointerCount	HandleCount
Process	125	2	0.016%	0x96E8180	1	0
				0x98A2960	1	0
Thread	2365	1	0.000%	0x9607618	1	0
Port	365	0	0.000%	NA	NA	NA
File	5820	0	0.000%	NA	NA	NA
Profile	685	0	0.000%	NA	NA	NA
Job	452	0	0.000%	NA	NA	NA
Session	178	0	0.016%	NA	NA	NA

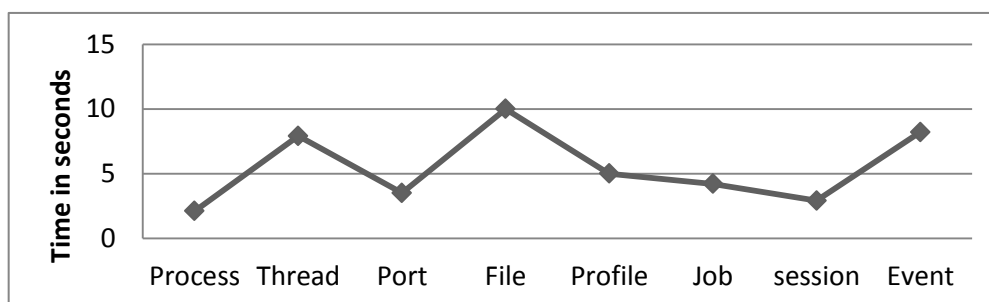


Figure 8-18. Dangling pointers detection time in seconds.

8.3.4 Memory Forensics

Analysing physical memory for rootkit infection evidence is a famous approach used by memory forensics experts to understand rootkits' behaviours. Brute force scanning is one of the approaches that enable such memory forensics analysis. Brute force scanning is an offline analysis approach of kernel memory images in order to uncover semantic information of interest for a specific rootkit or an object.

D-Hide, FP-Protect and D-Pointer are all online analysis tools and their main objective is detecting system rootkits in a near real-time fashion, rather than going in-depth in the details of

the detected rootkits and their effects on the underlying operating system. This is to avoid high performance overhead of the running security software. Brute force scanning is on the contrast enables getting detailed information about an infected runtime object. Implementing a brute force scanning tool is straightforward with the help of DIGGER's approach and OS-KDD generated *type-graphs*. Given a memory dump or a snapshot, and a signature for an infected object, our brute force scanning tool helps retrieving in-depth information about the running state of the infected objects. In particular, it builds detailed semantic information about the infected object including any *points-to* relations with the other benignant objects that have dereferenced the infected object at a specific point at system runtime.

We developed B-Force, as a proof-of-concept for our brute force scanning approach. B-Force works on snapshots or memory dumps. Memory snapshots can be easily created using the ESX hypervisor and its vSphere client. B-Force takes a signature for an infected object; the signature is a combination of the object's address and its pool tag. B-Force then scans the memory snapshot through the *security virtual machine* of *CloudSec++* to retrieve in-depth details about the object using the corresponding *type-graph* of the running kernel. Based on the *type-graph*, B-Force builds a detailed *infection-graph* using all the *points-to* relations (described in the *type-graph*) of the target infected object with the other running objects. Such *infection-graph* can be used by security experts for an in-depth analysis about rootkits to understand their behaviours and manipulation side of the running operating system. A sample of the *infection-graph* is shown in Figure 8-19. B-Force results (the *infection-graph*) are very accurate, and this mainly comes from the accuracy of KDD generated *type-graph* and DIGGER's approach to locate kernel runtime objects in memory.

We have implemented *B-Force* in C and it is deployed in the *security virtual machine*. However it is not like the previous tool that works online on a running virtual machine. B-Force works offline on a memory snapshot of a virtual machine. B-Force can also be used to uncover the presence of rootkits by using the rootkit signature instead of our pool tagging schema. However, we are not fans of using such signature-based detection approaches, as many of kernel data structures cannot be covered with a value-invariant schema, as discussed before.

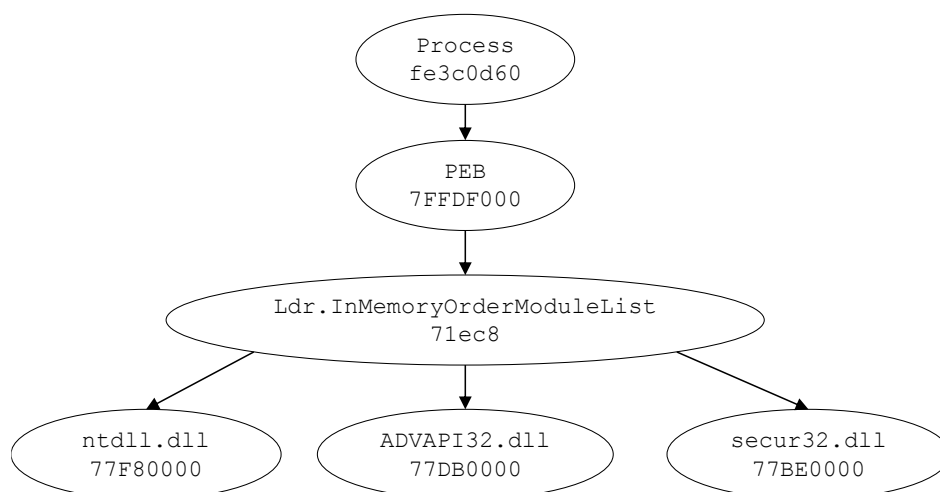


Figure 8-19. A sample infection-graph.

To evaluate B-Force efficiency, we used B-Force to analyse DMAOs to provide detailed forensic information about a malicious hacker's activity. Imagine a malicious hacker runs stealthy malware and then terminates it on a victim operating system. After the termination there may still exist for a non-trivial period of time some forensic data of interest in the memory pages of the deallocated objects. To demonstrate B-Force's efficiency in analysing DMAOs, we used some benchmark programs to run in three different memory snapshots, we then analysed the dead memory pages using B-Force to uncover some data of interest: user login information (GroupWise email client), chat sessions (Yahoo messenger), FTP sessions (FileZilla). We created 9 processes (three of these are the benchmark programs) and then performed some CPU-intensive operations using these processes. We then terminated these processes after 30 minutes, 10 minutes and 5 minutes in three different memory images – identified L, M and S, respectively. Then we created 4 different new processes 1 minute after termination. The memory images were then scanned using B-Force to generate *infection-graphs* for the terminated processes. We found that 3 from the terminated processes' physical addresses were overwritten by EPROCESS structure for new processes, while another three processes (from the terminated ones) still persisted in memory (at the same address in the memory).

We made the following observations. *First*, for the email client, we were not able to identify the login information (user name and password) for all of the memory images. For the ftp

client we were able to identify the server name and the server and client connection ports for the M image only, without any ability to locate the login credentials in all of the three images. For the chat benchmark application, we were able to locate the username and connection ports in the L image.

8.4 Summary

In this chapter, we introduced our proposed *virtualization-aware* security solution, *CloudSec₊₊*. In particular, we gathered our previously developed components and tools, discussed in chapters 5, 6 and 7, into a single security solution that has the ability to protect the hosted virtual machines in the IaaS platforms. *CloudSec₊₊* has the ability to systematically and effectively protect multiple operating system running instances in the same physical server against kernel data rootkits including zero-day threats. Key strengths of *CloudSec₊₊* include: (i) the systematic approach used to overcome the *semantic gap* problem and uncover kernel dynamic objects. (ii) The robust security approach used to protect the hosted virtual machines, where the security software is totally isolated from the running virtual machines. Such external security makes it harder for hackers to detect the installed security software and tamper with its behaviour. Moreover, external security provisioning model has the advantage of being easier to deploy across a wide variety of user operating systems. (iii) In *CloudSec₊₊*, cloud consumers enjoy the feature of getting feedback and enforce basic robust security polices of their hosted virtual machines *via* the cloud provider to ensure robust security for their hoisted virtual machines.

In this chapter, we also introduced a set of operating system kernel data integrity checking tools that have the ability to protect dynamic kernel data against zero-day threats that target to manipulate the generic pointers located in kernel dynamic data. The developed tools proved functional protection against various types of pointer manipulation threats such as hiding objects, dangling pointers and function pointer hooking. We also introduced a memory forensics tool that assists operating system's security experts to study and understand the behaviour of the newly detected zero-day threats on the running operating system.

Chapter 9

Conclusions and Future Work

In this thesis we reported our investigations into the *loss-of-control* security problem over hosted virtual machines in the IaaS platform, leading to the development of a robust security software system that can systematically and reliably protect their guest operating systems externally. In this chapter, we summarize our solutions to this research problem and outline key future work that can be done to extend and improve this work.

9.1 Key Conclusions

The main contribution of this research project is the development of a robust security solution to overcome the *loss-of-control* security problem of hosted virtual machines in the IaaS platform. In order to develop such a security solution, a number of new tools and mechanisms were introduced. Key strengths and limitations of these tools and mechanisms are discussed in the following sub-sections.

