Improving Digital Forensics and Incident Analysis in Production Environments by Using Virtual Machine Introspection

Benjamin Taubmann

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften

Eingereicht an der Fakultät für
Informatik und Mathematik der Universität Passau

Juli 2019

Gutachter: Prof. Dr. Hans P. Reiser
Prof. Dr. Nuno Miguel Carvalho dos Santos

Abstract

Main memory forensics and its special form, virtual machine introspection (VMI), are powerful tools for digital forensics and can be used to improve the security of computer-based systems. However, their use in production systems is often not possible. This work identifies the causes and offers practical solutions to apply these techniques in cloud computing and on mobile devices to improve digital forensics and incident analysis.

Four key challenges must be tackled. The first challenge is that many existing solutions are not reproducible, for example, because the corresponding software components are not available, obsolete or incompatible. The use of these tools is also often complex and can lead to a crash of the system to be monitored in case of incorrect use. To solve this problem, this thesis describes the design and implementation of *Libvmtrace*, which is a framework for the introspection of Linux-based virtual machines. The focus of the developed design is to implement frequently used methods in encapsulated modules so that they are easy for developers to use, optimize and test.

The second challenge is that many production systems do not provide an interface for main memory forensics and virtual machine introspection. To address this problem, this thesis describes possible solutions for how such an interface can be implemented on mobile devices and in cloud environments designed to protect main memory from unprivileged access. We discuss how cold boot attacks, the ARM TrustZone and the hypervisor of cloud servers can be used to acquire data from storage.

The third challenge is how to reconstruct information from main memory efficiently. This thesis describes how these questions can be solved by employing two practical examples. The first example involves extracting the keys of encrypted TLS connections from the main memory of applications to decrypt network traffic without affecting the performance of the monitored application. The *TLSKex* and *DroidKex* architecture describe two approaches to localize the keys efficiently with the help of semantic knowledge in the main memory of applications. The second example discusses how to monitor and document SSH sessions of potential attackers from outside of a virtual machine. It is important that the monitoring routines are not noticed by an attacker. To achieve this, we evaluate how to optimize the performance of the monitoring mechanism.

The fourth challenge is how to deal with the performance degradation caused by introspection in productive systems. This thesis discusses how this can be achieved using the example of a SIEM system. To reduce the performance overhead, we describe how to configure the monitoring routine to collect only the information needed to detect incidents. Also, we describe two approaches that permit the monitoring routine to be dynamically adjusted at runtime to extract more information if necessary so that incidents can be better analyzed.

Zusammenfassung

Hauptspeicherforensik und dessen Spezialform, die Introspektion von virtuellen Maschinen (VMI), sind leistungsstarke Werkzeuge für die digitale Forensik und können zur Verbesserung der Sicherheit von computergestützten Systemen eingesetzt werden. Allerdings ist der Einsatz davon in Produktivsystemen oft nicht möglich. Diese Arbeit identifiziert die Ursachen dafür und bietet praktische Lösungen an um diese Techniken in Cloud-Computing und auf mobilen Endgeräten einzusetzen um digitale Forensik und Vorfallsanalyse verbessert durchführen zu können.

Dazu müssen vier wesentliche Herausforderungen gelöst werden. Die erste Herausforderung ist, dass viele bereits vorhandene Lösungsansätze nicht reproduzierbar sind, zum Beispiel weil die entsprechenden Softwarekomponenten nicht verfügbar, veraltet oder nicht kompatibel sind. Ebenfalls ist die Verwendung dieser Werkzeuge oft komplex und kann bei fehlerhafter Verwendung zum Absturz des zu überwachenden Systems führen. Um dieses Problem zu lösen, beschreibt diese Arbeit das Design und die Implementierung von *Libvmtrace*, welches ein Framework für die Introspektion von Linux basierten virtuellen Maschinen ist. Der Fokus des erarbeiteten Designs liegt dabei darauf häufig verwendete Methoden in abgekapselten Modulen zu implementieren, sodass diese leicht für Entwickler zu verwenden, zu optimieren und zu testen sind.

Die zweite Herausforderung besteht darin, dass viele Produktivsysteme keine Schnittstelle bereitstellen die für Hauptspeicherforensik und die Introspektion von virtuellen Maschinen genutzt werden kann. Um dieses Problem zu adressieren, beschreibt diese Arbeit Lösungsansätze, wie eine solche Schnittstelle auf mobilen Endgeräten und in Cloud-Umgebungen umgesetzt werden kann die dafür entworfen sind, den Hauptspeicher vor unprivilegierten Zugriff zu schützen. Wir betrachten dazu, wie Kaltstartattacken, die ARM TrustZone und der Hypervisor von Cloud Servern für die Datenakquise von Speicher genutzt werden kann.

Die dritte Herausforderung ist die Fragestellung wie Informationen aus dem Hauptspeicher effizient rekonstruiert werden können. Diese Arbeit beschreibt anhand von zwei praxisnahen Beispielen, wie diese Fragestellungen gelöst werden kann. In dem ersten Beispiel geht es darum die Schlüssel von verschlüsselten TLS-Verbindungen aus dem Hauptspeicher von Anwendungen zu extrahieren, um den Netzwerkverkehr zu entschlüsseln, ohne die Performanz der überwachten Anwendung zu beeinträchtigen. Die *TLSKex* und *DroidKex* Architektur beschreiben dazu zwei Lösungsansätze, um die Schlüssel effizient mithilfe von semantischem Wissen im Hauptspeicher von Anwendungen zu lokalisieren. Das zweite Beispiel diskutiert wie man SSH-Sitzungen von möglichen Angreifern von außerhalb einer virtuellen Maschinen überwachen und dokumentieren kann. Dazu ist es wichtig, dass die Überwachungsroutinen nicht von einem Angreifer bemerkt werden. Um dies zu erreichen, evaluieren wir wie man die Performanz der Introspektion optimieren kann.

Die vierte Herausforderung ist wie man mit Leistungseinbußen, welche durch die Introspektion entstehen, in Produktivsystemen umgehen kann. Die vorliegende Arbeit diskutiert anhand von dem Beispiel eines SIEM Systemens, wie dies erreicht werden kann. Um die Leistungseinbußen zu reduzieren, beschreiben wir wie man die Überwachungsroutine so konfigurieren kann, dass nur Informationen gesammelt werden die benötigt werden, um Vorfälle zu erkennen. Ebenfalls, beschreiben wir zwei Ansätze, die es erlauben die Überwachungsroutine dynamisch zur Laufzeit anzupassen um gegebenenfalls mehr Informationen zu extrahieren damit Vorfälle besser aufgeklärt werden können.

Acknowledgments

First, I thank Hans P. Reiser for supporting me during the thesis and for all the discussion we had. I am really grateful, for his advice and to know that I can always count on him. I also owe him great thanks for giving me the chance to work on a fascinating topic in a research-friendly environment.

Second, I thank Nuno Santos for his valuable feedback and his willingness to support this thesis.

I also want to thank Juan David Parra for sharing the good and bad sides of publishing research. Moreover, I want to thank Stewart Sentanoe for supporting my research by building on top of and extending my implementations. Additionally, I am also very grateful to Siglinde Böck, who has always helped me to cope with non-scientific things in everyday life. Besides all that I want to thank all students and colleagues that contributed to my research: Omar Alabduljaleel, Michael Auer, Alexander Böhm, Dominik Dusold, Christian Frädrich, Miguel Guerra, Manuel Huber, Thomas Kittel, Stefan Kunz, Bojan Kolosnjaji, Henrich Pöhls, and Noëlle Rakatondravony.

Finally, I would like to thank my parents who made it possible for me to study computer science and to arouse my interest in science at an early age.

Contents

Contents vii

1 Introduction 1
1.1 Problem Statement . . . 2
1.2 Main Contributions . . . 3
1.3 Publications . . . 6
1.4 Structure of this Thesis . . . 8

2 Background 9
2.1 Virtualization . . . 9
2.2 The Xen Hypervisor . . . 12
2.3 Digital Forensics . . . 13
2.4 Memory Forensics . . . 14
2.5 Virtual Machine Introspection . . . 16
2.6 LibVMI . . . 18
2.7 TLS Internals . . . 21
2.8 Summary . . . 23

3 An Extensible Architecture For Memory Analysis 25
3.1 State of the Art . . . 26
3.2 Requirements of VMI-based Applications . . . 29
3.3 Design Goals . . . 31
3.3.1 Static Analysis . . . 31
3.3.2 Dynamic Analysis . . . 31
3.3.3 Network Traffic . . . 33
3.4 System Design . . . 34
3.5 Libvmtrace . . . 36
3.5.1 System Monitor . . . 36
3.5.2 Network Monitor . . . 41
3.5.3 Operating System Monitor . . . 42
3.5.4 Library and Process Monitor . . . 43
3.5.5 Plug-ins and Dynamic Reconfiguration . . . 43
3.5.6 Logging . . . 43
3.6 Evaluation . . . 44
3.6.1 Process List Extraction . . . 44
3.6.2 Breakpoint Performance . . . 44
3.6.3 Return Value . . . 44
3.6.4 System Call Tracing . . . 45

3.6.5 Process Monitor . . . 46
3.6.6 Accessing Virtual Addresses that are not Present in Physical Memory . . . 46
3.6.7 Stealthiness . . . 47
3.6.8 Network Tracing . . . 47
3.6.9 Compliance with Principals of Digital Forensics . . . 48
3.7 Summary . . . 49

4 Data Acquisition 51
4.1 State of the Art . . . 52
4.1.1 Main Memory Access on Mobile Devices . . . 52
4.1.2 VMI in Cloud Computing Environments . . . 53
4.2 Improving Cold-boot Based Data Acquisition . . . 55
4.2.1 System Design . . . 56
4.2.2 Implementation . . . 56
4.2.3 Evaluation . . . 58
4.3 Towards ARM TrustZone Based Monitoring . . . 61
4.3.1 Threat Model and Assumptions . . . 61
4.3.2 System Design . . . 61
4.3.3 Implementation . . . 62
4.3.4 Evaluation . . . 63
4.4 Bringing VMI to Cloud Environments . . . 65
4.4.1 Threat Model and Assumptions . . . 65
4.4.2 System Design . . . 66
4.4.3 Implementation . . . 67
4.4.4 Evaluation and Discussion . . . 69
4.5 VMI and Live Migration . . . 74
4.5.1 System Design . . . 75
4.5.2 Implementation . . . 79
4.5.3 Evaluation . . . 81
4.6 Summary . . . 83

5 Information Retrieval 85
5.1 State of the Art . . . 86
5.1.1 Decryption of TLS Communication . . . 86
5.1.2 SSH Honeypots . . . 88
5.1.3 Stealthiness of VMI . . . 88
5.1.4 Information Retrieval from Memory . . . 89
5.2 TLSKex: Content-based TLS Session Key Extraction from Virtual Machines . . . 91
5.2.1 System Design . . . 91
5.2.2 Implementation . . . 93
5.2.3 Evaluation . . . 96
5.3 DroidKex: Data structure-based Key Extraction from Mobile Phones . . . 99
5.3.1 System Design . . . 99
5.3.2 Implementation . . . 102
5.3.3 Evaluation and Discussion . . . 106
5.4 VMI-based SSH Honeypot . . . 110
5.4.1 Threat Model and Assumptions . . . 110
5.4.2 System Design . . . 111
5.4.3 Implementation . . . 111
5.4.4 Evaluation . . . 113
5.5 Summary . . . 116

6 VMI in SIEM Systems 119
6.1 State of the Art . . . 120

6.2 Threat Model and Assumptions . . . 121
6.3 System Design . . . 121
6.4 Implementation . . . 124
6.5 Evaluation . . . 127
6.6 Summary . . . 128

7 Conclusions 129
7.1 Contributions . . . 129
7.2 Future Work . . . 131

List of Abbreviations 133

List of Figures 135

List of Tables 137

List of Listings 139

Bibliography 141

1 INTRODUCTION

The Golden Age [of digital forensics] is quickly coming to an end.

Simson L. Garfinkel, 2010

Digital forensics has become an important and challenging part of many criminal investigations [CNB17]. It can be divided into *post-mortem analysis* and *live forensics* [Sic11]. While post-mortem analysis mainly operates on persistent memory such as hard disks, live forensics operates on ephemeral data typically stored in CPU registers and main memory. The main application of *post-mortem analysis* is to gather evidence from systems after an incident already happened, e.g., by taking snapshots from a hard disk to reconstruct deleted files. Today, these systems include not only desktop PCs, but also internet of things devices, such as smartphones and watches, fitness trackers, drones, speakers, and more. *Live forensics* is used to extract volatile data from running systems that does not manifest in persistent memory. A commonly used technique of live forensics is main memory forensics. Main memory forensics analyzes the content in main memory and, in addition to its application to live forensics, can also be used to proactively learn more about current or future attacks by collecting evidence from running systems, e.g., in security information and event management (SIEM) systems, honeypots or dynamic malware analysis.

Main memory forensics analyzes the state of a running operating system from the outside, e.g., by using a hypervisor that provides access to the system state of a virtual machine, i.e., the contents of the main memory and CPU registers. In general, memory forensics is often performed on snapshots that are not changed during analysis. A snapshot of memory can be taken in many ways, but it depends on the system being analyzed what data acquisition method is available. For example, it can be acquired by in-guest agents [Syl+12], cold boot attacks [Hal+09], by FireWire [Mar07], or by using the hypervisor to store the main memory of a virtual machine.

Memory forensics can be divided into *static* and *dynamic* analyses[1]. Static analysis is triggered by external events, such as a timer, or by a forensic analyst, and retrieves information from a snapshot or a running virtual machine. Dynamic analysis is triggered when a certain point in the control flow of the analyzed system is reached, e.g., to extract the parameters of a function call.

Virtual machine introspection (VMI) is a dynamic analysis performed on running virtual machines. It benefits from the *isolation* of the analyzing and the analyzed system provided by hardware virtualization and the hypervisor. It prevents an attacker with full control over the analyzed

[1] Jain et al. [Jai+14] defines them as asynchronous and synchronous to the execution of the analyzed system. Payne et al. [Pay+08] defines them as passive and active.

system from disturbing or attacking the analyzing system, e.g., by turning the monitoring off. In addition, it provides evidence of high value because it has an *untainted view* of the system state, making it difficult for attackers to hide information, as opposed to in-guest agents who need to trust the operating system, which could be manipulated by a (kernel-level) rootkit. Finally, VMI can be used to extract *ephemeral* data such as cryptographic keys required to decrypt network connections or persistent storage. Thus, More et al. [MT14] stated that "VMI, which has its roots in cloud-based technology virtualization, has the potential to change the security implementation in cloud environments".

1.1 Problem Statement

Although memory-based forensic approaches have several advantages, they also present a number of challenges in practice. This thesis aims to tackle the following four challenges.

(1.) *How should a virtual machine introspection frameworks be designed to be robust and powerful to help analysts develop application-specific information extraction tools?*

The first challenge is the lack of frameworks for virtual machine introspection. Since basic functions for the introspection of virtual machines (e.g., the handling of software breakpoints) are often complex and small implementation errors can lead to undefined behavior or even system crashes of the analyzed system, it is important to implement and test these functions once very well to be able to reuse them for the development of new applications. However, most of the available software solutions do not provide an easy-to-use interface for this purpose. For example, LibVMI does not provide an API for tracing (user-space) function calls. It only provides very rudimentary functions for handling low-level events generated by the CPU such as traps caused by software breakpoints. In addition, available software components for virtual machines introspection often only work with outdated software versions or are very complicated to use. Hence, the first research question is how to design a powerful VMI framework that can be used to rapidly develop new VMI applications.

(2.) *How to access the memory of production systems such as cloud environments or mobile devices while preserving the security of the overall architecture?*

The second challenge is the process of data acquisition. Since main memory can contain confidential information, hardware manufacturer of mobile devices try to protect it from unauthorized access as good as possible [Beu18], e.g., by using biometric data for authentication [Cox18] or by implementing a secure boot process that prevents booting, installing and using forensic analysis tools [CNB17; Win08]. On the other side, software vendors encrypt the data of their customers to protect it from unauthorized access [Ken18]. This is good for users, but it complicates forensic analysis that requires obtaining user-related data.

In public cloud environments, we face a similar problem, because they are not ready for virtual machine introspection. Legitimate cloud tenants who want to take advantage of VMI-based security solutions cannot implement them because there is no interface that gives them access to the main memory of their virtual machines required for VMI. The Xen hypervisor provides such an interface. However, this is only available in the most privileged domain Dom0 and grants access to all virtual machines running on this system. This interface is not enabled in common unprivileged virtual machines, to protect other virtual machines running on the same system from unauthorized access. Thus, the challenge of data acquisition is also a problem of implementing an access control mechanism around the VMI interfaces that grants only legitimate cloud tenants access to their virtual machines and not to others.

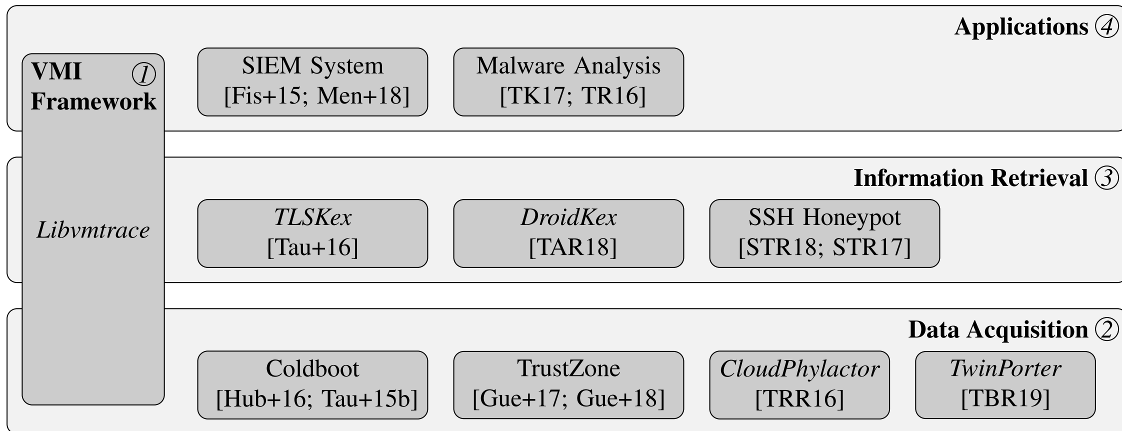


Figure 1.1: Contributions of the thesis

③. *How to locate and extract information efficiently in main memory when there is little or no semantic knowledge?*

The third challenge is information retrieval from the acquired data. It can be split into two sub-problems: the semantic gap problem and the performance of the data extraction. The semantic gap is the problem of extracting high-level information from low-level data sources [Dol+11]. This knowledge is required to locate data structures in memory and to parse/extract the information they hold. If the data structure layout is not given or slightly incorrect, e.g., due to updates of the monitored software, these approaches often fail to retrieve the correct information. Additionally, for live forensics, it is crucial that this process is implemented fast to minimize the performance impact on the analyzed system when the dynamic analysis requires to pause the analyzed system for the information extraction of transient data.

④. *How can VMI methods be used to improve the data collection in SIEM systems while keeping the overhead low?*

Live monitoring has in most cases a negative performance impact on the analyzed system. However, the overhead depends on many factors and is not always the same for different virtual machine introspection applications. A SIEM system requires to have a low-performance impact on production systems to detect intrusions. However, in case of an incident, it can be justified to perform forensic operations with higher performance impact to obtain more information. Thus, the fourth challenge is how to select VMI-based methods for SIEM systems to detect incidents and how to use expensive VMI-based methods to analyze incidents.

1.2 Main Contributions

The following four contributions address the research challenges. Figure 1.1 gives an overview of the contributions and assigns them to the problems.

The **first contribution** of this thesis is the design of a VMI architecture – *Libvmtrace*. It provides a framework for VMI-based monitoring of Linux based virtual machines and is the basis for several of the proof-of-concept implementations presented in this thesis. *Libvmtrace* encapsulates the functions required to trace the execution of virtual machines so that it is *easier for developers* to use. Additionally, this approach helps to reduce the risk of crashes of the analyzed system caused by errors in the implementation of VMI applications because the tracing methods can be tested independently. Furthermore, the encapsulation also helps to improve the performance of each module individually. Besides that, an additional contribution of *Libvmtrace* is to provide an implementation for *standard tasks of virtual machine introspection* that can be used to build

application-specific monitoring plug-ins. These plug-ins can be *activated and deactivated* on the fly, e.g., when a forensic investigator needs more information or when the virtual machine sends certain network packets. Finally, *Libvmtrace* implements an approach for *injecting commands* to virtual machines and for *accessing memory pages that are not present in physical memory*, e.g., when they are swapped out or not yet mapped. None of the available VMI frameworks provides all of these features.

The **second contribution** of this thesis addresses the research problem of data acquisition in cloud environments and on mobile devices. To tackle that problem for mobile devices, we improve state-of-the-art cold boot attacks for ARM-based systems by minimizing their footprint in memory. State-of-the-art solutions use a fully fledged Linux kernel for the data acquisition, which overwrites the data structures of the previously running system. However, these data structures contain important information for forensic operations. To solve this problem, we present the design of a framework for *cold boot attacks on smartphones with a minimal footprint in memory*. This approach increases the amount of unchanged memory and leaves the data structures of the previously running kernel untouched [Hub+16; Tau+15b]. We describe the implementation of a minimal bootable operating system with the goal of sending requested parts of memory via the serial console to the system running the forensic analysis. Furthermore, we describe our extension for the memory forensic tool volatility that requests memory on demand from a device running our minimal operating system. Thus, only the required data is transferred and not the entire content in the main memory of the smartphone, which would take much longer, especially via the serial console.

On top of digital forensics, we discuss how *virtual machine introspection can be used on mobile devices using the ARM TrustZone* in order to increase the security level of a running mobile device. In contrast to VMI-based approaches on virtual machines, there are no tools or interfaces that could be used to access the main memory of the normal world from the secure world. To achieve that, we propose to implement the interface of the de facto standard tool for VMI-based applications – LibVMI – in the secure world of the TrustZone so that developers can port already existing VMI-based applications to it [Gue+17; Gue+18]. Additionally, we implemented and evaluated that approach.

With the *CloudPhylactor* architecture [TRR16] we present a cloud architecture that *allows cloud tenants to perform VMI-based operations on their virtual machines*. To achieve that, we introduce the concept of monitoring virtual machines (MVM) and production virtual machines (PVM) and describe how the Xen security modules can be used to implement this concept. Monitoring virtual machines can be created and used by regular customers and provide VMI access to their production virtual machines. To achieve that, we implement policies for the Xen security modules that implement the concept of MVM and PVM and the mapping of the virtual machines to cloud tenants. To test whether our approach works in real-world environments, we extend the OpenNebula cloud management with this concept and discuss the performance difference of doing VMI from a monitoring virtual machine and not from the hypervisor.

One shortcoming of the *CloudPhylactor* architecture is that it does not cover the aspect of live migration. With the *TwinPorter* architecture [TBR19] we present a solution for that. *TwinPorter extends the live migration concept of Xen* so that a monitoring virtual machine and a production virtual machine are migrated synchronously to the same target cloud node. This approach ensures a minimal downtime of both virtual machines and guarantees that there is no point in time when the production virtual machine is running without monitoring during the migration.

The **third contribution** addresses the problem of efficient information retrieval based on the example of *extracting TLS session keys from communicating applications* and *reconstructing SSH sessions by monitoring function calls*. Our first approach to extract TLS session keys is the *TlsInspector* [Tau+15a] architecture, which has been improved and extended by *TLSKex* [Tau+16]. Both approaches extract the TLS master secret from the main memory of virtual machines. The contribution of them is that they provide an approach for locating the master secret in memory

that is independent of specific key exchange, encryption algorithm, client/server role and of the application's implementation. To achieve that, they monitor the network traffic of the virtual machine and start the extraction of the master secret after a TLS session is negotiated. To locate the key in memory *TLSKex* uses a novel brute-force approach that tests every byte sequence in memory whether it can be used to decrypt a message of the TLS communication correctly. To improve the performance, we apply several heuristics and optimizations to narrow the address range of where the key is searched to locate it in memory. However, the performance overhead of this brute-force approach is not acceptable for many real-world application scenarios.

The contribution of the *DroidKex* architecture is to improve the performance of the localization of the master secret in memory. To achieve that, the *DroidKex* architecture directly accesses the master secret using the semantic knowledge on how and where the monitored application stores this information. To obtain the semantic knowledge, it derives the data structure layout in advance based on memory snapshot and calculates a path by dereferencing pointers used by data structures. In contrast to *TLSKex*, *DroidKex* operates on Android applications.

In *Sarracenia* [STR18] we describe an SSH honeypot that uses virtual machine introspection to reconstruct SSH sessions. The contribution of the *Sarracenia* honeypot is to implement SSH session monitoring and improve the performance in order to make the monitoring stealthy for attackers so that they do not evade the analysis. *Sarracenia* uses semantic knowledge to extract the payload of SSH connections from the SSH daemon handling a connection. However, instead of reconstructing the data structure layout from memory snapshots like *DroidKex*, it uses the debugging information of the SSH binary. To improve the performance of the information extraction, we discuss which functions of the SSH daemon to monitor to minimize the interceptions caused by the monitoring. For this discussion, we compare the performance of *Sarracenia* with our preliminary architecture [STR17], which reconstructs SSH sessions using the parameters of the read and write system call.

The **fourth contribution** tackles the problem of *selecting VMI-based tracing for SIEM systems to maintain low monitoring overhead*. In [TK17] we discuss an architecture for efficiently analyzing malware samples using cloud resources provided by a private cloud that implements the CloudPhylactor [TRR16] architecture. We use this approach to generate malware traces in order to train a machine learning algorithm that aims to detect malware in virtual machines efficiently. The goal of this approach is to *select features for the detection mechanism that can be traced with minimal costs*.

The *CloudIDEA* [Fis+15; Tau+15c] and *DINGfest* [Men+18] architectures describe two approaches how the *VMI-based tracing in SIEM systems* can be adjusted at run-time in production environments in case of an incident. The contribution of these architectures is their approach to handle the monitoring overhead by keeping the total costs low. The *CloudIDEA* uses light-weight monitoring mechanisms to detect incidents and migrates potentially infected virtual machines to an isolated analysis environment with more resources to keep service level agreements and to ensure that malware does not spread in the cloud environment of a provider. In this environment, heavy-weight analysis mechanisms can be used to investigate suspicious behavior. The *DINGfest* architecture enables the forensic investigator to dynamically reconfigure the tracing by selecting only those tracing mechanisms that are adequate to react to potential infections.

1.3 Publications

Parts of this thesis have been already published in the following publications presented at peer-reviewed conferences and workshops as well as peer-reviewed journals. In particular, these are:

[Tau+15a] Benjamin Taubmann, Dominik Dusold, Christoph Frädrich, and Hans P. Reiser. "Analysing malware attacks in the cloud: A use case for the TLSInspector toolkit." In: *Proceedings of the Workshop on Security in Highly Connected IT Systems*. Sept. 2015

[Tau+15b] Benjamin Taubmann, Manuel Huber, Sascha Wessel, Lukas Heim, Hans P. Reiser, and Georg Sigl. "A lightweight framework for cold boot based forensics on mobile devices." In: *International Conference on Availability, Reliability and Security (ARES)*. Aug. 2015, pp. 120–128. DOI: 10.1109/ARES.2015.47

[Tau+15c] Benjamin Taubmann, Hans P. Reiser, Thomas Kittel, Andreas Fischer, Waseem Mandarawi, and Hermann de Meer. "CloudIDEA: Cloud Intrusion Detection, Evidence preservation and Analysis." In: *EuroSys poster*. Apr. 2015. URL: http://eurosys2015.labri.fr/posters/p37.pdf

[Fis+15] Andreas Fischer, Thomas Kittel, Bojan Kolosnjaji, Tamas K Lengyel, Waseem Mandarawi, Hans P. Reiser, Benjamin Taubmann, Eva Weishäupl, Hermann de Meer, Tilo Müller, and Mykola Protsenko. "CloudIDEA: A Malware Defense Architecture for Cloud Data Centers." In: *Cloud and Trusted Computing*. 2015. ISBN: 978-3-319-26148-5. DOI: 10.1007/978-3-319-26148-5_40

[Hub+16] Manuel Huber, Benjamin Taubmann, Sascha Wessel, Hans P. Reiser, and Georg Sigl. "A flexible framework for mobile device forensics based on cold boot attacks." In: *EURASIP Journal on Information Security* 1 (2016), p. 17. DOI: 10.1186/s13635-016-0041-4

[Tau+16] Benjamin Taubmann, Christoph Frädrich, Dominik Dusold, and Hans P. Reiser. "TLSkex: Harnessing virtual machine introspection for decrypting TLS communication." In: *Digital Investigation* 16 (2016), pp. 114–123. DOI: 10.1016/j.diin.2016.01.014

[TR16] Benjamin Taubmann and Hans P. Reiser. "Secure Architecture for VMI-based Dynamic Malware Analysis in the Cloud." In: *DSN fast abstract*. 2016. URL: https://hal.archives-ouvertes.fr/hal-01316519

[TRR16] Benjamin Taubmann, Noëlle Rakotondravony, and Hans P. Reiser. "CloudPhylactor: Harnessing Mandatory Access Control for Virtual Machine Introspection in Cloud Data Centers." In: *IEEE Trustcom/BigDataSE/ISPA*. Aug. 2016, pp. 957–964. DOI: 10.1109/TrustCom.2016.0162

[STR17] Stewart Sentanoe, Benjamin Taubmann, and Hans P. Reiser. "Virtual Machine Introspection Based SSH Honeypot." In: *Proceedings of the Workshop on Security in Highly Connected IT Systems*. Neuchâtel, Switzerland, 2017, pp. 13–18. ISBN: 978-1-4503-5271-0. DOI: 10.1145/3099012.3099016

[TK17] Benjamin Taubmann and Bojan Kolosnjaji. “Architecture for Resource-Aware VMI-based Cloud Malware Analysis.” In: *Proceedings of the Workshop on Security in Highly Connected IT Systems.* Neuchâtel, Switzerland, 2017, pp. 43–48. ISBN: 978-1-4503-5271-0. DOI: 10.1145/3099012.3099015

[Gue+17] Miguel Guerra, Miguel Correia, Benjamin Taubmann, and Hans P. Reiser. “ITZ: An Introspection Library for ARM TrustZone.” In: *Proceedings of INFORUM.* 2017

[Gue+18] Miguel Guerra, Benjamin Taubmann, Hans P. Reiser, Sileshi Yalew, and Miguel Correia. “Introspection for ARM TrustZone with the ITZ Library.” In: *IEEE International Conference on Software Quality, Reliability and Security (QRS).* July 2018, pp. 123–134. DOI: 10.1109/QRS.2018.00026

[Men+18] Florian Menges, Fabian Böhm, Manfred Vielberth, Alexander Puchta, Benjamin Taubmann, Noëlle Rakotondravony, and Tobias Latzo. “Introducing DINGfest: An architecture for next generation SIEM systems.” In: *Short Paper, GI Sicherheit.* 2018, pp. 257–260. ISBN: 978-3-88579-675-6. DOI: 10.18420/sicherheit2018_21

[TAR18] Benjamin Taubmann, Omar Alabduljaleel, and Hans P. Reiser. “DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps.” In: *Digital Investigation* 26 (2018), S67–S76. DOI: 10.1016/j.diin.2018.04.013

[STR18] Stewart Sentanoe, Benjamin Taubmann, and Hans P. Reiser. “Sarracenia: Enhancing the Performance and Stealthiness of SSH Honeypots Using Virtual Machine Introspection.” In: *Nordic Conference on Secure IT Systems.* Springer. 2018, pp. 255–271. ISBN: 978-3-030-03638-6. DOI: 10.1007/978-3-030-03638-6_16

[TBR19] Benjamin Taubmann, Alexander Böhm, and Hans P. Reiser. “TwinPorter – An Architecture For Enabling the Live Migration of VMI-based Monitored Virtual Machines.” In: *Accepted for publication at TrustCom’19.* 2019

1.4 Structure of this Thesis

This thesis is structured as follows:

Chapter 2 introduces the most important background knowledge for the remaining chapters. It discusses the basic concepts of virtualization and the Xen hypervisor that is the foundation for presented virtual machine introspection mechanisms. Besides that, this chapter discusses the most important aspects of digital forensics that are used for virtual machine introspection and memory analysis.

Chapter 3 describes the design and implementation of an architecture for virtual machine introspection and discusses the requirements of different forensic applications. Moreover, we describe how common virtual machine introspection operations can be implemented and evaluate the performance of our implementation.

Chapter 4 describes different approaches for data acquisition on mobile devices and in cloud computing environments. On mobile devices, we discuss how cold boot attacks can be improved to provide better results and discuss how the TrustZone can be leveraged to enhance the security of a system. Additionally, we introduced the *CloudPhylactor* architecture that allows IaaS cloud customers to use VMI operations on their virtual machines. The *TwinPorter* architecture extends this architecture to support live migration while a virtual machine is monitored.

Chapter 5 discusses the problem of fast and efficient information extraction based on two examples. The first example is the extraction of the TLS master secret from the main memory of communicating applications. We discuss two different approaches: *TLSKex* and *DroidKex*. *TLSKex* uses an optimized brute-force based approach to locate the master secret in main memory, while *DroidKex* directly accesses the information by following pointers from the stack of network related function calls. The second example is the *Sarracenia* architecture that aims at efficiently monitoring an SSH honeypot by extracting SSH sessions from a virtual machine.

Chapter 6 describes how to select VMI-based monitoring and data acquisition mechanisms for SIEM systems.

Finally, Chapter 7 concludes the thesis, summarizes the most important contributions, and presents open research challenges for future work.

2 BACKGROUND

In this chapter, we give the background information required to understand the remaining chapters of this thesis. While specific topics are discussed in the corresponding sections, this chapter gives an overview of generic topics used within more than one section. First, we explain in Section 2.1 the most important concepts of virtualization that are used for virtual machine introspection. Second, we discuss in Section 2.2 the Xen architecture and its most important components. Third, we introduce in Section 2.3 the theoretical foundations for digital forensics. Fourth, we discuss the main concepts of memory forensics (Section 2.4) and virtual machine introspection (Section 2.5). Fifth, the core components of the de-facto library for virtual machine introspection – LibVMI – are introduced in Section 2.6. Finally, in Section 2.7 we describe the internals of TLS-based communication.

2.1 Virtualization

Virtualization is the core concept required for virtual machines and virtual machine introspection. Hence, this section gives a brief introduction to the main principles of traditional virtualization technologies that virtualize a processor[1]. The concept of virtualization can be traced back to the 1960s and has evolved during the decades. It enables that several operating systems can concurrently exist on the same processor [CJ06]. “Virtualization refers [...] to the process of decoupling the hardware from the operating system on a physical machine” [CJ06] and helps to reduce costs of hardware and energy by maximizing the resource utilization. With the proliferation of modern, powerful CPUs with hardware virtualization, many software solutions such as VirtualBox, VMWare Workstation and Xen appeared on the market and opened the way for cloud computing.

Virtual Machine Monitor Types

The component that allocates and manages the resources of virtual machines on a physical machine is called virtual machine monitor (VMM). Already in 1973, Goldberg defined two types of “virtual computer systems” which are still used today [Gol73]:

- *Type 1:* The VMM runs on physical hardware. It is often called bare-metal “hypervisor” (Examples: Xen, VMWare ESX).
- *Type 2:* The VMM is an application in the host operating system and is also called hosted virtualization (Examples: KVM, VirtualBox).

[1] This section does not cover other virtualization approaches such as those used by Docker or the Java virtual machine

Virtual Machine Monitor Properties

The most important properties of a VMM have been defined by Garfinkel and Rosenblum [GR03]:

- *Isolation:* The software running in a virtual machine is isolated from the VMM and can not subvert the VMM or any other virtual machine running on the same physical machine.
- *Inspection:* The VMM has full control over the system state (CPU registers, memory, hard disk, etc.) of a virtual machine. Hence, since the VMM can see everything in a VM it is difficult for an attacker to evade the analysis.
- *Interposition:* The VMM can interpose the control flow of a virtual machine at certain virtual machine operations (e.g. executing privileged instructions).

The property isolation holds only in theory. In the past, there have been several successful attempts where a virtual machine was able to break the isolation [Com15a; Com15b; Com15c; Com15d; Com18a; Com18b; Com18c; Com19].

Requirements for Hardware-based Virtualization

Robin and Irvine [RI00] summarized the requirements discussed by Popek and Goldberg [PG74] that a processor should have so that it can provide hardware-based virtualization.

- *Requirement 1:* The method of executing non-privileged instructions must be roughly equivalent in both privileged and user mode. For example, a processor cannot use an additional bit in an instruction word or in the address portion of an instruction when running in privileged mode.
- *Requirement 2:* There must be a method such as a protection system or an address translation system to protect the real system and any other virtual machines from the active virtual machine.
- *Requirement 3:* There must be a way to automatically signal the VMM when a virtual machine attempts to execute a *sensitive instruction*, e.g., instructions that interfere with the state of the underlying VMM or host OS. It must also be possible for the VMM to simulate the effect of the instruction.

Virtual machine introspection mainly relies on the third requirement to trace the execution of virtual machines from the perspective of the VMM. It allows the hypervisor to investigate the state of the virtual machine whenever the virtual machine executes a sensitive instruction. A frequently used technique to monitor the use of non-sensitive operations such as function calls is to replace a regular instruction with a sensitive instruction temporarily.

Approaches for Virtualization

In 2000, Robin and Irvine concluded that the IA-32 architecture is unvirtualizable since several instructions violate the third requirement as it provides sensitive, unprivileged instructions that do not generate an interrupt or an exception [RI00]. Similarly, Penneman et al. [Pen+13] analyzed the virtualizability of the ARMv7-A architecture and stated that it is not classically virtualizable. Even so, there are approaches for using virtual machines on those architectures. The following sections discuss some of those techniques.

Hardware-Assisted Virtualization

Hardware-Assisted virtualization requires that the underlying processor implements the requirements mentioned above. Virtual machines running in hardware-assisted environments do not need to be adapted since the CPU is capable of signaling the VMM the use of sensitive instructions. However, it might be useful that the virtual machine implements the drivers for virtual hardware components such as network cards to improve the performance. The most prominent example for hardware-assisted virtualization is the Intel VT-x processor extension, which introduces two new

processor modes (VMX root and non-root) and ten new instructions to manage virtual machines. Guest systems are executed in VMX non-root mode while the hypervisor is running in VMX root mode [Int17; Uhl+05].

VT-x defines two transitions for the context switch between the two modes [Uhl+05]:

- *VM Entry:* the transition from VMX root operation to VMX non-root operation.
- *VM Exit:* the transition from VMX non-root operation to VMX root operation.

For each transition or context switch, the processor state is saved and loaded from the corresponding virtual-machine control structure (VMCS). The VMM defines in the *VM-execution control fields* for which sensitive instructions in the virtual machine the CPU should trigger a transition to the VMM.

Para-virtualization

Para-virtualization does not require that the hardware supports virtualization. *Para-virtualization (PV)* is a virtualization approach that provides a software interface to the guest operating system that is similar to the physical hardware but not equal. Thus, the guest does not directly execute sensitive instructions but requests the hypervisor to do it by invoking a hypercall which implements a similar concept as system calls. Hence, to run a virtual machine para-virtualized, the operating system and its drivers must be adapted to use the *hypercall* ABI of the hypervisor. This approach can provide better performance and lower virtualization overhead compared to hardware virtualization [Bar+03; CXZ08].

Other Virtualization Approaches

Besides, there are also different approaches to virtualize a system which are today mainly replaced by hardware virtualization. Bochs [Law96] is an *emulator* that does not execute the instruction in hardware but emulates them. Emulation is nowadays mainly required for debugging or for testing software on a different architecture. *Binary translation* was used by VMware Workstation and Virtual PC to run unmodified guest operating systems on x86 CPUs without hardware virtualization [AA06]. To achieve that, it scans at run-time the instructions for sensitive instructions and interprets them[2].

[2] VMWare does not support binary translation of 64-Bit guests `https://kb.vmware.com/s/article/1003945?lang=de`, accessed 2019-04-30

2.2 The Xen Hypervisor

Xen [Bar+03] is a type 1 or bare-metal hypervisor and one of the most important VMMs as it is used in many Infrastructure-as-a-Service (IaaS) cloud data centers[3]. Xen supports both hardware-assisted (HVM) and para-virtualized guest (PVM).

At boot, Xen starts the Domain 0 (Dom0) which is a privileged domain that manages other unprivileged domains (DomU). The Dom0 has access to the physical devices and provides virtual devices to DomUs. The guest has to implement a front-end driver and the Dom0 the back-end driver. In general, the Dom0 is a para-virtualized fully-fledged Linux virtual machine and in cloud infrastructures, it contains the node management software.