9.1.1 CloudSec

CloudSec enables reliable active and transparent monitoring for multiple concurrent hosted virtual machines, without placing any security code in their running operating systems. The key point behind implementing such a transparent active monitoring framework is moving the monitoring hooks from the virtual machine level to the hypervisor level by implementing our introspection framework on the ESX hypervisor using VMsafe libraries that facilitated achieving that. The hypervisor are the ideal location to host any security-critical code to ensure strict protection of the security solution. This is built on our assumption that the hypervisor is highly trusted and protected by the underlying hardware protection technologies, as discussed previously in chapter 5.

It is not just our *virtualization-aware* security solution that can benefit from *CloudSec*. Many other security software systems can benefit from *CloudSec* including virus scanners, kernel data and code integrity checkers, memory forensics tools, intrusion detection systems and firewalls. This is because nearly all of these security solutions share the same high-level operational flow that is supported by *CloudSec* – *i.e.* active, transparent and near real-time monitoring of the runtime operating system activities.

9.1.2 OS-KDD

While *CloudSec* was an effective IaaS security contribution, it suffered from manual effort to bridge the *semantic gap*. Based on such limitation, we introduced OS-KDD to enable systematic overcoming of the *semantic gap* problem, without a need to the expensive manual efforts or deep experience about an operating system’s kernel runtime data layout. OS-KDD can be considered the backbone of this research project. It solved most of the technical problems that relate to the complex implementation of the C-based operating systems *e.g.* Windows and Linux. OS-KDD enables systematic generation of an accurate *type-graph* that reflects statically the runtime data layout of an operating system’s kernel. The key point behind such systematic approach is performing field, flow and context-sensitive *points-to* analysis on the operating system’s kernel source code to disambiguate the generic pointers and get accurate estimation of their expected dereferencing operations at runtime, statically. OS-KDD scales to produce detailed and highly accurate *type-graphs* that solve the generic pointers problem of large-scale C programs, including C-based operating systems. This scalability and good performance was achieved by using abstract syntax trees as the basis for implementing our *points-to* analysis algorithm. The compact and syntax-free abstract syntax trees improves the time and memory usage efficiency of the analysis, as instrumenting abstract syntax trees is more efficient than instrumenting the machine code, as discussed before in chapter 6.

In summary, OS-KDD was a key solution for a number of technical problems in this research project, as follow: (i) OS-KDD enables systematic and accurate overcoming of the *semantic gap* problem for C-based operating system, without requiring deep experience about the

runtime kernel data layout or even the running kernel version. (ii) OS-KDD supports the implementation of DIGGER’s approach to enable computing a complete *object-graph* in a near-real time fashion that reflects the actual set of kernel dynamic objects at system runtime. (iii) OS-KDD is an essential factor to support performing systematic integrity checks on the generic pointers located in kernel dynamic objects. OS-KDD enables computing accurate constraint-sets on the pointers and pointer-compatible variables, during its analysis phases. These constraint-sets enable performing runtime integrity checks to detect memory errors and violations caused by manipulating the generic pointers located in kernel dynamic objects.

OS-KDD is not only beneficial for our research project; it can also be applied to many other security applications. Performing static analysis on the operating system’s kernel source code to extract robust type definitions for kernel data has several advantages that can be used in many other operating system security applications for both *virtualization-aware* and traditional *in-guest* security solutions. These advantages include: (i) OS-KDD supports implementing **systematic** operating system security solutions, without a need to deeply understand the complex kernel implementation details. Such feature gives the security software the maximum coverage to protect different operating system kernel versions without major modifications to the implementation of the security software itself. (ii) OS-KDD greatly minimizes the performance overhead of any dependant security software, as a major part of the analysis is performed offline. Static analysis saves a lot of time by instrumenting most of pointer dereferencing operations offline. (iii) Systematic static analysis maximizes the likelihood of detecting zero-day threats that could target kernel non-control data structures. Not like other researches that cover a fraction of kernel dynamic data to only cover around 28% of dynamic objects, OS-KDD enables uncovering and protecting all non-control data structures. This is because OS-KDD generates constraint-sets on the generic pointers located in these structures and thus enables detecting new rootkits that could exploit obscure data structures that have not been considered by current security solutions. (iv) *Type-Inference*. Declared data types of C pointers are unreliable indications of how the pointers are likely to be used at runtime, as C allows casting operations at system runtime with no restrictions. *Points-to* analysis is a powerful approach in type inference

to determine the actual runtime type(s) of an object statically, by analysing the usage of those pointers in the source code.

To the best of our knowledge, our *points-to* analysis algorithm is the first *points-to* analysis technique that depends on the abstract syntax trees to provide field, flow and context sensitive analysis. Buss *et al.* [191] had an initiative in performing *points-to* analysis based on the abstract syntax trees of the source code. However, their algorithm is field and context insensitive and does not scale to the large programs. To the best of our knowledge also, there is no similar research in the area of systematically solving the semantic gap except KOP [14]. KOP has an initiative in systematically computing a *type-graph* for kernel data; however KOP has a number of critical limitations, as discussed before in chapter 3.

9.1.3 DIGGER

DIGGER is a key to enabling *real-time* monitoring and protection for kernel runtime objects. DIGGER is a fast and robust object discovery approach that enables systemic discovery of kernel runtime objects from a trusted view. DIGGER is accurate and fast due to the utilization of a new *value-invariant* approach and an advanced *memory mapping* technique to work together in a single mechanism to deliver the required accuracy and high performance. The robust approach used in DIGGER to discover system runtime objects can be used in many other operating system applications that require locating system runtime objects such as kernel data integrity checkers, memory forensics tools, virtual machine introspection frameworks and network monitors. A key feature of DIGGER is the small and robust *value-invariant* signatures used to uncover the runtime objects. A key advantage of such signatures is that they are not tied to a specific data structure layout and are thus effective in different operating system kernel versions, where data structure layout change is likely to occur. Moreover, the very small size of the developed signatures significantly decreases the performance overhead of the security software.

DIGGER's implementation in this research project is limited to Windows operating systems. However, it is generic enough to be used in the different versions of the Windows operating system's kernels. The DIGGER's approach could be customized and implemented on

Linux and UNIX operating systems by utilizing the slab allocation mechanism of the Linux operating systems instead of the pool memory concept of Windows operating systems, as discussed previously in chapter 7. Recent research by Microsoft [255] was published after our DIGGER paper [133]. This research introduced a similar idea of using pool tags as a *value-invariant* approach to locate runtime objects. The research confirms our results and shows how effective our approach is in locating runtime objects from a robust source, as Microsoft has applied this approach to different operating system kernels.

9.1.4 CloudSec₊₊

Our final major component of this research project is *CloudSec₊₊*. *CloudSec₊₊* is capable of monitoring and protecting multiple concurrent virtual machines hosted at the same physical server and also supports migrating virtual machines across the hosting physical servers of an IaaS platform. The *CloudSec₊₊* architecture allows benefiting from the virtualization characteristics and runs in an isolated and controlled virtual machine as a security appliance to protect the other hosted virtual machines. The design of *CloudSec₊₊* has key advantages that enable efficient and high performance protection for the IaaS platforms. *First*, the deployment model of *CloudSec₊₊* makes the external security provisioning approach much easier to deploy across a wide variety of user operating systems. This is because *CloudSec₊₊* has the ability to protect different operating systems such as UNIX, Linux or Windows without relying on hard-coded offsets of a specific kernel version. *Second*, in our system design, *CloudSec₊₊* works at a hypervisor level, which means that the security virtual machine is deployed at a layer lower than the hosted virtual machines' layer. This enables complete control over the hosted virtual machines including their virtual hardware, while making it difficult for an attacker to even detect the security software. *Third*, *CloudSec₊₊* has the ability to perform real-time kernel integrity checks for operating system kernel dynamic data to defend against system zero-day threats that could target the non-control data structures. *CloudSec₊₊* employs a collection of security tools that enable detecting pointer manipulations across kernel dynamic data. We developed a number of security tools that are deployed in *CloudSec₊₊* to detect various operating system kernel data

attacks. This demonstrates the ability of our approach to provide effective external security support for virtualized operating systems in the IaaS cloud model. We developed four main tools: (i) D-Hide is a hidden objects detection tool that can detect any kind of stealthy malware not just limited to specific object types. A key feature of D-Hide is its ability to perform cross-view comparison without the need for an internal debugging tool to read the operating system's internal view. This feature enables deploying D-hide to monitor and protect hosted virtual machines in the IaaS platform, where the cloud providers do not have any access rights to the virtual machine's running operating system. (ii) FP-Hook is a function pointer hooking detection tool that can detect malicious function pointers manipulations that could happen in any routine located in the kernel dynamic objects. (iii) D-pointer has the ability to detect dangling pointers that might be happen because of operating system runtime errors or even a malicious hacker. D-pointer is not like previous approaches that require two phases of the analysis to detect the presence of a dangling pointer. D-pointer makes use of kernel data DIGGER's approach to easily detect dangling pointers and deallocate them from the kernel address space. (iv) B-Force is an offline analysis tool that has the ability to deeply analyse kernel runtime objects to enable further investigations about rootkits behaviours. B-Force is considered a powerful memory forensics tool because it has the ability to retrieve accurate and detailed information about a running object based on OS-KDD generated *type-graphs*.

9.2 Future Research

CloudSec₊₊ results have been encouraging enough to merit further investigation. The research in this thesis can be extended and enhanced to introduce more robust security features that can be deployed in *CloudSec₊₊*. Below, we summarize key research points in this project that can be enhanced for better performance and productivity.

OS-KDD does not employ enough intelligence to work efficiently on operating system's kernel updates and new patch releases. An interesting thing would be developing an extension for OS-KDD to enable locating kernel code and data layout changes and then only analyse the new changes instead of re-analysing the whole kernel's source code, to get an updated

type-graph that reflects the new changes in the runtime kernel data layout. Furthermore, one potential issue with OS-KDD is the high performance overhead. Despite the fact that OS-KDD works offline and needs to be run once for each kernel version, the analysis performance overhead is still very high. Improving the analysis performance by optimizing the analysis phases and using parallel computations could greatly improve OS-KDD's efficiency for more practical usage in different applications not just limited to operating system static analysis. Moreover, OS-KDD can be employed to support mobile security. Mobile operating systems such as Android and iOS are also exposed to similar vulnerabilities that benefit from manipulating pointers located in kernel dynamic objects at runtime. This is because these operating systems are developed using the C and Objective C programming languages. For example, the Android operating system is based on the Linux kernel with further architectural changes. The Linux kernel is implemented in C and thus it has the same problem of generic pointers and dynamic data. Furthermore, Mobile devices are being used more and more widely nowadays. Many applications have been developed for different objectives such as children's learning, eHealth and mobile banking applications. Regular users including children can unintentionally allow executing of a malicious code in memory by touching links or accepting commands they are not aware of. Such malicious code can allow the installation of backdoors such as key loggers in memory by modifying a generic pointer to dereference the malicious code. Thus, in addition to the risks of runtime memory bugs and errors of C-based operating systems, mobile operating systems have an even wider attack interface. This is because many of such systems' users could unintentionally facilitate installing rootkits that target the runtime memory.

Our implementation of DIGGER is Windows operating system-specific. However, we discussed, in chapter 8, that a Linux version can be implemented by utilization the slab allocation mechanism to implement DIGGER's approach in a similar way to the pool memory used in Windows operating systems. A practical implementation of DIGGER on Linux operating systems would further demonstrate the validity of our assumptions. In this research project, we focused on operating system kernel data protection. Kernel code with rootkits that develop return and jump-oriented programming e worth attention, especially with the support of

OS-KDD that provides precise analyse of pointers scattered around kernel text pages. In the evaluation results of FP-Hook, there was a small false alarm rate of around %0.04 in device drivers that relates to pointers were wrongly identified as function pointers. Further investigations into the reasons of false alarms could help to end or lower this rate. Also the developed rootkits to modify function pointers were not highly accurate in the implementation-side, causing runtime errors. Further investigations in the technical development phases of the rootkits in order to get an-error free environment is also a point for future work.

References

- [1] O. S. Hofmann, A. M. Dunn, and S. Kim, "Ensuring operating system kernel integrity with OSck," in *Proc. of 16th international conference on Architectural support for programming languages and operating systems*, California, USA, 2011, pp. 279-290.
- [2] Cisco Systems. (2010, 2013). *Cloud: Powered by the Network What a Business Leader Must Know*. Available: http://www.cisco.com/en/US/solutions/collateral/ns341/ns991/white_paper_c11-609220.pdf
- [3] R. Meulen. (2013, 2013). *Gartner Says Cloud-Based Security Services Market to Reach \$2.1 Billion in 2013*. Available: <http://www.gartner.com/newsroom/id/2616115>
- [4] A. S. Ibrahim, J. Hamlyn-Harris, and J. Grundy, "Emerging Security Challenges of Cloud Virtual Infrastructure," in *Proc. of 2010 Asia Pacific Cloud Workshop co-located with APSEC2010*, Sydney, Australia, 2010, pp. 10-16.
- [5] M. Prandini and M. Ramilli, "Return-Oriented Programming," *IEEE Security & Privacy*, vol. 10, pp. 84-87, 2012.
- [6] J. Pfoh, C. Schneider, and C. Eckert, "Exploiting the x86 Architecture to Derive Virtual Machine State Information," in *Proc. of Fourth International Conference on Emerging Security Information Systems and Technologies (SECURWARE)*, 2010, pp. 166-175.
- [7] T. Garfinkel and M. Rosenblum, "Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. of 2003 Network and Distributed Systems Security Symposium*, 2003, pp. 191-206.
- [8] T. Chiueh, M. Conover, M. Lu, *et al.*, "Stealthy Deployment and Execution of In-Guest Kernel Agents," in *Proc. of 2009 Black Hat*, USA, 2009, pp. 25-32.
- [9] A. Ranadive, A. Gavrilovska, and K. Schwan, "IBMon: monitoring VMM-bypass capable InfiniBand devices using memory introspection," in *Proc. of 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, Nuremberg, Germany, 2009, pp. 25-32.
- [10] K. Nance, M. Bishop, and B. Hay, "Virtual Machine Introspection: Observation or Interference?," *Journal of IEEE Security and Privacy*, vol. 6, pp. 32-37, 2008.
- [11] B. Dolan-Gavitt, T. Leek, M. Zhivich, *et al.*, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *Proc. of 2011 IEEE Symposium on Security and Privacy*, 2011, pp. 297-312.
- [12] M. Neugschwandtner, C. Platzer, P. M. Comparetti, *et al.*, "dAnubis - Dynamic Device Driver Analysis Based on Virtual Machine Introspection," in *Proc. of Seventh Conference on Detection of Intrusions and Malware & Vulnerability Assessment Bonn - Germany*, 2010, pp. 41-60.
- [13] K. Nance, M. Bishop, and B. Hay, "Investigating the Implications of Virtual Machine Introspection for Digital Forensics," in *proc. of 2009 International Conference on Availability, Reliability and Security*, Fukuoka, 2009, pp. 1024-1029.
- [14] M. Carbone, W. Cui, L. Lu, *et al.*, "Mapping kernel objects to enable systematic integrity checking," in *Proc of 16th ACM conference on Computer and communications security*, Chicago, USA, 2009, pp. 555-565.
- [15] S. Bahram, X. Jiang, and Z. Wang, "DKSM: Subverting Virtual Machine Introspection for Fun and

- Profit," in *Proc. of 29th IEEE International Symposium on Reliable Distributed Systems*, New Delhi, India, 2010, pp. 36-44.
- [16] B. Dolan-Gavitt, A. Srivastava, P. Traynor, *et al.*, "Robust signatures for kernel data structures," in *Proc. of 16th ACM conference on Computer and communications security*, Illinois, USA, 2009, pp. 566-577.
- [17] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic Inference and Enforcement of Kernel Data Structure Invariants," in *Proc of 2008 Annual Computer Security Applications Conference*, 2008, pp. 77-86.
- [18] S. Andreas, "Searching for processes and threads in Microsoft Windows memory dumps," *Digital Investigation*, vol. 3, pp. 10-16, 2006.
- [19] Z. Lin, J. Rhee, and X. Zhang, "SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures," in *Proc. of 18th Network and Distributed System Security Symposium*, San Diego, CA, 2011, pp. 110-122.
- [20] N. L. Petroni, T. Fraser, A. Walters, *et al.*, "An architecture for specification-based detection of semantic integrity violations in kernel dynamic data," in *Proc. of 15th conference on USENIX Security Symposium - Volume 15*, Vancouver, Canada, 2006, pp. 35-44.
- [21] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, *et al.*, "CloudSec: A Security Monitoring Appliance for Virtual Machines in the IaaS Cloud Model," in *Proc. of 2011 International Conference on Network and System Security (NSS 2011)*, Milan, Italy, 2011, pp. 124-136.
- [22] W.-J. Tsaur, Y.-C. Chen, and B.-Y. Tsai, "A New Windows Driver-Hidden Rootkit Based on Direct Kernel Object Manipulation," in *Algorithms and Architectures for Parallel Processing*. vol. 5574, ed: Springer Berlin / Heidelberg, 2009, pp. 202-213.
- [23] Z. Wang, C. Wu, and P. Yew, "On improving heap memory layout by dynamic pool allocation," in *Proc. of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, Toronto, Ontario, Canada, 2010, pp. 92-100.
- [24] B. Liang, W. You, W. Shi, *et al.*, "Detecting stealthy malware with inter-structure and imported signatures," in *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security*, Hong Kong, China, 2011, pp. 217-227.
- [25] Z. Lin, J. Rhee, C. Wu, *et al.*, "Discovering Semantic Data of Interest from Un-mappable Memory with Confidence," in *Proc. of the 19th Network and Distributed System Security Symposium (NDSS'12)*, San Diego, CA, 2012 pp. 56-62.
- [26] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, *et al.*, "Supporting Virtualization-Aware Security Solutions using a Systematic Approach to Overcome the Semantic Gap," in *Proc. of 5th IEEE International Conference on Cloud Computing*, Hawaii, USA, 2012, pp. 12-22.
- [27] M. Mock, D. Atkinson, C. Chambers, *et al.*, "Program Slicing with Dynamic Points-To Sets," *IEEE Transaction of Software Engineering*, vol. 31, pp. 657-678, 2005.
- [28] T. Bletsch, X. Jiang, V. W. Freeh, *et al.*, "Jump-oriented programming: a new class of code-reuse attack," in *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security*, Hong Kong, China, 2011, pp. 30-40.
- [29] D. Avots, M. Dalton, B. Livshits, *et al.*, "Improving software security with a C pointer analysis," in *Proc. of 27th international conference on Software engineering*, St. Louis, MO, USA, 2005, pp. 332-341.
- [30] M. Siff, S. Chandra, T. Ball, *et al.*, "Coping with type casts in C," in *Proc. of the 7th European*