XenBus

The *XenBus* is used for communication between domains and is mainly used by split drivers of para-virtualization [Suo19]. The XenBus is not designed for data transfer, such as network traffic. Such data should be exchanged via shared memory regions and virtual interrupts to signal new data [LL09]. Each domain can grant other domains access to their memory pages, and the corresponding configuration is stored in the *grant tables* of each domain.

Xenstore

The *Xenstore* is a shared database between all domains [Shi+07]. It allows high-level operations such as reading and writing in a hierarchically organized namespace and is similar to a file system. It can run as a process in Dom0 or as a separate stub domain. The Xenstore can be directly accessed in Dom0 but also from DomUs via the XenBus. The Xenstore implements a discretionary access control (DAC) based access control, and it is possible to grant each domain access to a specific node in the tree.

Xen Access Control

While old versions of Xen do not allow executing VMI related hypercalls in unprivileged domains, the implementation of mandatory access control (MAC) makes it possible to restrict the access between domains in a more fine-grained way, including the possibility to grant selected virtual machines (VMs) more permissions than regular DomUs.

Xen comes since version 4.3 with an implementation of the flux advanced security kernel (Flask) architecture, which uses the Xen security modules (XSM) interface [Cok08]. The Xen security modules control the access and communication of Xen domains, the Xen hypervisor and resources such as devices or memory, but not of applications and files. The XSM Flask implementation uses the same language for defining the policies as SELinux [LS01]. However, with a different set of rules, users, roles and types.

In contrast to DAC, the security of MAC is not defined by the owner of an object but by the "sensitivity [...] of the information contained in the objects" [Com85]. The decision of whether access is granted or not is based on rules (policies) and the security label of an object.

A precondition for MAC is that each subject (e.g., processes) and object (e.g., files, sockets, devices) must be labeled with a *security context* which consists of a user, role, type and optionally the security level/category. The policies are static and must be compiled before they can be loaded to a running system. It is not possible to add new rules or types at run-time without reloading all rules. Policy rules can prohibit the reloading of rules, in which case the application of new rules requires rebooting the system.

[3] This section is based on [TRR16].

2.3 Digital Forensics

Virtual machine introspection and memory forensics are valuable tools for digital forensics. There is a variety of other data sources that are available for digital forensics such as hard disks, storage devices, and network traffic and the analysis for each of the sources have different challenges [Gar10]. Different publications discuss methods of digital forensics [RRK17; Sic11], but in this thesis, we mainly use the definitions of Casey [Cas11].

In 2001 a group of experts defined the term *Digital Forensic Science* at the *Digital Forensic Research Workshop (DFRWS)* [Pal+01]:

> **Definition:** The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations.

Applications

Even so, Casey [Cas11] mainly describes digital forensics in the context of computer crime to provide evidence for law enforcement, digital forensics becomes more relevant for incident response in organizations or companies. They need to analyze and maybe even to report security incidents to authorities. Digital forensics does not seek to improve the security or protection level of a system [Pal+01]. However, it can be used for incident response to analyze past attacks and learn from them which can help to improve the security.

Casey [Cas11] describes the main concepts of digital forensics especially in the context of post-mortem crime investigations. The focus of the German “Leitfaden ‚IT-Forensik‘ ” [Sic11] is on detection and analysis of cyber attacks in company networks. In 2016, ENISA [Eur16] published a summary of the challenges and approaches to analyze cloud incidents.

Principles of Digital Forensics

There is a wide variety of tools and techniques that can be used to deduce evidence based on digital traces. However, they need to follow the principals of digital forensics so that they are suitable for use at court. These principals are soundness, authentication, chain of custody, integrity, objectivity and repeatability [Cas11].

The *forensic soundness* of evidence is preserved by not modifying it during the examination. However, in some situations, it can be acceptable to alter the analyzed object as long as the process is well documented and reproducible. Nevertheless, all modifications should be kept as small as possible. The *authentication* of evidence means that is was not manipulated (during the analysis or later), that it originates from its reported source and that the metadata is correct and accurate. The authenticity does not necessarily require that the acquired evidence is the same as the origin when it is technically not possible. For example, a memory snapshot can never be compared with the original contents in memory when they are continuously changing [Cas11]. The *chain of custody* is an aspect of the authentication and documents the origin of evidence and ensures that no unauthorized modification was made during the examination. Additionally, it is a proof about where the evidence was stored and who accessed it. *Integrity* is another aspect of authentication and makes sure that the evidence was not altered since it was collected. Technically, this can be implemented by using cryptographic primitives. *Objectivity* refers to the interpretation of evidence, which should not be biased by expectations or assumptions of the analyst. Moreover, it should be possible to *reproduce* the results of the analysis that led to a specific interpretation of the original data source.

2.4 Memory Forensics

Memory forensics is applied digital forensics that uses the contents in the main memory of a system as a data source to extract evidence out of it. It is “an important component of a forensic investigation as it can provide a wealth of transient system state information not available in persistent storage” [Osb13]. Usually, it operates on memory snapshots.

Main memory can contain valuable information such as cryptographic key material or confidential information. However, the required information can be hidden among several hundreds of gigabytes with temporal run-time data of programs or even with random data. Thus, for the process of locating information in main memory, it is crucial to restrain the search space, e.g., to the address space of a program.

Advantages and Application Areas

The main advantage and purpose of doing memory forensics is that main memory contains transient system states that are not stored on persistent storage, which can have several reasons. The first reason is that main memory provides better performance than persistent storage and thus it is used to store intermediate results and data structures of programs as well as the instructions that are executed by them. Another reason is that main memory is often considered as more secure than persistent storage since it loses its information quickly after turning a device off[4]. Because of that, key managers and other programs store unencrypted credentials often in plain text in main memory.

Hence, memory forensics is a valuable extension of traditional hard disk analysis. For example, it can provide cryptographic key material, which is for example required to analyze encrypted hard disks. Additionally, memory forensics is used for malware analysis and reverse engineering, e.g., to access packed or obfuscated code which is not available with static analysis only [KII10].

Data Acquisition

The data acquisition is the process of getting all relevant contents in main memory of a system. There are various possibilities to take a snapshot of a running system which can be grouped into three categories.

The first one accesses the memory from within the analyzed system. In most of the cases, it requires high system privileges since most parts of the address space are protected by the operating system against unauthorized access from applications. Thus, the operating system must be involved to access the full address space of a system. If it is compromised, an attacker can be able to interfere with the data acquisition and manipulate the data. A common tool for taking a memory snapshot is the LiME kernel module, which was initially called DMD [Syl12; Syl+12].

The second one accesses the memory from outside via the system bus [CG04; Mar07], debugging interfaces such as JTAG [Gur+15] or using cold boot attacks [Hal+09]. These approaches require that the analyzed system has an exposed interface that allows direct access to memory or that the memory modules can be transferred fast enough to another system and exploit the remanence effect of DRAM.

The third one uses the architecture of VMMs to access the memory of virtual machines from the isolated VMM. In this way, the data acquisition is still executed on the same host but separated from the analyzed system. This approach provides only forensic sound data as long as the VMM is not compromised. This approach will be discussed in the next section.

[4]Coldboot attacks and similar approaches have shown that this assumption is not always valid. Because of that, Tresor and TreVisor [MFD11; MTF12] store the keys required for hard disk encryption in plain in the debug registers of the CPU where they are secure against cold boot attacks.

Information Retrieval

To access and extract information from raw memory it must be known where and how an application stores it. In some easy cases it might be enough to scan the memory snapshot with regular expressions for specific information, e.g., to search for kernel versions or IP addresses. However, this approach has the disadvantage that the time for searching depends on the size of the snapshot and might also produce false positives, e.g., not only the host IP might be found but also the IPs of established connections.

Thus, to correctly identify information, semantic knowledge concerning the data structure layout is required. The identification and localization is one of the main challenges of memory forensics and was defined as the semantic gap problem:

> **Definition:** The semantic gap is the problem of extracting high-level semantic information from low-level data sources [Dol+11].

Jain et al. [Jai+14] suggest distinguishing between the *weak* and the *strong* semantic gap. The weak semantic gap can be bridged with information gained from source code or debugging data to interpret contents in main memory. The strong semantic gap is the problem of interpreting memory where no information about the data structure is given or even worse when they are obfuscated or try to mislead the analysis [Bah+10]. In the last decade, various solutions have been presented that attempt to solve the semantic gap problem in different use cases [HLM15; Jai+14; Pay+08; Xu+17].

It is important to note that due to the semantic gap problem, information retrieval cannot be considered as just data acquisition because it is necessary to analyze and interpret the contents in memory. If an attacker places manipulated data in memory, he can hide information or even fake the results of the retrieval process [Bah+10].

2.5 Virtual Machine Introspection

Virtual machine introspection is a particular form of memory forensics, which operates on virtual machines. Since it does not operate on a single snapshot but can access the contents of the virtual machine it opens the way to new analysis methods. For example, it allows tracing the execution of virtual machines and accessing data that resides only for a short time in memory.

In 2003 Garfinkel and Rosenblum defined the term virtual machine introspection [GR03]:

> **Definition:** We call this approach of inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it virtual machine introspection.

Virtual machine introspection has several characteristics which are shared with memory forensics. Both approaches work on main memory of systems and aim to extract information out of it and methods and programs developed for memory forensics usually also work for VMI.

Since VMI operates on live systems, it has different goals and acquisition strategies compared to standard memory forensics. VMI is not only used to analyze the system state after an incident, but it can also be used to increase the security level proactively. Hence, VMI is a continuous process of monitoring a virtual machine, while memory forensics is mostly applied to snapshots. VMI methods can be either *static* or *dynamic*. Static analysis is executed independently of the control flow of the virtual machine but is triggered by external events such as timers. Similar to traditional memory forensics static VMI-based analysis extracts information from the contents in main memory but only on live running virtual machines.

Dynamic analysis is executed synchronous to the control flow of the virtual machine and triggered by instructions of the monitored virtual machine, e.g., sensitive instructions. To intercept sensitive instructions in a virtual machine the hypervisor and the CPU must be configured, e.g., in the *Processor-Based VM-Execution Controls* data structure of Intel which contains the configuration for those events [Int17].

Compared to other tracing techniques executed within the analyzed system, VMI takes advantage of the functionality of VMMs, resulting in the following properties that make VMI interesting for multiple security relevant use cases:

- *Stealthiness:* VMI is often considered as stealthy since it is not possible to directly detect the monitoring. However, in most of the cases, it can be detected indirectly, e.g., by measuring the timing of intercepted functions or by finding software breakpoints in memory.
- *Untampered View:* Attackers in the analyzed system cannot alter the data which is acquired using VMI.
- *Protection:* Due to the isolation between the analyzed system and the analyzing system, the monitoring tool is secure against direct attacks.

In the past, there have been various works about applications for VMI [HLM15]. This is a short list of possible application areas of VMI:

- *Intrusion detection system*: [BR18b; GR03; Raj+18]
- *Intrusion prevention system*: [Sha+09]
- *Malware analysis:* [Hen+14; Len+14]
- *Memory forensics:* [BSM14; Fou; ZR15]
- *Virtual machines subverting*: [FLH13]
- *Virtual machines management and configuration*: [FL13; FZL14]
- *Honeypots:* [Len+12; STR17]

Every hypervisor provides a different API with different functionality to access and manipulate the system state of a virtual machine from outside. Since this interface is different for each hypervisor, developers need to manually port their VMI-based applications if they want to use it with a different virtualization solution.

VMI uses the same techniques as memory forensics for bridging the semantic gap to access information in main memory. In addition to that, it can trace the execution of a virtual machine by leveraging the use of sensitive instructions which are signaled to the VMM. The most important low-level operations for dynamic analysis are:

- *Writing to main memory:* Writing to the main memory with VMI can be used to modify the control flow of the virtual machine, to inject software breakpoints and arbitrary code. If this operation is used for digital forensics, it must be ensured that the analyzed virtual machine is modified as little as possible and that the modification is documented so that the original operation is not affected.
- *Writing to CPU registers:* The write access to registers can be used for VMI applications, e.g., to change the return value of functions that is sometimes stored in registers.
- *Monitoring of memory access:* Monitoring write access to main memory can be used to trace changes to data structures such as the system call table.
- *Monitoring of register access:* The most important application for this is the monitoring of the $CR3$ register where the operating system stores the pointer to the directory table base (DTB) of the currently active process on this CPU. Every time the scheduler of the operating system dispatches a new process, it modifies the CR3 register. By monitoring this register, it is possible to trace userspace process context switches.
- *Software breakpoints:* Software breakpoints are used to interrupt the control flow at arbitrary positions. A software breakpoint for the Intel architecture is implemented by replacing the original with the `INT 3` instruction, which can be handled by the hypervisor. When this instruction is reached the hypervisor can replace it with the original instruction and continue with the normal operation.

The most common operating system specific, high-level mechanisms for dynamic analysis that leverage the low-level tracing mechanisms are:

- *System call tracing* can be used to monitor the invocation of system calls of user-space applications.
- *Function call tracing* can be used to trace the use of user-space functions, e.g., library calls of libc.
- *Process list extraction:* The process list and the data structures of a process contain important information for virtual machine introspection. This includes for example, the process ID (PID), the memory mappings, the open file descriptors.

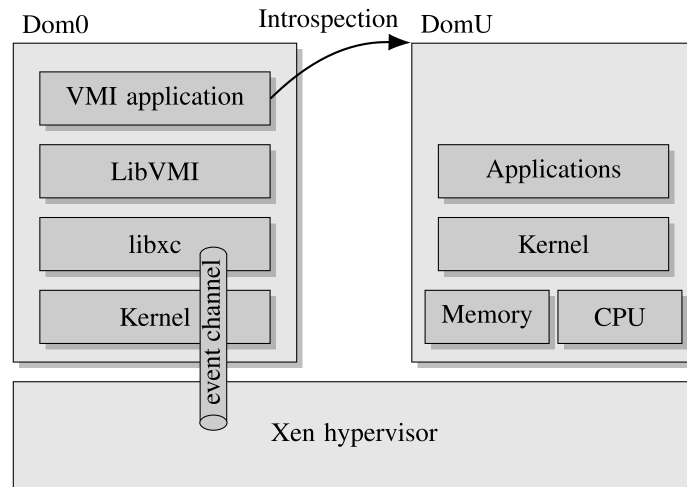


Figure 2.1: Common architecture for virtual machine introspection using Xen and LibVMI

2.6 LibVMI

LibVMI[5] is the defacto standard library for virtual machine introspection. It is actively maintained and provides all necessary functionality to implement VMI based applications. It is the successor of XenAccess [PAL07]. It works best in conjunction with the Xen Hypervisor and Intel CPUs. With KVM and AMD CPUs it does not provide event support, i.e., the ability to listen for traps that are for example caused by software breakpoints.

Listing 2.1 depicts the architecture used by LibVMI in conjunction with Xen. In order to communicate with the Xen hypervisor, it uses the Xen libxc library. The event channel of Xen is used by LibVMI to listen for events, such as caused by software breakpoints.

Functions

LibVMI provides functions for accessing the main memory and CPU registers as well as for tracing software breakpoints, memory (access) events and access to CPU registers. A short list of the most important functions of LibVMI is depicted in Listing 2.1. It shows the functions that are provided to translate virtual to physical addresses of a virtual machine and vice versa. Additionally, there are functions to read and write from/to memory and to access the CPU registers. Moreover, LibVMI allows pausing and resume the monitored virtual machine. Additionally, LibVMI provides functions to lookup the DTB of a PID. The DTB is unique for each process and is often easier to use because it is stored in the CR3 register when a process is running. The mapping of PID to DTB and vice versa is looked up in the data structure of a process in the kernel.

The functions of LibVMI are very low level. It does not provide functions to manage software breakpoints. They have to be inserted and managed by a developer using LibVMI.

Cache

To improve the performance, LibVMI uses several caches to prevent requesting the same information several times from the monitored virtual machine. For example, it caches address translations and PID to DTB lookups. The drawback of the caching approach is that the cache can in some situations contain outdated information, e.g., when a process is terminated. The application developer needs to call the flush operation of the corresponding cache when the information in the cache does not match the state in the monitored system anymore. Knowing whether the cache is

[5] `https://github.com/libvmi/libvmi`, Accessed 2019-06-17

outdated is not always possible in VMI-applications since it would require to monitor all access on the main memory. Thus, it is often better to flush the cache, even it might not be required, when it is necessary to get the current state of the system. On the other hand, the cache should only be flushed if really needed to avoid requesting data again from memory. Thus, the application developer needs to carefully decide when flushing the cache is really necessary.

Event Handling

LibVMI provides functions to trace the execution of a monitored virtual machine. To use it, a developer can register certain events with call back functions. If such an event occurs, LibVMI calls the registered call back function.

The most important event types are:

- Memory events can be used to monitor the read, write and execution operation of an address in main memory.
- Register events can be used to monitor the write access to certain registers, such as the CR3 register.
- Single step events are used to execute single CPU instructions.
- Interrupt events are generated when the CPU executes a software breakpoint.

VMIFS

VMIFS is a tool provided by LibVMI that maps the memory of a virtual machine into a file in the file system using fuse. With this approach the main memory can be used like a normal snapshot file, e.g., with rekall and volitility.

After discussing the most important concepts of memory forensics and virtual machine introspection, we will introduce the core concepts of the TLS protocol that are required for the use case discussion of TLS session key extraction in Chapter 5.

```
1  // initialization and deinitialization
2  status_t vmi_init(vmi_instance_t *vmi, vmi_mode_t mode, void* domain, uint64_t
       init_flags, vmi_init_data_t *init_data, vmi_init_error_t *error);
3  status_t vmi_destroy(prvmi_instance_t vmi);
4
5
6  // functions to access the VM state
7  status_t vmi_translate_kv2p(vmi_instance_t vmi, addr_t vaddr, addr_t *paddr);
8  status_t vmi_translate_uv2p(vmi_instance_t vmi, addr_t vaddr,   vmi_pid_t pid, addr_t *
       paddr);
9  status_t vmi_translate_ksym2v(vmi_instance_t vmi,       const char *symbol, addr_t *
       vaddr);
10 status_t vmi_read_va(vmi_instance_t vmi, addr_t vaddr, vmi_pid_t pid, size_t count, void
        *buf, size_t *bytes_read);
11 status_t vmi_read_pa(vmi_instance_t vmi, addr_t paddr, size_t count, void *buf, size_t *
       bytes_read);
12 status_t vmi_write_va(vmi_instance_t vmi, addr_t vaddr, vmi_pid_t pid, size_t count,
       void *buf, size_t *bytes_written);
13 status_t vmi_write_pa(vmi_instance_t vmi, addr_t paddr, size_t count, void *buf, size_t
       *bytes_written);
14 status_t vmi_get_vcpuregs(vmi_instance_t vmi, registers_t *regs, unsigned long vcpu);
15 status_t vmi_set_vcpuregs(vmi_instance_t vmi, registers_t *regs, unsigned long vcpu);
16 status_t vmi_pause_vm(vmi_instance_t vmi);
17 status_t vmi_resume_vm(vmi_instance_t vmi);
18 status_t vmi_pid_to_dtb(vmi_instance_t vmi, vmi_pid_t pid, addr_t *dtb);
19
20 // functions to handle for events
21 struct vmi_event {
22         ...
23         void *data;
24         event_callback_t callback;
25         uint32_t vcpu_id; /**< The VCPU relative to which the event occurred. */
26         union {
27                 reg_event_t reg_event;
28                 mem_access_event_t mem_event;
29                 single_step_event_t ss_event;
30                 interrupt_event_t interrupt_event;
31         ...
32         };
33         x86_registers_t *x86_regs;
34         ...
35 };
36
37 status_t vmi_events_listen(vmi_instance_t vmi, uint32_t timeout);
38 status_t vmi_register_event(vmi_instance_t vmi, vmi_event_t *event);
```

Listing 2.1: The most important functions of LibVMI to access the system state of a virtual machine and to handle events

2.7 TLS Internals

Transport layer security (TLS) is the successor of secure sockets layer (SSL) and is used to provide a secure communication channel [DA99][6]. TLS uses cryptographic certificates based on asymmetric cryptography for server authentication and – optionally – client authentication. Hence, all decryption attempts based on active man-in-the-middle intercepts with fake certificates can be detected, unless the interceptor has access to the private keys of the original endpoints.

After the authentication phase of TLS, the communicating entities negotiate a symmetric session key (master secret) using RSA encryption, Diffie-Hellman (DH) or elliptic-curve Diffie-Hellman (ECDH) algorithm. Nowadays, DH or ECDH should be the preferred way to negotiate a session key. In contrast to RSA based key negotiation, DH or ECDH ensure that an attacker, who as access to the private RSA key of one party, it is not able to decrypt the corresponding TLS connections. This is because it is not possible to extract the session key from the network traffic. This property is called perfect forward secrecy (PFS).

Finally, the communication between the endpoints is protected with symmetric encryption and message authentication based on symmetric keys derived from the master secret. TLS does not rely on a single fixed cryptographic algorithm; instead, it provides a flexible framework that supports a large variety of encryption algorithms. At the beginning of a TLS session, the endpoints negotiate which algorithms and parameters to use.

TLS records

Internally, TLS is composed of several sub-protocols. The lowest protocol layer is the TLS record protocol, which is used to exchange control and data messages between the communication partners. Each message of the record protocol contains the content type of a record, the TLS version, the length of a data fragment, and the data fragment (compressed, integrity protected and encrypted using the negotiated algorithms). At this layer, only the data fragment is encrypted, whereas all other fields (record type, TLS version, and length) are exchanged in plain text.

Key negotiation and derivation

During the key negotiation process (see Figure 2.2) the client and server select the cryptographic parameters of the upcoming encrypted network session. The client initiates the protocol by sending a TLS record with the content type client hello (CH), and the server responds with a server hello (SH), to negotiate the encryption algorithm and to exchange a client and server random value. Afterwards, the client and server define a premaster secret, for example by using the DH algorithm. In the initial handshake of TLS, no encryption is used, and thus the client and server random values are exchanged in plain text. If keys are renegotiated on an already encrypted TLS channel, the parameters are encrypted with the still active configuration.

The premaster secret, client random, and server random are used to calculate the master secret of a connection. The master secret is used together with the server and client random to compute the derived keys using a pseudo random function (PRF). Typically, they comprise a MAC secret key used to verify the integrity of a TLS record, the data encryption key used to encrypt the payload of a TLS record, and an initialization vector.

In the TLS protocol, a dedicated message is used to signal the transition to new cryptographic parameters. After completing the computation of the master secret, each communication partner sends a change cipher spec (CCS) message to the other endpoint and starts encrypting all subsequent messages using the new cryptographic parameters. The CCS payload message itself, which consists of a single constant byte, is encrypted using the previous parameters. Because the message is sent using the record layer protocol using a dedicated record type, it is possible to detect any CCS message in an encrypted TLS channel without decryption.

[6] This section is based on [Tau+16]

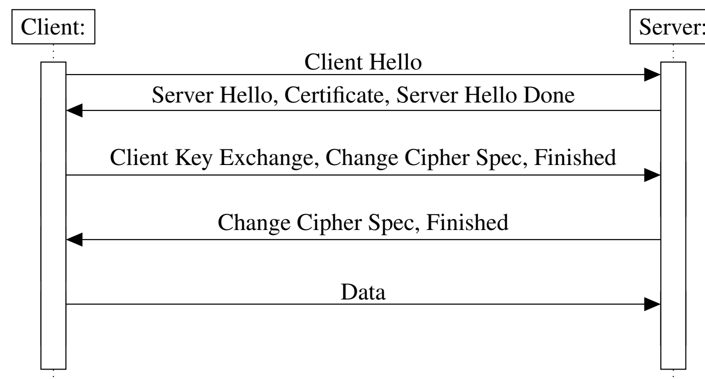


Figure 2.2: TLS handshake

Message Authentication Code Computation

TLS protects the integrity of each TLS record using a message authentication code (MAC). For not encrypted messages or messages encrypted with a standard stream cipher or CBC block cipher, TLS appends an HMAC [KBC97] to the payload data. The HMAC is keyed with a dedicated MAC secret derived from the master secret (as described above) and is computed over the implicit sequence number of the TLS record, the type of the record, the TLS version, the length and the unencrypted data of the record. The receiver can recompute the HMAC of a decrypted record and check whether it matches the sent one, to verify the integrity of a TLS record. For authenticated encryption with associated data (AEAD) ciphers, such as counter with CMC-MAC (CCM) or Galois/Counter Mode (GCM) an authentication tag is used to verify the integrity of a TLS record instead of a separated MAC.

Session resumption

TLS supports the resumption of previously established TLS sessions. In session resumptions, parameters and keys of a previous TLS session are reused instead of negotiating new values, resulting in a faster initial handshake. The session state includes the choice of cryptographic parameters as well as the values for client random, server random, and master secret. The session state can be stored either on the server and is identified using a unique ID that the client includes in its initial CH message (session ID) or on the client (session tickets) [Sal+08].

2.8 Summary

Virtualization is an important technology that empowers the proliferation of cloud computing. Hardware-based virtualization does not require changes to the guest system. However, the CPU needs to be able to trap to the hypervisor when a sensitive instruction is invoked.

Xen is a widespread bare-metal hypervisor and supports hardware and para-virtualization. The virtual machine Dom0 has access to physical devices and provides virtual devices to unprivileged DomUs.

Digital forensics is the process of retrieving evidence reliably from digital sources. The principals of digital forensics are forensic soundness, authentication, chain of custody, integrity, objectivity, and reproducibility.

Memory forensics is one technology used for digital forensics. It operates on main memory and can be used to retrieve information that does not manifest on persistent storage, e.g., passwords and encryption keys.

A particular form of memory forensics is virtual machine introspection. It analyzes the state of a virtual machine by looking at its main memory and the CPU registers. In addition to memory forensics, it can also be used to trace the execution of a virtual machine. To achieve that, normal instructions are replaced by sensitive instructions that cause the CPU to trap to the hypervisor. In that way, the hypervisor can intercept the control flow to analyze the system state of the virtual machine. Afterward, the original instruction is reinserted, and the virtual machine can be continued. LibVMI is the defacto library for virtual machine introspection and provides all necessary functions to implement these steps.

The TLS protocol is widely used to provide secure network connections. For each session, the client and server negotiate new cryptographic parameters. The most important one is the master secret, which is used to derive the session keys. The master secret is usually only stored in main memory of the communicating parties while the connection is active.

3 AN EXTENSIBLE ARCHITECTURE FOR MEMORY ANALYSIS

Virtual machine introspection and memory forensics are valuable tools for a wide range of application areas, such as criminal investigations, intrusion detection, and malware analysis. Because of that, there is already a wide variety of publications and tools for this purpose. However, their implementation is often no longer maintained or not available. Furthermore, such tools often do not support the operating system of the analyzed system or the hypervisor hosting the virtual machines. Hence, using them in production environments or even for research prototypes is often complicated or not possible.

The contribution of this chapter is that we first describe the requirements for a VMI framework that is generic enough to be used in various practical applications. We then discuss how such a framework must be designed in order to meet the requirements for as many application scenarios as possible. Furthermore, we show how the design can be implemented and evaluate it based on the performance of our *Libvmtrace* implementation.

Section 3.1 discusses related approaches. In Section 3.2, we describe the requirements of applications used for VMI and memory forensics. Section 3.3 discusses the most important functionality required by these applications. Next, we describe in Section 3.4 the generic design concepts of an architecture that can be used for VMI and Section 3.5 describes *Libvmtrace* that implements it. Section 3.6 evaluates the implementation of *Libvmtrace* concerning its performance and Section 3.7 summarizes this chapter.

	Linux	Windows	Inject Commands	Access Not-present Mem	Maintained	Dynamic Reconfiguration	Packet Inspection	Hypervisor
Rekall [Coh14]	✓	✓	✗	✗	✓	✗	✗	Snapshots
Volatility [Fou]	✓	✓	✗	✗	✓	✗	✗	Snapshots
LibVMI [Pay12]	✓	✓	✗	✓	✓	✗	✗	Xen, KVM
Drakvuf [Len+14]	✓	✓	✓	✗	✓	✗	✗	Xen
rVMI [PV17]	✓	✓	✗	✗	✗	✓	✗	KVM
libbdvmi [Coj15]	✗	✗	✗	✓	✗	✗	✗	Xen
StackDB [JHE14]	✓	✗	✗	✗	✗	✗	✗	Xen
Vprobes [VMw10]	✓	✓	✗	✗	✗		✓	VMware Workstation 8 & Fusion 3
Libvmtrace	✓	✗	✓	✓	✓	✓	✓	Xen

Table 3.1: Comparison of different virtual machine introspection frameworks

3.1 State of the Art

In this section, we compare the most prominent state-of-the-art frameworks for virtual machine introspection. We discuss them based on

- The support for the analysis of Linux and Windows systems.
- If they allow injecting commands to the guest system.
- If they provide access to memory regions of a process that have a valid virtual address but are not present in physical memory (e.g., swap).
- If they are maintained.
- If they can be dynamically reconfigured at run-time.
- If they can inspect the network traffic and trigger VMI analysis when certain network packets are seen.
- The hypervisor they support.

We chose these features as we required them for the implementation of our prototypes presented in the next chapters. The summary of this discussion is depicted in Table 3.1.

Volatility and Rekall [Coh14; Fou] provide a high amount of different plug-ins to analyze snapshots of Windows, Linux and MacOS based system. The focus of these frameworks is static analysis, e.g., for forensic investigations. Both are written in Python and do not support tracing the execution of a running system. They can be used to perform VMI on a live running system, e.g., by using vmifs of LibVMI [Pay12] but none of them is optimized for performance and hence should not be used to trace the execution of virtual machines.

Lengyel et al. [Len+14] presented in 2014 Drakvuf, which is still actively maintained. The primary use case of Drakvuf is malware analysis on Windows, and it uses LibVMI with Xen. In addition, Drakvuf uses the altp2m approach to hide software breakpoints from guests [Len16]. To achieve that it uses Intel EPT to create a shadow copy of the memory mappings of the guest holding the software breakpoint. If an attacker reads from the memory page with the breakpoint, Drakvuf changes the mapping back so that he reads the original contents in memory. Similarly, if a breakpoint is reached Drakvuf does not replace it with the original instruction but only switches the memory mappings so that the page with the original instruction is active. Drakvuf provides

means to inject code into Windows-based virtual machines. To do so, it uses the Windows functions *CreateProcess* and *ShellExecute* to execute commands. Similar solutions to inject code into a running system are process hollowing [Lei11] and process doppelgänging [LK17], which are mainly used to hide malicious activities in a system[1]. Drakvuf does not do deep packet inspection of the network traffic. Drakvuf implements different tracing plug-ins, but it is required to restart the Drakvuf process to select a different set at run-time.

The tool rVMI [PV17] was published in 2017 by FireEye and is similar to Drakvuf. Instead of using LibVMI and Xen, rVMI uses a patched version of KVM and is designed for malware analysis. This patch of the KVM functions in the Linux kernel makes it hard to use rVMI with newer KVM and qemu versions. In contrast to Drakvuf, the publicly available source code is not actively maintained, and only a little documentation is publicly available. It uses Rekall to parse the contents in memory and thus provides good support to analyze Windows and Linux guests. Since the analysis routines are mainly implemented in python, the performance for tracing a virtual machine can be considered lower as the performance of LibVMI and Drakvuf. However, due to the use of Python and Rekall, it provides an easy to use command line interface based user interface, which can be configured at run-time of the analyzed system.

Libbdvmi [Coj15] provides similar functionality as LibVMI and was published by Bitdefender and is designed for malware analysis. It is written in C++ and does not use the glibc data structures as LibVMI does. The open source code is not as actively maintained and documented as LibVMI. In contrast to LibVMI, it provides functions to map the address space of a virtual machine to the user space of an application using libbdvmi. Similar to LibVMI, it uses the Xen interface to inject traps causing a page fault in order to access not present memory regions. Libbdvmi does not provide any means to interpret the contents in memory to bridge the semantic gap. Neither for Windows nor for Linux. Hence, this needs to be handled by applications using libbdvmi.

Johnson et al. [JHE14] present StackDB, a debugging library with VMI support. It introduces different layers for interpreting the contents in memory to trace the kernel and user-space applications in a virtual machine. They use a unified layer for data acquisition and implement support for ptrace or VMI. In this way, they can use the same analysis code with VMI using (LibVMI and XenAccess) or the kernel debugging interface (ptrace). Hence, they make it easy for developers to implement new monitoring applications. However, they do not provide an interface for orchestrating different tracing mechanisms, such as monitoring network traffic and intercepting system calls at the same time. Additionally, they do not provide means to inject commands or to access data at virtual addresses that are not present in physical memory. The source code is available online, however, it is not maintained anymore since 2017.

Vprobes [VMw10] is the VMI framework of VMware and works with the Workstation and Fusion virtualization solutions. Vprobes uses the programming language Emmet to build VMI applications. Emmet supports synchronous and asynchronous tracing of virtual machines. Additionally, Vprobes supports to use semantic knowledge gained from debugging information and is also able to learn offsets based on function calls of the operating system at run-time[2]. The last available programming reference of Vprobes is from the year 2010. Vprobes can be used together with PktTrace in order to correlate VMI-based analysis with network packets. It is not clear whether Vprobes can be dynamically reconfigured.

Based on that discussion we conclude that there is a wide variety of different tools for VMI-related application scenarios. However, in many cases, they are not well documented, not maintained, or only work with deprecated hypervisor versions. This holds for libbdvmi, StackDB, rVMI and Vprobes. Only Drakvuf is well maintained and under active development. It provides a rich set of

[1] Process hollowing creates a new "innocent" process in suspended state, unmaps the memory and replaces it with the malicious code. Afterward, the process is resumed. Process doppelgänging is similar to process hollowing. Instead of unmapping the memory of a running process it stores the malicious code in a file. To hide traces in the filesystem, it uses the transactions of the NTFS filesystem. After the process is started and before the malicious file is written to memory, the transaction is rolled back. Thus, the malicious process is never written to the file system.

[2] `https://labs.vmware.com/vmtj/intrusion-detection-using-vprobes`, Accessed 2019-07-03

features for the dynamic analysis of Windows. However, it supports the analysis of Linux guests only momentarily and does not combine network analysis with VMI. The support of Linux is especially important for us because we use *Libvmtrace* mainly in application scenarios like cloud computing and on mobile devices. In both cases, the Linux kernel is mainly used.

3.2 Requirements of VMI-based Applications

The application areas where VMI and memory forensics can be used have different requirements. The following paragraphs discuss and summarize them.

Compliance with the principles of digital forensics: One of the most important aspects of research in the area of digital forensics is that data collection and analysis comply with the principles of digital forensics (see Section 2.3). For data acquisition, this means that the data obtained during the acquisition process is not altered by an attacker or analyst, and represents the original state as accurately as possible. The same applies to the analysis process, which must be carried out in a comprehensible and objective manner. In the specific case of VMI, changes to the analyzed system may be required (for example, by inserting breakpoints) to monitor its execution. To ensure that this does not violate the principles of digital forensics, the changes must be documented and kept as minimal as possible. Furthermore, it also means that the integrity of the collected data must be preserved from data acquisition to a possible court hearing.

Access control: Since the contents in main memory hold confidential information such as passwords and keys, it must be protected from unauthorized access. When a new forensic interface is implemented, the developers must ensure that only authorized entities can access it.

Resilience to attacks: Applications that analyze potentially compromised systems must be resilient against attacks targeting the interpretation routine of memory. In the past, it has been shown that these tools can be attacked by placing crafted data in the analyzed system [Bah+10; Ett17]. In general, there is no easy way to prevent that, however, to minimize the potential harm of a system that is overtaken by an attacker, the permissions of an analyzing system must be restricted so that at least no additional system can be affected.

Resilience to software bugs of VMI applications: Digital forensics should not affect the analyzed system in terms of performance and stability. Especially when it is applied to production systems, a crash caused by the monitoring is not acceptable. Because it is not possible to always prove the correctness of analysis software at least the monitoring framework should be implemented in such a way, that bugs in complex analysis tools can never lead to crashes of the analyzed system. For example, a crash is often caused when the analysis tool did not remove all software breakpoint before it was terminated or crashed. In that case, the framework needs to implement a safe shutdown procedure to ensure, that all breakpoints are removed and handled before it is terminated.

Performance: The impact monitoring on the analyzed system should be as little as possible to maintain the quality of the service and keep costs low. Differences in the performance mainly originate from the interception of the execution, either, because the virtual machine is paused to take a consistent snapshot of the system or because of the interception of sensitive instructions that pause the virtual machine during the analysis.

Stealthiness means that an attacker cannot derive information about whether the system is monitored or not. The stealthiness of monitoring is particularly important for use cases where malware behavior is to be analyzed, as it could change its behavior to evade analysis [Bal+10]. There are mainly two ways of how an adversary can detect VMI-based monitoring in the analyzed system. Either by searching for software breakpoints in memory that have been inserted by the monitoring or by measuring the run-time of potentially monitored functions. By using software breakpoints, the execution time of functions is increased by additional context changes and by the time required to analyze this state. Hence, it is possible to detect changes in the run-time if the average execution time is known. The impact on timing can be minimized by improving the performance of interception mechanisms, but currently it is an open research question to make VMI-based monitoring completely stealthy.

Platform independence: Writing VMI-based applications is complex and cumbersome and porting them to different platforms or operating systems should not require to re-implement everything

from scratch. Thus, functionality that is specific for certain architectures should be encapsulated in modules that can be replaced if the application needs to be ported to a different platform. For example, the basic accessor functions such as *read* and *write* should be implemented in the data acquisition layer that can be exchanged depending on the use case. So, in theory, it should not make a difference to extract the process list from a snapshot file or directly from the memory of a virtual machine. Similarly, operating system functionality should be encapsulated in a module.

Expandability and dynamic reconfiguration: Since VMI-based tracing has a significant performance impact, it is vital to activate only required tracing at run-time and to disable non-required tracing methods. Hence, a monitoring framework should let a user activate and deactivate tracings methods at run-time, e.g., to request more detailed tracing when suspicious behavior is detected. Additionally, it should be easy for developers to implement and integrate new tracing mechanisms.

Multiprocessor support: Many systems use more than one CPU, which needs to be considered when the execution of a virtual machine is monitored since multithreading can lead to race conditions.

3.3 Design Goals

In the following section, we introduce the basic functionalities that are required for the most common VMI related use cases.

3.3.1 Static Analysis

Static analysis extracts information from the main memory of the guest system. The most important operations are: reading memory, translating virtual and physical addresses and providing semantic knowledge about data structures. Writing data to main memory does not play an essential role in static analysis.

The *read* operation is the minimal requirement for doing memory forensics. It can be done on live running systems as well as on memory snapshots or paused systems. It requires two parameters: the amount of data to read and the physical or virtual address from where the data should be acquired. A special form of the read operation is implemented by the snapshot routine that stores all available memory of a system.

It is necessary to *translate virtual addresses to physical addresses* and vice versa to access the memory of pointers of a user-space process in the guest system. To translate addresses the page table from the guest system must be read and parsed accordingly.

Moreover, static analysis requires having *semantic knowledge* about data structures to extract the information they hold them from memory. This knowledge can be for instance derived from kernel source code or debugging information from vendors. The following standard routines of static analysis can be implemented using the *read* operation, address translation and semantic knowledge:

- Parse kernel data structures in memory such as the process list or kernel modules
- Get the memory mappings of processes
- Get the offset used for (kernel) address space randomization
- Parse binaries and kernel data structures to retrieve the logical address of functions and the data structure layout used in user space processes and the kernel
- Translate the logical address of a function/symbol to the virtual/physical address of where it is mapped in memory during execution. This requires to know the offsets used for kernel/user-space address space randomization [GL16].

In order to analyze applications that require a certain run-time environment, e.g., the Java virtual machine, an additional layer of interpretation can be required.

3.3.2 Dynamic Analysis

Dynamic analysis uses the routines of static memory analysis and extends them by calling them synchronously to the control flow of the analyzed system. Hence, it can be used to trace the execution of the analyzed system. For VMI related purposes, the following five methods are the most relevant: *interception, control flow manipulation, code injection, access to unmapped memory regions and file system access.*

Dynamic analysis depends on the hypervisor/CPU to trap at sensitive instructions and on the *write* operation that is used to insert breakpoints. Since the write operation changes the object of investigation, it should be handled carefully. In order to create forensic sound evidence, it can be necessary to document the use of this operation and use it only in a pre-defined way, e.g., to insert breakpoints [Cas11].

Interception

The control flow interception is an important mechanism to trace the execution of a system by inserting breakpoints, e.g., to monitor function calls. By using a breakpoint, the normal control flow of the analyzed system should not be affected, except for differences in the timing behavior.

If a breakpoint is inserted to the main memory of a virtual machine, it can be invoked by all process where the corresponding page is mapped. For example, if it is inserted in a shared library, the breakpoint can be reached by many processes. In contrast, a common debugger traces only a single userspace program. To trace only a single process using virtual machine introspection, it must be either checked which process is running, and the events of non-required processes are disregarded, or the breakpoints can be set/unset when the scheduler in the analyzed system dispatches a new process[3]. The same holds for system call tracing, which is not process-bound but monitors all system calls.