- software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, Toulouse, France, 1999, pp. 180-198.
- [31] D. R. Chase, M. Wegman, and K. Zadeck, "Analysis of pointers and structures," in *Proc. of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, White Plains, New York, USA, 1990, pp. 296-310.
- [32] V. Raman, "Pointer Analysis - A survey," Technical Report, University of California, http://classes.soe.ucsc.edu/cms203/Fall04/finalreports/raman_pointeranalysis.pdf2004.
- [33] L. Andersen, "Program Analysis and Specialization for the C Programming Language," PhD Thesis, University of Copenhagen, 1994.
- [34] B. Steensgaard, "Points-to analysis in almost linear time," in *Proc. of 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Florida, United States, 1996, pp. 32-41.
- [35] B. Hardekopf and C. Lin, "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *Proc. of 2007 ACM SIGPLAN conference on Programming language design and implementation*, California, USA, 2007, pp. 290-299.
- [36] H. Yu, J. Xue, W. Huo, *et al.*, "Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code," in *Proc. of 8th annual IEEE/ACM international symposium on Code generation and optimization*, Ontario, Canada, 2010, pp. 218-229.
- [37] J. Wang, X.-D. Ma, W. Dong, *et al.*, "Demand-Driven Memory Leak Detection Based on Flow- and Context-Sensitive Pointer Analysis," *Journal of Computer Science and Technology*, vol. 24, pp. 347-356, 2009.
- [38] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *Proc. of 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, GA, USA, 2009, pp. 226-238.
- [39] K. Hwang, S. Kulkareni, and Y. Hu, "Cloud Security with Virtualized Defense and Reputation-Based Trust Mangement," in *Proc. of Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, Chengdu, China, 2009, pp. 717-722.
- [40] T. Dillon, C. Wu, and E. Chang, "Cloud Computing: Issues and Challenges," in *Proc. of 24th IEEE International Conference on Advanced Information Networking and Applications*, Perth, Australia, 2010, pp. 27-33.
- [41] B. Kandukuri, R. Paturi, and A. Rakshit, "Cloud Security Issues," in *Proc. of IEEE International Conference on Services Computing*, Bangalore, 2009, pp. 517 - 520.
- [42] M. Christodorescu, R. Sailer, and D. L. Schales, "Cloud security is not (just) virtualization security," in *Proc. of 2009 ACM workshop on Cloud computing security*, Illinois, USA, 2009, pp. 97-102.
- [43] cisco Systems. (2009, 2011). *Cisoc Cloud Computing - Data Center Strategy, Architecture and Solutions*. Available: http://www.cisco.com/web/strategy/docs/gov/CiscoCloudComputing_WP.pdf
- [44] F. Hao, T. Lakshman, S. Mukherjee, *et al.*, "Secure Cloud Computing with a Virtualized Network Infrastructure," in *Proc. of 2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'10)*, Boston, MA, June 2010.
- [45] F. Lombardia and R. D. Pietro, "Secure virtualization for cloud computing " *Elsevier Ltd.*, 2010.
- [46] G. Lin, D. Fu, and J. Zhu, "Cloud Computing: IT as a Service," *IT Professional*, vol. 11, pp. 10-13,

- 2009.
- [47] L. M. Kaufman, "Can a Trusted Environment Provide Security?," *IEEE Security & Privacy*, vol. 8, pp. 50-52, 2010.
- [48] T. Bittman. (October 2009, 2010). *Server Virtualization: One path that lead to cloud computing* [Gartner RSA Core Research]. Available: <http://www.vmware.com/files/pdf/analysts/Gartner-server-virtualization-leads-to-cloud-computing.pdf>
- [49] T. Software, "IBM Point of View: Security and Cloud Computing," *IBM Research*, 2009.
- [50] W. Dawoud, I. Takouna, and C. Meinel, "Infrastructure as a service security: Challenges and solutions," in *Proc. of 2010 the 7th International Conference on Informatics and Systems*, cairo, Egypt, 2010, pp. 1-8.
- [51] K. Hashizume, D. Rosado, E. Fernández-Medina, *et al.*, "An analysis of security issues for cloud computing," *Journal of Internet Services and Applications*, vol. 4, pp. 1-13, 2013/02/27 2013.
- [52] M. Almorsy, J. Grundy, and I. Mueller, "An analysis of the cloud computing security problem," in *Proc. of the 2010 Asia Pacific Cloud Workshop 2010*, Sydne, Australia, 2010, pp. 26-31.
- [53] M. Jensen, J. Schwenk, and N. Gruschka, "On Technical Security Issues in Cloud Computing," in *Proc. of 2009 IEEE International Conference on Cloud Computing*, Bangalore, India, 2009, pp. 109-116.
- [54] B. Grobauer, T. Walloschek, and E. Stöcker, "Understanding Cloud-Computing Vulnerabilities," in *Proc. of IEEE Security and Privacy*, Claremont Resort, Berkeley, CA, 2010, pp. 1-8.
- [55] M. Yildiz, J. Abawajy, T. Ercan, *et al.*, "A Layered Security Approach for Cloud Computing Infrastructure," in *Proc. of 10th International Symposium on Pervasive Systems, Algorithms, and Networks*, Kaohsiung, Taiwan, 2009, pp. 763-767.
- [56] Alexander Lenk, Markus Klems, Jens Nimis, Stefan Tai and Thomas Sandholm, "What's inside the Cloud? An architectural map of the Cloud landscape," in *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, 2009, pp. 23-3.
- [57] M. Brock and A. Goscinski, "Toward a Framework for Cloud Security," in *Algorithms and Architectures for Parallel Processing*. vol. 6082, C.-H. Hsu, L. Yang, J. Park, and S.-S. Yeo, Eds., ed: Springer Berlin Heidelberg, 2010, pp. 254-263.
- [58] S. Qamar, N. Lal, and M. Singh, "Internet ware cloud computing :Challenges," *International Journal of Computer Science and Information Security*, vol. 7, pp. 112-129, March 2010.
- [59] I. M. a. C. Barton, "Cloud security technologies," *Elsevier Ltd.*, vol. 14, pp. Pages 1-6, April 2009.
- [60] K. Hwang, S. Kulkareni, and Y. Hu, "Cloud Security with Virtualized Defense and Reputation-Based Trust Mangement," in *Proc. of Eighth IEEE International Conference on Dependable, Autonomic and Secure Computing*, Chengdu, China, 2009, pp. 717-722.
- [61] E. Caron, L. Anh Dung, A. Lefray, *et al.*, "Definition of Security Metrics for the Cloud Computing and Security-Aware Virtual Machine Placement Algorithms," in *Proc. of 2013 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, 2013, pp. 125-131.
- [62] S. Berger, R. Cáceres, D. Pendarakis, *et al.*, "TVDC: Managing Security in the Trusted Virtual Datacenter," *ACM SIGOPS Operating Systems Review (Systems work at IBM Research)*, vol. 42, pp. 40-47, 2008.
- [63] F. Krautheim, "Private virtual infrastructure for cloud computing," in *Proc. of the 2009 conference on Hot topics in cloud computing*, San Diego, California, 2009, pp. 36-45.

-
- [64] (July 2011). *XEN : Security Vulnerabilities* [Common Vulnerabilities and Exposures]. Available: www.cvedetails.com/vulnerability-list/vendor_id-6276/XEN.html
- [65] (Sep 2011). *ESX : Security Vulnerabilities* [Common Vulnerabilities and Exposures]. Available: www.cvedetails.com/vulnerability-list/vendor_id-252/product_id-14181/Vmware-ESX.html
- [66] M. Carbone, D. Zamboni, and W. Lee, "Taming Virtualization," *IEEE Security and Privacy*, vol. 6, pp. 65-67, 2008.
- [67] R. Wojtczuk. (2008, May 2011). *Subverting the Xen hypervisor*. Available: http://www.invisiblethingslab.com/bh08/papers/part1-subverting_xen.pdf
- [68] devcentral. (2008, Sept 2012). *IOMMU*. Available: <http://developer.amd.com/community/blog/2008/09/01/iommu/>
- [69] *Intel® Virtualization Technology*.
- [70] A. Seshadri, M. Luk, N. Qu, *et al.*, "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," in *Proc. of 21st ACM SIGOPS symposium on Operating systems principles*, Stevenson, Washington, USA, 2007, pp. 335-350.
- [71] R. Riley, Y. Xuxian, and D. Xu, "Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing," in *Proc of the 11th international symposium on Recent Advances in Intrusion Detection*, Cambridge, MA, USA, 2008, pp. 210-210.
- [72] U. Steinberg and B. Kauer, "NOVA: a microhypervisor-based secure virtualization architecture," in *Proc. of the 5th European conference on Computer systems*, Paris, France, 2010, pp. 209-222.
- [73] T. Shinagawa, H. Eiraku, and K. Tanimoto, "BitVisor: a thin hypervisor for enforcing i/o device security," in *Proc. of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Washington, DC, USA, 2009, pp. 121-130.
- [74] D. G. Murray, G. Milos, and S. Hand, "Improving Xen security through disaggregation," in *Proc. of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environment*, Seattle, WA, USA, 2008, pp. 151-160.
- [75] Intel. (Jan 2014). *Trusted Compute Pools with Intel® Trusted Execution Technology (Intel® TXT)*. Available: <http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/malware-reduction-general-technology.html>
- [76] T. Mirzoev and B. Yang, "Securing Virtualized Datacenters," *International Journal of Engineering Research & Innovation*, vol. 2, 2010.
- [77] E. Elmroth and L. Larsson, "Interfaces for Placement, Migration, and Monitoring of Virtual Machines in Federated Clouds," in *Proc. of the 2009 Eighth International Conference on Grid and Cooperative Computing*, 2009, pp. 253-260.
- [78] Q. Li, Q. Hao, L. Xiao, *et al.*, "Adaptive Management of Virtualized Resources in Cloud Computing Using Feedback Control," in *Proc. of 1st International Conference on Information Science and Engineering (ICISE)*, Nanjing, China, April 2010, pp. 99 - 102
- [79] J. Wei, X. Zhang, G. Ammons, *et al.*, "Managing security of virtual machine images in a cloud environment," in *Proc. of the 2009 ACM workshop on Cloud computing security*, Chicago, Illinois, USA, 2009, pp. 91-96.
- [80] W. Wang, Y. Zhang, B. Lin, *et al.*, "Secured and Reliable VM Migration in Personal Cloud," in *Proc. of 2nd International Conference on Computer Engineering and Technology*, Chengdu, China, 2010, pp. 705-709
- [81] F. Hao, T. V. Lakshman, S. Mukherjee, *et al.*, "Enhancing dynamic cloud-based services using
-

- network virtualization," in *Proc. of ACM SIGCOMM Computer Communication*, York, NY, USA, 2010, pp. 67-74.
- [82] (2009, May 2010). *Making Virtual Machines Cloud-Ready* [White Paper]. Available: http://emea.trendmicro.com/imperia/md/content/uk/products/whitepapers/wp04_vm_cloudsecurity100528us.pdf
- [83] R. Schwarzkopf, M. Schmidt, C. Strack, *et al.*, "Checking Running and Dormant Virtual Machines for the Necessity of Security Updates in Cloud Environments," in *Proc. of 2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, Athens, Greece, 2011, pp. 239-246.
- [84] S. King, P. Chen, Y.-M. Wang, *et al.*, "SubVirt: Implementing malware with virtual machines," in *proc. of 2006 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2006, pp. 314-327.
- [85] D. D. Zovi, "Hardware Virtualization Rootkits," in *Proc. of 2009 BlackHat Conference*, USA, 2009.
- [86] J. Franklin, M. Luk, J. M. McCune, *et al.*, "Remote detection of virtual machine monitors with fuzzy benchmarking," *SIGOPS Operating System Review*, vol. 42, pp. 83-92, 2008.
- [87] R. Paleari, L. Martignoni, G. F. Roglia, *et al.*, "A fistful of red-pills: how to automatically generate procedures to detect CPU emulators," in *Proc. of the 3rd USENIX conference on Offensive technologies*, Montreal, Canada, 2009, pp. 2-2.
- [88] P. Royal, "Alternative medicine: The malware analyst's blue pill," in *Proc. of Blackhat Conference*, USA, 2008, pp. 1-1.
- [89] Monirul I. Sharif, Wenke Lee, Weidong Cui, Andrea Lanzi, "Secure in-VM monitoring using hardware virtualization," *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 477-487, 2009.
- [90] K. Vieira, A. Schuster, and C. Westphall, "Intrusion Detection for Grid and Cloud Computing," *IT Professional*, vol. 12, pp. 38-43, 2010.
- [91] Lorenzo Martignoni, R. Paleari, and D. Bruschi, "A Framework for Behavior-Based Malware Analysis in the Cloud," in *Proc. of the 5th International Conference on Information Systems Security*, Kolkata, India, 2009, pp. 178-192.
- [92] D. Alvim, S. d. Oliveira, and F. Wu, "Protecting Kernel Code and Data with a Virtualization-Aware Collaborative Operating System," in *Proc. of 2009 Annual Computer Security Applications Conference*, Honolulu, HI, 2009, pp. 451-460.
- [93] B. Jansen, H. Ramasamy, and M. Schunter, "Architecting Dependable and Secure Systems Using Virtualization," in *Architecting Dependable Systems*. vol. 5135, ed: Springer Berlin, 2008, pp. 124-149.
- [94] G. W. Dunlap, S. T. King, S. Cinar, *et al.*, "ReVirt: enabling intrusion analysis through virtual-machine logging and replay," *ACM SIGOPS Operating Systems Review*, vol. 36, pp. 211 - 224, 2002.
- [95] B. D. Payne, M. Carbone, and W. Lee, "Secure and Flexible Monitoring of Virtual Machines," in *Proc. of 23rd Annual Computer Security Applications Conference*, 2007, pp. 385-397.
- [96] J. Rhee, R. Riley, D. Xu, *et al.*, "Defeating Dynamic Data Kernel Rootkit Attacks via VMM-Based Guest-Transparent Monitoring " in *Proc. of 2009 International Conference on Availability, Reliability and Security*, 2009, pp. 74-81.

-
- [97] N. A. Quynh and Y. Takefuji, "Towards a tamper-resistant kernel rootkit detector," in *Proc. of the 2007 ACM symposium on Applied computing*, Seoul, Korea, 2007, pp. 276-283.
- [98] Koichi Onoue, Yoshihiro Oyama and Akinori Yonezawa, "Control of system calls from outside of virtual machines," in *ACM Symposium on Applied Computing*, Fortaleza, Ceara, Brazil, 2008, pp. 1116-1221.
- [99] D. Kienzle, R. Persaud, and M. Elder, "Endpoint Configuration Compliance Monitoring via Virtual Machine Introspection," in *Proc. of the 43rd Hawaii International Conference on System Sciences*, Honolulu, HI, 2010, pp. pp.1-10.
- [100] F. Baiardi and D. Sgandurra, "Building Trustworthy Intrusion Detection through VM Introspection," in *Proc. of The Third International Symposium on Information Assurance and Security*, Manchester, Germany, 2007, pp. 209-214.
- [101] A. Joshi, S. T. King, G. W. Dunlap, *et al.*, "Detecting Past and Present Intrusions through Vulnerability-Specific Predicates," in *Proc. of The twentieth ACM symposium on Operating systems principles*, Brighton, United Kingdom, 2005, pp. 91 - 104.
- [102] B. D. Payne, M. Carbone, M. Sharif, *et al.*, "Lares: An Architecture for Secure Active Monitoring Using Virtualization," in *Proc. of IEEE Symposium on Security and Privacy*, Oakland, CA, 2008, pp. 233-247.
- [103] F. Lombardi and R. D. Pietro, "KvmSec: a security extension for Linux kernel virtual machines," in *Proc. of 2009 ACM symposium on Applied Computing*, Honolulu, Hawaii, 2009, pp. 2029-2034.
- [104] M. Sharif, W. Lee, and W. Cui, "Secure in-VM monitoring using hardware virtualization," in *Proc. of The 16th ACM conference on Computer and communications security*, Chicago, Illinois, USA, 2009, pp. 477-487.
- [105] A. V. Konstantinou, T. Eilam, M. Kalantar, *et al.*, "An architecture for virtual solution composition and deployment in infrastructure clouds," in *Proc. of the 3rd international workshop on Virtualization technologies in distributed computing*, Barcelona, Spain, 2009, pp. 9-18.
- [106] H. Yin, Z. Liang, and D. Song, "HookFinder: Identifying and understanding malware hooking behaviors," in *Proc. of Network and Distributed Systems Security Symposium (NDSS)*, San Diego, California, USA, 2008.
- [107] H. Yin, P. Poosankam, S. Hanna, *et al.*, "HookScout: Proactive Binary-Centric Hook Detection," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. vol. 6201, ed: Springer Berlin / Heidelberg, 2010, pp. 1-20.
- [108] A. Baliga, V. Ganapathy, and L. Iftode, "Detecting Kernel-Level Rootkits Using Data Structure Invariants," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 1-10, 2010.
- [109] C. Kruegel, W. Robertson, and G. Vigna, "Detecting Kernel-Level Rootkits Through Binary Analysis," in *Proc. of the 20th Annual Computer Security Applications Conference*, 2004, pp. 91-100.
- [110] A. Ibrahim, M. Shouman, and H. Faheem, "Surviving cyber warfare with a hybrid multiagent-base intrusion prevention system," *IEEE Potentials*, vol. 29, pp. 32-40, 2010.
- [111] H. Yin, D. Song, M. Egele, *et al.*, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proc. of the 14th ACM conference on Computer and communications security*, Alexandria, Virginia, USA, 2007, pp. 116-127.
- [112] Carsten Willems, Thorsten Holz, Felix Freiling, "Toward Automated Dynamic Malware Analysis Using CWSandbox," *IEEE Security and Privacy*, vol. 5, pp. 32-39, 2007.
-