Since tracing can produce a significant overhead, a good choice can be to restrain the tracing to specific processes by removing/inserting breakpoint at every process switch. We distinguish between those two modes [STR17]:

- *System-wide* tracing means that the system calls of all processes are traced.
- *Process-bound* tracing intercepts the system calls of only specified processes. To achieve that it inserts/removes breakpoints whenever a new process is scheduled. By intercepting each process context switch, we introduce another overhead.

The decision for one of those two modes depends on their overhead. Process-bound tracing should only be selected when the overhead for monitoring context switches and inserting/removing breakpoints is lower than tracing system calls of not required processes.

Control Flow Manipulation

Control flow manipulation changes the normal operation of a program and does not require to insert any code, but it manipulates values. For example, it can be used to change the control flow by modifying the return value of functions or by modifying the content of variables so that a different branch in the execution is taken.

Code Injection

Code injection is the most general control flow manipulation mechanism and can be used to instruct the analyzed system to execute arbitrary code. It is more complicated than the two approaches mentioned before, as it requires to modify the control flow so that the injected instructions are executed without affecting the original state and control flow. Command execution can be used, for example, to implement a lie detector that compares the system state observed from commands running in a system with the system state extracted with VMI methods.

Accessing Virtual Addresses that are not Present in Physical Memory

A common problem of memory analysis is that some areas belong to the virtual address space of a process that are not present in physical memory, e.g., areas that are not often accessed and swapped out at run-time or areas of binaries/files that have not been loaded, to minimize the required space in physical memory. As a consequence, those areas cannot be directly accessed from memory and are not available for static analysis. Hence, there needs to be a mechanism to access those memory areas.

[3] This approach only works for a single-core CPU. In order to use it on a multi-core CPU the altp2m approach can be used where each processor has its own page table. Thus pages that include the breakpoints can be activated for each CPU individually based on the fact whether it runs a process that should be traced.

File System Access

Forensics often requires to access and analyze files and recover files from storage, which is mostly done by accessing the raw storage medium, e.g., the hard disk, the image or the network storage. However, this approach does not work for file systems that reside only in memory such as *tmpfs* or shared memory. Similarly, files of encrypted storage cannot be accessed, when the cryptographic key is not known.

3.3.3 Network Traffic

The network traffic of a virtual machine can contain information concerning the system state. Additionally, it provides the raw network traffic of connections. Extracting the same information with virtual machine introspection is possible, but can require a higher tracing overhead. Thus, network traffic can be considered as another source of information to augment the process of forensic analysis.

3.4 System Design

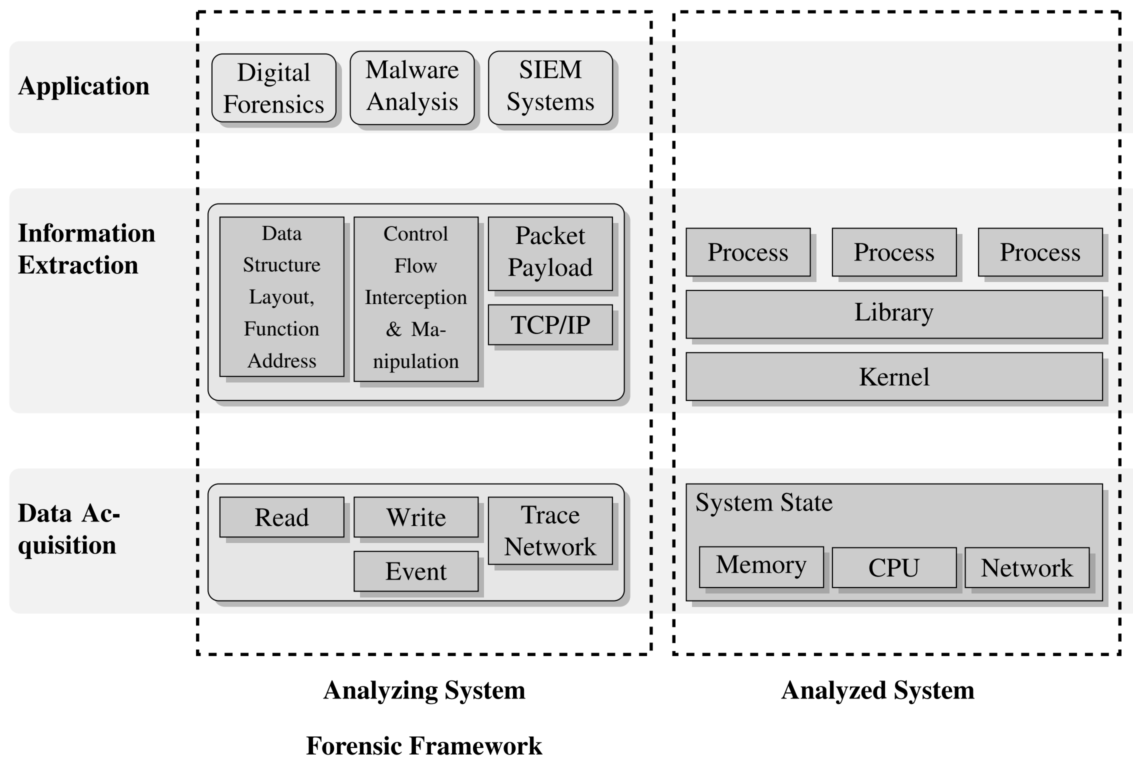


Figure 3.1: Generic design for a forensic framework supporting memory forensics and virtual machine introspection

In this section, we introduce a software architecture that satisfies the application requirements and provides common functionalities of virtual machine introspection. The application requirements, especially the stability and portability requirement, suggest the use of a layered architecture that allows testing and optimizing each layer to improve the performance and stability of the framework. We use the architecture depicted in Figure 3.1 to analyze and trace the kernel, library and user-space processes of the analyzed system.

The layered architecture makes the porting of applications to different platforms and operating systems easier because only the required data acquisition driver or operating system interpreter must be replaced. By providing a programming interface for forensics functions, developers can benefit from it when they build new VMI-based applications.

The core concept of this architecture is designed for virtual machine introspection. However, it can also be employed for other use cases, such as memory analysis from the secure world of the ARM TrustZone or even as an in-guest agent.

Data Acquisition

The data acquisition layer provides access to the state of a system stored in main memory and the CPU registers. Additionally, it provides functions for dynamic analysis and to trace the execution of the analyzed system and its network traffic.

Information Retrieval

The information retrieval layer builts on top of the data acquisition and comprises functions for static and dynamic analysis. For static analysis it provides functions to rebuild high-level information from the acquired system state, e.g., to extract the process list or the memory mappings of a process. In order to bridge the semantic gap, rekall profiles can be used as well as the debugging and meta-information of binaries.

For dynamic tracing, it provides functions to resolve the address of function calls and to inject breakpoints on them. Additionally, it provides an event-based programming interface that invokes call back functions when a breakpoint is reached.

Moreover, similar to the functions for virtual machine introspection, it provides routines that introspect network connections to retrieve basic information from packets such as the source/target IP address and port.

Applications

Applications that are implemented in the highest layer use the API provided by the lower layers to extract information for specific use-cases, such as malware analysis or intrusion detection. To achieve good results they can combine use-case specific information gained from VMI and network analysis to accomplish minimal tracing overhead.

One important aspect is that applications can be implemented as plug-ins. Those plug-ins can be activated and deactivate at run-time, which allows analysts to select only required tracing plug-ins at run-time to lower the overall tracing overhead.

3.5 Libvmtrace

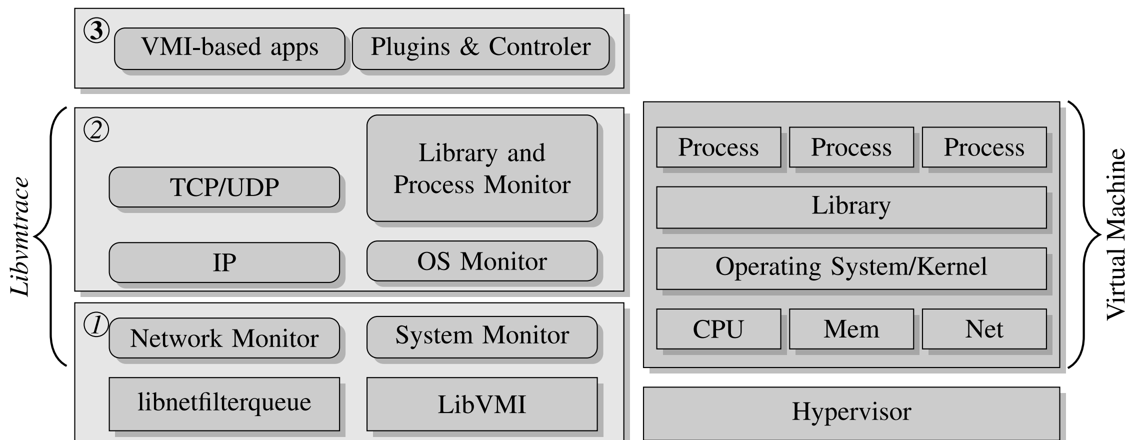


Figure 3.2: Architecture of *Libvmtrace*: the left side represents libvmtrace and the right one the layers of an analyzed virtual machine. The components implemented by *Libvmtrace* have rounded corners. *Libvmtrace* can run in the Dom0 or a dedicated monitoring virtual machine (see Section 4.4). Layer 1 is responsible for data acquisition, layer 2 for information retrieval and in layer 3 are VMI applications.

Libvmtrace aims to implement the architecture described in the previous section. It is mainly designed to be used for VMI and in conjunction with LibVMI and the Xen hypervisor. The focus of *Libvmtrace* is to analyze Linux based virtual machines since the Linux kernel is widely used in cloud data centers and also on mobile Android-based devices. The design of the *Libvmtrace* architecture corresponds to the structure of conventional systems with hardware, operating system, and applications. The different layers of *Libvmtrace* are depicted in Figure 3.2 and discussed in the next sections.

The key aspect of this library is to provide a stable framework that abstracts VMI related operations to make them easy to use, stable without crashing itself or the analysis system and with good performance. For example, LibVMI provides the means to set up and insert software breakpoints. However, it requires several steps to manage them. Thus, the main idea of *Libvmtrace* is to abstract those tasks and to make them easy to use, well tested and easy to adapt for different use cases.

Additionally, *Libvmtrace* should be able to run different tracing mechanisms (plug-ins) at the same time. Hence, it can be necessary to deliver events such as caused by a breakpoint to different plug-ins that monitor the same breakpoint. Starting two tracing applications/processes for a single virtual machine is not possible since they could overwrite the breakpoints of each other and events would not be correctly processed. Thus, *Libvmtrace* implements a monolithic architecture where the tracing is handled in a single process that manages and delivers the events by invoking callback functions of plug-ins.

3.5.1 System Monitor

On the lowest layer of *Libvmtrace* is the *System Monitor*, which implements the data acquisition. As a basis for *Libvmtrace*, we use LibVMI, as it provides a rich set of features for VMI-based analysis. One goal of this architecture is to support different platforms and to provide an exchangeable interface for the acquisition process on multiple systems. Encapsulating the interface of LibVMI with a more generic one that suits other platforms is complex and introduces additional overhead to the system when all data types must be converted at run-time. Thus, *Libvmtrace* currently

only supports the introspection of Xen VMs by using LibVMI[4]. Additionally, the System Monitor synchronizes the access to the virtual machine, manages breakpoints, provides functions to access memory regions that are not present in physical memory, and implements functions to inject code to a running virtual machine.

Locking

Often it makes sense to use different threads in the VMI application for performing memory analysis. The most common case is that one thread is regularly extracting the process list while another one is handling the events of LibVMI. However, some data structures used for VMI, such as the *Libvmtrace* cache for process lists, can be used simultaneously by each thread and therefore require synchronization.

Additionally, some VMI operations must be executed without being interrupted by another VMI thread to avoid that both processes change state in contradicting ways, for example, if one process is stopping a virtual machine and the other one resumes it. This holds for operations that manipulate the state of the virtual machine, e.g., pause or resume it.

To solve these two problems of synchronizing the access to data structures and VMI operations, *Libvmtrace* implements a simple to use locking mechanism in the System Monitor. This lock must be acquired every time a process is using shared data structures (e.g., the process list cache) or when a process wants to use any LibVMI function[5]. After it is finished, the lock must be released.

Breakpoints

Additionally, the System Monitor provides routines that simplify the management of breakpoints to interrupt the control flow of the analyzed system by implementing the complex steps in a software module with an easy to use interface. To achieve that, *Libvmtrace* uses the concept of software breakpoints to intercept the control flow of a virtual machine. In contrast to a hardware breakpoint, a software breakpoint replaces the original instruction in memory with the `INT 3` instruction. When the CPU is invoking it in the context of a virtual machine, it traps to the hypervisor, and the VMI-based application that runs the analysis. Afterward, it needs to restore the original instruction and resume the execution.

Software breakpoints can also be used by applications running in the monitored virtual machine, e.g., by debuggers. In that case, the hypervisor can ignore it and has to forward the interrupt to the operating system of the virtual machine. If the interrupt is caused by a breakpoint from the VMI-based monitoring, it must not be injected to the virtual machine.

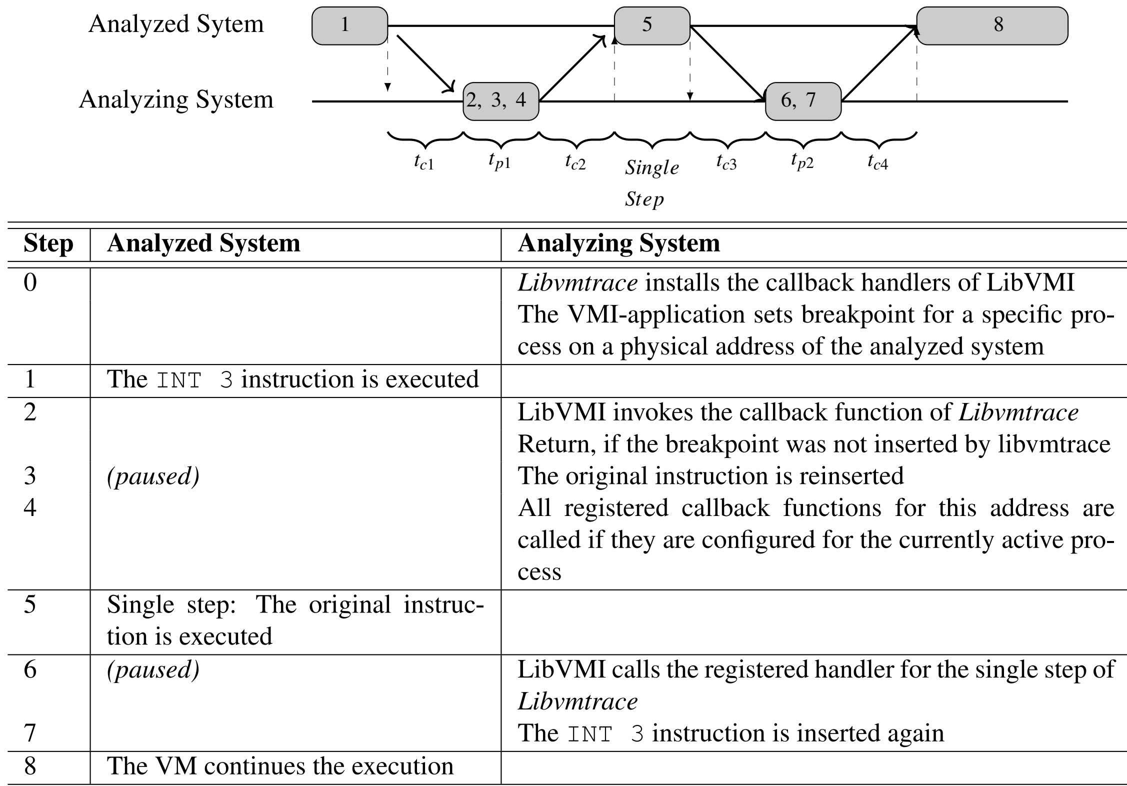
Whenever a breakpoint is invoked, the steps in Figure 3.3 are processed by *Libvmtrace*. The steps 6 to 8 are only required if the breakpoint should be re-inserted. The time t_{bp} when the analyzed system is paused for a breakpoint can be split into these components (see Figure 3.3):

$$t_{bp} = t_{p1} + t_{p2} + t_{c1} + t_{c2} + t_{c3} + t_{c4} \tag{3.1}$$

- t_{p1}: Time to retrieve information from the analyzed system when sensitive instruction is reached
- t_{p2}: Time to retrieve information from the analyzed system after sensitive instruction was executed
- $t_{c1..c4}$: Time spent on context switches and delivering events from the hypervisor the monitoring application

[4]Nevertheless, parts of *Libvmtrace* are used for our *DroidKex* architecture as an in-guest agent on ARM devices to take memory snapshots of Android applications using the kernel debugging interface *ptrace* (see Section 5.3).

[5]This is a coarse-grained locking mechanism that could be enhanced in the future to improve the performance of complex analysis that uses many threads



Step	Analyzed System	Analyzing System
0		*Libvmtrace* installs the callback handlers of LibVMI The VMI-application sets breakpoint for a specific process on a physical address of the analyzed system
1	The `INT 3` instruction is executed	
2 3 4	 *(paused)*	LibVMI invokes the callback function of *Libvmtrace* Return, if the breakpoint was not inserted by libvmtrace The original instruction is reinserted All registered callback functions for this address are called if they are configured for the currently active process
5	Single step: The original instruction is executed	
6 7	*(paused)*	LibVMI calls the registered handler for the single step of *Libvmtrace* The `INT 3` instruction is inserted again
8	The VM continues the execution	

Figure 3.3: Breakpoint handling mechanism and the time that causes the overhead

Our approach of using software-based breakpoints works only under certain conditions. The first limitation is that this approach only works when the monitored system has only one CPU. Otherwise, it can occur that, for example, one CPU is executing the single step operation, while the other CPU executes the same instruction without getting monitored. Another limitation of this approach is that the software breakpoints can be read or even be manipulated by the monitored system. However, the modular architecture of *Libvmtrace* allows implementing and selecting a different breakpoint mechanism without changing the application logic.

Lengyel [Len16] proposed a solution that solves both limitations by employing Intel's extended page tables (EPT). With this approach, each CPU has a shadow copy of the original page tables. Instead of writing and re-writing the software breakpoint, it switches mapping of a page the contains the software breakpoint. Thus, this approach can be applied to multi-core virtual machines. Additionally, if someone is reading from that page, the mapping can be changed to the unaltered view for a single-step, which can be used to hide software breakpoints from applications that search for software breakpoint in memory.

Accessing Virtual Addresses that are not Present in Physical Memory

Libvmtrace implements a mechanism to access memory regions that are currently not present in physical memory, i.e., memory areas that a currently swapped out, e.g., when not enough physical memory is available, or not yet loaded to memory, e.g., when not all parts of a memory mapped file have been accessed.

There are different approaches on how this problem can be solved, e.g., by directly reading from the physical location such as the swap partition or an ELF binary. The advantage of this approach is that it does not require any help from the analyzed system. However, this approach is cumbersome, since it requires first of all having access to the storage where the information is stored (e.g., swap

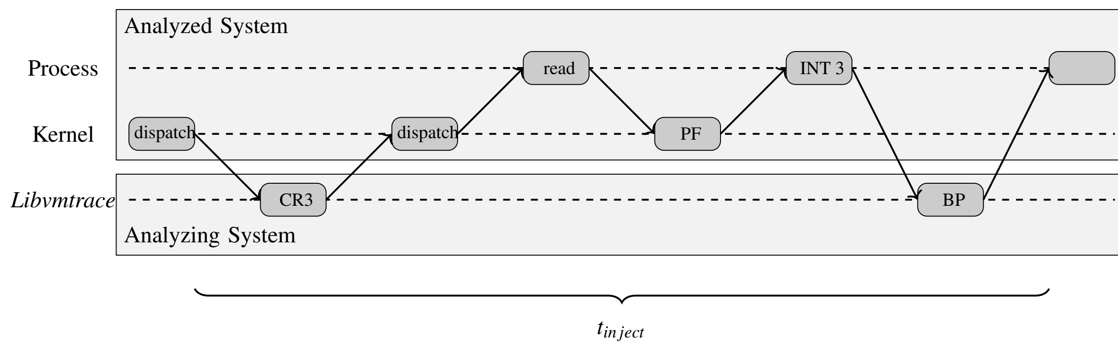


Figure 3.4: Steps required for the injection of the read access

```
1 push rax              ; save the content of register rax
2 mov  rax, 0xdeadbeef ; address of the request page
3 mov  rax, [rax]      ; request the memory
4 pop  rax              ; restore the register rax
5 int  3                ; software breakpoint
```

Listing 3.1: Injected instructions that trigger a page-fault that loads a certain virtual address to physical memory

partition, swap file, memory mapped file, or ELF binary). Additionally, it requires the knowledge to interpret the data, e.g., the structure of a ELF-binary. Encrypted storage makes this approach even more complicated.

Another approach is to inject a page fault trap in the analyzed system using the hypervisor when the user space application is currently running. The requested address is set in the CR2 register and informs the operating system which address to load. When the operating system of the analyzed system handles the trap, it loads the required page to memory. This approach has been lately implemented in LibVMI[6].

Libvmtrace inserts a read instruction that accesses the target address into the process to which the address is assigned. This instruction generates a page fault and forces the memory management of the guest operating system to load the requested page into memory. The limitation of this approach is that it requires that the target process is dispatched while the analysis is running. To employ this approach, we have to address the following challenges:

- Make sure the address is a valid virtual address of that process
- Make sure that the control flow of the target process remains intact
- Find a location in the control flow in the analyzed system where the injected code is executed

Before a read operation is injected, it is necessary to check whether the requested page has a valid virtual address of the process. The validation can be done either by checking the memory tables of the process or by checking whether it is in the area of the mapped memory mappings in *task_struct* of the process. Inserting an operation that reads from a non-accessible address can cause a SEGFAULT in the corresponding application.

[6] https://github.com/libvmi/libvmi/commit/34ec2e5df0c0d0eba4d835dae8fa49f38215c440, Accessed 2019-07-02

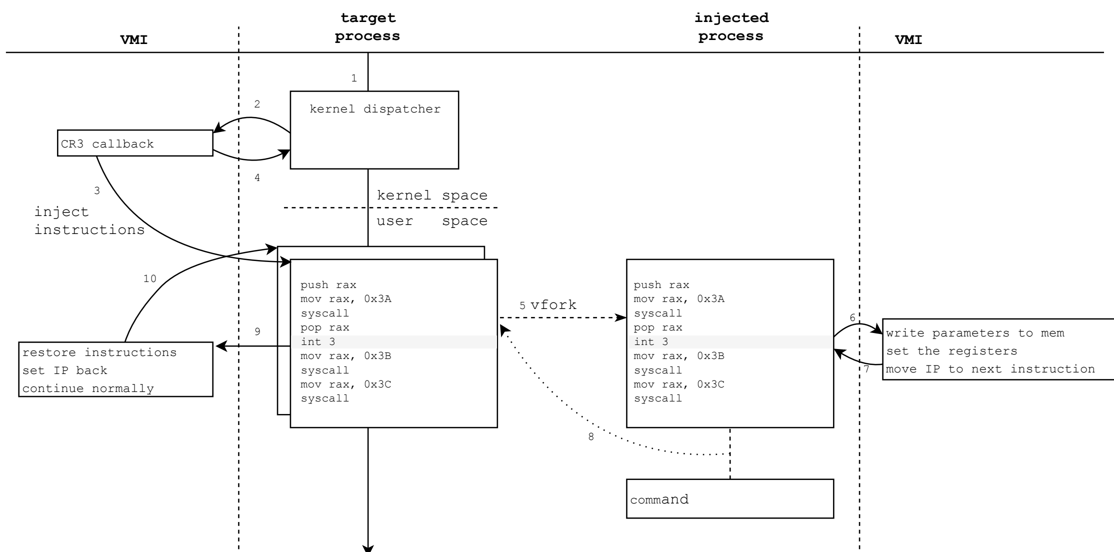


Figure 3.5: *Libvmtrace*: steps required for injecting code into the context of a running process

To ensure that the control flow of the target process remains intact, we save the state of the CPU that is modified by the code injection and restore it later. Otherwise, it leads to undefined behavior or crashes. To achieve that, we surround the actual read operation with instructions that save the content of the modified registers to the stack (see Listing 3.1)[7]. Additionally, we restore the original instructions that are overwritten by the injected code after the execution of the injected instructions and set the instruction pointer back to the point where it was intercepted.

To get the location in the target process where the code can be injected, *Libvmtrace* implements an approach that monitors changes to the CR3 register written by the operating system when it dispatches another process. We use the return address in user-space, i.e., the location where the execution resumes, to inject the operations there. As return address we use the value that is stored in the instruction pointer, which is stored in the task structure of a process in the kernel. The instruction pointer points to the next instruction in the user space of a process when it gets dispatched.

In case the process is dispatched due to a signal, the signal handler of the process is invoked first. However, after the signal handler exists, the kernel returns to user space and the injected instruction at the return address is executed [Bar00].

Figure 3.4 depicts all the steps from the dispatching of the target process until the target process continues with its normal operations. The time t_{inject} estimates the time when the analyzed system does not execute its normal operations due to the injection.

Command execution

The command execution is implemented similarly to the injection of the read operation. Instead of using a single *move* instruction, we insert instructions that execute a command. For this purpose the *system* function or a combination of the system calls $fork$ and *exec* can be used. However, in both cases the child process is started with the same address space of the parent. Since the kernel uses a copy-on-write approach, pages are only copied to the new address space when they

[7] This could of course overwrite possibly relevant forensic data. Thus, this approach should only be used when its safe to overwrite contents below the stack pointer, which is usually the case

```
1 push rax               ; store rax
2 mov  rax, 0x3A         ; move the syscall number of vfork to rax
3 syscall                ; invoke the vfork syscall
4 pop  rax               ; restore rax
5 int  3                 ; software breakpoint
6 mov  rax, 0x3B         ; move the syscall number of execve to rax
7 syscall                ; invoke the execve syscall
8 mov  rax, 0x3C         ; move the syscall number of exit to rax
9 syscall                ; invoke exit
```

Listing 3.2: Implanted instructions required to inject a command in a 64-bit Linux virtual machine

are altered. As a consequence, both the child and the parent process have the same VMI software breakpoints in memory, which can result in race conditions when a breakpoint is removed or added.

To avoid concurrency, we use $vfork$ instead of $fork$, which pauses the parent process until it either exits or successfully starts a new process with *execve*. Both functions are system calls that can be called by using the *syscall* instruction so that it is not necessary to know the function address in a library. Only the parameters of the system call must be placed in the address space of the memory, either on the heap or the stack. The parameters are passed as pointers in the registers. In our implementation, we store the parameters for the *execve* call below the stack pointer of the child process, when the breakpoint is reached. Afterward, we set the registers so that the values are passed to the *execve* system call. To handle the error case when the *execve* call returns, we insert an *exit* call at the end. When the breakpoint in the parent process is reached, we restore the original instructions and set the instruction pointer to the start of the replaced instructions. The injected instructions are depicted in Listing 3.2, and the control flow is depicted in Figure 3.5. When the injected process terminates, the parent process will receive the *SIGCHLD* signal. Unless the parent process implements a handler function for this signal, it will be ignored.

The approach of injecting code can be extended to access the contents of the file system, for example, by injecting a process that reads the requested file and maps the contents to memory. Hence, the VMI application can read the contents from the file directly from memory. If the file is too big to be mapped to memory, it must be possible to instruct the injected program to load only certain areas, e.g., by writing to variables or a shared memory region of the process.

3.5.2 Network Monitor

Besides memory access, the data acquisition layer provides the means to monitor the network traffic of virtual machines. Thereby, it is possible to extract information from the network traffic or to trigger memory analysis when certain network packets occur, e.g., to identify the process of a new network connection. We distinguish between passive sniffing and active interception of network packets. The difference between those approaches is that passive sniffing is decoupled from the control flow of the analyzed system, mainly caused by processing time and analysis of the network traffic. Because of that packets are usually analyzed a significant time after the analyzed system sent them. In some situations, this is not acceptable, e.g., when VMI analysis operates on short living data that needs to be executed while a TCP connection is established and data is exchanged. If the analysis is triggered too late the data might be deleted already (see Chapter 5.2).

One approach for achieving a higher level of synchronicity is to forward packets only if they were analyzed and if necessary corresponding call back functions are executed. The downside of this approach is that it can affect network bandwidth and latency. As a consequence, if the analysis takes too long, network connections can expire, and packets are re-transmitted.

The Network Monitor of *Libvmtrace* employs the library libnetfilter_queue to process packets before they are delivered[8]. It uses BPF filters that operate in the kernel and are used to decide whether a packet is forwarded, dropped or requires further analysis [MJ93]. Packets the match the filter rule used by libnetfilter_queue can be analyzed by a user space applications. For this purpose, the user space application waits for the payload of each packet on a certain file descriptor, which is the communication interface with the kernel. The application can then inspect each packet that matches the filter and decide whether the packet should be forwarded or not.

3.5.3 Operating System Monitor

The Operating System Monitor aims to be a generic interface to access and trace the most commonly used data structures and functions of an operating system in main memory, such as the process list and the properties of each process, e.g., the memory mapping, the established network connections, and system calls. This interface is implemented in a separate class for each analyzed operating system.

To retrieve information from the main memory of the operating system, we use rekall profiles instead of manually computing offsets of data structures. Thus, if only slight changes in the data structures are made, e.g., due to updates, it is sufficient to generate and use a new rekall profile.

Additionally, a class implementing the Operating System Monitor interface should implement methods to trace the execution of system calls, since they are operating system specific. System call monitoring is an important mechanism to study the behavior of a system, e.g., for dynamic malware analysis, because system calls are the interface between the operating system and processes of the user space. One approach to invoke a system call from a userspace process is to use the interrupt 0x80 on Intel-based systems to trigger a system calls and store the number of the system call in the rax register. However, in the past multiple approaches have evolved that optimize the performance compared to the interrupt based approach [PSE11]. Since operating systems are often backward compatible, they support the old way of system calls execution. Internally, the same functions are called even when a different calling mechanism is used.

The Linux implementation of the Operating System Monitor sets breakpoints on the function that is called by each system call, and the address of the corresponding function for each system call is derived from the system call table. This approach allows us to selectively trace system calls and extract the parameters at the beginning of a function call.

Another important aspect of system call tracing is the extraction of function parameters. In some cases, it is easy to extract the parameters, when they are integers or bytes and can be stored directly in registers or on the stack (depending on the calling convention). The extraction of complex data types, such as the `struct sockaddr` of the `connect` call, is cumbersome. Since Linux provides numerous system calls and each with different data types as parameters, our implementation extracts only parameters of the most important system calls for the analysis.

To get the return value of a function, a breakpoint should be set to the last instruction of the function. However, determining the address of the last instruction is in most of the cases not easy since a function can return at multiple places in the control flow. To retrieve the return value of system calls, which is stored in the rax register, we set the breakpoint to the next instruction after the function returns. This address is stored by the `call` instruction on the stack when the function is invoked. If a system call returns values in addition to the value in the rax register, a separate handler for the particular system call must be implemented. For example, *Libvmtrace* implements a handler for the `read` system call, as it stores the read values in a buffer of the calling userspace application.

[8] For *TLSKex* we used a different approach that did not yet use libnetfilter_queue. Instead, we implemented a user-space network bridge that reads network packets from the monitored virtual machine, processes it, and afterward injects it to the outgoing device. However, one of the main disadvantages of that approach is that it does not integrate very well to the Xen infrastructure. For example, in our test setup, we need to compute the checksum of each packet in the software bridge because the virtual machine did not correctly set them.

3.5.4 Library and Process Monitor

The library and process monitor provides functions to trace the execution of user space applications and the use of library function calls. This mainly includes resolving the address of functions and the interpretation of the data structures of an application. This is similar to the information extraction and tracing of the operating system, but also has two more challenges:

- Data structures and function symbols must be derived from applications.
- The virtual address of functions in memory must be computed since the linker relocates them to implement address space randomization.

In order to derive the semantic knowledge of the data structures layout, we use the debugging information stored in the DWARF format of executable and linking format (ELF) files, created by the compiler at compile time. This, of course, requires that the application provides this information.

In order to trace a function call and to set a breakpoint on it, the virtual address of the function must be known. The logical address of a function is stored in the ELF file headers in DWARF format [Com+10] and the virtual address of a function in the address range of a process is defined by the linker that loads a binary to memory when a process is started. In order to get the virtual address of a function, the logical address and the mappings of the ELF sections in memory are required. To retrieve the information of where a binary is mapped, we use the information that is stored in the Linux kernel for each process. The approach of using the information of the ELF headers requires that the compiler actually stores it and that it is not stripped for obfuscation. In case of shared libraries, at least the exported functions and variables must not stripped. Otherwise, other processes cannot use them. Often, Linux maintainers also provide the debugging information of ELF headers for the applications in their distribution.

3.5.5 Plug-ins and Dynamic Reconfiguration

Libvmtrace allows building different tracing plug-ins, e.g., to trace function calls of applications and their system calls. These plug-ins can be used for various reasons such as simple system call tracing or extracting the process list periodically. To reconfigure the tracing system *Libvmtrace* implements a controller that receives commands from clients to activate and deactivate tracing plug-ins. Each plug-in needs to implement at least three functions: initialization, de-initialization and the execution of plug-in specific commands.

In Chapter 6, we describe how VMI can be used in the *DINGfest* SIEM system. In this architecture, we use Kafka as a central message broker and storage location. Hence, the communication of *Libvmtrace* with clients is established using the Kafka server. However, other communications channels should be easy to add.

3.5.6 Logging

VMI-based monitoring generates a vast amount of tracing information, and different analysis mechanism might need to access the data. Thus, the traces must be stored for further processing in such a way that the logging does not affect the performance of the tracing and that the data is stored in a generic format that is accessible from different components. To achieve high performance, we separate the tracing from the logging module. Every time a new log entry is produced, it is written to a shared memory region. Then, the log entry is processed by another process that is sending the log entry to a configured target, e.g., to standard output or a Kafka server. The standard target is Kafka since it is optimized for high throughput and different accessors libraries are available so that the data can be forwarded easily to other databases such as ElasticSearch. To make the data accessible for different analysis, we store all log entries in json format.

3.6 Evaluation

To evaluate our architecture, we measure the performance of the standard functions and their impact on the analyzed system. All measurements presented in this section are executed from a monitoring virtual machine in one CPU and 4 GB main memory. The analyzed system is running Ubuntu 16.04.1 LTS, has one CPU and 1 GB of main memory. The virtual machines are pinned to different cores of the physical CPU (Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz) using Xen 4.11.1-pre.

3.6.1 Process List Extraction

The extraction of the process list is a standard feature of our framework and can be performed either asynchronously to the execution of the analyzed system, e.g., to monitor the running process, or as part of a complex synchronous analysis. To access a non-changing state, the analyzing system pauses the analyzed system for the extraction of the process list. Hence, performance losses in the analyzed system are the result. To quantify the slowdown, we measure how long the extraction of the process list takes and how the performance of the analyzed system is affected. To do so, we run a process and measure the time of a function that is doing a compute-intensive operation. Meanwhile, we extract the process list of the analyzed system every 1000, 100 and 10 ms. Without extracting the process list, the computation takes 14.7 s ± 0.04 s on average for ten times. When extracting the process list every 1000, 100 and 10 ms the computation time increases as respectively: 14.7 s ± 0.02 s, 14.8 s ± 0.01 s, and 16.2 s ± 0.02 s. During the measurement, the system was idle and thus the amount of processes can be considered as relatively constant. Hence, the time required to extract the process list can be also considered as constant.

3.6.2 Breakpoint Performance

The use of breakpoints decreases the performance of the analyzed system. The overhead mainly depends on the number of context switches between the analyzed and analyzing system and the time for inserting/removing the software breakpoint to memory.

We measure the overhead caused by our breakpoint mechanism, by comparing the run-time of a function call against the run-time of an intercepted function call. The time t_{bp} quantifies the time when the virtual machine is paused due to a breakpoint. This is mainly due to context switches between the analyzed system and the analyzing system and the time require to inspect the system state. To measure the time t_{bp} we use a program in the analyzed system that executes the `getpid` system call[9] 1,000,000 times and measure the total run-time. We run this program with and without a breakpoint on the system call `getpid`, which can be considered to have a constant execution time and does not depend on input parameters. The breakpoint is always reinserted after it was reached. The breakpoint handler simply returns and does not further analysis. Thus, we can get the minimal time that is required by a breakpoint. For this measurement, we only monitor the invocation of the `getpid` system call and do not monitor retrieve its return value.

The mean run-time in 100 iterations of the program calling `getpid` for 1,000,000 times without using a breakpoint is 0.36 s ± 2.75 ms. We measure a run-time of 270.82 s ± 2.1 s when the system call is intercepted. Hence, each breakpoint delays the execution for $t_{bp} = 0.27ms$.

3.6.3 Return Value

Libvmtrace sets an additional break point on the instruction that is executed after the analyzed function returns, to get the return value of a function call. We measure the impact of inserting an additional break point the same approach as in the preceding section (100 iterations with 1,000,000 `getpid` calls) and extend it to extract the return value stored in the RAX register after the system

[9] For our measurement we use the `syscall` function to invoke `getpid` to avoid that the compiler uses the vdso based function call

	Idle	Web server	Compile	Compute
Top 10	io_getevents 199	gettimeofday 251% ‖ 15809	lstat 192% ‖ 275070	read 3% ‖ 1815
	gettimeofday 75	recvfrom 184% ‖ 11520	rt_sigprocmask 124% ‖ 173207	clock_gettime 2% ‖ 1355
	futex 70	poll 113% ‖ 7125	open 101% ‖ 141354	close 2% ‖ 1252
	poll 42	clock_gettime 113% ‖ 7113	read 57% ‖ 78284	open 2% ‖ 1219
	nanosleep 20	sendto 84% ‖ 5109	close 50% ‖ 67395	fstat 2% ‖ 1158
	clock_gettime 10	access 48% ‖ 2805	rt_sigaction 42% ‖ 59305	lstat 1% ‖ 1116
	select 10	stat 26% ‖ 1523	brk 41% ‖ 55399	mmap 1% ‖ 719
	wait4 9	close 15% ‖ 746	fstat 37% ‖ 48609	stat 1% ‖ 546
	inotify_add_watch 7	read 14% ‖ 724	lseek 34% ‖ 47556	access 0% ‖ 485
	stat 1	fcntl 12% ‖ 709	mmap 34% ‖ 47084	mprotect 0% ‖ 461
All (count)	454	4469% ‖ 59319	97793% ‖ 1267245	77.0% ‖ 15407
All (time)	10 s ± 0 s	45.69 s ± 1.4 s	978.93 s ± 4.71 s	26.1 s ± 0.1 s
Base	10 s ± 0 s	1.77 s ± 17.8 ms	39.88 s ± 159.9 ms	14.71 s ± 16.2 ms

Table 3.2: Overhead caused by tracing all and the top ten most used system calls for each use case. The results show the overhead in time (in percent) and the amount of intercepted system calls.

call returns. When we extract the return value of the `getpid` system call, each function call is on average delayed by $0.56ms \approx 2 \star t_{bp}$. If functions have parameters that are more complex to extract, the overhead becomes bigger.

3.6.4 System Call Tracing

We use the following use cases to measure the overhead caused by tracing specific system calls:

- *Idle system:* we monitor the system only running standard Ubuntu background jobs for ten seconds. In this case, we only measure the regular system call activity of background jobs by counting the amount of invoked system calls.
- *Compute:* we run a program that is performing computation expensive operations. To get the overhead, we measure the execution time of the computation.
- *Compiling:* we connect to the system via SSH, extract a zip file that holds the libvmi source code, and compile it afterward. To get the overhead, we measure the execution time required for all these steps.
- *Web server:* the analyzed system is running apache2, and the content management system WordPress. A client on a different system sends 100 requests using the Apache HTTP server benchmarking tool `ab`. To get the overhead, we measure the time until all requests are processed.

Each of them uses different system calls and hence tracing one specific system call is more expensive in one use case. For example, a web server opens many network connections, while compiling source code does not open any. The measurements in Table 3.2 depict the overhead for intercepting the top ten most expensive system calls and the count of interception. The values are the average of running the use case and the corresponding tracing for ten times.

The ten most expensive system calls and their overhead is depicted in Table 3.2. For the idle use case, we do not measure the overhead in time, since we do not run anything in the virtual machine where we could measure the run-time. We only measure the system for 10 seconds with and without tracing and count the number of system calls.

In the web server use case, we monitor many network related system calls and the monitor of the corresponding file descriptors. In the compile use case, we monitor a different set of used system calls. They are more related to reading and opening files, which is the case when a large number of files are compiled. The web server and the compile use case make great use of system calls. Hence, tracing a single system call can have an overhead of 250% or 192%. Tracing all system calls can have an overhead of almost 4500% or 10,000%.

	Idle	Web server	Compile	Compute
base [time]	10 s ± 0 s	1.77 s ± 17.8 ms	39.88 s ± 159.9 ms	14.71 s ± 16.2 ms
tracing [time]	10 s ± 0 s	10.48 s ± 0.9 s	430.54 s ± 4.2 s	18.86 s ± 1.7 s
tracing [overhead]	N/A	492.56%	979.49 %	28.27 %
tracing [count]	1,110 ± 58	129,585 ± 11,051	5,812,845 ± 9,975	60,327 ± 20,226

Table 3.3: Overhead to the analyzed system when monitoring changes to the CR3 register

In contrast, the compute use case does not require many system calls. Hence, the tracing has only a little impact on the performance.

3.6.5 Process Monitor

Monitoring changes to the CR3 register can be used to trace the currently active process. The amount of userspace context switches depends on the type and amount of currently active processes. To estimate the impact caused by the monitoring, we measure the overhead caused by intercepting write access to the CR3 register for the use cases introduced in the preceding section. Table 3.3 summarizes the measured overhead for these scenarios. Based on these results we conclude that tracing modifications of the CR3 register can be significantly expensive and should only be used when necessary.

3.6.6 Accessing Virtual Addresses that are not Present in Physical Memory

To access memory regions that have a valid virtual address but are currently not present in physical memory, we inject a read access instruction into the corresponding process. In this section, we measure the overhead to the analyzed process that is caused by our approach. These factors mainly have an impact on the performance of the analyzed system:

t_{active} The time until the process gets dispatched

t_{inject} The time to inject and handle the breakpoint directly after the target process gets active (see Section 3.5.1). This time also includes the time to load the requested page to main memory.

t_{parse} The time to parse the *task_struct* data structure of the operating system to find the location where to inject the code

t_{valid} The validation on whether the address is mapped by the application in order to prevent a segmentation fault

To measure the impact on a real system, we use the following test setup: the analyzed system is running a program that first maps a file to memory and afterward performs a mathematical computation in a loop. The analyzing system injects a read instruction to the mapped file while the analyzed system is performing the computation. The hard disk of the virtual machine is a qcow image file that located on an SSD (Samsung SSD 860 EVO 1TB).

In this setup we measure the following times and compute the average over 100 runs:

- The time $t_{parse} + t_{valid}$ that is required for identifying the location where the code is injected and whether it is a valid address. The localization is done before the computation in the analyzed system is started. In our test setup, we measure in the analyzing system 2.958 ms ± 0.044 ms. It is executed asynchronously to the analyzed system and thus only has a little performance impact on it.
- The time from calling the function that injects the read access until the request page is in memory ($t_{activate} + t_{inject}$). It depends on the analyzed system because the process must become dispatched and active. The longer it takes until the scheduler activates it, the longer it takes to inject the read instruction. In our case, the program is active most of the time

since it is doing computation and the rest of the system is idle. In our test setup, we measure in the analyzing system 3.814 ms ± 1.184 ms until a required page is loaded. This time also depends on the performance of the storage where the data is stored.

- The run-time of the computation in the analyzed system without injection is 14.6652 s ± 0.014 s. With injection, we measure 14.6654 s ± 0.009 s. The difference is 0.2 ms ± 25 ms.

In our evaluation, we measure only the time for accessing a single memory page. Our approach can be extended to inject multiple read instructions at the same time, to access more than one page. In that case, the overhead for costly monitoring context switches is only required once.

The results show, that our approach of accessing not present memory regions has only little overhead. To improve performance, the approach could also be extended to load more than a single page to physical memory at once. However, as shown in the previous section, the monitoring of the CR3 register has a significant impact on the performance. Thus, if the target process gets not quickly dispatched the performance impact increases. Additionally, a limitation of this approach is that the target process needs to become active at all. If not, the memory can not be loaded.

3.6.7 Stealthiness

The presence of VMI-based monitoring can be detected in various ways. While a full analysis of the stealthiness has already been discussed [Tuz+18], we only discuss the most important ones that affect the tracing in our architecture.

The easiest way to detect VMI-based tracing is by searching for software breakpoints in the main memory of a VM. Memory events can be used to detect read access to pages that contain breakpoints, to hide them. In that case, the original page can be restored so that the read access does not see the software breakpoint. However, this approach slows down the read access, which can be used as an indicator to detect monitoring.

The timing measurements of this section show that VMI-based tracing adds significant overhead to the monitored system. The analyzed system can detect the overhead by measuring the timing of function or system calls. Hence, timing behavior is currently the best indicator to detect whether a system is monitored. Future work needs to address the problem of minimizing the overhead caused by VMI-based tracing. This could be for example achieved by integrating parts of the VMI code directly into the hypervisor, which would help to minimize the amount of context switches and to reduce the overhead.

3.6.8 Network Tracing

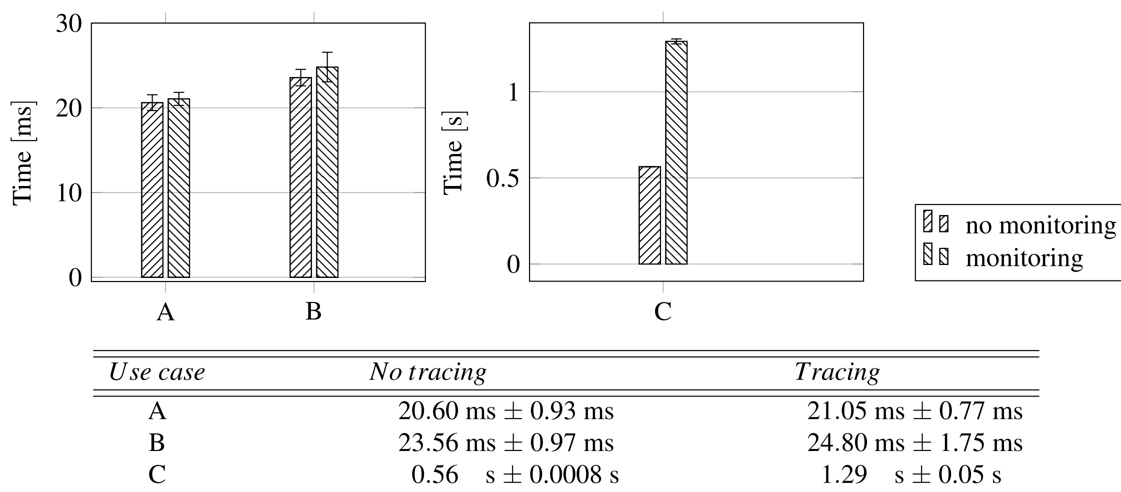
To quantify the overhead caused by introspecting the network, we measure three use cases:

A. We use the webserver benchmarking tool `ab` to establish 1000 https connections sequentially to a static website. For each connection we measure the total time of each connection.

B. Same as A but with 4 parallel connections.

C. We use `curl` to download a large file (60 MB) using https for ten times and measure the time.

For each use case we measure the time with and without monitoring. The packet monitoring is executed on the same host as the client (`ab` and `curl`). The server is located in the same lab network. The introspection routine in *Libvmtrace* calls an empty dummy call back handler for each https packet that matches the filter (here: packets that are sent/received on port 443). The callback handler only returns and does no further inspection on each packet. Hence, we only measure the overhead that is caused by the filter mechanism in the kernel and the time that is used to involve user space applications in the inspection process.



Use case	*No tracing*	*Tracing*
A	20.60 ms ± 0.93 ms	21.05 ms ± 0.77 ms
B	23.56 ms ± 0.97 ms	24.80 ms ± 1.75 ms
C	0.56 s ± 0.0008 s	1.29 s ± 0.05 s

Figure 3.6: Overhead of monitoring https network connections

Figure 3.6 depicts the results of the measurement. Based on them, we can see a inspection overhead of about 2% for short connections (A) and 130% (B) for connections where a file is downloaded. For many monitoring applications the overhead can be reduced by using specific filter rules that accept packets in the kernel so that they are not analyzed in the user space. For example, it may be sufficient to check only the first packets of a connection and ignore the following packets of a session to extract the URL of an http connection. After obtaining the URL of the GET request, a new connection-specific rule can be inserted into the kernel to prevent further examination of that particular connection.

3.6.9 Compliance with Principals of Digital Forensics

The most important principals for a VMI-based framework are forensic integrity and forensic soundness. The term forensic integrity refers to the integrity of the evidence after the acquisition. The associated problem of preserving integrity of log data in cloud environments is for example discussed by Dykstra et al. [DS13]. *Libvmtrace* currently does not implement means that enable a forensic investigator to verify the integrity of acquired traces. However, the logging mechanism could be for example extended to calculate hashes over log messages at run-time.

The term forensic soundness means that the evidence acquisition is reliable, repeatable, and documented. *Libvmtrace* helps to accomplish that because the core parts are implemented in separate modules that can be re-used for different purposes. This helps to minimized software flaws in VMI applications. Additionally, this modular design concepts helps to document the implementation. Besides, forensic soundness also means to keep the modifications to the analyzed system as minimal as possible. The most critical operations are the break point, the code injection, the access to non-present memory, and the write function because they alter the system state. Thus, these functions should be used only if really necessary to keep the modifications as small as possible. To support the forensic soundness of the evidence gained with our framework, *Libvmtrace* could be extended to log the use of these operations.

3.7 Summary

In this chapter, we introduced the *Libvmtrace* framework. The main contribution of it is that it *simplifies the implementation process of VMI applications* for monitoring Linux-based virtual machines on Xen using LibVMI. To achieve that, *Libvmtrace encapsulates frequently used operations* such as inserting and handling breakpoints required for tracing the execution of a virtual machine and to analyze its system state from outside. With this approach, we make sure that it gets *easier to port VMI applications using this framework* to new platforms as only platform specific parts of it must be adapted. The concept of *plug-ins* helps to dynamically reconfigure the tracing and to enable or disable tracing mechanisms at run-time or to trigger certain operations remotely, e.g., to take a snapshot. Moreover, another contribution of *Libvmtrace* is that we propose a mechanism *to access memory regions that are currently not mapped*, e.g., when they are swapped out or not yet mapped to memory. Additionally, in contrast to other VMI frameworks, *Libvmtrace* allows combining VMI with *network monitoring*, so that VMI operations can be triggered when the monitored virtual machine sends/receives certain network packets.

In this chapter, we also *measured the performance* of *Libvmtrace*. Based on these measurements we can show that polling-based monitoring approaches can have only little performance on the impact of the monitored system as long as the polling interval is chosen appropriately. Additionally, the measurements show that the breakpoint mechanism can have a significant impact on the performance of the analyzed system if it is reached often during the analysis (0.27 ms per breakpoint invocation). Hence, VMI-based tracing is not stealthy against an detection mechanism that measures the time of monitored function calls. Similarly, system call tracing can also have a negative impact on the performance (from 77% in the use case that runs computation intensive operations to up to 97793% in the use case that compiles source code). Thus, for production systems that are monitored it is important to trace only system calls that cause low tracing overhead, i.e., are less frequently called. Nevertheless, the injection of read instructions to memory areas that are not present in physical memory has only very little impact on the performance of the analyzed system. Similarly, the deep packet inspection of the network monitoring can be implemented efficiently when many packets are already pre-filtered in the kernel. Thus, network monitoring can support the VMI-based monitoring (see Section 5.2).

The practicability of *Libvmtrace* is demonstrated in the following chapters. They show that the *Libvmtrace* framework can be used to implement new VMI applications, such as an SSH honeypot (see Section 5.4), a TLS session key extraction approach from main memory (see Section 5.2), and for data acquisition in a SIEM system (see Chapter 6).

To support future VMI-based research, we published the source code of *Libvmtrace*[10].

[10] `https://github.com/libvmtrace/libvmtrace`

4 DATA ACQUISITION

Access to the system state is the prerequisite for memory forensics and virtual machine introspection. However, in most of the systems the contents in main memory are protected against unauthorized access with different techniques that hinder the data acquisition process. In this chapter, we discuss approaches that improve the process of data acquisition for mobile devices and cloud environments.

One approach for data acquisition of main memory on mobile devices is to use a cold boot attack. For this approach, the device under analysis is rebooted, and a new analysis operating system is started. Since the memory is often not wiped during the restart, the contents from the previously running operating system are still in memory. Frost [MS13] starts a fully-fledged Linux kernel for the analysis. However, this approach overwrites parts of the previously running operating system, and thus they are not available for the analysis. The first contribution of this chapter is a framework for cold boot attacks on mobile devices that does not overwrite the data structures of the previously running operating system.

The TrustZone is an extension for ARM-based CPUs [PS19], and it introduces two modes: the normal world, which runs the normal operating system, and the secure world, which run security services such as a key store. Memory assigned to the secure world is not accessible from the normal world. This property makes the secure world an exciting target for memory forensics because by executing the monitoring in the secure world it is protected from the normal world. However, there is currently no framework for that purpose. Hence, the second contribution of this chapter is the discussion about how the interface of LibVMI can be ported to the secure world of TrustZone.

Public cloud providers do not provide an interface for VMI-based analysis to cloud tenants. Hence, customers cannot use security solutions that benefit from virtual machine introspection. The third contribution of this chapter is the *CloudPhylactor* architecture, which discusses an approach on how to grant cloud tenants the access to functions required for performing virtual machine introspection on their virtual machines. Additionally, it introduces the concept of monitoring and production virtual machines. To do so, we leverage the Xen security modules to grant a second virtual machine of the same customers the permissions for VMI operations. With this approach, we move VMI from the most privileged operating system to a virtual machine with restricted access. By restricting the permissions of the VMI application we increase the security of the overall system.

The *CloudPhylactor* architecture has a drawback that it does not support live migration of virtual machines while they are monitored. The fourth contribution of this chapter is the *TwinPorter* architecture that addresses this shortcoming by extending the live migration approach of Xen in such a way that the monitoring and the production virtual machine are migrated at the same time.

In Section 4.1, we describe state-of-the-art techniques for the subsequent sections. In Section 4.2, we present the architecture that uses a minimal bare metal application for cold boot attack based data acquisition on mobile devices. The second approach in Section 4.3, is a pro-active method to increase the security level of mobile devices by implementing the monitoring application the ARM TrustZone. In Section 4.4, we describe the *CloudPhylactor* architecture that allows cloud customers to use VMI on their virtual machines in a cloud environment. Section 4.5 extends the concept of the *CloudPhylactor* architecture so that it is possible to live migrate a pair of monitoring and monitored virtual machine to a new cloud node. Section 4.6 summarizes this chapter.

4.1 State of the Art

This section provides an overview of state-of-the-art techniques used for data acquisition on mobile devices and in cloud computing environments.

4.1.1 Main Memory Access on Mobile Devices

Coldboot attacks provide access to the main memory of a system, but since cold boot attacks require restarting the system, they can only be used once to take a snapshot. Thus, cold boot attacks are not feasible for monitoring running systems but instead, they are eligible for situations where only physical access to a device is given, and the credentials or functionality is missing that allows accessing the main memory, for example in crime investigation scenarios. The first cold boot attack was described by Halderman et al. [Hal+09] on desktop PCs. The Frost architecture by Müller et al. [MS13] ported this approach to a mobile device [MS13]. To obtain the memory of the system, they boot a fully-fledged Linux kernel with the lime kernel module [Syl+12]. The problem of their solution is that the analysis system overwrites the data structures of the previously running kernel that are necessary for the analysis, e.g., to get the list of processes and their memory mappings.

Wächter et al. [WG15] study the practicability of common data acquisition on Android devices. As a result they state that the acquisition was only successful on one out of eight devices in the stock configuration. This is mainly caused, because stock phone are not rooted and have a locked boot loader.

Sun et al. [Sun+14] use the secure world of the TrustZone for the data acquisition process. Similarly to coldboot attacks, the TrustZone provides an isolation between the analyzed system and the analyzing system. They leverage the secure world TrustZone for the data acquisition to get a trustworthy snapshot. The memory is sent via the serial port to a remote workstation that analyses the contents. This approach is fine for static analysis that does not intercept the control flow of the normal world. However, they do not discuss how dynamic tracing could be implemented with this approach.

TZ-RKP is the approach used by Samsung Knox architecture to monitor the integrity of the Android kernel running in the normal world from the secure world [Aza+14]. In general, this approach could be also used for forensic investigations and other security related solutions. However, it is a commercial product and the source code is not available.

SPROBES [GVJ14] is similar to TZ-RKP and aims to protect the operating systems in the normal world running on mobile devices using the Android TrustZone. SPROBES can monitor the control at any place in the normal world. They use an instrumentation technique that is similar to the monitoring of sensitive instructions of the Intel architecture by the hypervisor. To achieve that, they inject *smc* instructions that induce a context switch to the secure world. The key aspect of their approach is to ensure that the *smc* instruction is not removed by attackers in the normal world.

T2Droid [Yal+17] uses tracing modules in the normal world to monitor system calls and API calls of Android applications. The integrity of the tracing modules is monitored from the secure world

to make sure that they are not compromised. The traces are processed in the secure world by a classifier to detect malware. The limitation of T2Droid is that the dynamic analysis mechanisms are running in the same environment as Android and are vulnerable to attacks.

Duarte et al. [Dua+18] present SafeChecker. It allows a remote verifier to obtain integrity proofs of function execution in the normal world. The authors use a similar approach as T2Droid to monitor function and system calls of applications and process them later in the secure world. In order to verify the program execution they use the approach of SNARKs [Ben+13].

4.1.2 VMI in Cloud Computing Environments

There have been various attempts that bring VMI to IaaS cloud environments and run forensic operations on virtual machines. LiveCloudInspector [ZR15] extends the cloud management interface so that cloud customers can start a limited set of VMI-based operations, i.e., volatility commands, on their machines. To achieve that it installs the required forensic tools on each cloud node. Whenever a user is issuing a forensic command using the LiveCloudInspector interface, the framework executes the required tool on the corresponding cloud node. The architecture also ensures that a user cannot perform any VMI operation on a VM that does not belong to him. One drawback of this approach is the limited amount of commands and that it does not allow the execution of code from a customer. That is caused by the fact that the operations are executed in the most privileged context of the cloud node that has access to all VMs running on it. Thus, if one user would be able to break out and gain access to the most privileged context, it would be possible to access all data. Another problem of this approaches is that the core part of the VMI functionality is running in the most privileged domain (such as the Dom0 VM on a Xen hypervisor). VMI-based systems heavily depend on the interpretation of data extracted from a guest system [Bah+10]. If an attacker can exploit flaws in routines that interpret the contents in memory, he might be able to gain access to the privileged domain that has full access to all VMs running on the same physical node [CER03; Com04].

CloudVMI [BSM14] provides another approach for VMI in cloud environments. It implements a service that uses remote procedure calls around LibVMI. Thus, a developer can use the same interface as LibVMI, but the actual command is sent over the network to the VMI service. The service is checking the access rights and executes the command afterwards. The advantage of this approach is that applications using LibVMI do not need to be adopted since they use the same API. The downside of this approach is that the network latency slows down that management of synchronous events, e.g., when a breakpoint is reached. Then, the virtual machine is paused for the time of the callback is invoked and the time that is required to encapsulate and send the callback command over the network. The introduced latency has a high impact on the performance of the monitored virtual machine, which is a show stopper for VMI-based monitoring that requires to handle these events in a very short time. The CryptVMI architecture [YC14] has presented a similar idea.

If the cloud provider does not implement any method to access the memory of a virtual machine from outside, nested virtualization can be used. Beham et al. [BVR13] describe a feasible way to install a VMM into a virtual machine and then to perform VMI-based operations on the nested virtual machines with the goal of implementing an intrusion detection system. The drawback of this approach is that the use of nested virtualization and especially the complex memory management harms the average performance on a virtual machine.

Furnace [BR18a] is a framework that allows cloud tenants to run VMI tools on their virtual machines and combines the concept of *CloudPhylactor* and CloudVMI. In contrast to *CloudPhylactor*, it executes the VMI applications of cloud tenants in sandboxes running in the Dom0 instead of in dedicated virtual machines. The sandbox that is running customer code is implemented with Linux namespaces and restricted by SELinux and Seccomp-BPF. To execute VMI operations, it needs to use inter-process-communication to send commands to a privileged container that is running LibVMI. Furnace does not address the problem of VMI and live migration. Besides, Bushouse et

al. [BR18b] discuss how the in-guest agents of the forensics framework GRR [GRR] can be moved to the Dom0 to use virtual machine introspection.

Using VMI in cloud environments has been discussed by various researchers and has been implemented in several ways. For example, Martini et al. [MC12] describes an integrated conceptual digital forensic framework for cloud computing. Poisel et al. [PMT13] discusses how data from the hypervisor can be used for forensic purposes in cloud environments.

Shaw et al. [Sha+14] introduces the concept of forensic virtual machines. They are using the Xen hypervisor to implement and evaluate this approach. However, they do not restrict the permissions of a FVM fine-grained and proposed that more research in this field is needed.

Aderholdt et al. [AL13] present an approach for migrating the state of a VMI-based intrusion detection system while the monitored virtual machine is migrated. Hence, by migrating the state of an application, they need to extend the application logic to send its state. Additionally, they only support asynchronous monitoring, i.e., they only extract the list of kernel modules, the process list and check the integrity of the kernel and its modules. Because of that, their architecture does not allow tracing the execution of the monitored system, e.g., to monitor the invocation of system calls.

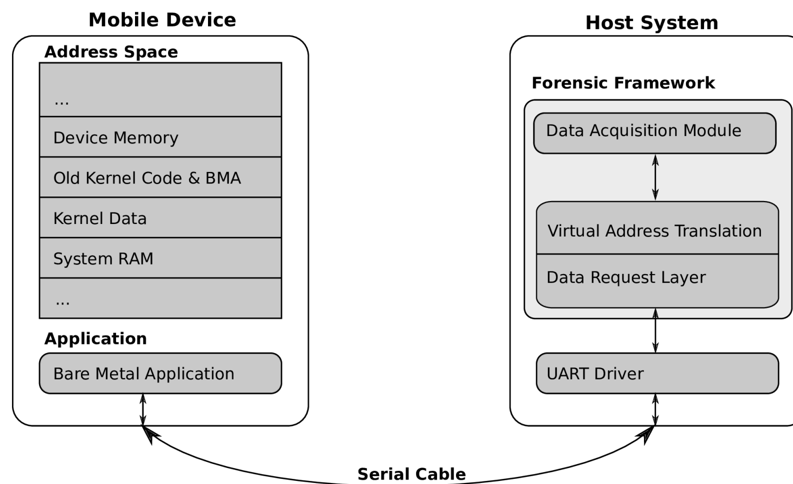


Figure 4.1: Architecture for cold boot based data acquisition over a serial cable [Tau+15b]

4.2 Improving Cold-boot Based Data Acquisition

Mobile devices such as smartphones and tablets contain much information about their owners such as pictures, and documents. Thus, these devices are of particular interest for forensic investigators as well as for attackers. Since these devices often have a high-security level provided by to the ARM architecture, the data acquisition process for memory forensics is a complicated task and cannot be accomplished on a standard device, since a regular Android user account does not have the permissions to access the whole main memory.

A common way for main memory acquisition on systems with physical access but without having any privileges on the system is a cold boot attack. Cold boot attacks take advantage of DRAM's remanence effect, which causes the data not to disappear immediately, but to degrade over time when the power supply is interrupted. Hence, a device that is restarted fast (cold booted), still contains the data of the applications and operating system that was running before. The time to degrade all data can be extended by cooling the DRAM module, so that it is even possible to move the modules to another computer and access the contents afterwards.

Since it is often not possible to unsolder a DRAM module from a smartphone and put it into another system, we have to start an analysis tool on the same system, which leads to the alternation of memory when the analysis system is loaded to memory. Müller et al. [MS13] use a fully fledged operating is used for that purpose. However, it overwrites the contents of the previously running operating system and might even initialize devices and destroys their state and contents in memory.

To minimize the memory footprint of the analysis system and to improve the amount of available data for forensic investigations, we show how a minimal system can be used for accessing data of a cold booted system for forensic purposes. The challenges of this approach are first of all to implement a minimal bare metal application (BMA) for a mobile phone. Additionally, a driver for this operating system is required that allows transferring commands and information between the forensic workstation and the mobile device via the serial bus. The complex analysis logic is implemented on the forensic workstation while the minimal operating system remains small. For the analysis, we use volatility as it provides a large set of analysis plug-ins. To integrate it into our architecture we extend it so that it transparently sends requests for memory pages to the mobile device via the serial interface. The following discussion is largely based on [Tau+15b] and [Hub+16].

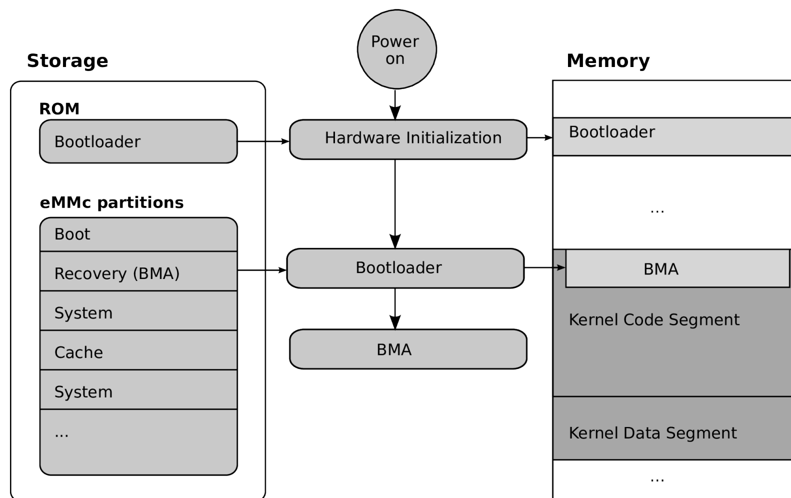


Figure 4.2: Boot procedure starting the bare metal application from the recovery partition on a cold booted device [Tau+15b]

4.2.1 System Design

The primary goal of the approach described in [Tau+15b] and [Hub+16] is to keep the memory footprint, i.e., the amount of data that is modified due to the started program, as small as possible. To achieve that, we implement a bare metal application for the data acquisition process, while the memory analysis is executed on a different system. Hence, the data contents in the main memory are transferred from the mobile device to the analysis system or forensic workstation. The program code is mapped to the kernel text code segment of the previously running kernel that was mapped read-only. It contains only the kernel image, which can be easily reconstructed from the persistent storage. This way, we ensure that important run-time data, e.g., the kernel heap is left intact for the analysis. Besides the small footprint, our bare metal application does not initialize devices, and thus the contents in their memory regions also remain unmodified.

The proposed architecture has two components: a minimal bare metal application and the forensic analysis tool that requests memory from it (see Figure 4.1). To transfer the data between those systems, the serial interface is used because it is easy to implement a driver for it and the driver requires only a few instructions, which keeps the memory footprint low.

This approach allows taking a full memory snapshot. However, it takes approximately 42 hours to transfer the data of a 2 GB main memory snapshot. In some situations, a full snapshot is not required, e.g., when only parts of the memory are required. Thus, we modified the analysis tool volatility, to request only the required data for the analysis on demand from the mobile device for the analysis procedure.

4.2.2 Implementation

To test our approach, we implemented a BMA for the Samsung Galaxy S4 GT-I9505 smartphone and installed it to its recovery partition before the analysis. That allows us to boot it directly after a power cut-off by triggering the recovery boot mode that is started after pressing the volume up and power button. To establish the connection between the test device and the host system, a conventional PC, we use the Samsung Anyway Jig, which is a universal maintenance tool[1]. This adapter allows accessing the serial interface of the smartphone via the Micro-USB port.

[1] www.xda-developers.com/what-is-the-samsung-anyway-jig

Bare Metal Application

The BMA provides the following functionalities:

- Initializing the driver of the serial interface
- Receiving and parsing commands to get the specified amount of memory from physical address
- Returning the contents

Volatility Extension

The analysis tool is an extended version of volatility, which instead of using a snapshot file, requests on-demand data from the mobile device via the serial interface. Additionally, it uses a cache to improve the performance, which avoids requesting the same address after it was already transferred.

With the concept of address spaces (AS), volatility provides the possibility to translate addresses. In our case, this is the translation of virtual to physical addresses or determining the correct position of an address in a memory dump. Depending on the use case, Volatility allows combining and stacking address spaces. We applied the ARM Address Space in our concept for virtual to physical address translation. According to our architecture, we extended Volatility with the implementation of an AS that requests data over the serial port, instead of from a memory dump file. Furthermore, we created a Volatility profile for the specific device under analysis.

We implemented a new Volatility AS for data acquisition via the serial port. We call this AS the Serial Address Space, which resembles the file address space. The latter requests the contents from a local memory dump file. Instead, the Serial Address Space implements the protocol required to request data from the BMA via the UART interface. The AS opens the serial port upon its initialization. Consequently, when a plug-in requests data from an address, the AS directly passes the request to the BMA on the target device. The AS writes the dump command with the start and end address of the request to the UART interface. The BMA then reads and parses the dump command. In the next step, the BMA writes the requested data back to the UART interface. The AS then reads the requested data and returns it to overlying ASs. As some address requests are frequently made, such as for reading the page tables, we equipped the Serial Address Space with a cache that stores the data from former requests. In case a request occurs again, we immediately return the data from the cache instead of consulting the BMA. This likely decreases the duration of the analysis many times over. Figure 4.1 depicts the different layers used by a standard Volatility plug-in, such as linux pslist. The plug-in retrieves the list of running processes by following the linked list of task structures, called task struct, in the Linux kernel. Since our target system is on the ARM architecture, Volatility makes use of the ARM Address Space for virtual to physical address translation. The Serial Address Space sends the request from the layer above to the BMA via the UART interface. The BMA reads the requested memory and returns it via the UART interface to Volatility.

Device Specific Realization

We selected the Samsung Galaxy S4 GT-I9505 device for our specific implementation case. This device is a commonly used mobile phone and provides a serial port. Since we deployed the CyanogenMod Android distribution on our target device, we utilized the corresponding kernel source for the Volatility profile. Our solution is easy to port to other devices that offer a serial interface. We integrated the BMA onto the device's recovery partition after the short power cut-off, respectively the hardware reset of the device. We then boot from the recovery partition in order to launch the BMA.

Figure 4.2 depicts the boot procedure of the Galaxy S4 device in our use case scenario from the point of pressing the power button until the system is started. First, the hardware gets initialized, and the instructions of the bootloader, which is stored in the read-only memory are loaded and

executed. Then, the bootloader checks whether buttons are pressed and boots the corresponding mode. The device supports three different modes: normal, recovery and download. The *normal mode* started the regular Android OS with the kernel from the boot partition when no buttons are pressed. The *recovery mode* starts the system stored on the recovery partition when the *Volume Up* and *Power* buttons are pressed. The recovery mode is mainly used to update, install repair an Android system. The *download mode* allows to directly write data to partitions via USB and is triggered via the *Volume Down* and *Power* button. For writing data to the recovery partition, the bootloader of the device must be unlocked. However, the unlock procedure erases the data partition that stores all user data to prevent attacks that install a malicious kernel for the extraction of confidential information.

For our implementation, we install the BMA to the recovery partition, so that we can directly boot it using the recovery mode after the normal Android was running. To deploy the BMA on the recovery partition of the mobile device, we wrapped the BMA into an Android boot image. The device's bootloader is thus capable of loading this image. For this purpose, the image contains the configuration for the mapping of the BMA in memory at runtime. Our configuration maps the BMA to the address 0x80208000, where the code segment of the formerly running Linux kernel is located (see Figure 4.2). Furthermore, the configuration defines the top of the stack of the BMA at address 0x80209400, i.e., also in the kernel code section (see Figure 2). In doing so, we only overwrite about 5 kB of memory completely located in the kernel code segment. This segment solely contains constant data deduced from kernel code. To launch the BMA, we flashed the generated Android boot image to the recovery partition of the device with the tool heimdall.

4.2.3 Evaluation

To evaluate our approach, we measure the information loss caused by the cold boot attack and the time of standard forensic routines on our test device. We evaluate our proposed framework on the Samsung Galaxy S4 device. We measure the amount of data that degrades during the cold boot attack and show the feasibility of our approach. Additionally, we demonstrate an application of our approach by comparison of the Volatility analysis on a traditional LiME memory dump to a cold boot based analysis using our framework. We also assess the framework's aspects of implementation.

Information Loss

One important practical aspect of cold boot attacks is the loss of information caused by a power cut off. The amount and the location of degradation is unpredictable and can cause problems for the analysis system. For example, if the pointers of a linked list get corrupted, it can not be traversed anymore, which might lead to wrong results of the analysis. Thus, analysis tools using cold boot attacks for data acquisition must be able to detect and handle corrupted data. We evaluate the loss of information with our architecture considering three aspects: the decay of information through the device restart, the duration of the analysis, and the BMA's size.

Decay based on the cold boot attack

Before rebooting the device, we wrote an array of 10,000 known bytes to main memory with a kernel module printing the physical start address of the array. Afterward, we read the contents of the array's physical address and count the unimpaired bytes. We executed the analysis several times. In case of the fast cold boot attack, we re-inserted the battery as fast as possible. In case of the slow cold boot attack, we re-inserted the battery after approximately 1 second. For the reset-based cold boot attack we press the power button press for a few seconds while the phone is still running, which causes a hardware-based reset. We executed both attacks at a temperature of approximately 20° Celsius. The measurements in Table 4.1 show that an analyst can expect an information loss of less than one percent.

Attack	min	max	average
Fast Cold Boot	9,983	9,998	9,991
Slow Cold Boot	26	9,521	3,379
Reset Cold Boot	9,998	10,000	9,999

Table 4.1: Number of successfully retrieved bytes of a 10,000-byte array with different cold boot attacks [Tau+15b]

The depicted results mainly show that a non-reset-based cold boot attack did not always return reliable results as it depends on multiple factors like the temperature and the abilities of an adversary. In case of the fast cold boot attack, we retrieved 9,991 of the 10,000 known bytes in average. Despite the only weak degradation, the application of Volatility plug-ins became difficult. When conducting a slow cold boot attack, the degradation proceeded quickly. In this case, the successful application of Volatility plug-ins got infeasible. However, the reset-based attack provided the better results during our tests. This scenario is more reliable as it does not depend on how fast the battery can be re-inserted. In most cases, we retrieved all of the bytes successfully and had occasional bit flips only in few test cases. This shows that even in case of the reset attack, where we did not remove the battery at all, the memory occasionally decays. In normal cases, where we retrieved all of the bytes correctly, the application of Volatility plug-ins was always successful. In our test set-up, it was also possible to successfully extract data with our architecture from the device when it was restarted twice, e.g., for first installing the BMA. Nevertheless, additional data might decay during the second reboot.

Decay during the analysis

As we do not completely save the data in main memory at once, we rely on the data to remain intact on the target device during the whole analysis process. To prove that this requirement is given, we executed our cold boot attack-based data acquisition multiple times 15 minutes after the device has booted the BMA. As expected, we retrieved exactly the same unaltered data between the test requests. For this reason, we may assume that the data does not decay any further as long as the memory is supplied with power and the DRAM modules are refreshing the capacitors.

Information loss based on the size of the BMA

Another important source of data loss is the amount of data in RAM that the BMA occupies. The boot image containing our BMA has a size of 4 kB and the stack size is 1024 bytes. Thus, we overwrite no more than 5 kB of memory. This is less than the previously running kernel's size. The kernel code segment forms a set of constant and known bytes. The segment does not change between different runs of the system. This means that the overwritten bytes do not impair relevant data as the code segment of the formerly running kernel is mapped to this address range.

Volatility Plug-ins

In Table 4.2 we measure the execution time of standard volatility plug-ins in comparison to the runtime on a pre-captured snapshot of the same system. We created a memory dump on the running phone with the LiME kernel module right before we executed the cold boot attack. Afterward, we executed the Volatility plug-ins on the memory dump, as well as in conjunction with the BMA running on the phone. The plug-in linux_pslist extracts a list of running processes. The resulting entries of our measurements differed between the dump file (230 entries) and the cold boot analysis (225 entries) by solely five more threads. This comes from the LiME kernel module creating these threads during the acquisition process. The analysis with the BMA took 24.23 seconds, whereas the LiME dump analysis took 3.54 seconds. The plug-in linux_iomem extracts the map of the system's memory for physical devices. The results received from the cold boot-based analysis were equal to the memory dump analysis. As in the scenario before, the runtime of the plug-in was

Plug-in	Results (LiME Dump)	Results (BMA)	Time (LiME Dump)	Time (BMA)
linux_pslist	230 entries	225 entries	3.54 s	24.23 s
linux_iomem	138 entries	138 entries	2.18 s	08.97 s
proc_maps (init)	9 entries	9 entries	2.00 s	06.03 s
dump_maps (init, heap)	340.0 kB	340.0 kB	1.98 s	35.84 s
dump_maps (init, stack)	139.3 kB	139.3 kB	1.94 s	07.02 s
proc_maps (rild)	156 entries	156 entries	5.38 s	18.55 s
dump_maps (rild,heap)	380.9 kB	380.9 kB	2.05 s	41.48 s
dump_maps (rild,stack)	139.3 kB	139.3 kB	2.06 s	08.29 s

Table 4.2: Different volatility plug-ins and their run-time using a dump file and the bare-metal application [Tau+15b]

longer in case of the BMA application. Compared to the LiME dump analysis with a duration of 2.18 seconds, the analysis with the BMA took 8.97 seconds. This emerged in every test case.

The plug-in proc maps returns the memory mappings of a single process. This renders results similar to contents in /proc/<pid>/maps. For our measurements, we requested the mappings of the init and the rild process. The latter is responsible for the Radio Interface Layer (RIL) functionality of an Android phone. In both cases, the measurements returned exactly the same results: 9 entries in case of the init process and 156 entries in case of the rild process. We finally requested the stack and heap memory segments of the rild and the init process with the plug-in dump maps. The amount of data in bytes was for both processes the same for the stack and heap. The data of the stack of the init process turned out to be consistent between the two data acquisition methods. The same holds for the rild process. In every test case, the required time for executing a plug-in, which operates on the memory using the BMA, was significantly higher. This emerged as a result of the low transfer rate of the UART interface.

A full dump of the 2 GB RAM of the test device required about 42 hours. For our purposes, the low transfer rate was acceptable, since all plug-ins terminate within less than 45 seconds.

The measurements show that doing memory forensics via the serial interface is possible and that the run-time for most of the standard forensic operations, e.g., getting the process list, is below one minute. However, when it comes to analysis that requires iterating over the full address space, the analysis logic of those plugins should at least be partially implemented in the minimal operating system, e.g., the search operation that locates specific strings in memory.

Limitations

The approach of using a cold boot attack only works on devices with an unlocked boot loader. If the device is still locked, it must be unlocked, which often requires to wipe the data partition containing the personal data. Thus, the described approach is only feasible for a limited set of application areas when an unlocked phone is given or when it can be unlocked without information loss.

Nevertheless, this approach can be useful to investigate the contents in memory that are usually not accessible even when having root permissions, such as memory areas that are used by devices or the TrustZone that are made inaccessible after the initialization by the kernel.

Finally, this approach requires that the target device provides an interface that is suitable for transferring data to an analysis host and implementing the corresponding low-level driver. In [Hub+16] we discussed the required steps to port it to the Nexus 5 device.

4.3 Towards ARM TrustZone Based Monitoring

Cold boot attacks can only be used to investigate past incidents or the current state of a system. Since they require to reboot a system, they can not be used to trace a system to improve security permanently. Virtual machine introspection can do that since it leverages the properties of virtualization.

The ARM TrustZone processor extension [ARM09] provides another way to achieve a similar level of isolation on mobile devices. It introduces two different processor modes: the normal and the secure world. The *normal world* is designed to run the fully fledged (Android) operating system. The *secure world* runs services with higher security requirements, e.g., a key manager. This approach accepts that an attacker can take over the operating system in the normal world, without losing control over the data that is maintained in the secure world of the TrustZone. The fact that programs running in the *secure world* have access to the full address space of the *normal world* makes the secure world of the TrustZone an exciting place to deploy memory forensic based monitoring approaches that attempt to increase the security of the normal world.

For example, the secure world of the TrustZone can be used to access whole system state in the main memory of the normal world without relying on kernel level functions of the normal world that could be manipulated by a rootkit. Moreover, the full analysis framework, i.e., the interpretation of contents in main memory can be moved to the secure world.

This section is based on [Gue+17; Gue+18] and describes the design of the ITZ library which implements parts of the interface of LibVMI in the secure world of the TrustZone. With such an interface in the secure world, it should be possible to port already existing VMI-based applications from cloud environments to the TrustZone. The contribution of this section is the discussion and evaluation of the system design of this approach.

4.3.1 Threat Model and Assumptions

Before going into the details of the system design, we discuss the threat model and trust assumptions. We assume that the normal world and all of its components can be compromised by an attacker. Hence, we also assume that all communication coming from the normal world to the secure world cannot be trusted.