- [113] J. Rhee, R. Riley, and D. Xu, "Kernel malware analysis with un-tampered and temporal views of dynamic kernel memory," in *Proc. of 13th international conference on Recent advances in intrusion detection*, Ontario, Canada, 2010, pp. 178-197.
- [114] C. Xuan, J. Copeland, and R. Beyah, "Toward Revealing Kernel Malware Behavior in Virtual Execution Environments," in *Proc. of the 12th International Symposium on Recent Advances in Intrusion Detection*, Saint-Malo, France, 2009, pp. 304-325.
- [115] P. Dewan, D. Durham, H. Khosravi, *et al.*, "A hypervisor-based system for protecting software runtime memory and persistent storage," in *Proc. of the 2008 Spring simulation multiconference*, Ottawa, Canada, 2008, pp. 828-835.
- [116] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," in *Proc. of 2010 IEEE Symposium on Security and Privacy*, 2010, pp. 380-395.
- [117] L. Litty and D. Lie, "Manitou: a layer-below approach to fighting malware," in *Proc. of the 1st workshop on Architectural and system support for improving software dependability*, San Jose, California, 2006, pp. 6-11.
- [118] M. Laureano, C. Maziero, and E. Jamhour, "Intrusion Detection in Virtual Machine Environments," in *Proc. of the 30th EUROMICRO Conference*, Rennes, France, 2004, pp. 520-525.
- [119] X. Jiang and X. Wang, "'Out-of-the-Box' monitoring of VM-based high-interaction honeypots," in *Proc. of the 10th international conference on Recent advances in intrusion detection*, Gold Coast, Australia, 2007, pp. 198-218.
- [120] N. L. Petroni and M. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proc. of 14th ACM conference on Computer and communications security*, Alexandria, Virginia, USA, 2007, pp. 103-115.
- [121] N. L. Petroni, T. Fraser, J. Molina, *et al.*, "Copilot - a coprocessor-based kernel runtime integrity monitor," in *Proc of the 13th conference on USENIX Security Symposium - Volume 13*, San Diego, CA, 2004, pp. 13-13.
- [122] J. Wilhelm and T.-c. Chiueh, "A forced sampled execution approach to kernel rootkit identification," in *Proc. of the 10th international conference on Recent advances in intrusion detection*, Gold Coast, Australia, 2007, pp. 219-235.
- [123] B. Demsky and M. Rinard, "Automatic detection and repair of errors in data structures," in *Proc. of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Anaheim, California, USA, 2003, pp. 78-95.
- [124] A. Lanzi, M. Sharif, and W. Lee, "K-Tracer: A System for Extracting Kernel Malware Behavior," in *Proc. of the 16th Network and Distributed System Security Symposium*, San Diego, CA, 2009.
- [125] Z. Wang, X. Jiang, W. Cui, *et al.*, "Countering Persistent Kernel Rootkits through Systematic Hook Discovery," in *Proc. of the 11th international symposium on Recent Advances in Intrusion Detection*, Massachusetts, USA, 2009.
- [126] Z. Wang, X. Jiang, W. Cui, *et al.*, "Countering kernel rootkits with lightweight hook protection," in *Proc. of the 16th ACM conference on Computer and communications security*, Chicago, Illinois, USA, 2009, pp. 545-554.
- [127] X. Jiang, X. Wang, and D. Xu, "Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction," in *Proc. of 14th ACM conference on Computer and*

- communications security*, Virginia, USA, 2007, pp. 128-138.
- [128] M. Nanavati and B. Kothari. (2010, Nov 2010). *Hidden Processes Detection using the PspCidTable* [Technical Report]. Available: http://helios.miel-labs.com/downloads/process_scan.pdf
- [129] Bugcheck. (2006, Apr 2011). *GREPEXEC: Grepping Executive Objects from Pool Memory*. Available: <http://uninformed.org/?v=4&a=2&t=sumry>
- [130] I. Sutherland, J. Evans, T. Tryfonas, *et al.*, "Acquiring volatile operating system data tools and techniques," *SIGOPS Operating System Review*, vol. 42, pp. 65-73, 2008.
- [131] R. Riley, X. Jiang, and D. Xu, "Multi-aspect profiling of kernel rootkit behavior," in *Proc. of the 4th ACM European conference on Computer systems*, Nuremberg, Germany, 2009, pp. 47-60.
- [132] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Antfarm: tracking processes in a virtual machine environment," in *Proc. of the annual conference on USENIX '06 Annual Technical Conference*, Boston, MA, 2006, pp. 1-1.
- [133] A. Ibrahim, J. Hamlyn-Harris, J. Grundy, *et al.*, "Identifying OS Kernel Objects for Run-Time Security Analysis," in *Network and System Security*. vol. 7645, L. Xu, E. Bertino, and Y. Mu, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 72-85.
- [134] D. Beck, B. Vo, and C. Verbowski, "Detecting Stealth Software with Strider GhostBuster," in *Proc. of the 2005 International Conference on Dependable Systems and Networks*, 2005, pp. 368-377.
- [135] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "VMM-based hidden process detection and identification using Lycosid," in *Proc. of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, Seattle, WA, USA, 2008, pp. 91-100.
- [136] K. A. Z. Ariffin, A. K. Mahmood, and J. Jaafar, "Investigating the PROCESS block for memory analysis," in *Proc. of the 11th WSEAS international conference on Applied computer science*, Penang, Malaysia, 2011, pp. 21-29.
- [137] A. Schuster, "Pool allocations as an information source in Windows memory forensics," in *Proc. of International conference on IT-incident management and IT-forensics*, Stuttgart, Germany, 2006, pp. 112-121.
- [138] A. Schuster, "The impact of Microsoft Windows pool allocation strategies on memory forensics," *Digital Investigation*, vol. 5, pp. S58-S64, 2008.
- [139] B. Mariusz. (2005, June 2011). *An introduction to Windows memory forensic* [Technical Report]. Available: http://forensic.seccure.net/pdf/introduction_to_windows_memory_forensic%.pdf
- [140] B. Chris. (2005, Aug 2005). *MemParser*. Available: <http://www.dfrws.org/2005/challenge/memparser.html>
- [141] G. G. M and M. Robert-Jan. (2005, Aug 2011). *Kntlist*. Available: <http://www.dfrws.org/2005/challenge/kntlist.html>
- [142] LMH. (2006, June 2012). *the Month of Kernel Bugs (MoKB)*. Available: <http://projects.info-pull.com/mokb/>
- [143] E. D. Berger and B. G. Zorn, "DieHard: probabilistic memory safety for unsafe languages," in *Proc. of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, Ottawa, Ontario, Canada, 2006, pp. 158-168.
- [144] T. Jim, J. G. Morrisett, D. Grossman, *et al.*, "Cyclone: A Safe Dialect of C," in *Proc. of the General Track of the annual conference on USENIX Annual Technical Conference*, 2002, pp. 275-288.
- [145] C. Cowan, C. Pu, D. Maier, *et al.*, "StackGuard: automatic adaptive detection and prevention of

- buffer-overflow attacks," in *Proc. of the 7th conference on USENIX Security Symposium*, San Antonio, Texas, 1998, pp. 5-5.
- [146] P. Akritidis, C. Cadar, C. Raiciu, *et al.*, "Preventing Memory Error Exploits with WIT," in *Proc. of 2008 IEEE Symposium on Security and Privacy*, 2008, pp. 263-277.
- [147] J. S. Foster, M. Fhndrich, and A. Aiken, "A theory of type qualifiers," in *Proc. of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, Atlanta, Georgia, USA, 1999, pp. 192-203.
- [148] O. Ruwase and M. S. Lam, "A Practical Dynamic Buffer Overflow Detector," in *Proc. of the 11th Annual Network and Distributed System Security Symposium*, 2004, pp. 159--169.
- [149] J. Yang, T. Kremenek, Y. Xie, *et al.*, "MECA: an extensible, expressive system and language for statically checking security properties," in *Proc. of the 10th ACM conference on Computer and communications security*, Washington D.C., USA, 2003, pp. 321-334.
- [150] Linux. *Sparse* [Linux Project]. Available: <http://elinux.org/Sparse>
- [151] M. Das, S. Lerner, and M. Seigle, "ESP: path-sensitive program verification in polynomial time," in *Proc. of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, Berlin, Germany, 2002, pp. 57-68.
- [152] B. Blanchet, P. Cousot, R. Cousot, *et al.*, "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software," in *The Essence of Computation*. vol. 2566, ed: Springer Berlin Heidelberg, 2002, pp. 85-108.
- [153] R. Jhala and R. Majumdar, "Software model checking," *ACM Computer Survey*, vol. 41, pp. 1-54, 2009.
- [154] Microsoft. (May 2013). *The SLAM Project*. Available: <http://research.microsoft.com/en-us/projects/slam/>
- [155] T. A. Henzinger, R. Jhala, R. Majumdar, *et al.*, "Lazy abstraction," in *Proc. of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Portland, Oregon, 2002, pp. 58-70.
- [156] J. C. Reynolds, "Separation Logic: A Logic for Shared Mutable Data Structures," in *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science*, New Orleans, LA, USA, 2002, pp. 55-74.
- [157] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," in *Programming Methodology*, D. Gries, Ed., ed: Springer New York, 1978, pp. 89-100.
- [158] M. Bozga, R. Iosif, and S. Perarnau, "Quantitative Separation Logic and Programs with Lists," in *Automated Reasoning*. vol. 5195, A. Armando, P. Baumgartner, and G. Dowek, Eds., ed: Springer Berlin Heidelberg, 2008, pp. 34-49.
- [159] H. Tuch, G. Klein, and M. Norrish, "Types, bytes, and separation logic," in *Proc. of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Nice, France, 2007, pp. 97-108.
- [160] N. Marti, R. Affeldt, and A. Yonezawa, "Formal verification of the heap manager of an operating system using separation logic," in *Proc. of the 8th international conference on Formal Methods and Software Engineering*, Macao, China, 2006, pp. 400-419.
- [161] R. Kolanski and G. Klein, "Mapped Separation Logic," in *Verified Software: Theories, Tools, Experiments*. vol. 5295, ed: Springer Berlin Heidelberg, 2008, pp. 15-29.
- [162] F. Mehta and T. Nipkow, "Proving Pointer Programs in Higher-Order Logic," in *Automated*