We assume that the processor works as expected and provides isolation between the secure and normal world. Additionally, we assume that the operating system used in the secure world is not compromised and trustworthy. This also includes, that the boot process is secure and starts the trusted operating system in the secure world.

4.3.2 System Design

The primary goal of the ITZ library is to provide the same API as LibVMI in the secure world to be able to re-use code that analyzes the state of virtual machines for the system running in the normal world. The proposed architecture has its components divided among the two worlds as shown in Figure 4.3. The architecture is built on top of the microkernel architecture of Genode [Gen], including the Trustzone VMM.

Normal World

The normal world runs a fully-fledged operating system, e.g., Linux or Android. For the interaction with the secure world, the kernel in the normal world implements a driver that allocates a shared memory region that can be accessed by the secure and normal world and serves as a buffer to exchange data such as commands and their results. The shared memory region is only used by the secure world to return data. Every time the secure world needs to retrieve data from the normal world, it can access the memory directly.

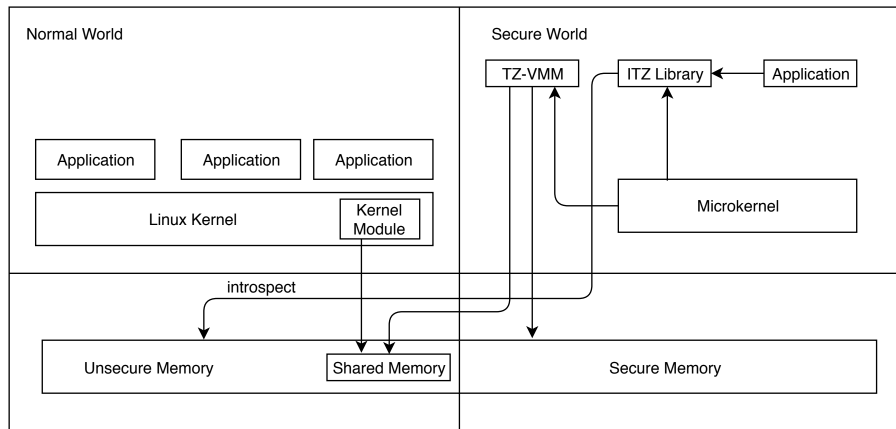


Figure 4.3: The TrustZone system architecture including the ITZ library [Gue+18]

In order to invoke functions of the secure world, the *smc* instruction is used. It switches the context from the normal world to the secure world.

Secure World

The secure world runs the Genode microkernel that provides basic resource management functions (memory, CPU, I/O). Additionally, it runs the virtual machine monitor of the microkernel that manages the shared memory buffer and communication with the normal world.

Moreover, the secure world runs the ITZ library and on top applications that do the introspection, such as extracting the process list.

4.3.3 Implementation

This section describes the implementation of the discussed architecture. For the implementation, we use in the secure world the microkernel from Genode labs [Gen]. In the normal world, we run a modified Linux kernel in version 2.6.53.3 that implements the kernel module required for the shared memory.

System Configuration

The prototype of the ITZ library is implemented on the Freescale NXP i.MX53 Quick Start Board, which contains an ARM cortex-A8 processor with TrustZone, 1GB DDR3, an SD/MMC card slot, two USB adapters, and other interfaces and chips.

The i.MX53 QSB has a mechanism to assure that memory regions are protected from unauthorized access. This works as part of the dynamic random-access (DDR) memory controller, through the multi-master multi-memory interface (M4IF). This interface guarantees that a configurable range of memory is protected and used exclusively by the secure world. The i.MX53 QSB has its 1GB of RAM split into two memory banks, RAM 0 and RAM 1. Each memory bank has 512MB of memory that can be configured and used. For our work, and due to the limitations of the M4IF, that only allows for up to 256MB of memory to be protected in each bank, we choose to protect the initial 256MB of the first memory bank [Fre12]. This process is done at boot time and assures that 256MB of memory are reserved for our secure world, while the rest of the bank is used for the normal world OS.

World	Component	Lines of Code
Normal world	Linux kernel	9132 K
Secure world	Microkernel	20 K
Secure world	Genode OS framework	10 K
Secure world	TZ VMM	3 K
Secure world	ITZ Library	2 K

Table 4.3: Code Base Size of the Components [Gue+18]

The ITZ Library

The secure world runs the ITZ library which provides a subset of the API of LibVMI. For accessing the memory of the normal world it uses the TZ VMM. The most important functions are the read, write, and translate functions. The translate functions are required to translate virtual addresses of the normal world to physical ones and vice versa[2].

4.3.4 Evaluation

To evaluate the concept of monitoring the normal world from the secure world, we will discuss the feasibility of porting LibVMI to the secure world. A deeper analysis of the performance of the ITZ library together with introspection application has been done by Guerra et al. [Gue+18].

Portability and Code Size

One of the main problems of porting LibVMI to the secure world is missing software dependencies. The genode secure world implementation does not include the Linux library glibc, which is heavily used by LibVMI, e.g., to implement the cache[3]. Porting all required libraries to the secure world is not an easy task and increases the size of the code base installed to the secure world, which results in a higher chance of containing a flaw that can be compromised by an attacker. Moreover, if this code should be migrated to another TrustZone implementation, such as the OP-TEE operating system [Lin], all the libraries have to be migrated there as well. Table 4.3 summarizes the size of the code base for all the components of our architecture. The lines are counted using the tool cloc[4].

Validity of Data coming from the Secure World

Besides, if an application invokes the introspection function of the secure world it cannot make sure that the return value is not altered. This is because the whole system in the normal world could be compromised and there is no direct way to validate if the result comes from the application running in the secure world or from a malicious component in the normal world that overwrote the results. One way to tackle this problem is to sign the results with cryptographic primitives, which of course only works under the assumption that the secure world has access to a private key that is not available in the normal world.

Future Work

Using the ARM TrustZone for increasing the security of normal world seems to be a good idea. Unfortunately, the TrustZone is a very closed environment, and it is usually not possible to deploy any code there on common end-user devices without breaking security mechanisms, which makes the TrustZone a very unattractive target for research activities. However, the OP-TEE operating

[2] A more detailed discussion of the internals can be found in [Gue+18]

[3] The authors of LibVMI provide a lite version of glib that only brings the required functions of LibVMI for porting it to systems that do not have their version of glib (`https://github.com/libvmi/glib_lite`, Accessed on 2019-07-08)

[4] cloc - `https://github.com/AlDanial/cloc`, Accessed on 2019-07-08

system seems to provide a stable environment for future research since it has already been ported to several existing systems such as the Raspberry Pi3. Thus, porting the ITZ library to the OP-TEE operating system and extending it seems to be a good start for further research.

The ITZ-library does not support synchronous monitoring of the normal world. A possible mechanism how this could be implemented is described by Ge et al. [GVJ14].

4.4 Bringing VMI to Cloud Environments

In this section, we discuss the *CloudPhylactor* architecture, which aims to provide VMI-based monitoring mechanisms to cloud tenants. The discussion is based on [TRR16]. There are different reasons for using virtual machine introspection in cloud environments either as customers or as providers. Cloud providers may want to detect malware in the virtual machines of their customers to make sure that it does not infect the virtual machines of other customers, which could harm the reputation of the provider [Fis+15]. Additionally, cloud providers might be forced to give VMI access to law enforcement, e.g., to take a snapshot of a suspicious tenant. Cloud customers might want to have VMI access on their virtual machines to run a VMI-based virus scanner or to monitor the integrity of running software components. However, there are no public cloud providers that offer their customers access for VMI-based operations to their virtual machines.

Several significant problems make it impractical to use VMI in current cloud environments. First, in most (if not all) real cloud infrastructures, there is a *lack of access* for the cloud customer to VMI functionality. In addition, there is no established *billing model* for using VMI services. The few available research prototype solutions for the access problem are insufficient due to *lack of functionality* and *lack of performance*. And finally, *security risks* are an additional issue for enabling VMI in production cloud infrastructures.

The CloudPhylactor[5] architecture is an approach to solve these problems by leveraging MAC. In particular, we show how the implementation of the Flask architecture of the Xen hypervisor can be used to grant a dedicated monitoring VM the rights to run VMI on other guest VMs. This enables cloud customers to use the Xen API for VMI on their VMs.

Additionally, the access of these monitoring VMs can be restricted to a single guest VM or a subset of all VMs. Thus, even if an attacker can gain access to the monitoring VM, he does not gain full control over the physical cloud node. Thereby, we increase the security of VMI-based monitoring approaches. In contrast to other cloud forensic solutions like CloudVMI [BSM14] and LiveCloudInspector [ZR15], CloudPhylactor does not introduce significant overhead to the monitoring process, as it can access the Xen interface directly and is therefore not restricted to use another more limited API. Moreover, this approach does not require fundamental changes to the cloud management software.

4.4.1 Threat Model and Assumptions

VMI requires powerful permissions and can, therefore, pose a potential security threat to cloud environments. This section describes the two most important attack vectors to VMI-based monitoring tools in cloud environments, which use a VMI component installed in the most privileged virtual machine of the cloud node, such as LiveCloudInspector [ZR15] or CloudVMI [BSM14]. Users can interact with those tools using their API, which could be exposed via the cloud management software (LiveCloudInspector) or directly by using a network connection to the tool inside the Dom0 (CloudVMI).

The first attack vector are flaws in the VMI framework that can be exploited regular. If an attacker is able to inject malicious code into the VMI-based framework (e.g., by exploiting buffer overflows), he could gain access to the cloud node or use the monitoring tool to access virtual machines of other users. Additionally, flaws in the authentication module could be used to bypass the authentication to access the virtual machines of other users.

The second attack targets the interpretation of main memory of virtual machines. It requires an attacker who has full control over a virtual machine (e.g., an internal attacker) and can modify all entries in memory of a VM. By placing crafted data in memory, it is possible to attack the interpretation routine. Similar attacks already exist for Tripwire [Com04] and Snort [CER03] and other forensic software [Ett17]. They show that an attacker can interact with a target system in

[5] The name CloudPhylactor is a combination of Cloud and the Greek word $\varphi\upsilon\lambda\alpha\kappa$ (phulak), which means watch or guard.

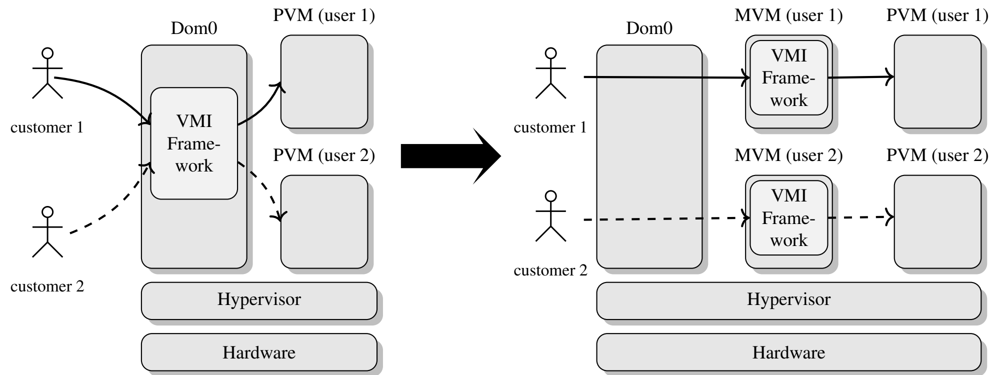


Figure 4.4: The left side represents the most used architecture: all VMI-based operations are executed in the privileged domain. The right side represents the *CloudPhylactor* architecture where a dedicated monitoring virtual machine monitors a production virtual machine. [TRR16]

a way such that the collected monitoring data exploits vulnerabilities in an intrusion detection system. In the case of VMI, a flaw in the interpretation of the monitoring framework can be used by an attacker to get access to the system where the monitoring framework is running. If the monitoring tool is installed in Dom0, an attacker gets access to the virtual machines of other users or to the system running in Dom0.

VMI-based systems heavily depend on the interpretation of data extracted from a guest system. Bahram et al. [Bah+10] demonstrate that manipulating a VMI-based system by changing in-guest data structures is feasible, e.g., for placing misleading or wrong information to fool the analysis tool or to hide traces [Bah+10]. The problem of incorrect interpretation of the state of a VM that has been manipulated by an attacker is known as the *strong semantic gap* and needs to be addressed by the monitoring tool [Jai+14]. This problem, which concerns the validity of the results obtained by the monitoring tool, is outside the scope of the *CloudPhylactor* architecture. Our focus is on preventing any malicious impact outside of the monitoring tool.

A general threat to cloud computing are malicious cloud administrators. This threat is out of the scope of the *CloudPhylactor* architecture.

4.4.2 System Design

Our proposed architecture – *CloudPhylactor* – allows cloud customers to perform VMI-based operations from a dedicated monitoring virtual machine on their production virtual machines. Furthermore, it minimizes the risk of privilege escalation, when an attacker exploits flaws in the monitoring software and gains control over the virtual machine that runs the VMI software. For this purpose, we move the VMI-based operations into a dedicated virtual machine with limited access to other domains (see Figure 4.4). In the following, we call a virtual machine with normal permissions *production virtual machine (PVM)* and a virtual machine with permissions to run VMI *monitoring virtual machine (MVM)*. Additionally, we use the term *user* in the context of a cloud customer and not for different entities such as administrators and forensic investigators. Instead, we are discussing the permissions of VMs and not entities that are using them. The mapping of entity roles to policy users is out of the scope of the *CloudPhylactor* architecture.

The concept that we are describing is independent of a specific hypervisor. However, only Xen supports MAC or at least a fine-grained access control that supports to define that a VM has permissions to access another VM's main memory. Thus, we use the terminology of Xen in the architecture description.

Monitoring Virtual Machine

An MVM must have at least the permissions for these operations on another domain for static VMI, e.g., to extract the process list:

- *read access on main memory* to extract low-level information, e.g., to extract the process list of the operating system.
- *read access on CPU registers* to extract the current CPU state, e.g., to determine the process that is currently running on a CPU by accessing the CR3 register.

To trace the execution of a VM and do dynamic analysis, writing permissions are required, which includes:

- *write access to main memory* to modify program code, e.g., to insert necessary software breakpoints to trace the execution of a process or to inject code. A software breakpoint is an instruction that causes the CPU to throw an interrupt.
- *write access to CPU registers* to change CPU state, e.g., to change return values.
- *access the Xen event channel;* it allows monitoring VM events such as interrupts of a VM that can be caused by software breakpoints.

An MVM should contain only the necessary tools to minimize the attack surface [Sha+14]. The operating system of an MVM can be either a fully-fledged Linux system with several forensic tools or a minimal domain, where the kernel is the forensic tool. The last approach helps to minimize the number of additional resources and reduces the attack surface to a minimum.

Mandatory Access Control

The goal of *CloudPhylactor* is to grant MVMs the permissions to run VMI-based operations on PVMs by using the MAC of Xen. To do so, we introduce a new Flask type for MVMs. The Flask type is part of the label that is assigned to a VM when it is launched, and it cannot be changed during run-time. This label also includes the name of a Flask user. We use it to constrain the access of MVMs so that they can access only PVMs of the same user but not of others. The enforcement of the policies is performed at run-time by the Xen hypervisor. As Flask users are part of the policy that is static, all users must be known before they are compiled. We create dummy users in the Flask database and map cloud customers to them, to handle new users at run-time. The mapping process is performed by the cloud management.

Cloud Management

The cloud management is the interface between the user and the cloud infrastructure, e.g., Xen. It provide a user interface to allow users to set the type (PVM or MVM) of a VM and the group of PVMs that can be monitored by an MVM. When a VM is launched, the cloud management has to compute the security label of it, based on the user input. Additionally, it has to manage the mapping from cloud users to dummy flask users.

4.4.3 Implementation

For our implementation of the *CloudPhylactor* architecture, we use OpenNebula 4.12.3 as the cloud management platform and Xen 4.5 as the hypervisor. To use MAC for our architecture, we define a new policy type for MVMs and its permissions on PVMs. Then we assign domains to users so that an MVM is able to access only domains of the owner but no others. Additionally, we generate Flask users and map them to cloud customers. We will discuss these steps in detail in the following paragraphs.

Policies

To grant an MVM access to the main memory of other VM, we introduce a new domain type – `domU_monitoring_t` in the Xen Flask policies and derive it from the permissions of a regular DomU. Furthermore, we define rules that grant an MVM the permissions to execute VMI-based operations on domains of the type `domU_t` (see Section 4.4.2).

User management

When a domain is started in the context of `domU_monitoring_t`, it can perform all granted operations on every domain running in the context of `domU_t` even if the VM belongs to another customer. Usually such global access is unwanted, and instead, access should be restricted on a per-customer basis. To achieve that, we restrict VMI permissions to the domains that belong to the same user or only a subset of his domains if a more fine-grained restriction is required.

As the policies are static and cannot be modified, all users must be known before the rules are compiled. Otherwise, rules must be adapted, compiled and loaded whenever a new user registers. As the user set of cloud providers is changing permanently and reloading of rules can introduce security and performance drawbacks, this approach is not feasible for production environments.

To avoid the generation of Flask users at run-time, we create a set of dummy users and map cloud users to them. The mapping can be *cloud wide* or *cloud node local*. A cloud-wide mapping maps each customer to the same Flask user on all nodes, which requires that all cloud nodes must have at least as many Flask users as real cloud customers. This approach does not scale for huge installations, as every cloud node has to provide Flask users for all cloud users. Cloud node local mapping solves this problem. As the mapping is local, a user might be mapped to a different Flask user on every cloud node or might not be mapped at all when he has no VM running on this node. In both cases, the cloud management must handle the mappings.

Additionally, not all monitoring domains of one user should be able to monitor all PVMs of that user. If VMs contain data of different levels of clearance and an attacker can subvert their monitoring domain, he can also gain access to data with a higher level of clearance. To solve this problem, VMs can be assigned to sub-users where access is only granted in between sub-users. For example, one cloud customer has the user ID `1`. Then the sub-users would be `cloud_customer_1_{0..n}`. By choosing a sub-user ID, the cloud customer can define the group of VMs that can be monitored by an MVM as it is only able to monitor other VMs with the same sub-user ID. More fine-grained policies, e.g., those that only grant read access would be also possible.

Another approach is the usage of Multi-level security (MLS)/Multi-category security (MCS) policies. This concept assigns each unit (i.e., domain) a sensitivity level and a category. It is based on the model of Bell and LaPadula [BL73; Han06]. However, the MLS and MCS support of Xen 4.5 is incomplete.

Cloud Management

We extend OpenNebula so that a cloud customer can create monitoring machines by adding two flags to the template of a VM. One flag defines whether a VM is an MVM and the other one defines the sub-user ID. Both flags are translated in the Xen deployment configuration to the corresponding security label. Furthermore, we implement a mapping mechanism from OpenNebula to users in the Flask policies. To achieve that, we use a one-to-one mapping that translates the OpenNebula user ID to a Flask user with the same ID. The computed security label of a VM has the form:

```
customer_{userid}_{class}:vm_r:{type}
```

The variable `userid` is defined by the cloud management and derived from the ID of an OpenNebula user. The `class` variable and the `type` of a VM is defined by the customer in the template

variables of a VM. Moreover, the cloud customer can use the user interface to choose on which cloud node his VM shall be launched to place the MVM on the same cloud node where the PVM is running.

XenStore

The Xenstore holds the domain name of each VM, which is required to perform translation from a domain name to a domain id. As a VM can only be addressed with the domain id and not the name we modify the permissions of the Xenstore entries so that a cloud customer can perform the translation for his VMs. Hence, we grant a monitoring VM the read access to the domain name entry of all domains that belong to the same user. We perform this job with a script every time a VM is launched.

Monitoring VM

Another part of the *CloudPhylactor* architecture is the MVM. In our implementation, it is a fully fledged Linux that contains all necessary libraries to communicate with the Xen hypervisor. Additionally, we have installed tools such as LibVMI and Volatility to run standard forensic operations [Fou]. Finally, the MVM must be paravirtualized. Otherwise, some hypercalls required for VMI are not available to the guest system.

4.4.4 Evaluation and Discussion

In this section, we measure the performance of the *CloudPhylactor* architecture and discuss its security. All described tests are executed on a server with an Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz CPU with six cores and 32 GB RAM.

Performance

To determine the performance difference between the access from Dom0 and an MVM, we run four tests that mimic the behavior of often used real-world VMI operations. These are:

1. A four-byte read operation commonly used to retrieve integer values or pointers
2. A four byte write operation commonly used to insert software breakpoints
3. A read operation that requests a full page (4096 Bytes), e.g., to take a snapshot of process memory
4. Extraction of PVM's process list

These are real-life use case scenarios that can be used to detect suspicious processes or for doing forensic operations on main memory. Even if more complex VMI-based monitoring tools are used, they rely on the same operations as the extraction of the process list requires, i.e., the read operation to extract data structures. Using more complex VMI operations would partially conceal and clutter with noise the observed overhead values. The extraction of the process list is not trivial and uses most common LibVMI features for parsing and traversing kernel data structures, and we consider it representative for a VMI tool that analyses guest kernel data structures.

We measure the performance of the VMI operations for the four test cases in different configurations. The VMI operations are either executed (as done traditionally) in Dom0, or (using our *CloudPhylactor* architecture) in MVM. We also use different mappings of virtual machines to physical CPUs to find out if such mapping has an impact on performance, e.g., due to some cache effects.

In total, the following five configurations are used, and VMI operations are executed from:

A **Dom0:** PVM is pinned to the same CPU as Dom0

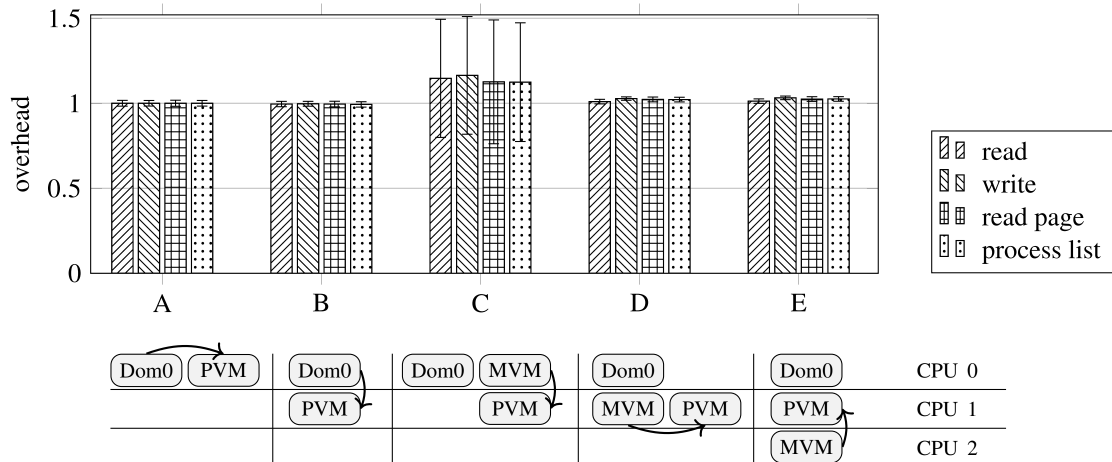


Figure 4.5: Relation between the time that is required for the VMI access methods of the test cases B to E compared to test case A [TRR16]

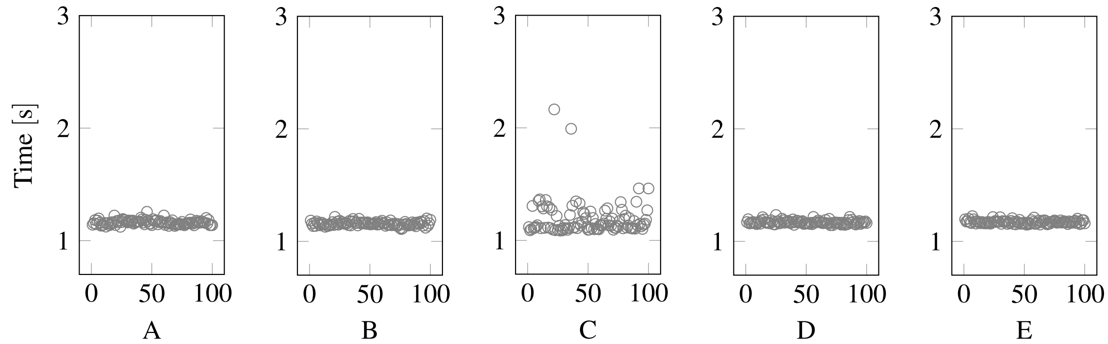


Figure 4.6: The timing of 300 read-page operations measurements for the five different configurations of Figure 4.5

B **Dom0:** PVM is pinned to a different CPU then Dom0

C **MVM:** Dom0 and MVM are pinned to the same CPU; PVM is pinned to another CPU

D **MVM:** PVM and MVM are pinned to the same CPU; Dom0 is on another CPU

E **MVM:** Every virtual machine is pinned to another CPU

To prevent the caching from LibVMI, we disable this feature for all measurements. During this measurement the PVM is mostly in idle state. For the access time, we measure the time that is required for 10,000 operations and take the average and standard deviation of 100 runs. Figure 4.5 shows the results of this measurement. We use configuration A (VMI on Dom0, on the same CPU as PVM) as a baseline, and for all other configuration, we plot the relative execution time of the VMI operations (smaller values are better). We can conclude that even in the worst case the VMI performance loss is less than 3%. The overhead of about 17% percent for the test case C is caused by the fact that Dom0 and MVM are pinned to the same CPU. Thus, the VMI access is delayed by the execution of Dom0. Figure 4.6 depicts the detailed results of the read page operation for each configuration. The time of each measurement is the average time of 100,000 read page operations in 300 iterations. The measurements show that the average access times of all configurations are similar. However, the configuration C has multiple outliers, which are caused by the fact that the Dom0 and the MVM share the same CPU. In this case, when Dom0 gets active (e.g., due I/O operations), it can pause the MVM and cause delays of the read operation.

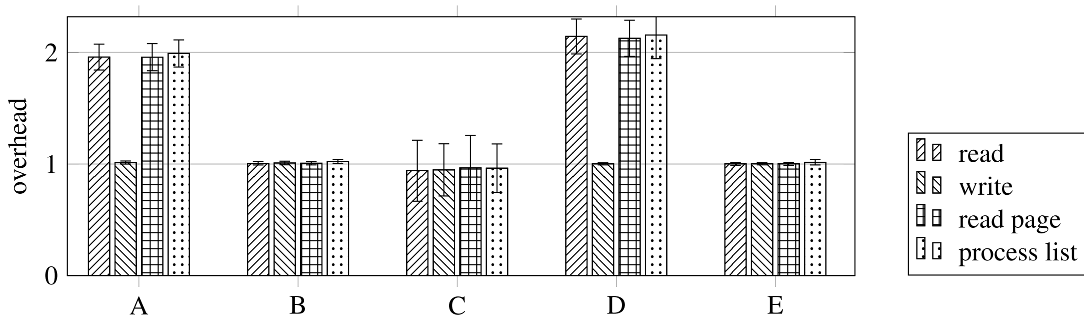


Figure 4.7: Relative access time of the VMI access methods when the PVM is running a Benchmark compared to test case of Figure 4.5 [TRR16]

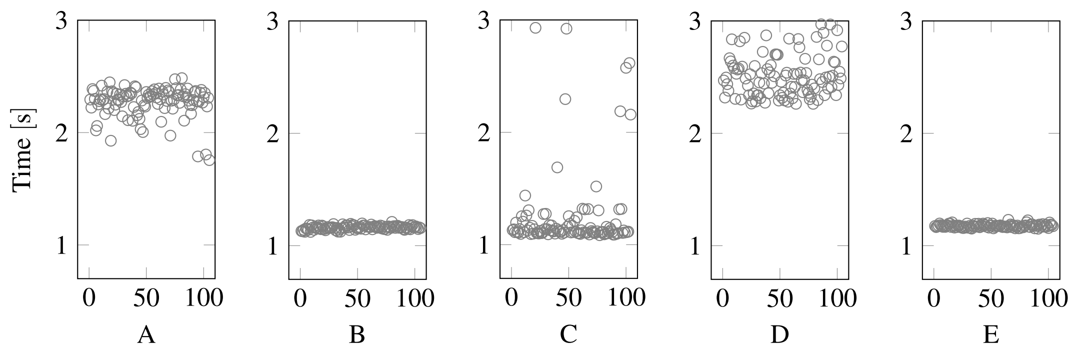


Figure 4.8: The absolute timing of the read page operation for the five different configurations of Figure 4.7

The low access latency is an advantage compared to the CloudVMI [BSM14] solution as their approach uses remote procedure calls. Thus, depending on where the measurements are executed – on the same or a remote cloud node – they experience a high overhead up to two hundred percent, which causes a negative performance impact when events are used, e.g., to monitor software breakpoints. In that case, the execution of the PVM is paused until the event is processed by the monitoring tool. Thus, when the execution is delayed by the latency of the network, the performance of the PVM decreases.

To evaluate the impact that is caused by the placement of the virtual machines when a virtual machine is polluting the CPU cache, we run the same measurements while the PVM is running a benchmark. To achieve that we run the Dhrystone and pipe-based context switching benchmark of the byte-unix benchmark suite[6] in a loop in the PVM. The results of the relative access times are depicted in Figure 4.7, where we use the measurements of Figure 4.5 as a baseline. When the virtual machine running the VMI operations shares the CPU with the PVM (case A and C), we have a significant impact on the read, readpage and process list operation. Hence, there is no big difference when the VMI application is executed on Dom0 or in the MVM. In case C we again have a bigger standard deviation than in the other cases, because the Dom0 can interrupt the MVM during the measurement.

The write access is not delayed because of the measurement method: we pause the PVM for the write operation to avoid inconsistency of the virtual machine state as we write the value that has been read before. Thus, the PVM is not scheduled during the VMI operation and does not affect the time for the VMI access. The other operations do not pause the PVM. Thus, the VMI

[6] Byte-unixbench - `https://github.com/kdlucas/byte-unixbench`, Accessed on 2019-07-08

Users	1,000	10,000	50,000
Compile	0.20 s±0.03 s	2.90 s± 0.14s	50.93 s± 2.33s
Load	0.0035 s±0.01 s	0.30 s± 0.02s	5.63 s± 0.20s
Size	96 KB	860 KB	4.2 MB

Table 4.4: Average time in seconds and standard deviation to compile and load policies measured in relation to the amount of users [TRR16]

operation can be interrupted by the PVM that is doing the CPU intense benchmark.

Size of Policy Database

Another aspect that might influence the performance of VMI-based operations is the size of the policy database. The Xen hypervisor checks for each access if it is allowed or not. To do that, it looks up each decision in the database. To check if the lookup operation for an access operation takes longer with a bigger database, we run the same measurements as in Section 4.4.4 but with a bigger policy database, i.e., 10,000 and 50,000 users instead of 1000 users. Depending on how the lookup operation is implemented, it could take longer to find the policy for an access operation in the set of policies. Figure 4.9 and 4.10 shows the overhead for these measurements. Figure 4.11 and Figure 4.12 depict the results of the individual measurements for the read page operations. Based on these results, we conclude that there is no significant difference at run-time between a big policy database and a small one.

Additionally, we measure the size of the database and the time that is required to compile and load the policy database. The results in Table 4.4 show that the size correlates linearly with the number of users but the time does not. These measurements show that both global and local user mapping is feasible. However, the policy database should not become too big. Otherwise, it is not possible to load it anymore to main memory. In that case, it is necessary to switch to a local mapping to keep the policy database small on each cloud node (see Section 4.4.3).

Resources

Compared to LiveCloudInspector our approach requires – in the worst case – a separate MVM for each PVM instead of only a monitoring process. However, one MVM can monitor more than one PVM. Thus, additional resources such as main memory, CPU time and storage are necessary for each MVM. The amount depends on the configuration and the memory that is required by the processes that are running in the VM. The MVM can be stripped down to a minimal system, e.g., use only a very minimal kernel [Ma18]. Moreover, with VMI on Dom0, it is impossible to attribute CPU and memory consumed by monitoring to a customer. With MVM, the cloud customer can rent an MVM of appropriate size (small for lightweight VMI monitoring, larger for more complex processing of VMI data), and the billing infrastructure of the cloud can be reused without adding any additional complexity.

Security

Attack Impact: In Section 4.4.1, we describe the risks that come along with the installation of an analysis tool in a privileged VM including the access to another VM or the execution of commands in the privileged domain. By moving the analysis tool to a dedicated MVM we protect the cloud management infrastructure and other cloud customers. In the worst case, an attacker is only able to instrument the analysis tool in an MVM and execute commands in this domain or to access the PVM of the same customer. However, he is not able to attack other parts of the cloud infrastructure.

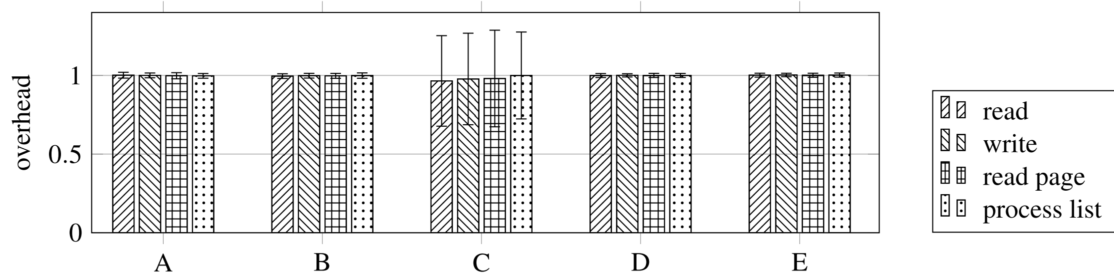


Figure 4.9: Relative VMI access time with 10,000 flask users compared to the same measurement with 1,000 users in Figure 4.5 [TRR16]

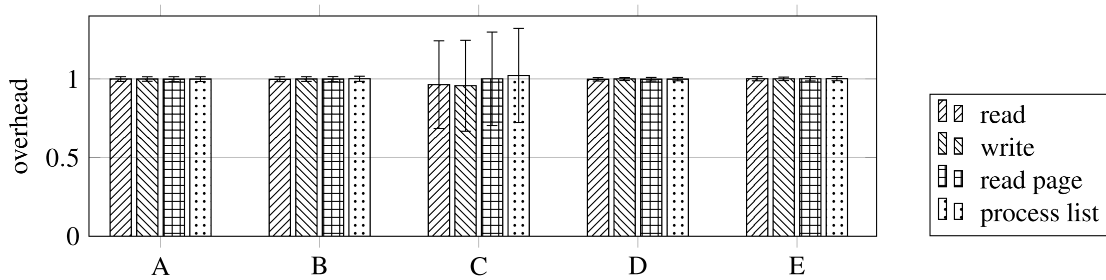


Figure 4.10: Relative VMI access time with 50,000 flask users compared to the same measurement with 1,000 users in Figure 4.5 [TRR16]

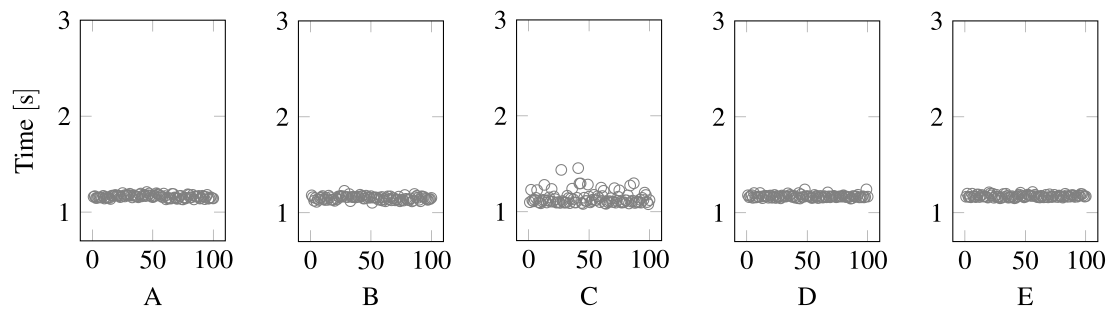


Figure 4.11: The timing of the read page operation for the five different configurations of Figure 4.9

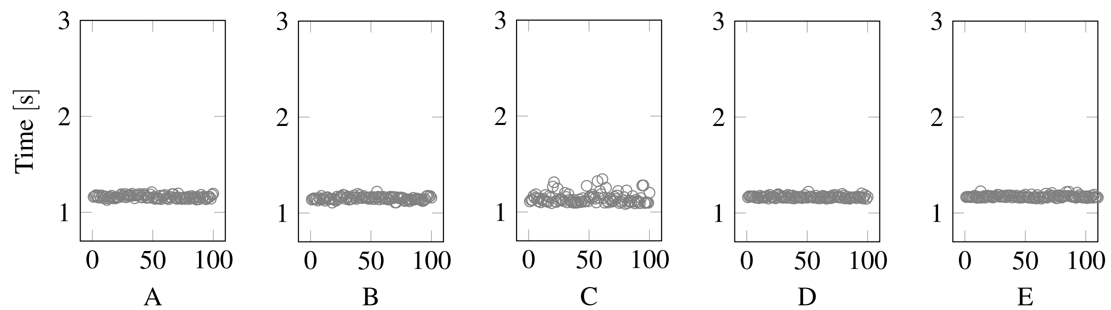


Figure 4.12: The timing of the read page operation for the five different configurations of Figure 4.10

4.5 VMI and Live Migration

In this section, we discuss the *TwinPorter* architecture that extends *CloudPhylactor* to support live migration of monitored virtual machines. This discussion is based on [TBR19]. Live migration is the dynamic relocation of a virtual machine without the need for shutting down and restarting the virtual machine. In cloud computing environments, live migration of virtual machines from one host to another is sometimes necessary, for reasons such as load balancing or maintenance operations. For example, if the overall system load is low, the cloud management system can migrate virtual machines to a subset of all hosts and shut down the remaining hosts to save energy. The main advantage of live migration is that it minimizes the down-times of virtual machines since it does not require restarting the virtual machine.

CloudPhylactor 4.4 enables virtual machine introspection for cloud customers by introducing dedicated monitoring virtual machines that cloud customers can access. Since the CloudPhylactor architecture always deploys the MVM and PVM on the same cloud node, the MVM can access the PVM without introducing any significant overhead that could be caused by sending the commands and results of virtual machine introspection via network. However, the *CloudPhylactor* architecture does not handle the live migration of virtual machines while they are monitored.

To support live migration, the monitoring application must be migrated jointly with the monitored virtual machine. In the CloudPhylactor architecture, the apparently simple solution of independently migrating the PVM and the MVM is insufficient. First of all, careful synchronization between the migration of monitoring application and monitored virtual machine is mandatory, to avoid fatal failures of the monitoring application because of access to the (not yet or no longer locally available) PVM. Second, asynchronous monitoring requires the monitoring application to be aware of the migration, because the application needs to re-attach itself to the VMI-interface of the hypervisor. Third, synchronous monitoring intercepts the control flow in the PVM and uses callback functions that are triggered by the interception mechanisms. The interception mechanisms and callback functions need to be re-initialized after migration. Otherwise, the control flow in the PVM might trigger the invocation of a callback handler that does not yet (or no longer) exists, and in the worst case cause a failure of the monitored system. Hence, the live migration of monitored virtual machines must be carefully synchronized among all components.

TwinPorter extends the concept of CloudPhylactor to support live migration. To achieve that it migrates both MVM and PVM at the same time to the same cloud node. With this approach, we ensure that the MVM and PVM are always executed on the same cloud node and that operations required for virtual machine introspection are not sent via network, which makes sure that no additional overhead is introduced. In addition, our approach allows synchronous and asynchronous tracing of the PVM in the MVM. To achieve that, we implement a synchronization approach between the monitoring application and the migration tools on the source and target hypervisors. To keep the downtime of the PVM minimal and to make sure that there is no point in time during live migration where no VMI-based monitoring is possible, we extend the concept the pre-copy base live migration of Xen.

The main contributions of this section that discusses the *TwinPorter* architecture are:

- A novel live migration protocol that enables coordinated migration of an MVM and a PVM such that PVM downtime is minimized and continuous VMI monitoring of the PVM by the MVM is guaranteed;
- The support for synchronous and asynchronous VMI-based monitoring;
- The non-invasive design and implementation of this protocol for LibVMI-based introspection on the Xen hypervisor;
- An experimental evaluation of the performance and the downtime induced by our approach.

4.5.1 System Design

We make a few assumptions about the environment. We define these assumptions as generic as possible, to make them applicable to potentially different virtualization environments. Additionally, we explain the problems for monitoring a virtual machine that is live migrated in detail.

Local dedicated monitoring virtual machines. Our first assumption is that we have a concept for VMI-based monitoring of a target virtual machine (PVM) using a dedicated monitoring virtual machine (MVM) on the same host. This assumption yields the requirement of migrating both virtual machines together in a coordinated way.