-
- Deduction – CADE-19*. vol. 2741, ed: Springer Berlin Heidelberg, 2003, pp. 121-135.
- [163] J.-C. Filliâtre and C. Marché, "Multi-prover Verification of C Programs," in *Formal Methods and Software Engineering*. vol. 3308, ed: Springer Berlin Heidelberg, 2004, pp. 15-29.
- [164] J. Berdine, C. Calcagno, B. Cook, *et al.*, "Shape Analysis for Composite Data Structures," in *Computer Aided Verification*. vol. 4590, ed: Springer Berlin Heidelberg, 2007, pp. 178-192.
- [165] D. Distefano, P. W. Hearn, and H. Yang, "A local shape analysis based on separation logic," in *Proc. of the 12th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, Vienna, Austria, 2006, pp. 287-302.
- [166] O. Lee, H. Yang, and K. Yi, "Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis," in *Programming Languages and Systems*. vol. 3444, ed: Springer Berlin Heidelberg, 2005, pp. 124-140.
- [167] M. Češka, P. Erlebach, and T. Vojnar, "Generalised multi-pattern-based verification of programs with linear linked structures," *Formal Aspects of Computing*, vol. 19, pp. 363-374, 2007/08/01 2007.
- [168] P. Fradet and D. L. Mtayer, "Shape types," in *Proc. of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Paris, France, 1997, pp. 27-39.
- [169] N. Klarlund and M. I. Schwartzbach, "Graph types," in *Proc. of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Charleston, South Carolina, USA, 1993, pp. 196-205.
- [170] A. Miller and M. I. Schwartzbach, "The pointer assertion logic engine," in *Proc. of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, Snowbird, Utah, USA, 2001, pp. 221-231.
- [171] J. Foster, M. Fähndrich, and A. Aiken, "Polymorphic versus Monomorphic Flow-Insensitive Points-To Analysis for C," in *Static Analysis*. vol. 1824, ed: Springer Berlin Heidelberg, 2000, pp. 175-198.
- [172] R. Ghiya, D. Lavery, and D. Sehr, "On the importance of points-to analysis and other memory disambiguation methods for C programs," in *Proc. of ACM SIGPLAN 2001 conference on Programming language design and implementation*, Utah, US, 2001, pp. 47-58.
- [173] C. Lattner, A. Lenharth, and V. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proc. of 2007 ACM SIGPLAN conference on Programming language design and implementation*, California, USA, 2007, pp. 278-289.
- [174] D. Liang and M. J. Harrold, "Efficient points-to analysis for whole-program analysis," in *Proc. of the 7th European software engineering conference*, Toulouse, France, 1999, pp. 199-215.
- [175] R. Nasre, "Scaling Context-Sensitive Points-To Analysis," PhD, Computer Science and Automation, Indian Institute of Science, BANGALORE, 2012.
- [176] A. Rountev and S. Chandra, "Off-line variable substitution for scaling points-to analysis," in *Proc. of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, Vancouver, British Columbia, Canada, 2000, pp. 47-56.
- [177] M. Sridharan and R. Bod, "Refinement-based context-sensitive points-to analysis for Java," in *Proc. of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, Ottawa, Ontario, Canada, 2006, pp. 387-400.
- [178] G. Xu, A. Rountev, and M. Sridharan, "Scaling CFL-Reachability-Based Points-To Analysis Using Context-Sensitive Must-Not-Alias Analysis," in *Proc. of the 23rd European Conference on*
-

- Object-Oriented Programming*, Italy, 2009, pp. 98-122.
- [179] M. Emami, R. Ghiya, and L. J. Hendren, "Context-sensitive interprocedural points-to analysis in the presence of function pointers," in *Proc. of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, Orlando, Florida, USA, 1994, pp. 242-256.
- [180] M. Das, B. Liblit, M. Fähndrich, *et al.*, "Estimating the Impact of Scalable Pointer Analysis on Optimization," in *Static Analysis*. vol. 2126, ed: Springer Berlin Heidelberg, 2001, pp. 260-278.
- [181] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proc. of ACM SIGPLAN 2004 conference on Programming language design and implementation*, Washington DC, USA, 2004, pp. 131-144.
- [182] N. Heintze and O. Tardieu, "Ultra-fast aliasing analysis using CLA: a million lines of C code in a second," in *Proc. of ACM SIGPLAN 2001 conference on Programming language design and implementation*, Utah, USA, 2001, pp. 254-263.
- [183] M. F. hndrich, J. Rehof, *et al.*, "Scalable context-sensitive flow analysis using instantiation constraints," in *Proc. of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, Vancouver, British Columbia, Canada, 2000, pp. 253-263.
- [184] E. Nystrom, H.-S. Kim, and W.-m. Hwu, "Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis," in *Static Analysis*. vol. 3148, ed: Springer Berlin Heidelberg, 2004, pp. 165-180.
- [185] A. Deutsch, "Interprocedural may-alias analysis for pointers: beyond limiting," in *Proc. of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, Orlando, Florida, USA, 1994, pp. 230-241.
- [186] R. O'Callahan and D. Jackson, "Lackwit: a program understanding tool based on type inference," in *Proc. of the 19th international conference on Software engineering*, Boston, Massachusetts, USA, 1997, pp. 338-348.
- [187] R. Ghiya and L. J. Hendren, "Connection analysis: a practical interprocedural heap analysis for C," *International Journal of Parallel Program*, vol. 24, pp. 547-578, 1996.
- [188] D. Liang and M. Harrold, "Efficient Computation of Parameterized Pointer Information for Interprocedural Analyses," in *Static Analysis*. vol. 2126, ed: Springer Berlin Heidelberg, 2001, pp. 279-298.
- [189] R. Tarjan, "Depth-first search and linear graph algorithms," in *12th Annual Symposium on Switching and Automata Theory*, 1971, pp. 114-121.
- [190] D. Y. a. C. T. A. Chan, "Refinement of rule-based intrusion detection system for denial of service attacks by support vector machine," in *Proc. of International Conference on Machine Learning and Cybernetics*, 2004, pp. 79-85.
- [191] M. Buss, S. A. Edwards, Y. Bin, *et al.*, "Pointer analysis for source-to-source transformations," in *Proc. of 5th IEEE International Workshop on Source Code Analysis and Manipulation*, 2005, pp. 139-148.
- [192] D. J. Pearce, P. H. J. Kelly, and C. Hankin, "Online cycle detection and difference propagation for pointer analysis," in *Proc. of Third IEEE International Workshop on Source Code Analysis and Manipulation*, 2003, pp. 3-12.
- [193] D. J. Pearce, P. H. Kelly, and C. Hankin, "Efficient field-sensitive pointer analysis for C," in *Proc. of 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Washington DC, USA, 2004, pp. 37-42.

-
- [194] M. Hind and A. Pioli, "Which pointer analysis should I use?," in *Proc of 2000 ACM SIGSOFT international symposium on Software testing and analysis*, Portland, Oregon, United States, 2000, pp. 113-123.
- [195] M. Hind and A. Pioli, "Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses," in *Proc. of the 5th International Symposium on Static Analysis*, 1998, pp. 57-81.
- [196] J.-D. Choi, M. Burke, and P. Carini, "Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects," in *Proc. of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, Charleston, South Carolina, USA, 1993, pp. 232-245.
- [197] R. Hasti and S. Horwitz, "Using static single assignment form to improve flow-insensitive pointer analysis," in *Proc. of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, Montreal, Quebec, Canada, 1998, pp. 97-105.
- [198] J. Zhu, "Towards scalable flow and context sensitive pointer analysis," in *Proc. of the 42nd annual Design Automation Conference*, Anaheim, California, USA, 2005, pp. 831-836.
- [199] T. Tok, S. Guyer, and C. Lin, "Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers," in *Compiler Construction*. vol. 3923, ed: Springer Berlin Heidelberg, 2006, pp. 17-31.
- [200] Y. Chen, R. Venkatesan, M. Cary, *et al.*, "Oblivious Hashing: A Stealthy Software Integrity Verification Primitive," in *Proc. of 5th International Workshop on Information Hiding 2003*, pp. 400-414.
- [201] VMware. (2009, Oct 2010). *Virtual Appliances: A New Paradigm for Software Delivery* [White Paper]. Available: www.vmware.com/files/pdf/vam/VMware_Virtual_Appliance_Solutions_White_Paper_08Q3.pdf
- [202] Trusted Computing Group. (Mar 2011). *Trusted Platform Module*. Available: http://www.trustedcomputinggroup.org/developers/trusted_platform_module/
- [203] Intel. (Mar 2011). *Intel Trusted Execution Technology*. Available: <http://www.intel.com/technology/security/>
- [204] (June 2011). *Trusted Computing Base* [Wikipedia]. Available: http://en.wikipedia.org/wiki/Trusted_computing_base
- [205] W. A. Arbaugh, D. J. Farber, and J. M. Smith, "A secure and reliable bootstrap architecture," in *Proc. of the 1997 IEEE Symposium on Security and Privacy*, 1997, p. 65.
- [206] Bryan D. Payne, Reiner Sailer, Ramón Cáceres, Ron Perez and Wenke Lee, "A layered approach to simplified access control in virtualized systemsA layered approach to simplified access control in virtualized systems," *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 12 - 19, 2007.
- [207] B. D. Payne, "Improving host-based computer security using secure active monitoring and memory analysis," PhD, Georgia Institute of Technology, Georgia Institute of Technology, 2010.
- [208] VMware. (Sept 2010). *Vmware vmsafe security technology*. Available: <http://www.vmware.com/technical-resources/security/vmsafe.html>
- [209] B. Barney, L. Livermore, and N. Laboratory. (Sept 2010). *POSIX Threads Programming*. Available: <https://computing.llnl.gov/tutorials/pthreads/>
- [210] Intel. (2011, June 2011). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Available: www.intel.com/Assets/PDF/manual/253668.pdf
- [211] VMware. (2009, Jan 2013). *Understanding Memory Resource Management in VMware vSphere®*
-