A potential alternative is to run the VMI-based analysis framework as a normal process in the most privileged domain of the hypervisor. However, in this case, the VMI application itself would have to be responsible for transferring its state to the target host, since there is no generic solution for live migration of processes to another system. Thus, encapsulating the analysis application in a virtual machine minimizes the effort for migrating the state of the analysis.

Remote VMI operation, for example, supported by CloudVMI [BSM14], are an alternative to local VMI operations. Remote VMI eliminates the need for simultaneously migrating MVM and PVM, but it adds significant latency. For example, the CloudVMI authors show that a cross-host extraction of the process list in a sample use case takes more than three times longer than the same local VMI operation. CloudVMI supports only asynchronous VMI operations. We know of no system that supports synchronous VMI operation with a remote VMI tool. Probably the network latency would have a too disastrous impact on PVM performance. Due to these disadvantages, remote VMI operation is not an acceptable approach.

Fine-grained VMI access control. The second assumption is that we have fine-grained per-user access control for VMI operations. There is a static assignment of permissions that allow an MVM to access exactly one PVM. If we limited VMI operations to entities with privileged access to the hypervisor, such as the root user of a Dom0 for the Xen hypervisor, no access control would be required. However, as soon as ordinary cloud users should be enabled to use VMI applications directly, fine-grained access control is necessary that limits the privileges of some user's MVM to access only PVM's of the same user.

Live VM migration. The third assumption we make is that the virtualization environment supports live migration. The normal live migration of Xen is implemented using a pre-copy strategy, which uses these consecutive steps [Cla+05]

1. *Reservation:* The selected target host reserves the resources for the virtual machine to be migrated.
2. *Pre-Copy:* The source host transfers all memory pages of the virtual machine to the target host, while the virtual machine is still running. Afterward, in additional iterations, it sends updates for dirty pages (i.e., memory pages that have changed after the transfer) to the target.
3. *Stop-and-Copy:* The virtual machine is suspended on the source host, and the remaining dirty pages are transferred to the target.
4. *Commitment:* The target host signals the successful transfer.
5. *Activation:* The migrated virtual machine is resumed on the target host and removed on the source host.

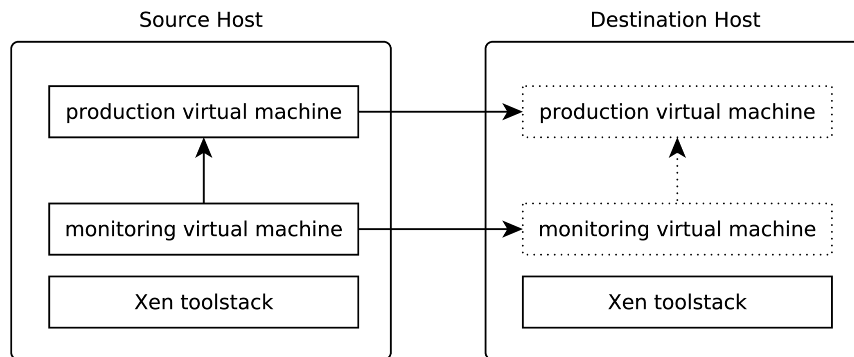


Figure 4.13: Migration of a PVM and MVM [TBR19]

Goals

Based on such an environment, we aim at achieving the following goals with *TwinPorter*:

- Ensure that the PVM never runs without the being actively monitored, i.e., we must not pause the MVM on the migration source before we pause the PVM, and we must not resume the PVM before the MVM. For synchronous VMI, we must make sure that there is always a callback handler available for monitored events in the PVM.
- Support synchronous and asynchronous operations of virtual machine introspection.
- Minimize the impact on the downtime of the PVM caused by VMI-based monitoring.
- Keep the changes to the hypervisor as small as possible.
- Fully transparent live migration of the PVM. The PVM is not required to be aware of VMI monitoring (in fact VMI operations should be stealthy and in the ideal case not noticeable by the PVM), nor of any differences compared to simple migration of a virtual machine.
- Automatically define access permissions such that VMI is permitted on the target system with the same permissions as we had on the source system.
- Do not increase the VMI overhead.

Problem Description

The parallel migration of the MVM and PVM has the advantage that the state of the monitoring application is implicitly migrated to the target node. However, this introduces the problem to synchronize the process of live migration among the involved components.

To support asynchronous VMI-based monitoring of live migrated virtual machines, the application needs to re-initialize the communication interface with the hypervisor. Otherwise, VMI-operations may fail or cannot be executed, and the system is not monitored anymore. Hence, the application must be aware of the migration of the monitored system.

Migrating VMI-applications that employ synchronous VMI-based monitoring is even more complicated because it requires that the correct callback handlers for software breakpoints are registered. Software breakpoints for VMI on Intel-based systems are implemented by replacing the original instruction with the `INT 3` instruction. The execution of this instruction causes a trap to the hypervisor which then can be handled by inserting the original instruction. When the monitored system executes such a software breakpoint while the VMI-application is not registered and does not handle the software breakpoint with inserting the original instruction, the monitored system may crash because the monitored system is not aware of the original instruction and does not know

how to handle the software breakpoint. Depending on where the breakpoint is, the corresponding userspace application or the kernel may crash.

A potential approach to solve these problems is to use the standard tools for live migration and migrate both PVM and MVM and pause the PVM during the migration. However, depending on the size of the main memory of both virtual machines, the downtime can increase significantly. Hence, the live migration approach should be extended so that the PVM is still executed during the pre-copy phase and that it is only paused for the stop-and-copy phase where no monitoring is possible because in this phase it cannot be ensured that the PVM and MVM are always running on the same system.

Concept

Ideally, the migration of the MVM should happen entirely transparently for the VMI. However, during our investigation, it turned out that fully transparent migration of MVMs is faced with the problem that VMI operations in the MVM require initialization of a hypervisor's VMI functionality, and additional initialization of callback and interception points for synchronous VMI. Such hypervisor-internal state is not migrated automatically by standard migration mechanisms of a hypervisor, and thus fully transparent migration would imply significant internal modifications to the hypervisor, which contradicts our goal of minimal changes to the hypervisor. For these reasons, it is acceptable for us to require a migration-aware VMI application that actively interacts with our migration infrastructure to satisfy the goals above.

We want to ensure that the PVM is never executed without being monitored. This means on the one hand that on the source host, the MVM must not be stopped before the PVM has been stopped. In the pre-copy phase of PVM's live migration – which usually takes the most time of the migration – the VMI application still needs to be running in the MVM. On the other hand, on the target host, the PVM must not be resumed before the VMI application in the MVM has received its state from the source host and completed initialization.

The whole process of the migration needs the following components, besides the common tools of the virtualization software:

- A migration tool (source) that supports the migration of the MVM and PVM in parallel informs MVM about migration and sends the target node the configuration of both VMs, including the access permissions between them.
- A migration tool (target) that sets the permissions of the MVM on the target host and informs the MVM about the current state of the migration.
- A fast communication interface between the migration tools (source and target) and the MVM to inform about the current state of the migration.

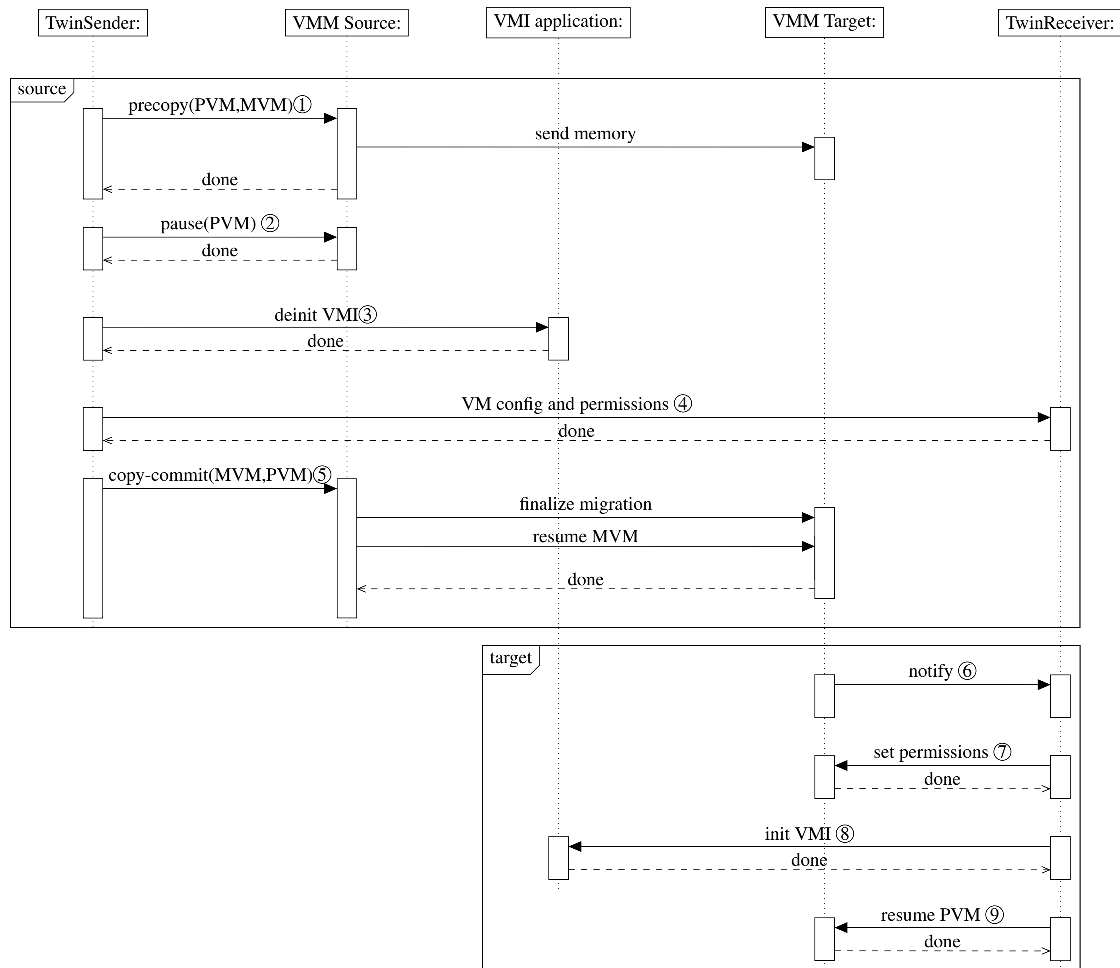


Figure 4.14: The interaction of the components during migration. The upper part is executed while both machines are executed on the source node. The lower part is executed when both virtual machines are running on the target node [TBR19]

The general steps for migration are the following (see Figure 4.14):

① The migration tool starts the migration of both virtual machines simultaneously on the source host; the target host needs to allocate enough resources for the PVM and MVM.

② The migration tool on the source host pauses the PVM after the pre-copy phase is finished.

③ The migration tool on the source informs the MVM to deinitialize the VMI-based monitoring.

④ The migration tool on the source sends the permissions of the virtual machines to the target.

⑤ The migration tool on the sources finalizes the migration. The PVM remains paused on the target host while the MVM is automatically resumed.

⑥ The target VMM informs the migration tool on the target host about the finished migration.

⑦ The migration tool on the target host sets the permissions of the MVM on the PVM.

⑧ The migration tool on the target informs the MVM that the migration is completed and that it can re-initialize the monitoring.

⑨ After the MVM acknowledged the re-initialization, the migration tool resumes the PVM.

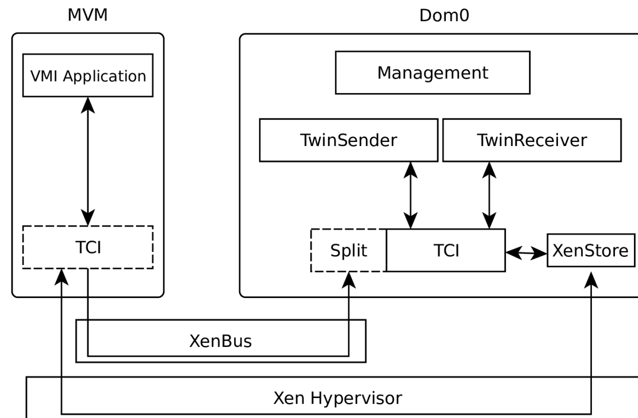


Figure 4.15: The communication channels of the *TwinPorter* architecture [TBR19].

4.5.2 Implementation

This section discusses the components of the *TwinPorter* implementation and their interactions (see Figure 4.15). The proof-of-concept implementation uses the Xen hypervisor. Xen [Bar+03] is a type 1 hypervisor and is located between the operating system and the actual hardware. It supports para- and hardware virtualization. The most privileged virtual machine that is started at boot and manages the other virtual machines is call Dom0. Other, unprivileged virtual machines are called DomU.

Components

The *TwinPorter* architecture requires four components. The first component is the *VMI application* which is performing virtual machine introspection based analysis of a PVM in the MVM. In addition to common virtual machine introspection based applications, it needs to be able to pause its analysis routine during the migration and to process messages that signal the start and end of a migration.

The second component is the *TwinCommunicationInterface (TCI)*. It serves as the communication interface between the VMI application and the tools handling the migration. The TCI has two components: an unprivileged frontend driver in the MVM, and a privileged backend driver in Dom0. If the migration tool in the Dom0 wants to inform the VMI application in the MVM about state changes, the migration tool needs to send the new state information to the backend driver. Afterward, the backend driver delivers this information to the frontend driver in the MVM using the XenBus. Then, the frontend driver sends a *SIGUSR1* signal to the corresponding process of the VMI application so that it can immediately react to the new state. The sending of state changes from the VMI application to the migration tool works similar but in reverse order.

The third component is the *TwinSender (TS)*. It is required to initiate the migration of the MVM and PVM and is executed in the most privileged domain (Dom0) of the source host. When the migration is started, it informs the VMI application via the TCI about the upcoming migration so that it can react to it. Additionally, it sends the permissions of the MVM required for VMI operations on the PVM to the target cloud node. Internally, the TS uses a modified version of the command line tool *xl* of Xen that supports parallel migration and behaves according to the steps described in Section 4.5.2. To achieve that, it needs to support the parallel migration of virtual machines and to implement the modified protocol for live migration (see Figure 4.14). The normal `xl` command does not support the migration of two VMs in parallel since it uses a user-space file lock in the reservation phase to ensure that enough resources are available for a virtual machine. This prevents the parallel migration of two virtual machines. To achieve parallel migration of two

virtual machines, we remove the lock operation in the prototype of *TwinPorter* to support parallel migration. For production use, the locking mechanism in the reservation phase should be extended to ensure that the target host allocates enough resources for the PVM and the MVM before the live migration starts.

The fourth component is the *TwinReceiver (TR)*. It is executed on the target host and waits for incoming virtual machines. The communication between the TR and TS is established via a TCP connection. As soon as the migration process is completed, the TR sets the permissions of the MVM on the PVM in the XenStore, which is a storage location for all virtual machines on a Xen system and contains their configuration and is used by split drivers. Additionally, the TR informs the VMI application via the TCI about the finished migration, so the VMI-application in the MVM can re-initialize introspection.

Permissions

TwinPorter uses the same concept as CloudPhylactor for defining the VMI permissions using the Xen security modules. The decision on whether access to a resource is granted is implemented using mandatory accesses control and is based on a set of policies and the labels assigned to a resource. These policies can be used to define the permissions of each virtual machine very fine-grained and cannot be changed at run-time [RVJ09]. However, a complete set of policies can be reloaded, unless the active policies prevent it. The label of a resource such as a virtual machine cannot be changed at run-time. During migration, the label of a virtual machine can be changed, but the proof-of-concept implementation currently does not do that. The permissions of entries in the XenStore are set for each entry and are independent of the security label for virtual machines used for the Xen security modules.

The Migration Process

The migration process of *TwinPorter* extends the default migration process for Xen by implementing the steps described in Section 4.5.1. As long as no migration is ongoing, virtual machine introspection can be performed without any limitations. The request for a live migration is generated, for example, by the cloud management and executed by the TS on the source node. Then, as a first step, the pre-copy phase of the migration is started which transfers the contents of the MVM and PVM to the target node. After the pre-copy phase of the migration is finished, the TS pauses the PVM. Additionally, the TS signals the VMI application via the TCI that a live migration for the given machine is in progress and that it should de-initialize the virtual machine introspection hooks and structures. As soon as the VMI application has called the LibVMI de-initialization routine, the VMI application informs the TR via the TCI that the migration can be finalized.

After the migration of both virtual machines is finalized, the TR restores Xenstore permissions for the TCI driver and resumes the monitoring virtual machine on the target host. Then, the TR tells the VMI application that the migration is finished and that it should re-initialize VMI functions. After the re-initialization, the VMI application continues with the normal monitoring and informs the TR that re-initialization has been completed and that the production virtual machine can be resumed. Then, the TR resumes the PVM and signals that to the PVM to finish the migration process.

Step	Async. Monitoring [s]	Sync. Monitoring [s]
pre-copy + pause PVM	16.42±0.5	16.43±0.48
stop VMI	0.03±0.0	0.05±0.04
send config	0.01±0.0	0.01±0.0
finalize migration	2.98±0.17	3.28±0.16
restart VMI	0.15±0.01	0.2±0.39
resume PVM	0.05±0.01	0.06±0.01

Table 4.5: Time required for the migration steps [TBR19]

4.5.3 Evaluation

The evaluation described in this section is executed on two machines with the following identical specifications: The CPU is an Intel(R) Core(TM)2 Quad Q9300, model 23, family 6 and consists of four cores, each of them having a processor base frequency of 2.50 GHz. Its instruction set is 64-bit, and it supports Intel VT-x and VT-d. Each host is equipped with 8 GB of RAM As hard disk, a TOSHIBA MQ01ABF050 with 500 GB size is used. The hard disk image of both virtual machines is stored on the sending host and exported via NFS to the receiving host. In the Dom0 of both physical hosts and in the MVM the *TCI* kernel module is loaded.

Each virtual machine has one vCPU, and the swap space is deactivated during the evaluation. The MVM is para-virtualized and has 512 MB of main memory while the PVM is hardware-virtualized and has 1024 MB of main memory.

VMI Application

We evaluate the downtime during migration, with two standard use cases of VMI. First, we run a VMI application for asynchronous monitoring based on the LibVMI process-list example that extracts the list of processes running in the PVM in a loop and pauses for 500 μs afterward. Second, we use a VMI application for synchronous monitoring that traces write access to the CR3 register which is a standard method for VMI-based monitoring and can be used to trace userspace context switches in the monitored virtual machine [JAA06]. Both VMI applications react to changes of the live migration, which are signaled via the *TCI* in such a way that the VMI application disables the process list extraction during migration and calls the functions `vmi_destroy` of LibVMI. After the migration is finished, the VMI application calls `vmi_init` and continues extracting the process list.

Normal Migration

To get a baseline for how long the migration takes in our environment, we measure the time that is required to migrate only the PVM by migrating the (idling) PVM ten times. For each migration, we measure the time of all migrations steps, i.e., the run-time of the xl command. In average, it takes 18.18±0.46 s to migrate the PVM from the source to the target node.

To estimate the downtime of the PVM during migration when the PVM is not reachable via network, we send a ping request every 0.01 s and consider the longest time frame without response as the downtime. Due to the measurement method, this downtime includes the time required for reconfiguring the network infrastructure when the VM is deployed on another system, and the time required for refreshing the ARP tables. In our test setup, we measure a downtime of 2.09±0.12 s in which the PVM does not respond to ICMP messages during migration.

TwinPorter Migration

To evaluate the live migration approach of *TwinPorter*, we measure the time required by all the steps in the TS and TR. To do so, we migrate the PVM and MVM in parallel for five times and

Migration	Time [s]
normal migration (only PVM, no mon.)	18.18±0.46
normal downtime (only PVM, no mon.)	2.09±0.12
parallel migration (PVM & MVM, async mon)	19.64±0.24
parallel downtime (PVM & MVM, async mon.)	3.83±0.20
parallel migration (PVM & MVM, sync mon.)	20.03±1.08
parallel downtime (PVM & MVM, sync mon.)	4.08±0.58

Table 4.6: Total time required for the migration and the corresponding downtime

measure the time required for each step. Table 4.5 shows the time: required for the pre-copy phase of both virtual machines; to stop the monitoring in the VMI application; to send the config of the virtual machines via network; to finalize the migration of both virtual machines; to reactivate the monitoring and the time required to resume the PVM. The total migration time with asynchronous monitoring is 19.64±0.24s and with synchronous monitoring 20.03±1.08s.

The downtime of the PVM using ping with asynchronous monitoring is 3.83±0.20 s and with synchronous monitoring it is 4.08±0.58 s. Hence, the overhead introduced by *TwinPorter* compared to normal migration of the PVM is about 2 s. All migration times are summarized in Table 4.6.

Limitations

TwinPorter requires stopping the VMI-based analysis in the MVM before the migration can be finalized. Hence, in the worst case, a malicious cloud user that owns an MVM could stall the migration of his PVM by not stopping the VMI-based analysis.

4.6 Summary

In this chapter, we introduced novel memory acquisition techniques for mobile devices and cloud computing. Each of the discussed data acquisition techniques leverages the features of the underlying hardware and hence is suitable for specific applications.

The first contribution, the framework for cold boot attacks on mobile devices, enhances the current state-of-the-art of cold boot attacks on ARM by minimizing the footprint of the analysis system on the analyzed system. With this approach, it gets possible to analyze the kernel data structures of the previously running operating system.

The second contribution addresses the problem that the implementation of security-related memory forensics applications for the TrustZone is complicated. This is primarily a problem because there is no framework, similar to LibVMI, available that at least provides the basic routines for accessing the main memory of the normal world from the secure world. We addressed that by implementing the interface of LibVMI in the secure world of the TrustZone.

The third contribution of this chapter addresses the problem that cloud tenants cannot use VMI-based operations on their virtual machines. The *CloudPhylactor* architecture tackles this problem by introducing the concept of monitoring virtual machines that have the permissions to run VMI operations on production virtual machines of the same customer. Besides, the cloud management is extended to give users the ability to start monitoring virtual machines to monitor their production virtual machines.

The fourth contribution deals with the problem of the *CloudPhylactor* architecture that the live migration of monitored systems is not possible. The *TwinPorter* architecture extends the *CloudPhylactor* architecture so that virtual production machines can be migrated in cloud environments. Additionally, the *TwinPorter* architecture ensures that there is no time interval during migration where the production virtual machine is running not monitored. For this purpose, both the production and the virtual monitoring machine are migrated simultaneously. To achieve this, we have extended Xen's live pre-copy-based migration approach to ensure minimal downtime of the productive virtual machine and it does not run unmonitored.

5 INFORMATION RETRIEVAL

Information extraction is one of the most important and challenging tasks of virtual machine introspection. It comprises three aspects: bridging the semantic gap, the timing of when to extract the information, and the performance of the extraction routine.

The first and most significant challenge is to bridge the semantic gap. Obtaining the semantic knowledge required to interpret the contents in memory is still challenging in practice. Thus, the associated research question is how to obtain the semantic knowledge of the data structure of applications.

Second, the timing aspect is vital for the extraction of transient data, i.e., information that resides only for a short time in memory. Otherwise, if the information extraction is started too early or late, it may fail because the data is not present (anymore) in memory. Hence, the associated research question is what are potential trigger mechanisms for starting the VMI-based analysis?

Third, the performance of VMI-based analysis can have a significant impact on the performance of the analyzed system. Traditional memory forensic tools operate memory snapshots and thus the performance of the extraction process is usually not important. However, VMI-based analysis tools have to extract information efficiently which mainly originates in the dynamic analysis, i.e., the process of extraction information when a sensitive instruction is executed. In that case, the virtual machine is paused until the data extraction is finished. The associated research task is to optimize the performance of the information extraction process.

Since the optimization of the extraction routines is application-dependent, we discuss how these questions can be addressed using two examples: the extracting of TLS session keys from memory and the implementation of a VMI-based SSH honeypot.

The remainder of this chapter is as follows: First, Section 5.1 gives an overview of the state-of-the-art approaches concerning information retrieval from memory, TLS decryption, and SSH honeypots. Then, Section 5.2 describes the *TLSKex* architecture that extracts the session key from virtual machines using a brute-force approach. Section 5.3 optimizes the key extraction of *DroidKex* by deriving the semantic knowledge of the data structures holding it and demonstrates the feasibility of the approach on Android applications. Section 5.4 discusses two approaches of how to implement a VMI-based SSH honeypot that uses control flow interception to monitor the behavior of attackers. Finally, Section 5.5 summarizes the approaches described in this chapter.

	Approach	Performance	Access	Cert. pinning	Security Level
Proxy	mitmproxy [Mit]	middle	Network	no	low
	Haystack [Raz+16]	middle	Android App	no	low
Decryption	ssldump [Ive14]		Private Key/Log	yes	good
	Wireshark [Wir15]		Private Key/Log	yes	good
Key Extraction	*TLSKex* [Tau+16]	low	Virtual Machine	yes	good
	TeLeScope [Car16]	low	Virtual Machine	yes	good
Control Flow	Cuckoo [Jur15]	high	Windows Process	yes	good
	jsslkeylog [Mic]	high	Java VM	yes	good
	Frida Pinning Bypass [Pie17]	middle	Android Process	yes	low
	DroidKex [TAR18]	middle	Android Process	yes	good

Table 5.1: Comparison of different TLS decryption solutions [TAR18]

5.1 State of the Art

This section describes state-of-the-art approaches for decryption of TLS connections, SSH honeypots, and information retreival from memory.

5.1.1 Decryption of TLS Communication

This section is based on [TAR18] and discusses different approaches and related work that addresses the problem of extracting cryptographic key material, decrypting network communication and recomputing data structures. There are three approaches for decrypting network communication: the usage of a man-in-the-middle proxy that modifies the communication traffic, a passive knowledge-based approach that decrypts network connections by having access to the session key, and the interception of the application control flow to access the key material. Table 5.1 shows a comparison between implementations of the three approaches. The comparison is made based on performance overhead at run-time, in which layer the key extraction occurs, the ability to overcome certificate pinning, and the impact on the security of the encrypted connections.

Man-in-the-Middle Proxy

The usage of man-in-the-middle proxies is the most common approach to decrypt TLS communication. An example of this approach is mitmproxy [Mit], which operates on network level. Thus, it does not require access to or patching any of the communicating parties. It modifies network traffic to use self-generated certificates allowing it to de-/encrypt communication between the two entities. Haystack [Raz+16] is an application that can be installed on a regular Android device. It redirects all traffic of applications using a VPN tunnel to a man-in-the-middle proxy to decrypt their TLS traffic. For these approaches, it is necessary, that the application installs a self-signed certificate. If an application uses key pinning, i.e., check the server certificate, this approach is not feasible.

By actively modifying the crypto parameters of a TLS connection, man-in-the-middle approaches lower the security of the communication. Carnavalet et al. [Xav16] summarize the weaknesses of widely used TLS Proxy implementations. US-cert published an alert concerning the usage of these systems [Cer17]. Durumeric et al. [Dur+17] also investigated the impact of proxy solutions on the security of TLS connections.

Knowledge Based

Another way to decrypt TLS communication is by knowing the cryptographic key material that is used for the symmetric encryption during a session. Wireshark [Wir15] and SSLdump [Ive14] are passive approaches that monitor the network traffic and decrypt TLS connections when appropriate key material is provided.

There are two ways to negotiate session keys with TLS, either using asymmetric encryption or by using the key exchange algorithms such as Diffie–Hellman (DH) and Elliptic curve Diffie–Hellman (ECDH) protocol. In the first case, it is sufficient to know the corresponding private key of the server to extract and decrypt the session keys used in encrypted sessions following the TLS handshake. The private key can be obtained for example by extracting it from the hard disk of the server that uses the key. Saxon et al. [SBH15] describe a way how to extract RSA keys from virtual machines efficiently. If the position of a key in a memory dump is not known, heuristics help to identify potential keys. Shamir et al. [SS99] describe theoretical approaches to find RSA cryptographic keys efficiently using stochastic information. Klein [Kle06] uses an alternative approach by searching for sequences of the ASN.1 encoding, which is commonly used to store such keys.

When Diffie-Hellman (DH) or Elliptic curve Diffie-Hellman (ECDH) is used to negotiate the symmetric key, it is not possible to obtain the symmetric session keys used in TLS sessions even if the private key of any peer is known. In such a case, session keys can be acquired if the application logs them or by extracting the master secret from the application's memory.

Control Flow Interception

The Cuckoo sandbox [OM13] instruments the pseudo random function PRF function of the local security authority subsystem service (LSASS) process of Windows to extract the server random (SR) and client random (CR) in addition to the MS [Jur15]. The lsass process is handling the TLS encryption in Windows. The PRF functions gets as a parameter the server and client random as well as the master secret which is used to derive the individual session keys. Hence, by intercepting the function call, the master secret and the server and client random can be extracted. However, this method does not work when an application comes with its own TLS implementation and hence does not use crypto services of the operating system.

Key material can also be extracted by intercepting the function calls of crypto libraries. For example, SSLKeyLog [Mic] intercepts SSL function calls using Java agents mechanism. Since Dalvik/ART environments do not support Java agents, SSLKeyLog cannot be deployed in Android. Wu [Wu16] intercepts OpenSSL function calls to extract the MS from the memory of applications and relies on the fact that the exact layout of the data structures is known. He uses the GNU debugger (GDB) to intercept the control flow of applications.

Cipolloni [Pie17] describes an approach with the Frida framework that manipulates the control flow of Android applications so that it accepts the self-signed certificate of a man-in-the-middle proxy to bypass the problem of certificate pinning. As described before, this method can lower the security of encrypted communication.

Lee and Wallach analyzed the lifetime of the TLS master secret in Android applications [LW18]. For finding the master secret in main memory, they use a format based approach that uses knowledge about the surrounding data in the `SSL_Session` structure of BoringSSL, which allows them to locate the master secret in seconds from gigabytes of a memory image. They found out that the keys might remain recoverable in main memory even after a connection is terminated. They also proposed changes to Android that mitigate those problems.

All of these approaches that manipulate the control flow require semantic knowledge about the monitored application such as the data structure layout and the functions that are used by the application. Thus, these solutions often do not work for malware analysis, when the application is obfuscated or when it uses its own TLS library.

	Concept	Interaction	Shell commands	Stealthiness	File Change Detection	sftp and scp	Session Reconstruction	Playable Log	Port Forwarding	Extracted information
Kojoney [Cor06]	Emulation	○	◔	○	◔	○	◔	○	○	◔
Cowrie [Oos14]	Emulation	◑	◕	○	◕	●	●	●	◑	◑
SSHHiPot [McM16]	MiTM	●	●	●	○	●	○	○	●	◑
ssh-mitm [Tes17]	MiTM	●	●	●	○	●	●	○	●	◑
Sarracenia	VMI	●	●	●	●	●	●	●	●	●

Table 5.2: Comparison of SSH honeypots. A full circle means that a feature is supported or has high performance, and an empty circle means the opposite. [STR18]

5.1.2 SSH Honeypots

This section discusses state-of-the-art approach of honeypots and is based on [STR18]. The main problem of honeypots is to provide an environment that cannot be distinguished from normal systems while still providing in-depth traces of attacks. Hence, honeypots can be classified as high, medium, and low interaction based on the amount of features they mimic. Low interactions honeypots, for example, do not provide detailed traces since they only provide a very reduced set of feature of the original system. Hence, they are easy to detect by attackers. However, they can provide alerts for potential misuse. Figure 5.2 summarizes the features of the most common SSH honeypots.

Kojoney [Cor06] is a low interaction SSH honeypot and provides only a rudimentary interface to an adversary. It logs the username, password, and the entered commands. Cowrie [Oos14] is a medium interaction SSH honeypot and extends Kojoney with some additional features that make it behave more like a real Debian system. For example, it allows creating files and provides commands that emulate a real system. Both, Cowrie and Kojoney are written in python and do not provide an attacker access to a real system.

SSHHiPot [McM16] and SSH-mitm [Tes17] are a high-interaction SSH honeypots. They implement a man-in-the-middle SSH proxy and redirect an attacker to a real system, which makes it difficult for an attacker to detect the monitoring. The downside of this approach is that the system state is not monitored. For example, if an attacker downloads binaries over a TLS secured connection and deletes them directly after the execution, this approach fails to reconstruct the downloaded binary.

Lengyel et al. [Len+12] implemented VMI-Honeymon which is a hybrid honeypot architecture. It combines a low-interaction honeypot to collect malware and a high-interaction honeypot (sandbox) to analyze the captured malware. To analyze the honeypot, they use volatility to scan the main memory periodically. However, this approach does not provide means to dynamically trace the activities of an attacker. Furthermore, they do not discuss how VMI can contribute to make the anlaysis stealthy.

5.1.3 Stealthiness of VMI

While, at first glance, VMI-based analysis is isolated from the analyzed virtual machine due to the virtualization layer, the tracing can often be detected and is not stealthy. The stealthiness is critical for the implementation of honeypots and the analysis because malware and attacker can behave differently when running in a monitored environment to evade the analysis [Bal+10; HR05]. Testing whether the system is monitored can be done by searching for artifacts, such as breakpoints in memory, or by testing whether the system is emulated [RKK07]. Holz and Raynal [HR05] describe different approaches on how attackers can detect whether a system is a honeypot based on the emulation technique.

Another approach for attackers to detect the presence of monitoring tools is to measure the perfor-

mance of certain function and system calls because tracing function calls reduces performance so that the total runtime is longer. To analyze this, Tuzel et al. [Tuz+18] discuss the stealthiness of VMI-based tracing using LibVMI and the Xen hypervisor and measured timing changes of memory access instructions at the run-time of a program caused by VMI operations. The result of this work is that the presence of monitoring can be detected.

Miramirkhani et al. [Mir+17] describe another approach that analyzes how wear-and-tear artifacts can be used to detect whether a system is a sandbox or a real-world system of a user. Such artifacts are for example the recently opened files or the amount of currently running processes. The absence of those traces can be a indication whether a system is a real used system or a sandbox environment. Even if a tracing technique is completely secret, this approach would allow malware to evade the analysis based on how the environments looks.

In the past years, many approaches for malware analysis have been presented [BY17]. Barecloud [KVK14] addresses the problem of evasive malware that checks the presence of virtualization. To detect whether malware behaves differently on a normal PC and a virtual machine, they run the malware sample on a bare-metal computer without any active tracing, but they stored the changes to the disk. Additionally, they run the sample in Anubis [Int], Ether [Din+08] and Virtualbox and compared the behavior of the malware in the different systems. This approach requires many resources (physical and virtual machines) for tracing and thus does not scale very well for many samples. Additionally, they only detect that there is a difference in the behavior, but they do not provide means to actually analyze malicious behaviour.

5.1.4 Information Retrieval from Memory

There are different approaches and techniques on how to bridge the semantic gap to locate data in the main memory. A survey of different approaches that bridge the semantic gap is presented by Jain et al. [Jai+14] and Xu et al. [Xu+17]. Jain et al. [Jai+14] distinguish between four different techniques to bridge the semantic gap:

- *Hand-crafted data structure signatures* are obtained with experts knowledge.
- *Automated learning and reconstruction* use source code analysis and debugging information. Those approaches often use a training phase to deduce semantic information in several iterations.
- *Code implanting* injects code to running a virtual machine to retrieve information. It does not interpret the contents in memory but relies on the fact that the injected code can retrieve correct information and that it does not rely on possibly tampered functions (of the kernel).
- *Process outgrafting* relocates in-guest agents to a dedicated virtual machine, which has access to the main memory of the monitored system.

The most prominent approach for information retrieval of information stored in main memory is implemented by Volatility [Fou] and Rekall [Coh14]. Both tools employ semantic knowledge stored in profiles that are obtained from debugging information, which is, for example, stored in the DWARF format of Linux binaries [Com+10]. Their information retrieval approach does not work when no profile is available or contains wrong information, e.g., when the operating systems or application is not known or in a different version. Additionally, they are not optimized for performance and are not suitable for online monitoring and the extraction of ephemeral data, when the analyzed system requires to be paused for the extraction.

Sigpath [Urb+14] uses a data-centric approach to compute graphs of data structures in memory of unknown applications, which means that they do not need the source code, debugging symbols, or API information from the target program for the analysis. Hence, this approach is classified as automated learning and reconstruction. They create several snapshots of the application and compute the graph based on them. This approach mostly works on simple data structures that can be accessed by following a static path. Data structures such as lists, unions, and arrays are

not supported. Additionally, this approach does not work when the application uses obfuscation techniques that aim to randomize the data structure layout [LRX09]. They applied the approach for game hacking and memory forensics. In both cases, the required information stays longer in the memory of the analyzed application. However, they do not discuss an approach that could be used to trigger the extraction of ephemeral information.

Dolan-Gavitt et al. [Dol+11] present Virtuoso, which uses an instruction-centric approach to narrow the semantic gap. Jain et al. [Jai+14] classify their approach as process outgrafting. Virtuoso monitors the instructions of a process in several iterations and isolates those that are used to access the required information. Their approach seems to be promising, even so, it is very complex since it requires tracing applications on instruction level. Unfortunately, they do not provide the source code of their implementation.

5.2 TLSKex: Content-based TLS Session Key Extraction from Virtual Machines

The extraction of TLS session keys from the main memory is a good example to discuss the challenges of VMI (the semantic gap, performance, and timing) and possible solutions. In this section – which is mostly based on [Tau+16] – we present the *TLSKex* architecture that implements that approach. The extraction of TLS sessions key from the main memory of applications has several advantages compared to state-of-the-art TLS analysis solutions. For example, it can be applied when a man-in-the-middle based proxy approach is not feasible, e.g., when the client application implements certificate pinning. Such a solution can work in a *non-intrusive* and *universal* way. Being *non-intrusive* implies the following two requirements:

- *No active manipulation of communication:* The communication should be monitored passively without modifying the contents of the communication (we do not exclude a possible impact on the timing of messages).
- *No modification of the monitored application:* The decryption should work without internal modifications to the communicating applications (such as exporting the session key to a file).

Being *universal* implies the following four requirements:

- *Independence of specific key exchange:* The decryption key extraction should work for any key exchange algorithm.
- *Independence of encryption algorithm:* The decryption and key extraction should work independently of a specific cryptographic algorithm.
- *Independence of client/server role:* It should work for local applications that operate as a server, as well as for local client applications that connect to a remote server.
- *Independence of the implementation:* The key extraction should work for every TLS implementation. The only assumption we make is that the application stores the master secret in a consecutive 48 byte array.

To implement such a solution, the following research challenges must be solved: First, how can the session key of a TLS connection be located in main memory, when no semantic knowledge about the applications is given, e.g., for malware analysis? Second, when should the extraction routine be started and how can it be determined that the session key of a TLS connection is in memory? Third, how does the extraction affect the monitored system and how can the performance be improved?

With *TLSKex*, we present an architecture that extracts the TLS session keys from the main memory of virtual machines. The main idea of *TLSKex* is to take a snapshot of the application in a virtual machine after the application negotiated a new TLS connection. For the identification of the key in memory, we use a brute-force approach, which iterates over the snapshot and decrypts the first encrypted TLS record of the corresponding connection with each byte sequence. Each byte sequence serves as a master secret to derive the required session keys. The key was found when the message was successfully decrypted. To detect a new TLS connection, we monitor the network traffic of the analyzed system. To improve the speed, we implemented and evaluated several heuristics that allow ignoring large portions of the contents in the main memory to limit the search space. This section is based on [Tau+16].

5.2.1 System Design

This section presents all the components that are necessary to decrypt TLS secured connections (see Figure 5.1) The TLS decryption process can be separated into two stages – an on-line and an off-line part. The on-line functions must be executed synchronous to the TLS communication and include capturing of the network traffic, detecting TLS sessions, and taking a snapshot of the

memory in which the key is stored, e.g., a snapshot of the whole virtual machine main memory. The off-line part can be executed later, whenever the decrypted content of a TLS connection needs to be accessed. The process of extracting the master secret from memory uses the information captured by the online parts.

Network Logging

Network logging is responsible for capturing the network traffic of all TLS connections that shall be monitored. This task can be executed by any standard network logging tool, such as a dedicated device on a promiscuous network switch port or a local capturing process on the host running a virtual machine.

Trigger Mechanism

The acquisition of virtual machine memory must be triggered at the right point in time when the key material is present in main memory. The master secret and the derived key material is available as soon as the TLS handshake for key negotiation has finished and the key calculation has been executed. According to the TLS protocol, a CCS message is sent when a node is ready to use new key material in subsequent messages. Hence, the right moment for taking a snapshot can be detected by monitoring the network traffic for TLS records that contains the CCS message.

During a TLS session, the key material can be renegotiated. This means that the key extraction routine must be triggered each time when new cryptographic parameters are exchanged. We can detect the subsequent CCS messages in the network traffic even if these messages themselves are encrypted because they are sent with a unique payload type in the TLS record layer header, which is not encrypted (as explained in the previous section).

Memory Acquisition

The memory acquisition must be performed synchronously to the triggered snapshot request and network traffic. If it is performed asynchronously, the connection or the program might be terminated, and the key could be gone. Thus, it is essential that the memory acquisition is executed synchronously. Additionally, it is important to take the snapshot fast to decrease the impact on the timing of network communication.