- 5.0. Available: http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf
- [212] David A. Solomon, "The Windows NT Kernel Architecture," *Computer*, vol. 31, pp. 40-47, 1998.
- [213] Microsoft. (Dec 2010). *Windows Symbol Packages*. Available: <http://msdn.microsoft.com/en-us/windows/hardware/gg463028>
- [214] Microsoft. (Sept 2010). *Debugging Tools For Windows*. Available: <http://msdn.microsoft.com/en-us/windows/hardware/gg463009>
- [215] A. Danehkar. (2007, May 2011). *Injective Code inside Import Table*. Available: <http://www.codeproject.com/KB/system/inject2it.aspx>
- [216] (Jul 2010). *RemoteDLL*. Available: <http://securityxploded.com/remotedll.php>
- [217] (2009, Jan 2012). *NiIllusion V1.0*. Available: http://www.mobile-download.net/Soft/Soft_11734.htm
- [218] (Feb 2011). *FU Rootkit*. Available: <http://www.rootkit.com/project.php?id=12>
- [219] (Jul 2010). *FUTo Enhanced Rootkit*. Available: http://www.openrce.org/articles/full_view/19
- [220] (2011, Jun 2011). *BlindBossKey Pro 1.4.2* Available: http://www.mobile-download.net/Soft/Soft_14986.htm
- [221] M. Rhys. (2008, Jan 2011). *NtOpenProcess hook*. Available: <http://somebastardstolemyname.wordpress.com/2008/10/04/c-ntopenprocess-hook>
- [222] Jetamay. (2007, Jun 2010). *Writing drivers to perform kernel-level SSDT hooking*. Available: <http://www.mpgn.net/forum/31-c-c-programming/67067-programming-drivers-perform-ssdt-hooking.html>
- [223] (Feb 2011). *A *REAL* NT Rootkit, patching the NT Kernel*. Available: <http://biblio.l0t3k.net/rootkit/en/P55-05.txt>
- [224] S. Kim and B. Chanana. (2009, Sep 2010). *An Introduction to VMsafe . Architecture, Performance, and Solutions*. Available: <http://www.vmworld.com/docs/DOC-2406>
- [225] T. Garfinkel and M. Rosenblum, "When virtual is harder than real: security challenges in virtual machine based computing environments," in *Proc. of the 10th conference on Hot Topics in Operating Systems*, Santa Fe, NM, 2005, pp. 20-20.
- [226] F. Baiardi, D. Maggiari, and D. Sgandurra, "PsycoTrace: Virtual and Transparent Monitoring of a Process Self," in *Proc. of 17 th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Weimar, 2009, pp. 393 - 397.
- [227] (Jan 2013). *Context-free grammar* [Wikipedia]. Available: http://en.wikipedia.org/wiki/Context-free_grammar
- [228] J. Brauer, R. Huuck, and B. Schlich, "Interprocedural Pointer Analysis in Goanna," *Electronic Notes Theory Computer Science*, vol. 254, pp. 65-83, 2009.
- [229] E. M. Nystrom, H.-S. Kim, and W.-m. W. Hwu, "Importance of heap specialization in pointer analysis," in *Proc. of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Washington DC, USA, 2004, pp. 43-48.
- [230] E. Bendersky. (2011, June 2012). *pycparser: C parser and AST generator written in Python* Available: <http://code.google.com/p/pycparser/>
- [231] Microsoft. (Feb 2011). *Parallel Extensions*. Available: <http://msdn.microsoft.com/en-us/gg465239>
- [232] GraphViz. (March 2011). *DOT Language*. Available: www.graphviz.org/doc/info/lang.html
- [233] (May 2013). *SLOCCount* [Tool]. Available: <http://www.dwheeler.com/sloccount/>
-

-
- [234] M. Russinovich, D. Solomon, and A. Ionescu, *Windows Internals, 5th Edition*: Microsoft Press, 2009.
- [235] (May 2012). *PoolMon* [Tool]. Available: <http://msdn.microsoft.com/en-us/library/windows/hardware/ff550442%28v=vs.85%29.aspx>
- [236] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 1, 6th Edition*: Microsoft Press, 2012.
- [237] A. S. Ibrahim, J. C. Grundy, J. Hamlyn-Harris, *et al.*, "Supporting Operating System Kernel Data Disambiguation using Points-to Analysis," in *Proc. of 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, 2012, pp. 234-237.
- [238] J. Bonwick, "The slab allocator: an object-caching kernel memory allocator," in *Proc. of the USENIX Summer 1994 Technical Conference*, Boston, Massachusetts, 1994, pp. 6-6.
- [239] *Slab Allocator*. Available: <https://www.kernel.org/doc/gorman/html/understand/understand011.html>
- [240] K. Bhargavan, R. Corin, and d. Fournet, "Secure sessions for Web services," *ACM Trans. Information System Security*, vol. 10, p. 8, 2007.
- [241] M. Bartoletti, P. Degano, G.-L. Ferrari, *et al.*, "Semantics-Based Design for Secure Web Services," *IEEE Trans. Software Engineering*, vol. 34, pp. 33-49, 2008.
- [242] Qi Yu, Xumin Liu, Athman Bouguettaya *et al.*, "Deploying and managing Web services: issues, solutions, and directions," *The VLDB Journal*, vol. 17, pp. 537-572, 2008.
- [243] W. Yan, Z. Jinjing, and W. Huaimin, "Implicit Detection of Hidden Processes with a Local-Booted Virtual Machine," in *Proc. of International Conference on Information Security and Assurance*, Busan, Korea, 2008, pp. 150-155.
- [244] (May 2013). *NX bit* [Wikipedia]. Available: http://en.wikipedia.org/wiki/NX_bit
- [245] T. Mandt, "Kernel Pool Exploitation on Windows 7," in *Proc. of Black Hat*, Arlington, VA, USA, 2011.
- [246] A. Bush. (2007, Jun 2012). *Explorations with adore-ng* Available: <http://ab-rtfm.blogspot.com.au/2007/07/explorations-with-adore-ng.html>
- [247] M. IvanLeFou. (2008, sept 2011). *Stealth hooking : Another way to subvert the Windows kernel*. Available: <http://phrack.org/issues/65/4.html>
- [248] A. Bassov. (2006, July 2012). *Hooking the kernel directly*. Available: <http://www.codeproject.com/Articles/13677/Hooking-the-kernel-directly>
- [249] I. Dobrovitski. (2003, Jun 2012). *Exploit for CVS Double Free() for Linux Pserver*. Available: <http://seclists.org/bugtraq/2003/Feb/42>
- [250] J. Afek and A. Sharabani, "Dangling pointer: Smashing the pointer for fun and profit," in *Proc. of Black Hat Conference*, USA, 2007, pp. 1-1.
- [251] (2014, Jan 2014). *Dangling pointer* [Wikipedia]. Available: http://en.wikipedia.org/wiki/Dangling_pointer
- [252] P. Akritidis, "Cling: A memory allocator to mitigate dangling pointers," in *Proc. of the 19th USENIX conference on Security*, Washington, DC, 2010, pp. 12-12.
- [253] D. Dhurjati and V. Adve, "Efficiently Detecting All Dangling Pointer Uses in Production Servers," in *Proc. of the International Conference on Dependable Systems and Networks*, Philadelphia, PA, USA, 2006, pp. 269-280.
- [254] D. Dhurjati, S. Kowshik, V. Adve, *et al.*, "Memory safety without runtime checks or garbage
-

collection," in *Proc. of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, San Diego, California, USA, 2003, pp. 69-80.

- [255] W. Cui, M. Peinado, Z. Xu, *et al.*, "Tracking rootkit footprints with a practical memory analysis system," in *Proc. of the 21st USENIX conference on Security symposium*, Bellevue, WA, 2012, pp. 42-42.