There are several ways how the time required to take a snapshot can be decreased. For example, LibVMI supported copy on write snapshots of virtual machines. However, this feature is currently not available anymore[1]. The other way to decrease the time is to reduce the size of the snapshot. However, this requires contextual knowledge about the guest operating system, for example, which process is communicating and where its memory is located in physical memory.

Key Extraction

There are several ways how the master secret can be obtained from a memory snapshot. In contrast to RSA keys, there is no standardized way to store the master secret of a TLS session. Thus, there is no general approach to find it with a simple pattern matching approach. However, there are several other ways of how this can be achieved.

One way is to parse the structures of a process to find the TLS session structure, which requires contextual knowledge about the program. Thus, this approach is not feasible for unknown programs such as malware. However, this approach is very fast because no complex computation is required. The search routine only needs to follow pointers in memory.

A similar alternative is to search for well-known TLS session structures of different implementations that include the master secret. These sessions structures often contain values such as the TLS

[1] LibVMI `https://github.com/libvmi/libvmi/commit/3a53dcde860a0f9f8bbde83a05671a7d44d6c56a`, Accessed on 2019-07-08

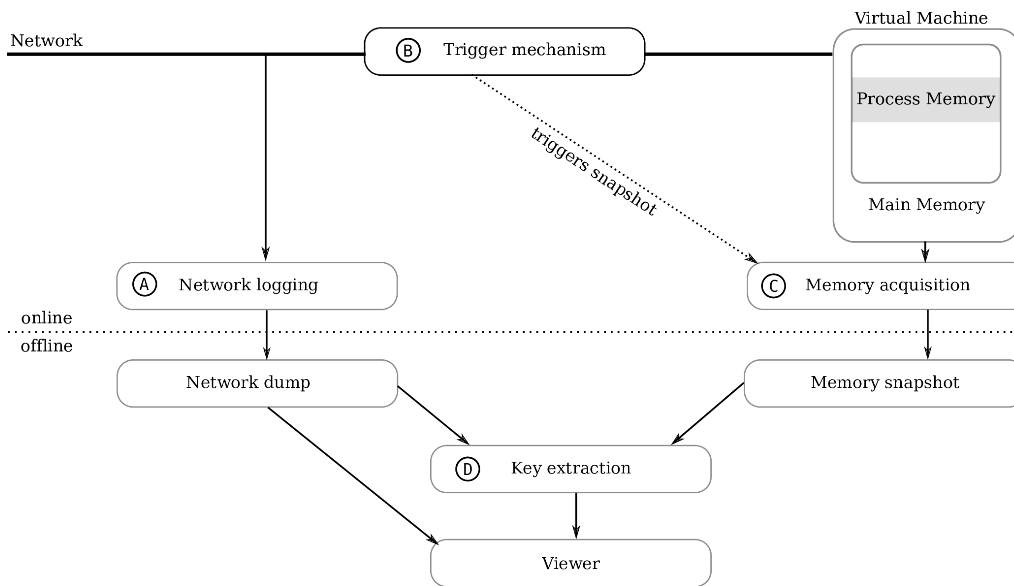


Figure 5.1: The main steps for decrypting TLS connections: network logging, TLS detection, memory acquisition, and key extraction [Tau+16]

version or the IP address of the communication partner. Thus, these structures can be found easily when some parts of them are known [Hom13]. This approach requires searching for parts of the key structure in the whole address range of a process, but no complex computation is required to identify the key. However, the key structure must be known a priori. If malware uses an unknown TLS implementation, this approach does not work.

When no a priori knowledge about a process is given, there is still the option to try every byte sequence as a potential master secret. This approach is slow as the testing of a byte sequence includes the key derivation and the decryption of a data block. Thus, first of all, the size of the snapshot should be reduced to memory areas that potentially contain the key. For example, in most of the cases, it would not make sense to search the master secret in the read-only mapped text segment of a process. With a high probability, it is stored in a memory region that is writeable as it is negotiated dynamically at runtime. This approach can be further optimized with heuristics that filter byte sequences that are no possible keys, e.g., by checking the entropy.

After a potential master secret is found in memory, it must be tested whether it belongs to the corresponding connection. This can be achieved by decrypting a TLS record and verifying the HMAC that is included in every TLS record. As the HMAC is computed over the sequence number of a TLS record, this number must be known. Thus, the first encrypted message should be used as it always has sequence number zero.

5.2.2 Implementation

TLSKex implements the concepts described in the previous section. It is a framework written in the programming language C that uses LibVMI to access the memory of a virtual machine running under the Xen hypervisor.

TLSKex is designed with the focus on the following goals: It shall obtain the master secret of a virtual machine without active manipulation of the TLS channel itself, and without manipulation of the communicating application. *TLSKex* shall work independently of the specific key exchange mechanism and selected cryptographic algorithms and independent of whether a client or a server application runs within the virtual machine.

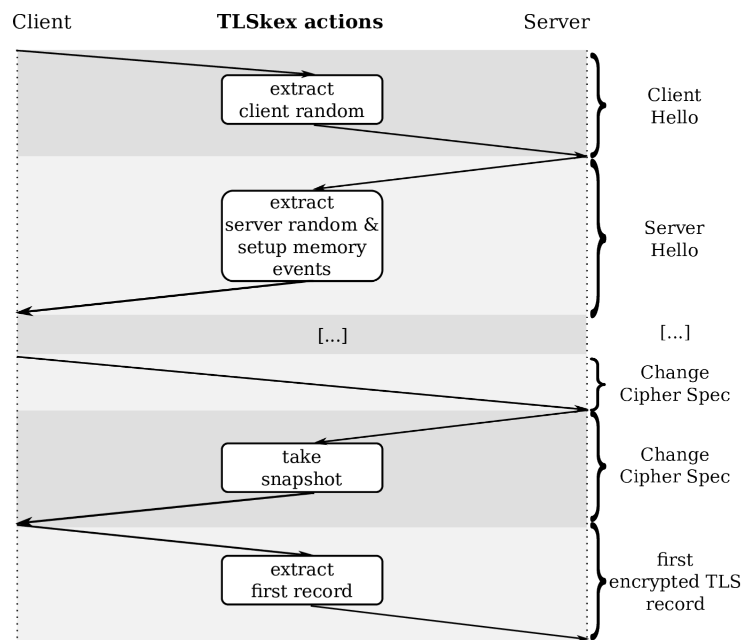


Figure 5.2: TLS key negotiation process and the corresponding TLSkex actions [Tau+16]

Trigger Mechanism

We need to trigger the snapshot process after observing a network packet containing a CCS message, but before the connection is closed and the master secret removed from main memory. It is not sufficient to monitor the network passively and trigger the key extraction process asynchronously. With such a passive approach, triggering the creation of the memory snapshot might take longer than the lifespan of the TLS connection, and thus fail to capture the master secret. This is primarily a problem for very short living connections. Instead, we implemented an *active network monitoring* approach.

Thus, TLSkex includes an active network monitoring component that is able to analyze every packet coming from or to the monitored virtual machine. The network analyzer forwards packets only after they have been inspected. It is equipped with two virtual TUN network interfaces. One interface is bridged to the virtual machine, and the other interface is bridged to the rest of the network. In general, the network analyzer receives a packet from one interface, analyzes its content, and writes it to the other one.

The network analyzer must recognize various types of TLS messages. These messages together with the corresponding actions of the network analyzer are depicted in Figure 5.2. The most important message is the CCS because it triggers the memory snapshot[2]. As both communication partners send a CCS message, the snapshot is triggered when the monitored virtual machine sends it. The corresponding packet is not passed to the destination interface until a memory snapshot is taken. Additionally, the client and server hello messages must be monitored. They contain the client and server random, which is extracted from the network packets.

Memory Acquisition

Every time the trigger mechanism detects a new TLS connection, the master secret that is present in virtual machine memory needs to be recorded. This can either be achieved by directly searching

[2]The CCS and the first encrypted TLS record do not need to be sent in two separate packets. They can also be transferred in the same network packet

the memory for the key, or by taking a snapshot of (parts of) the virtual machine and extract the key from the snapshot later when the communication needs to be decrypted.

In both cases, the snapshot process has to be executed fast to keep the delay impact on the connection as low as possible. Therefore, we decrease the time required to take the snapshot by minimizing its size. Thus, we have to find out where the master secret is stored in the main memory of a virtual machine. First of all, we restrict the snapshot to the memory of the process that handles the TLS session. Therefore, we parse the kernel task structures of each process to find the process that handles the connection. To get this information we parse the file descriptor table of each process in the task structure and compare the source and destination IP/port combination. For this purpose, we wrote a custom utility extracting the required information from the guest Linux kernel as we found existing tools like Volatility or Rekall as too slow for that time-critical operation.

Additionally, we consider only those pages of a process that are mapped writeable and anonymous. Anonymous pages do not have a reference to a file in their description structure. This usually holds for the heap and stack of a process as they are allocated dynamically.

Furthermore, we decrease the size of the snapshot by considering only pages that have been altered between the establishment of the connection and the key negotiation of the session key. To do so, we register memory access handlers that monitor the process memory between the sending/receiving of the SH record and the CCS message. The occurrence of the SH marks the last point in time where the session key is not existing and the CCS message indicates that that key was computed. Thus, the key can be found in pages that have been modified or newly allocated during this period.

This approach does not work, when the session key is stored in main memory before the connection is established, e.g., for resumed sessions. In this case, we can assume that we have captured the first key negotiation and do not need to extract it again. If this is not the case, only a snapshot of the whole address space of a process works.

Key extraction

TLSkex implements a brute force approach to find TLS master secrets in main memory. To achieve that, it takes each 48-byte sequence in the snapshot as a master key and checks whether its derived *write_key* can successfully decrypt a TLS record of the connection. To test whether the decryption was successful we compute the HMAC of a decrypted TLS record and compare it with the given one. If they match, the key is correct. All necessary parameters, e.g., the server random, are extracted from the network flow by the proxy component.

As the key validation process is slow, it is vital that we do not test every byte sequence as a master secret. Hence, we have implemented several strategies to pre-check whether a byte sequence is a potential key. The first optimization is that we assume that a key is stored four bytes aligned in memory. This increases the speed by a factor of four. The second optimization is that we look at the stochastic properties of a byte sequence. As the master secret is generated by a pseudo-random function, we assume that it contains about the same amount of zero and one bits and the amount is distributed binomially. For example, a 48-byte sequence that is randomly generated and binomial distributed has with a probability of about 90 percent between 176 and 208 one bits. The parameter μ is the expected count of one bits in the master secret. The TLS master secret is 48 bytes long and the probability p that a bit is one is 0.5. Thus, μ is 0.5 x 48 x 8 = 192.

$$\sum_{\mu-k}^{\mu+k} \binom{n}{k} p^k * (1-p)^{n-k} >= 0.89 \qquad (5.1)$$

$$k = 16, \mu = 192, p = 0.5$$

Process	total	anon&write able	new	modified	dumped	$\mathbf{t}_{snap_start}$	$\mathbf{t}_{snap_stop}$	$\mathbf{t}_{search}$
Apache2	72090	3715	0	26	26	4.3 ms	4.4 ms	30 ms
Curl	38264	3438	15	19	34	3.3 ms	4.0 ms	2 ms
Wget	22813	1378	16	13	29	4.0 ms	3.5 ms	2 ms
s_client	6114	152	9	22	31	0.4 ms	0.6 ms	8 ms

Table 5.3: Amount of mapped and changed memory pages (4096 bytes) of different processes during the key negotiation procedure and the time to prepare (t_{snap_start}) and take (t_{snap_stop}) a differential snapshot; t_{search} denotes the time to extract a key from a snapshot [Tau+16]

Thus, we first test bytes sequences in the snapshot that have between 176 and 208 one bits. If we do not find a key with these properties, we increase the borders and iterate again over the snapshot. Another heuristic to minimize the search space is to check whether a byte sequence consists only of ASCII characters. If so, the highest bit of each byte must be unset. The probability that such a 48-byte sequence with only ASCII characters is a key is about 0.5^{48}, which is negligibly small. The number of required characters could also be reduced, but the chance that a key gets pre-eliminated increases. The last heuristic that is implemented in TLSkex is to check whether an eight-byte sequence contains either only one bits or only zero bit. This can be tested easily as the bit counting function takes eight bytes as input and returns the number of one bits. The probability that a key contains at least one sequence with 64 zero or one bits is also negligible ($2 \cdot 48 \cdot 0.5^{64} \approx 10^{-15}$). *TLSKex* combines all three heuristics to decrease the search space as much as possible without eliminating too many potential keys. If the key was not found, the heuristics can be switched off in order to search again for the key in the snapshot. After we have found the session key of a connection, we write it into a key log file that serves as in input for Wireshark to decrypt TLS streams.

Network Logging & Traffic Decryption

TLSKex extracts only the master secret of TLS connections from main memory. It does not save the corresponding network traffic. To save the network traffic standard tools like tcpdump [TCP] can be used. Depending on the use case, the network sniffer should store only TLS connections to save space.

Moreover, *TLSKex* does not decrypt TLS sessions directly. It only extracts the TLS master secret from main memory. However, it can be used together with Wireshark to decrypt TLS encrypted network connections based on the extracted master key.

5.2.3 Evaluation

In this section, we measure the time to take a snapshot of several programs and the time that is required to extract the key out of it. Additionally, we discuss the limitations and stealthiness of *TLSKex*. All measurements in this section are executed on a machine with an Intel(R) Core(TM) i5-2500 CPU @ 3.30GHz and 4 GB of RAM. The hypervisor is Xen in version 4.4.1 out of the Debian stable repository. The dom0 and guest operating system is a 64-Bit Debian Linux with kernel version 3.16.0-4.

Memory Acquisition

To measure the throughput of the memory acquisition process we read the whole main memory of a virtual machine using LibVMI. To do so, we requested each page sequentially by starting from the guest physical address 0. For a guest system with 1024 MB of main memory the snapshot took 2.4 seconds, with 256 MB it took 0.6 seconds. These results do not include the time for storing the data on persistent storage. It only includes the time for the read operation, but not for the write operation.

	a						b	c	d
Process	**k=1**	**k=2**	**k=4**	**k=8**	**k=16**	**k=32**	**no string**	**not all 0 / 1**	**combined (k=16)**
key included	8.12	16.2	31.6	58.5	89.7	99.9	$1-10^{-15}$	$1-10^{-19}$	87.7
Apache2	0.10%	0.28%	0.64%	1.27%	2.33%	4.26%	85.49%	43.54%	1.69%
Curl	0.15%	0.45%	1.04%	2.11%	3.50%	4.75%	77.53%	10.55%	3.32%
Wget	0.15%	0.46%	1.06%	2.15%	3.60%	4.91%	78.10%	10.68%	3.38%
s_client	0.054%	0.18%	0.49%	0.96%	1.89%	3.40%	56.52%	37.35%	1.63%

Table 5.4: First row: probability that a key is not eliminated by the heuristic. Other rows: percentage of a memory snapshot that contains a 48 byte long and four byte aligned sequence with: a) $192 \pm k$ one bits, b) the byte sequence is not an ASCII string c) no 8 byte sequence with only zero or only one bits d) a to c combined [Tau+16]

Table 5.3 depicts the total amount of memory pages of a sample set of processes, the number of pages that are mapped anonymous and writeable and the amount of pages that are allocated (new) and altered (modified) during the key negotiation process and dumped as a snapshot. The time t_{snap_start} describes the time that is required to set the memory events and t_{snap_stop} the time that is required to take the snapshot and find newly allocated pages. The time that is required to extract the session key out of a snapshot is denoted t_{search}. All values in this table are based on single run measurements to provide the dimension of the timing values. The Apache2 process acted as a server, the other ones as clients.

The first notable observation in Table 5.3 is that the differential snapshot is tiny compared to the size of the anonymous and writeable pages. For example, the differential snapshot of the Apache2 process is about one percent as big as the snapshot of the anonymous and writeable mapped pages. This decreases the search space for the master secret dramatically.

Additionally, we can see that the time required to take a snapshot does not only correlate to its size. It also depends on the size of the address space of an application. For example, the differential snapshot of the s_client process has about the same size as the other ones. However, it is six to ten times faster than the other ones because its address space is smaller. This is caused by the fact, that we iterate over the address space to set/remove the memory events to detect modifications on each page.

Key Extraction

In Table 5.3 we present the time that is required to extract the TLS master secret from the memory dump of different processes. The different times to find the key by having the same snapshot size are caused by the position of the cryptographic key and the entropy of a snapshot. For example, if the key was stored in the heap of the process, it is at the beginning of the process dump and is found faster. However, when the key is on the stack, it is in the rear part of the memory dump, and it takes longer to find it. Additionally, the time depends on the number of bytes that are filtered out by our heuristics.

Our brute-force implementation was able to test about 131 thousand keys per second. This means that every sequential 48-byte sequence of a memory snapshot with a size of 131 KB can be tested in one second. The low throughput is mainly caused by the key derivation of TLS and the decryption process. However, this can be executed offline and does not affect the monitoring process.

In Table 5.4 we show how the performance of different heuristics can decrease the size of the search space of a memory snapshot by selecting only byte sequences that have the characteristics of a random encryption key. The first row shows the computed probability that a key matches the heuristic.

The first six columns show how many four-byte-aligned 48 byte sequences have between $192-k$ and $192+k$ one bits. For k=16 only about two to four percent of the memory meet this condition.

However, there is a chance of about 90% that a random master key satisfies this condition. In other words, we find the master key with a probability of about 90% if we just search the small part of the memory that matches this condition. The bit counting heuristic is implemented very efficiently. Thus, the pretesting reduces the time for finding the key dramatically.

In Table 5.4 we depict as well the performance of two other heuristics. The first one is testing whether a byte sequence is not an ASCII string. This is accomplished by testing if the highest bit of each byte is set. Depending on the application, this simple heuristic reduced the time for the key search in our experiments by between 56% and 85%. Another approach is to test whether each of the eight-byte blocks, where we count the bits in, has either 0 or 64 one bits. This reduced the search space by between 10% and 44%, again depending on the application process. The last column shows how much of the memory snapshot contains potential keys when all presented heuristics are combined.

Network Proxy

Two factors mainly influence the performance of the network: (1) The overhead of each packet caused by the deep packet inspection proxy that analyzes the contents of the TCP headers to check if it belongs to a TLS connection. We currently ignore this overhead in our proof-of-concept implementation as it is a constant factor that affects all packets. (2) The overhead that is caused by analyzing the TLS records and the corresponding actions, e.g., the acquisition of the snapshot depends on the TLS records that are included in a TLS record. For example packets with only application, data records are simply forwarded. The only packets that are delayed noticeable are packets with a SH and outgoing CCS records. Packets with a SH record are delayed by t_{snap_start} and CCS messages by t_{snap_stop}. Both values depend on the size of the address space of a process and the amount of changed pages.

Limitations

We have made several assumptions to increase performance. For example, we only take a snapshot of the process that handles the connection. However, malware might spawn a dedicated crypto process that runs the encryption routine. In that case, our approach would not work. Thus, it might be better to save a snapshot of the whole virtual machine. A malware might also obfuscate the key in memory, e.g., by shifting the byte order to hide it from *TLSKex*. However, a human operator might notice this problem when the key was not extracted and can implement a custom strategy for the specific use case.

Another way to circumvent the automatic key extraction process of TLSkex is to use a slightly changed version of the TLS protocol, e.g., by modifying the default numbers of some commands. However, this requires that the client and the server use the same modified protocol version.

TLSkex trusts the kernel structures of the guest to be uncorrupted and reliable and uses them for example to extract memory mappings of a process. Thus, a guest system might foil VMI based analysis by placing crafted data structures in memory [Bah+10]. This is a general problem of VMI based analysis and is out of the scope of this paper.

Finally, TLSkex could serve as a DoS vector, for example, when many TLS connections are established, and many snapshots must be taken. Thus, it is important to find ways to minimize the overhead of TLSkex in the future. Additionally, we have to investigate how this problem can be circumvented in practice.

5.3 DroidKex: Data structure-based Key Extraction from Mobile Phones

The *TLSKex* architecture – presented in the preceding section – is a generic approach to decrypt TLS based communication by extracting the TLS session keys from main memory of processes with unknown application logic. The main disadvantage of this approach is that the data acquisition and the brute-force key extraction is too slow for production environments, even though it was already optimized using several heuristics. This holds especially for applications that have a big address space and use multiple TLS connections at once.

The *DroidKex* architecture aims to address the problem of performance by using two assumptions about the application being analyzed. The first assumption is that the application is using the Boring/OpenSSL or libnspr library for TLS connections. However, the concrete data structure layout – which can change among versions – is not known. This assumption holds especially for most of the Android applications. The second assumption is that the monitored application uses network related system calls (send, recv, write and read) for communication. Hence, instead of monitoring the network traffic we can monitor system calls as a trigger mechanism. Additionally, we can use the parameters on the stack of an application invoking a system call as a starting point for looking up the TLS session related data structures.

The main idea of the DroidKex architecture is to improve the key extraction process of TLSkex by directly accessing the TLS master secret instead of using a brute-force approach. To do so, it computes the data structure layout of data structures that store it in main memory by deriving it from memory snapshots using a depth search approach. To get a pointer to the data structure holding the metadata of TLS connections, we intercept the control flow of the analyzed application while it is handling the connection. In this way, we can also ensure that the key is in memory when we attempt to access it. DroidKex works on Android applications, even so, this approach can be used with virtual machines. This section is based on [TAR18].

5.3.1 System Design

This section discusses the general approach of the DroidKex architecture, the assumptions under which it works and the components that are required for the proof-of-concept implementation and the requirements on the target device.

Approach and Goals

The goal of the *DroidKex* architecture is to extract the ephemeral information required for decrypting a TLS connection of an Android application from its main memory, namely the MS that is required to derive symmetric session keys. The key extraction is executed synchronously to the control flow of the application, i.e., the MS is extracted during a TLS session. Otherwise, there is no guarantee that the MS is still in main memory because an application might free or overwrite it directly after the connection terminates. Thus, we intercept all network related *send* and *receive* functions of an application. If they are handling a TLS connection (the interception framework resolves the remote address of a file descriptor), we extract the corresponding MS by dereferencing pointers that point to a structure holding the cryptographic key material that is passed to the functions of the crypto library. Even though the layout of the data structures is known since the implementation of OpenSSL and BoringSSL is open source, the exact layout of the data structures is unknown. This is caused by the fact that it changes based on the used compiler and compiler settings as well as the version of the library.

To address this problem, we follow a precomputed path starting with pointers on the calling stack to a data structure holding the MS. Such a path is computed during the *training phase* in which we run an application several times and let it initiate several TLS connections. For each connection we take a snapshot of the application memory. To bootstrap this approach, we locate the MS in

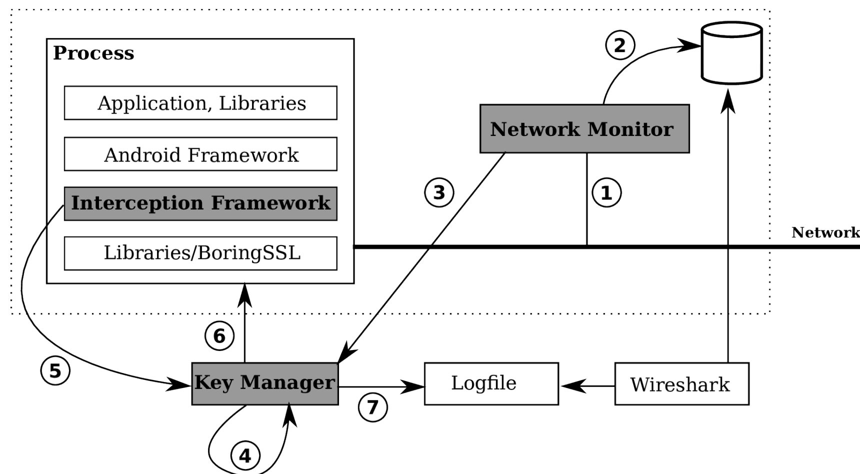


Figure 5.3: The components of *DroidKex* architecture and their interactions [TAR18]

a snapshot using the brute force testing method of *TLSKex* [Tau+16]. Afterward, we compute for each snapshot a path from the calling stack to the master secret. The computation of a path is discussed in more detail in Section 5.3.2. Finally, in the training phase we select the path that works best among all snapshots.

Assumptions

The first assumption we make is that all crypto libraries directly call network related *sending* and *receiving* functions (`read`, `write`, `receive`, `recvmsg`, `sendmsg`, `sendto`, `recvfrom`) and a starting point leading to the data structure holding the cryptographic key material is still on the stack.

The second assumption we make is that the crypto libraries do not use complex data structures such as linked lists, trees or hashmaps to store the MS. We assume that we can find one path from the start to the MS that always passes or visits the same data structures in the same order. Finding approaches that regenerate the layout of complex data structures to extract information efficiently should be addressed in the future.

Finally, we assume that each application can use different (versions of) libraries and different cryptographic parameters (e.g., the SSL/TLS version) resulting in the fact that different paths for the same application can be required. We try to learn these paths in the training phase. If we cannot learn a path, e.g., because the library was not used in the training phase, we are not able to extract the corresponding MS.

Components

The network monitor monitors all TCP connections of the application ① and captures traffic for later analysis ② (see Figure 5.3). It parses all packets to detect messages that are used to set up a new TLS connection. During the negotiation of a TLS connection, it extracts all parameters that are required to validate a master secret belonging to a connection by decrypting the first record. The parameters are CR and SR, the cipher and the first data record. Additionally, it extracts the source and destination IP address/port of a connection, which later allows the key manager to link a network packet with an intercepted function call. Afterward, it sends this information to the key manager ③.

The interception framework is a combination of Python and JavaScript tools that uses the Frida toolkit [Rav]. Frida [Rav] is a dynamic instrumentation toolkit and can be used to analyze the execution of applications running under Windows, macOS, GNU/Linux, iOS, Android, and QNX.

It allows injecting JavaScript-based analysis code to a running process to analyze the current state of the execution. The purpose of the interception framework is to extract the MS of each connection by intercepting network connection related function calls of an application. Every time a function call is intercepted, it follows the pre-computed paths with the goal to locate a MS. Whenever a potential MS is found, it is sent to the key manager ⑤. Additionally, it sends for each function call all values on the stack within a certain range above the stack pointer. During run-time, it polls the list of connections with a valid key so that function calls of these connections are not intercepted.

The key manager validates and stores the extracted keys of TLS connections in a file that can be used by Wireshark for analysis ⑦. The key manager combines the information about TLS negotiations with the data received from the interception framework to validate keys coming from the interception framework. To validate a key, it attempts to decrypt the first data record of the TLS connection with the extracted MS. In addition, the key manager stores the session ID and the corresponding master secret of the connections to improve the key extraction process in case the session is resumed. If a succeeding connection uses the same session ID, it is more likely that the same master key is used. Consequently, there is no need to go through the key extraction process since the MS is already known and can be looked up.

Furthermore, the key manager takes a snapshot of an application ⑥ in the training phase, which is required to compute paths. To do so, it waits a defined time interval after the network monitor reports that the application negotiated a new key ④ (see Figure 5.4). If it does not receive a key belonging to the corresponding connection, the key manager takes a snapshot[3]. The time difference between receiving the information from the network monitor and the interception framework depends on many factors. The first one is caused by the workload of the key manager that processes the messages from the other components sequentially. Thus, if one message takes longer to process (for example the validation of several extracted keys), the next message is delayed, i.e., the time difference between receiving a key and verifying it is increasing. Additionally, the communication overhead affects the timing. In our proof-of-concept implementation, all components send messages via TCP sockets. Due to the architecture of Frida, one part of the interception framework is running on a separate analysis PC, which increases the latency of the communication as well. Another reason for the late reception of keys is the fact that some applications do not directly communicate after the key exchange and the key is not extracted when no network function call is executed. This can be a problem when the data structures of the crypto library are not fully initialized after the key exchange. To address this problem, the key manager waits for a particular time until it takes a snapshot.

The exact timing for a snapshot is a trade-off between generating more overhead or losing the MS of a connection. The snapshot can be taken either directly after the network monitor sees the second CSP message, which indicates that both parties computed the MS, or a bit later. In the first case, it can happen that the interception framework did not yet send the MS to the key manager, e.g. because it is not copied to the final data structure of a TLS connection. Then, the key manager would take a snapshot and interrupt the application even though we would receive the MS later. However, if the snapshot would be taken too late, the MS might not be in the memory anymore.

Consequently, in the *training phase*, we choose a short timeout for taking a snapshot after seeing the second CSP message since finding a MS is more important than maintaining the usability of an application. In the *normal mode*, we only intercept the control flow to extract the MS and do not take snapshots. If there has been a network connection from which we were not able to extract the MS, we need to compute the missing path. In that case, we manually switch back the *DroidKex* prototype to training phase and try to compute the missing path.

To evaluate the impact of taking snapshots in the training phase on the time of the message processing, we measured the time difference between the key negotiation and the time when the corresponding MS is validated for two cases. In the first case a snapshot is taken when a key is

[3] During a TLS connection the parties can renegotiate the parameters. Since we never saw this during our testing, we do not consider this for our approach.

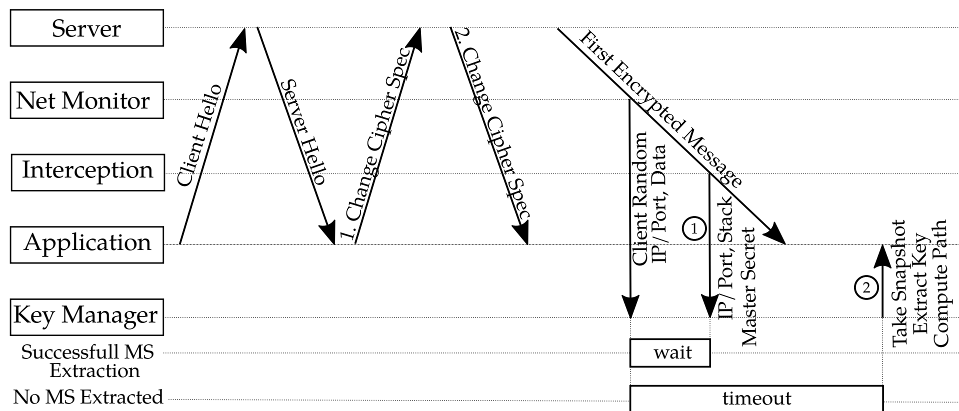


Figure 5.4: The timing of messages during a TLS connection with all components in two distinct cases. In the first case the control flow of receiving a data message is intercepted and the key is sent to the key manager. In the second case the master secret was not extracted successfully (the first message is not delivered) and after a timeout the key manager takes a snapshot of the application [TAR18]

received late (0.5 to 3.5 s after the negotiation). In the second case, we do not take a snapshot. In both cases, we measured the time after computing the paths for each application. In average we measured a time of 0.7 s to receive the MS in normal mode from the interception framework after the network monitor detected the second CSP message (case 2). By taking a snapshot 0.5 to 3.5 s after the negotiation, we measured a time of 1.11 s (case 1). Thus, the process of taking snapshots earlier delays the validation process by about 0.41 s. The impact on the timing is bigger when more snapshots are taken, e.g., when several snapshots are taken in the case of a missing path.

Requirements

The only requirement for deploying *DroidKex* on a target device is to have root permissions and to be able to disable SELinux, which is required by Frida. No additional adjustments to the Android runtime system are required, simplifying deployment on devices for which full device-specific source code is not available.

5.3.2 Implementation

There are two different ways to locate the MS in main memory: *scan the memory address space sequentially* or *follow pointers* in data structures that lead to it. The first option can be either signature-based or by testing all byte sequences as potential keys to decrypt one TLS record [Kle06]. Unfortunately, the MS of TLS connections does not have a defined format such as PKCS12, which is easy to identify. Thus, the signature-based approach does not work. However, the brute force based approach works when the MS is stored as a byte sequence in the main memory of an application. On the other hand, the derivation of session keys and the decryption of a message is computationally intensive and slows down the whole approach, which makes it infeasible for runtime key extraction.

The second option interprets the data structures in main memory and follows pointers to the data structure holding the MS. It requires: (1) a start point from where on we can follow pointers (2) knowledge about where the pointers are stored in each data structure (how to get the knowledge is discussed in the next section). A start point can be defined in two ways. The first one is a global symbol or variable in a binary. Such a variable (if it exists) is usually not exported in a binary file. Especially, crypto libraries try to encapsulate and hide internal functions and data structures to prevent data leakage. The second option for a start point are the pointers to the arguments

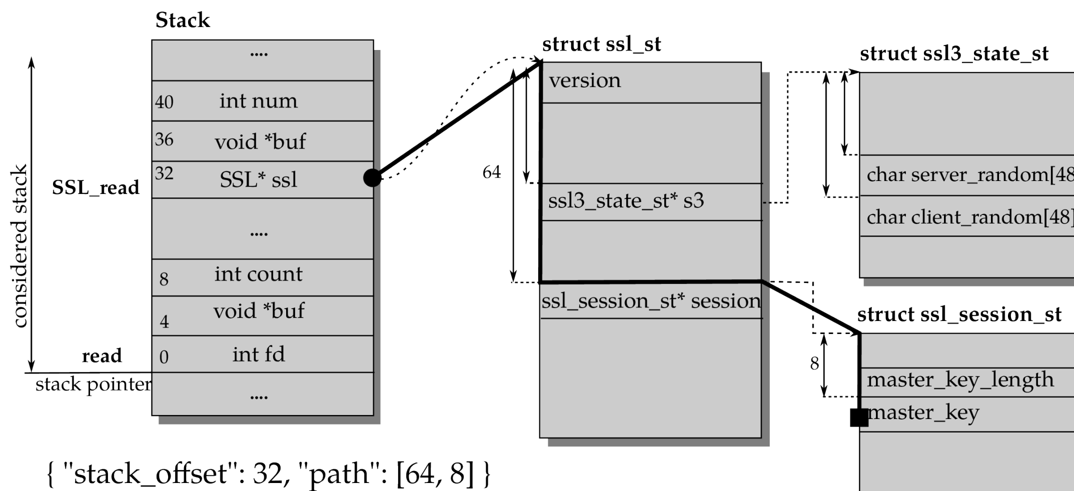


Figure 5.5: The contents on the stack when the `read` function is called by the `SSL_read` of OpenSSL function. The path from the starting point – the SSL pointer (black dot) to the MS (black square) – is marked bold and the corresponding. The computed path and the offset on the stack are noted on the bottom left side [TAR18]

passed to a function. They can be extracted from the stack by intercepting the function call, which requires that the function symbols are exported and knowledge of the utilized functions.

DroidKex implements the second approach of control flow interception since the run-time cost for the key extraction does not depend on the size of the main memory of the application. Instead, it depends on the amount of memory access operations for following a path and the overhead added to intercept a function call. To do so, *DroidKex* intercepts all network related function calls for *sending* and *receiving* data and checks based on the file descriptor whether they belong to a TCP connection on port 443. We assume that an entry point leading to the MS is in a certain range above the stack pointer of the intercepted function. We could also directly intercept crypto library functions, but we aim at finding a generic approach that also works if the function names are not known or not available, e.g., when the symbols are stripped from a binary.

To improve the performance, we reduce the number of considered elements on the stack by saving the position of a good start point in relation to the stack pointer. Additionally, we minimize the extraction attempts by intercepting only I/O calls of connections of which we do not yet have a valid MS. In Figure 5.5, we depict the stack layout at the point in time when the `SSL_read` function of OpenSSL/BoringSSL calls the `read` function. To extract the MS, we need to know the position of the pointer to the SSL struct in relation to the stack pointer and the path from the SSL struct to the MS.

Path Computation

For computing a path in the *training phase*, we use a data-centric approach, which means that we only consider the contents in main memory and not the instructions of a program [Dol+11]. We define a path as a list of integers. Each integer defines the offset in a data structure where the pointer to the next data structure is stored. The last element in the list stores the offset to MS. (for examples see Figures 5.5 and 5.6). The most important steps to compute a path are: taking a snapshot of the process holding the information, identifying the position of the MS, computing the path based on the snapshot and finally validate the data path. These steps are discussed in more detail in this section.

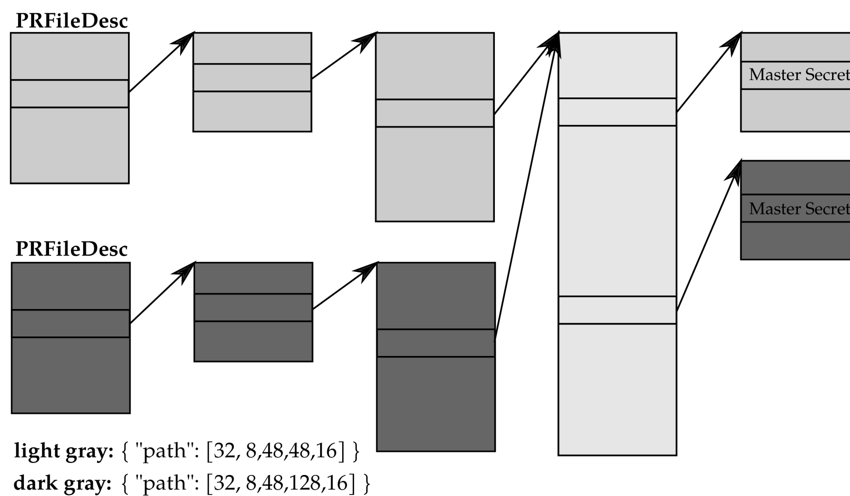


Figure 5.6: The path from the PRFileDesc struct to the MS in the memory of the Firefox application that uses the libnspr library [TAR18]

Prerequisite

In order to compute a path to a MS, three components are required. The first one is a *snapshot of the application* while the MS is in the memory. The second one is a *starting point* for a path. For this purpose *DroidKex* uses the values on the stack of each *send* and *receive* functions handling the corresponding networking connection. The third one is the *location of the MS (endpoint)*. To find a MS in a snapshot, we use the brute-force approach of *TLSKex* [Tau+16].

Data Structure Layout

DroidKex aims at extracting the MS of applications using either the OpenSSL/BoringSSL or libnspr crypto library. In both cases, a pointer to a data structure containing the cryptographic key parameters of a connection must be passed by the application to the functions that handle the encryption of payloads. For OpenSSL/BoringSSL the struct is called `SSL` and for libnspr `PRFileDesc` (see Figure 5.5 and 5.6). The path from the `SSL` struct to the MS is always the same. The path to the MS from a `PRFileDesc` struct (see Figure 5.6) is not the same since it uses a global array where all the cryptographic key material is stored. Thus, one element in the path is always different but within a defined range (the size of the array). Since these libraries are the most common options for Android applications, we only consider these two different types of paths for *DroidKex*.

Linking Information

While an application is running, it can start several TLS connections. Consequently, we need to link a snapshot with the stack of the corresponding function call and the information from the network monitor. Additionally, the content on the stack of multiple function calls belonging to the same connection must be grouped. To do so, we use the file descriptor from the argument of the monitored network functions to resolve the corresponding TCP connection by looking up the source and destination IP address & port. By having this information, we can combine the information extracted by the network monitor (first data record, CR) with the information of the function call (content on the stack) by matching source and destination IP address & port. If we would intercept functions that do not have a file descriptor (e.g., the `SSL_read` function of BoringSSL), the task of linking a function call to a network connection has to be established differently.

```
[8,256,96,16,40] → [-1,256,96,16,40] [8,-1,96,16,40] [8,256,-1,16,40] [8,256,96,-1,40] [8,256,96,16,-1]
[8,256,96,12,40] → [-1,256,96,12,40] [8,-1,96,12,40] [8,256,-1,12,40] [8,256,96,-1,40] [8,256,96,12,-1]
[8,256,96,36,40] → [-1,256,96,36,40] [8,-1,96,36,40] [8,256,-1,36,40] [8,256,96,-1,40] [8,256,96,36,-1]
[8,256,96,48,40] → [-1,256,96,48,40] [8,-1,96,48,40] [8,256,-1,48,40] [8,256,96,-1,40] [8,256,96,48,-1]
[8,296,96,48,40] → [-1,296,96,48,40] [8,-1,96,48,40] [8,296,-1,48,40] [8,296,96,-1,40] [8,296,96,48,-1]
[256,96,12,40]   → [-1,96,12,40]     [256,-1,12,40]  [256,96,-1,40]   [256,96,12,-1]
```

Figure 5.7: Path expansion: the left side shows sample paths extracted from Mozilla Firefox; the right side shows the expanded paths; the rectangle highlights the selected path [TAR18]

Path Computation

After having a snapshot together with the start and end point of a path in the memory, we can start computing paths in it. To calculate possible paths, we perform a depth search from start pointers (contents on the stack) to the MS. To do so, we dereference the first n bytes above the address where the pointer pointed to and test whether each pointer points m bytes before the MS, i.e., to a struct containing it. If it does not, we recursively continue searching by dereferencing again the first n bytes of the next structure and test if one of them points before the master secret. We continue that search until the path is longer than a defined value i (1) or the path reaches an address that has been visited before (2). If we found a path that ends m bytes before the master secret, we store that path and continue searching.

The values n, m and i have to be adapted to the assumed size of the data structures. By choosing them too small, the correct path cannot be found. By choosing them too big, the depth search takes longer, and many false positives are found, i.e., when we search across the borders of a data structure and continue searching in the data structure mapped above the dereferenced one. The value $stack_size$ is the amount of data above the stack pointer that is assumed to contain a pointer to the first data structure of the path. The worst case complexity of the approach is $\mathcal{O}(stack_size * n^i)$. However, most of the data values on the stack and in the data structures cannot be dereferenced, which removes possible paths and lowers the search time.

To improve the speed, we store for each address in memory that we visit the computed paths. Thus, if an address is reached again, we do not need to compute the path again and can take it from the cache. This approach also helps to mitigate getting stuck in loops, i.e., caused by pointers to the same data structure or lists.

Data Path Selection

The outcome of the depth search is that we usually have several paths for each snapshot and we have to decide which of them to choose[4]. The selection is implemented as a heuristic based iterative approach: first, we run the application with a set of paths (in the initial phase the set is empty, which means that we take a snapshot for each TLS connection). If we did not extract the MS of all TLS connections, we compute the paths out of the corresponding snapshots and select two paths that work for most of the snapshots we took. If two paths have the same coverage, we take the shorter path first. If both paths have the same size, we select one arbitrarily. Then, we repeat step one to check if the selected paths are enough to decrypt all connections of an application within a time frame or if more paths are required or if the path does not provide working MS. We iteratively repeat the two steps in the training phase until we can decrypt all connections of an application within a time frame. If we did not see any TLS connections during the run-time, we start the application again.

If a path did not extract any MS within a certain amount of rounds, we remove it from the set of paths. In each iteration, we remove paths that did not successfully extract a MS in the last four rounds and add it to a blacklist so that it is not selected in the future. Not working path can be caused for example when data structures are mapped closely together, which produces paths that

[4]We cannot take all paths and remove not working paths because too many possible paths slow down the extraction of the key and the normal operation of an application.

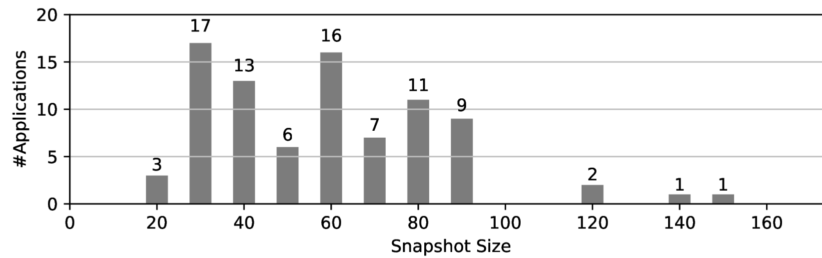


Figure 5.8: Number of applications with the same memory size [TAR18]

are only valid in that special case or when pointers on the stack point to ephemeral data structures that are used only during initialization. We progress with that approach (repeat step 1 and 2) until we can find a MS for each connection of an application and all paths were able to extract at least one MS.

Mozilla Firefox

The approach of finding a path by counting exact occurrences only works in the case of OpenSSL and BoringSSL paths, i.e., when the offsets in the data structures are always the same. In the case of Mozilla Firefox, the paths are different since they contain one offset that is different for each path (the array). Thus, this approach of finding exactly the same path does not work directly. To use it, we need to adapt it by searching a path that is the same except for one element. For example, the outcome of the path computation of four snapshots are the paths shown in Figure 5.7 on the left side.

To find a generic path that extracts all secrets of Firefox, we have to define a path that can handle the array (see Figure 5.6). To achieve that, we have to solve two problems: First, we have to find a group of paths that have only one different entry at the same position, i.e., the position of the array. Second, we have to identify at which position in the array the key of a certain connection is stored to extract from main memory.

To address the first problem, we expand all computed paths by inserting a variable (`-1`) at each position (see Figure 5.7). Afterward, we use the approach from above and choose the path that is the same in all four cases (here: `[8,256,96,-1,40]`). To address the second problem, we implement a simplistic approach that iterates over all elements in the array until the last element stored in the array is 0 or the index grows beyond a defined size. Then, the key manager has to identify the correct MS out of the extracted values.

5.3.3 Evaluation and Discussion

In this section, we measure the overhead and effectiveness of the *DroidKex* prototype, and discuss the limitations of our approach. To do so, we measure the time required for taking a snapshot and extracting the MS with the approach of intercepting the control flow and follow paths to it. In Section 5.3.3, we measure the size of common application snapshots, the time required to take it and the time to find the MS of a TLS connection in it. In Section 5.3.3, we measure the time required for computing the paths. In Section 5.3.3, we measure the time that is added to a network function call that is required whenever the control is intercepted to extract the MS.

All measurements in this section are executed on a Nexus 5x with Android 6.0.1. For testing, we chose a set of 86 applications from the top hundred free applications in the Google Play store that initiated TLS connections after starting them without any interaction. In each round, we execute the application for 90 s and take a snapshot 0.5 s after the key manager processes the message that the TLS session is negotiated. We do not take a snapshot 3.5 s after the negotiation to maintain usability of the application in the case when multiple snapshots are triggered.

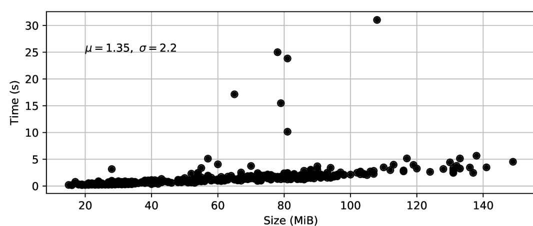


Figure 5.9: Time to take the snapshot of an application in relation to its size. Every dot represents one application [TAR18]

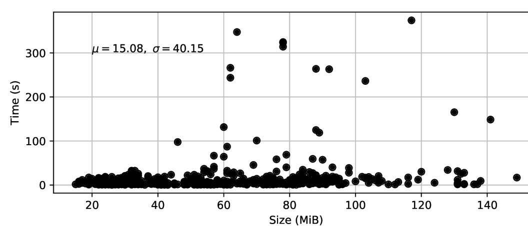


Figure 5.10: Time to extract the key material of a snapshot. Every dot represents an application snapshot [TAR18]

Extraction from Snapshot

DroidKex takes a snapshot only of memory areas that are likely to contain the MS, e.g., the heap and stack. To provide an estimation of the time for taking a snapshot and extracting the MS, we run the steps of the training phase with 86 applications. To access the process memory, we stop and resume the process execution using `ptrace` and access the memory using the mapped process memory file in the `proc` file system. In Figure 5.8 we depict the histogram of the size of the snapshots for each application. Figure 5.9 shows the relation between the size of the snapshot and the time required to take it; each dot represents one snapshot. Despite the outliers a linear correlation can be asserted. The outliers can be caused by background activities or other applications that are active while the snapshot is taken. Those activities could stall the process of taking a snapshot, e.g., by consuming a large amount of CPU time. Additionally, it could be also possible that the CPU is running with a low frequency to save battery.

Figure 5.10 depicts the time for finding the MS of a TLS connection in a snapshot. The time for taking a snapshot and extracting the MS depends mainly on the size of the address space and the position of the MS. Two facts cause the big variance in time for searching a MS: First, the MS can be located in the beginning or at the end of a snapshot, and we stop when we have found it. Second, the brute-force approach tests the entropy to enhance speed before actually decrypting the first record with it since the key derivation and decryption is computational intensive [Tau+16]. Thus, the search is slower on snapshots with high entropies, e.g., when they contain compressed images.

Based on these measurements, we can conclude that the approach of *TLSKex* that takes a snapshot of the application memory and extract the MS of it is not feasible for permanently monitoring applications that have multiple TLS connection at the same time, which is the case for most of the modern Android applications. The time required for taking a snapshot and extracting the MS is so high that application can not be used normally anymore if multiple connections are initiated at the same time. Thus, an approach like the one of *DroidKex* with less overhead is required to extract the MS of the main memory of applications.

Path Computation

To measure the performance of the path computation algorithm described in Section 5.3.2, we compute the paths for the 86 applications and measure the number of iterations and the required time until the algorithm converges. For these measurements, we use a struct size of $n = 700$ and use a limit of $i = 3$ iterations for the depth search.

Figure 5.11 depicts the time that is required to compute the path of a snapshot compared to the number of pointers from the stack collected from the network function calls of a connection to the MS. The figure shows that the computation time depends on the number of possible start points to the path computing algorithm.

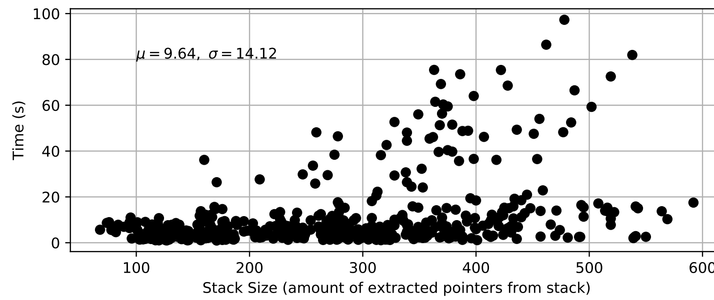


Figure 5.11: Time to compute the data path compared to the size of distinct dereference-able pointer on the stack collected from I/O calls [TAR18]

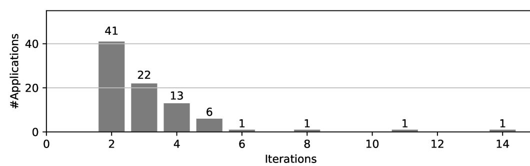


Figure 5.12: Number of iterations required to compute paths [TAR18]

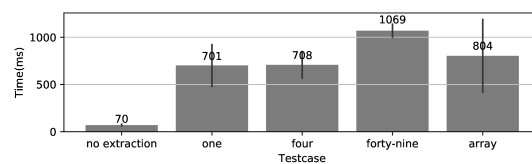


Figure 5.13: Time for a single TLS https GET request with a different number of extraction paths [TAR18]

Figure 5.12 shows the number of iterations that are required to compute the path for an application. The number of iterations defines the number of times an application was started, and during the runtime, a connection was found where the control flow interception did not provide a valid MS. In most of the cases, we were able to find valid offsets with one to four iterations.

During our evaluation, we extracted 12 distinct paths. Since about 95% of the applications only use one or two paths, they can be used for most applications on the same device without recomputing them.

Control Flow Interception

The overhead added by intercepting the control flow is caused by two components: the overhead of the Frida interception mechanism and the time required to extract the MS. To measure the impact of Frida, we call one native C++ function from Java that is similar to calling BoringSSL functions. Without Frida intercepting the control flow, the average time for doing 10,000 JNI calls 100 times is 31.58 ms ± 1.94 ms. After activating the tracing of the same function, it takes 90.56 ms ± 2.04 ms, which is an overhead of 5,9 μs per function call.

In order to measure the time for the interception overhead added to regular TLS connections, we implemented a sample application that initiates 100 TLS connections to the same server with and without intercepting the network function calls. Without extracting the MS, the average time for an https GET request is 70.9 ms ± 17.0 ms).

To measure the impact of different paths on the extraction process, we determine the optimal path of our sample application and modify it so that one, four and forty-nine offsets on the stack are used as a starting point[5]. Additionally, we tested one path that includes an array with an assumed size of 128 entries. For each possibility, we measure the average time for an https GET request (see Figure 5.13).

[5] The path computation found forty-nine possible entries on the stack that could lead to the MS. However, only one was working for data extraction.

The reasons for the longer run-time of a connection are: intercepting too many function calls that are not required, performance of the extraction routine and probing of none-required values on the stack and in arrays. Intercepting all network function calls of an application causes a high overhead since these functions are also used for other I/O operations such as file access. If we would only intercept one crypto library function, the overhead would be less since we would not need to search for an entry point on the stack and take the argument of the corresponding function, e.g., `SSL_read`. However, with this approach, we would also lose the link to a certain network connection since we could not link a function call to a network connection based on its IP address. Additionally, this approach is less generic and requires knowing all TLS related function calls. Nevertheless, the impact of the extraction routine decreases for long living connections, since after successfully acquiring the MS we stop the extraction of the corresponding connection.

We also assume that it is possible to improve the overhead by optimizing the proof-of-concept extraction routine, e.g., by implementing the extraction routine in a low-level language and not in JavaScript, which is required by Frida.

The performance of using a path that includes an array can be improved by implementing an approach that finds a path to the index value so that not all values of the array have to be tested.

Discussion and Limitations

By using Frida for control flow interception, we are aware that we can only analyze applications that work when Frida is attached, e.g., applications that do not prevent the use of debuggers. Because of that, we can only trace applications that have one process since Frida has problems with tracing multiple processes.

The approach of *DroidKex* would be infeasible when the access to memory holding the key material is not granted to the key manager such as using a secure keystore instead of the main memory. Such a key store can be implemented for example using the TrustZone of the ARM architecture. However, to the best of our knowledge, no TrustZone application provides such a service. Another approach is that parts of the TLS protocol implementation are integrated into the Linux kernel to improve the performance [Edg15]. In that case, the kernel has more control over the keys and could protect them from malicious user-space applications. However, then the kernel could also easily log all SSL keys without the help of an application.

5.4 VMI-based SSH Honeypot

The approaches *TLSKex* and *DroidKex* do not require full semantic knowledge about the monitored application. The approach of *TLSKex* only requires that the TLS session key is stored in a 48-byte array while *DroidKex* requires that the application uses OpenSSL/BoringSSL or libnspr. Due to this assumption, *DroidKex* has less analysis overhead compared to *TLSKex*. However, since *DroidKex* intercepts all network related system calls and attempts to extract the session keys for each call, the overhead is still noticeable. This is especially a problem for honeypots because an adversary may test whether he is connected to a monitored environment to evade analysis [Bal+10; Che+08]. Hence, the research question is how to minimize the overhead of VMI-based monitoring so that attackers do not notice that they are connected to a monitored system. To achieve that, we discuss in this section two approaches that demonstrate how the selection of monitored functions affects the overhead for monitoring. In this context, we compare the overhead for monitoring system calls with the monitoring of function calls of the SSH daemon to trace the interactions of an adversary with an SSH honeypot. The following discussion is based on [STR17] and [STR18].

A honeypot should achieve three goals. First of all, a honeypot should provide in-depth tracing of an attack, either to get and analyze new attacks or to detect attackers in a running network and to protect other systems by eliminating this threat [Hoo09; JS11]. Second, it should store the activities and modified/download files of an attack for further analysis. The data should be forensic sound, and an advisory should not be able to disable the tracing or manipulate the logs. Third, the monitoring should be stealthy to analyze attacks that aim to evade the analysis [Bal+10]. Hence, the tracing overhead of a honeypot should be as low as possible to be stealthy against timing attacks.

VMI-based monitoring is particularly suited to build honeypots since it allows monitoring all activities within in a honeypot. In addition, due to the virtualization layer, the monitoring is not directly accessible for an attacker. Finally, VMI-based tracing is stealthier than a typical in-guest agent, which could be detected by an adversary having full control over the system.

The primary goals of our honeypot is to log all system activities of an attacker including the standard features of a common SSH service:

- Username and password
- Interactive session
- File transfer
- Port forwarding

5.4.1 Threat Model and Assumptions

For our honeypot implementation we make several assumptions about the behavior of an adversary. First, we only trace the activities of an attacker when he has access to a system. We do not cover attacks that target that SSH service, e.g., with buffer overflows. Second, we assume that adversaries do not run timing based VMI detection. In the future we need to discuss whether the timing behavior of intercepted functions can be used to detect VMI based monitoring especially in cloud environments where several virtual machine coexist on the same physical server. Third, we assume that we have full control over the executed SSH binary and know that layout of all required data structures, which is required to extract information from main memory.

We assume that an attacker does not disable or circumvent the VMI-based tracing by exploiting the strong semantic gap problem. For example, even by replacing the SSH daemon with a different one, the tracing would not work anymore, when the addresses of the required functions cannot be resolved anymore. We do not consider attacks that actively put crafted data to main memory that subverts VMI based memory analysis.

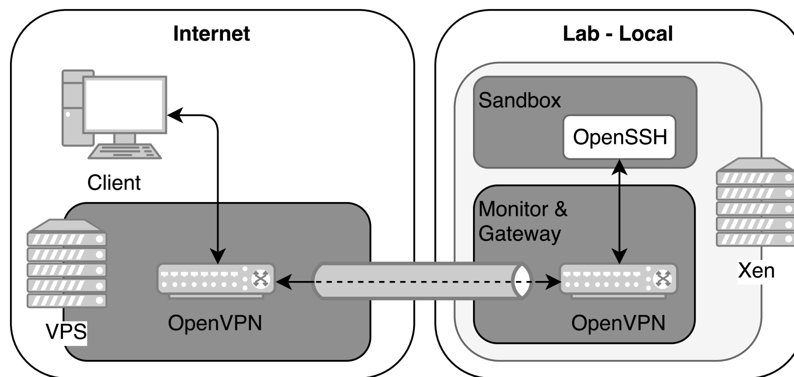


Figure 5.14: The *Sarracenia* system architecture including the network connection to redirect network traffic from a cloud virtual machine to get real attacks from the internet [STR18]

We also assume that an attacker does not bypass or attack the isolation provided by the virtualization in order to attack monitoring.

5.4.2 System Design

For the honeypots described in [STR17] and [STR18] we use a similar overall architecture that leverages the *CloudPhylactor* architecture. The design is depicted in Figure 5.14. We mainly use two virtual machines: a sandbox and a monitoring virtual machine. For the honeypot we use a regular Linux virtual machine where the attacker can connect to and a monitoring virtual machine that monitors it. We have full control over this virtual machine and have the semantic knowledge to bridge the semantic gap to interpret the required data structures in main memory of at least the SSH daemon and the kernel. In a monitoring virtual machine, we install the tracing tools (*Libvmtrace* and the SSH monitoring application) required to monitor the honeypot sandbox virtual machine. The permissions of the monitoring virtual machine on the sandbox are defined by the Xen security modules.

Since we can not deploy the honeypot in a public cloud infrastructure, we redirect the SSH traffic from virtual machines in publicly available cloud data centers to get real attacks.

5.4.3 Implementation

In the following sections we describe the implementation of the system call based (*Sarracenia* α) and the function call based (*Sarracenia*) honeypot. We discuss how the selection of monitored functions affects the overhead for monitoring. The first approach is *Sarracenia* α. It uses the parameters of the `read` and `write` system call to reconstruct SSH sessions. The second approach is *Sarracenia*. It uses the parameters of function calls of the SSH daemon to reconstruct SSH sessions.

System-call based: *Sarracenia* α

In [STR17] we use a system call based approach to monitor the activities of an attacker in the SSH honeypot, which leverages the fact that the SSH daemon reads and writes the unencrypted payload of an SSH session from/to a file descriptor. We intercept the system calls `read` and `write` and parse and save their parameters. This procedure allows us to reconstruct an SSH session, including user credentials, transferred files, and even port forwarding data. However, the `read` and `write` system call are also used for other activities. Thus, the challenge of this approach is to implement filters that extract the required information from the stream of data.

To reconstruct the contents of an SSH session from other read and write system calls, we need to

	Name	1	2	3	4	5	needed	intercepted
System Call	clone	✓					396	400
	sys_exit_group	✓					1	450
	exec			✓			364	366
	write					✓	2134	2213
	seek					✓	0	0
	close					✓	5644	6221
Function	kex_derive_keys		✓				1	1
	auth_password		✓				1	1
	sshbuf_get_u8			✓	✓		129	129
	ssh_packet_send2_wrapped			✓	✓		59	59
	channel_connect_to_port				✓		0	0

Table 5.5: Function and system calls that are traced for (1) detection of new SSH connections, (2) extraction of user credentials, source IP address and port, session keys of an authentication process, (3) reconstruction of SSH session, i.e., entered commands, (4) data of TCP port forwarding, and (5) modification of the file system [STR18]

filter out the significant ones. The *Sarracenia* α honeypot does it based on the PID of the SSH daemon handling the connection, the system call type (read or write) and the file descriptor. To obtain these values for a SSH session, we leverage the fact that at the beginning of a connection the contents of the `/etc/motd` file are sent to the client. Hence, we can filter the system calls by searching for the contents of the file. By identifying a write system call that has the contents of this file as an argument, we can retrieve the PID of the corresponding process and the file descriptor number used for the session. Afterward, we can filter the system calls based on this PID and the file descriptor number to reconstruct the data that is shown in the SSH terminal.

The advantage of this approach is that it is generic and does not require detailed knowledge about the implementation, e.g., the function names. The disadvantage of this approach is that many system calls are intercepted that are not required because they do not contain any relevant information about the state of an SSH connection or do not even belong to an SSH daemon. Thus, the whole approach was not very efficient because every intercepted system call degrades the performances of the monitored system.

Function-call based: *Sarracenia*

To improve the ratio of intercepted functions and function calls that contain relevant information *Sarracenia* [STR18] uses a different approach, which intercepts the control flow of the honeypot system only at positions, where we can assure that the required data is available. For example, it intercepts the function that is validating the user credentials to extract the username and password used by an attacker. Additionally, it intercepts the functions that are used to de-/encrypt the data transmitted during an SSH session. The following OpenSSH SSH functions are intercepted: `kex_derive_keys`, `sshbuf_get_u8`, `ssh_packet_send2_wrapped`, `channel_connect_to_port`. The parameters of these function calls are used to derive different information.

Tracing the functions of the SSH daemon allows reconstructing the activities of an SSH session, but it does not provide information about the files that are changed during an interactive session or when files are downloaded from other servers. To trace those activities, we monitor system calls that interact on files. The intercepted function and system calls and the amount of how often they are used during a regular login process are depicted in Table 5.5. The `clone` call is used to detect new connections since each SSH session is handled by a new child process. The `sys_exit_group` call is used to clean the cache of open files, which is used to derive the path from a file descriptor, and to stop tracing a running session. From the `exec` call, we extract the executed commands

during an SSH session. The `write` call is used to get the data that should be written to open files of a process. Whenever a process invokes `close`, the file descriptor is removed from the cache.

This is just one possible configuration of functions that allows reconstructing SSH sessions and information can be extracted from different calls. For example, the path from a file descriptor passed to a `write` operation can be either extracted from a preceding `open` call or by deriving the information from the kernel data structures handling file descriptors. In the first case, two different system calls must be traced. In the second case, only one call must be traced, which lowers the tracing overhead. However, it requires the parsing of kernel data structures and semantic knowledge about the layout of the data structure.

Sarracenia uses the two modes of operation of *Libvmtrace* for the tracing:

- *Process-bound:* Breakpoints on system calls are attached and detached dynamically based on the process that is running. To do this, we monitor write access to the CR3 register that holds the addresses of the DTB. Whenever a new process is dispatched, the content of this register is updated with the DTB of the next process. Thus, we can control that a specific process is monitored. This requires a VM context switch at every process change in order to check whether breakpoints should be set or not. Additionally, the breakpoints must be written to memory or removed with the original instruction if the new process should be monitored or not.
- *System-wide:* All breakpoints are set from the beginning when the monitoring is started which means that all processes are traced. This does not require a context switch for every process change. However, it results in more context switches for system calls at run-time.

5.4.4 Evaluation

In [STR18] we measured the performance of the function-call based approach. The implementation is optimized for performance and thus can be used as a general example for the impact of VMI-based tracing on the analyzed system.

Performance

We measured the performance impact on the analyzed system, i.e., the honeypot, with different tracing mechanisms in three use cases [STR18]:

A. **Simple command:** Execute `ls -alh`.

B. **I/O intensive test:** Download a file with 2MB size using `wget`.

C. **I/O and CPU intensive test:** Compile the Jansson library[6].

We used four different tracing mechanisms that extract a different amount of information and thus have different effects on performance:

1 **Without tracing:** The baseline to measure the execution without tracing

2 **System-wide tracing - SSH functionalities:** `sys_clone`, `sys_exit_group`, and all OpenSSH functions are monitored. This does not monitor any change to the file system.

3 **Process-bound tracing (whitelist) - with file change detection:** all system calls of OpenSSH functions are monitored. Only the file-system related system calls of `wget, curl, sftp` and `scp` .

4 **System-wide tracing - with file change detection:** all system calls and OpenSSH functions (see Table 5.5) are monitored.

[6] `https://github.com/akheron/jansson`, Accessed 2019-05-13

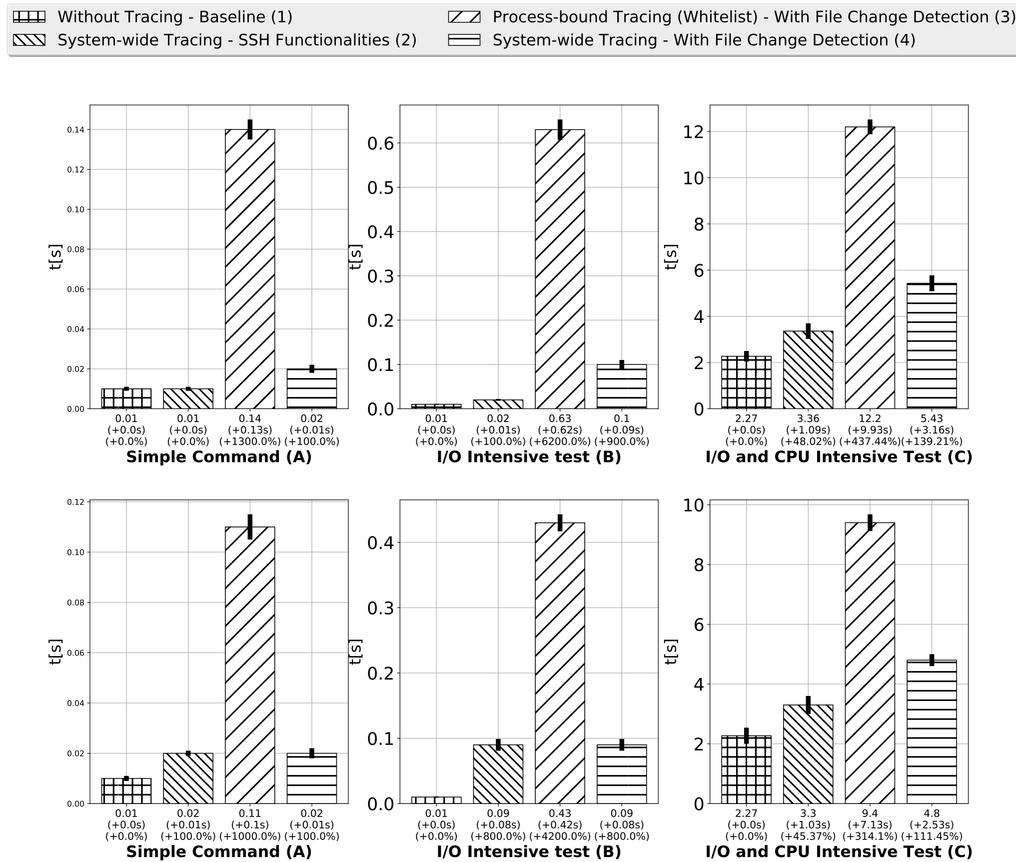


Figure 5.15: Overhead of client's execution time based on different scenario and configuration where (1) is without monitoring as the baseline and (2) to (4) are monitored by *Sarracenia* using pure `INT 3` (first row) and altp2m (second row) [STR18]

Additionally, we measured the impact of the breakpoint implementation (pure `INT 3` and altp2m) on the performance. Figure 5.15 depicts the results of the measurements. Based on the timings it can be said that the tracing system calls has a more significant impact on the system performance compared to tracing function calls, which is mainly because the traced functions of the SSH are called less frequent than the system calls.

Efficiency

To measure the effectiveness of the system call based approach, we counted all intercepted calls during a login process and calculated the ratio of system calls that are required to extract the user credentials. In [STR17] we computed that 1.95% and 8.99% of the intercepted read and write system calls are required. Thus, the effectiveness of this approach is low and has a high performance impact.

By intercepting only the userspace ssh function `auth_password`, we achieve higher effectiveness since we make sure that this function is only intercepted when a new password is validated. Thus, the overhead of the function based approach is lower. In Table 5.5 the effectiveness of this approach is measured in the use case C and tracing in configuration 4. The SSH session of the measurement took 10.377 s. Based on these values it can be concluded that function based tracing is more effective since only required functions are traced. However, in some cases when it is necessary,

system calls must be traced even, so the effectiveness is lower since many processes of the system use them. The process bound tracing did not improve the performance (see Figure 5.15).

A detailed discussion about the attacks we got from the internet is done in [STR18].

Stealthiness

There are at least three approaches to detect whether a system is a honeypot or being monitored: operational analysis (execute arbitrary commands), system level fingerprinting (timing benchmark), detecting monitoring agent. We tested these approaches against *Sarracenia* and Cowrie. Since *Sarracenia* provides a fully fledged Linux system an attacker can execute and install any required tools. Thus, he can not see any difference to a nor- mal system in contrast to Cowrie which only provides some limited amount of commands. As discussed in Section 3.6.2 the overhead added to a single monitored function is about 0.27 ms, which can be used to detect the presence of the introspection. This becomes noticeable when the same (monitored) system call is invoked multiple times and the timing of an untraced system is known. Reducing the impact of VMI-based tracing mechanisms is an ongoing research topic and reducing it to improve the stealthiness must be addressed in future work. Nevertheless, to the best of our knowledge when virtualization, e.g., in cloud computing, is used it is common that functions are delayed since several virtual machines share the same resources. *Sarracenia* does not require any agent inside the honeypot. Thus, an adversary is not able to directly detect any monitoring component. Hence, we can consider our approach as more stealthy than other SSH honeypot approaches that use an in-guest agent in the honeypot to record the traces of an attacker.

Limitations

One limitation of both approaches is that attackers can perform many operations and flood the tracing. For example, creating a file with random content and deleting it afterward multiple times creates a considerable amount of overhead and requires much storage.

5.5 Summary

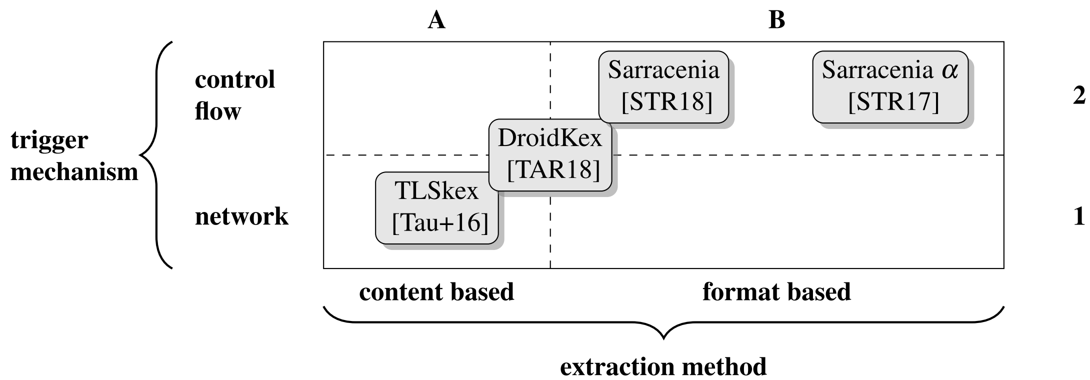


Figure 5.16: Information Extraction Approaches

In this chapter, we discussed the challenges of data extraction from volatile main memory, namely the performance, the semantic gap problem and the timing that triggers the information retrieval based on the example of extracting TLS session keys from the main memory of applications and the SSH honeypot. Figure 5.16 classifies the methods presented in this chapter based on their information retrieval method and the trigger mechanism that is used to start the extraction.

TLSKex extracts the master secret of TLS connections from applications in a virtual machine. The first contribution of *TLSKex* is to trigger the key extraction routine, *TLSKex* monitors the network traffic of a virtual machine and takes a snapshot of the corresponding application when it initiates a new TLS connection. The second contribution of *TLSKex* is to use a content-based approach to identify the TLS master secret in memory. To do so, it decrypts an encrypted TLS message with every byte sequence in the snapshot of an application and checks whether it was successful. Hence, this approach does not need to know the data structure format that holds the session key. Even so, we optimized the brute-force approach by minimizing the search space we could not apply it to applications that initiate many new connections at the same time and have a large address space because of the low performance. The low performance is mainly caused by the large amount of memory access operations and the high run-time of the decryption routine that tests whether a byte sequence is a valid key.

The main contribution of *DroidKex* is to increase the performance of the extraction routine by deriving and using the format of the data structures holding the key. This approach is required when the exact data structure layout is not given. To trigger the key extraction it monitors the control flow of the application. Whenever an application is calling a network related send or receive function, *DroidKex* follows a pre-computed path of pointers from the stack of the corresponding function call to the master secret. The second contribution of *DroidKex* is the way how such a path is obtained in the training phase using several snapshots of the same application. This approach minimizes the amount of necessary memory access operations and calls of the decryption routine. In contrast to *TLSKex*, we were able to run *DroidKex* with Android applications that have large address space. Hence, the performance overhead of *DroidKex* reached a level where it is feasible to monitor regular Android applications.

The main contribution of the SSH honeypot *Sarracenia* is the discussion on how to minimize the performance overhead required for monitoring SSH sessions using VMI. To achieve that, *Sarracenia* uses the semantic knowledge of the data structures of the OpenSSH daemon derived from the debugging information to extract and rebuild SSH sessions by placing breakpoints on function calls. Hence, the information extraction routine of *Sarracenia* is format based. In the early version *Sarracenia* α we use the data of the read and write system call, to achieve the same functionality. It reconstructs SSH sessions based on the strings passed to the system calls. Afterward,

the parameters are filtered and parsed to reconstruct the session information. With these two approaches, we compared the overhead for monitoring system calls and function calls to retrieve information. It turned out that in this case monitoring function calls is more efficient and less resource intensive. Additionally, we measured the difference in overhead for process-bound tracing compared to system-wide tracing of function calls. We also conclude, that VMI-based tracing is suited for honeypots and can provide better stealthiness than traditional in-guest agents. However, the performance overhead of the monitoring is still noticeable by attackers, and thus the VMI-based monitoring is not entirely stealthy. So in the future, the performance aspects of VMI-based tracing needs to be discussed to conceal the monitoring overhead from adversaries.

6 VIRTUAL MACHINE INTROSPECTION IN SIEM SYSTEMS

In the previous chapter we have discussed two applications areas where virtual machine introspection can be used in practice. The *TLSKex* and *DroidKex* architecture can be used for example to analyze the encrypted communication of malware. The main application area of the *Sarracenia* honeypot is to record the activities of attackers. The focus of the discussion was on how to improve the performance of the information extraction routine.

However, in certain systems the overhead generated by VMI-based mechanisms might still be too high, or detailed tracing of user interaction might not be permitted due to data protection regulations, e.g., in SIEM systems. Thus, such detailed tracing should be only used if necessary, e.g., in case of suspicious behavior. In contrast to common intrusion detection systems that often monitor only a single system, SIEM systems aim to protect large scale production environments of companies including desktop computers, servers, and virtual machines [KS06]. The primary goal of SIEM systems is to increase the overall security level of an infrastructure. To achieve that, SIEM systems collect different kind of log data of an infrastructure in a central storage. This data is then used to detect attacks or anomalies that could be a potential security incident [BMZ14; Mil+11]. Additionally, that data can be used to analyze past incidents.

A SIEM system can benefit from VMI and memory analysis because of the higher level of forensic soundness and the isolation introduced by virtualization, which makes the data acquisition process less vulnerable to attacks. Additionally, VMI can be used as an additional source of information to detect inconsistencies between different sources that could be caused by malicious software. Thus, VMI can be used for both use cases of SIEM systems: incident detection and forensic analysis. The contribution of this chapter is a discussion on how to solve these the two research questions: how can SIEM systems benefit from VMI-based tracing mechanisms even so some operations have significant performance overhead? And how can the challenge of performance impacts be tackled in SIEM systems?

First, in Section 6.1, we discuss related state-of-the-art approaches of security relevant VMI-based applications. In Section 6.2, we describe the design of an architecture that uses VMI-based tracing techniques in combination with other log files to detect security incidents in virtualized infrastructures. Section 6.4 shows how parts of the architecture can be implemented. Section 6.5 evaluates the approach. Finally, Section 6.6 summarizes this chapter.

6.1 State of the Art

Various publications address applications of digital forensics and virtual machine introspection. In the following section, we want to discuss the publications that are most related to our approaches.

One of the first VMI related publications by Garfinkel et al. [GR03], which was published in 2003, already proposed to use VMI for intrusion detection systems. To detect incidents, they present four different policy modules: a lie detector that compares the system state acquired by an in-guest agent and by VMI, a integrity checker that compares the memory of a binary in memory with the data stored in the ELF binary, a signature detector that scans the memory for known malware signatures and finally a detector that checks the userspace process for the use of raw sockets. Their approach checks periodically the state of the system but does not analyze the execution. Additionally, the authors monitor the access to the memory location where the system call table is stored. This allows them to be immediately informed when a rootkit attempts to modify this section in memory. However, the authors to not monitor the execution of the analyzed system, e.g., to trace system calls.

CRIMES [Raj+18] is a security framework for cloud infrastructures that aims to detect new attacks and to minimize their harm. To lower the performance overhead caused by tracing, they execute a virtual machine normally and buffer all in- and outputs such as disk writes and network packets. After a certain time interval (epoch), they check the integrity of the system. If a potential incident was detected forensic tools can be used to find the reason for it. In case of an incident the checkpoint created before the last epoch can be replayed.

Cohen et al. [CBC11] introduce the GRR Rapid Response framework that supports monitoring and live-forensics in enterprise networks. They use in-guest agents for the data acquisitions, which must be installed on all clients. A central scalable entity collects, combines and processes the log files. Bushouse et al. [BR18b] extend GRR by implementing agents for the forensics framework GRR [GRR] that use VMI.

Agarwal et al. [Aga+18] proposes to compare network traces gained from a sensor on a host that could be infected by malware and a dedicated trustworthy host that is only connected to the network via the port mirroring interface. This approach is similar to the lie detector proposed by Garfinkel et al. [GR03] and can be used to detect malware that actively hides communication. Instead of VMI, he uses network traffic. Hence, they can not monitor the control flow within a system, which is required to analyze incidents.

Borisaniya et al. [BP19] present an intrusion detection system for cloud environments using VMI-based system call tracing with Nitro [PSE11]. However, they do not measure the overhead caused by the monitoring and do not explain how to efficiently select system calls that reduce the tracing overhead.

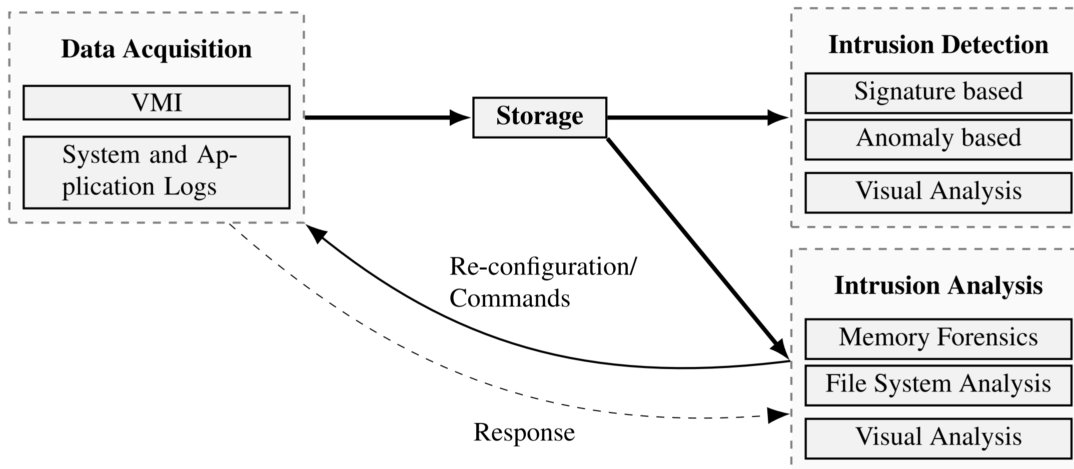


Figure 6.1: The data flow and interactions in a SIEM system using VMI as an additional data source

6.2 Threat Model and Assumptions

In this chapter we make the following assumptions. We assume that an adversary can compromise a virtual machine and all components running in it. This is the most relevant attack vector since virtual machines have often an exposed interface to the outside that could be exploited. However, we do not cover attacks that target the hypervisor [PSL13; Rak+17], the CPU [Com18d; Com18e; Van+18; Wei+18] or other parts of the infrastructure such as hardware components (e.g., switches) or software parts outside the monitored virtual machine (e.g., the cloud management software) [Som+11]. Additionally, we do not address attacks that target the VMI interpretation routine.

6.3 System Design

In this section, we describe a generic architecture of a SIEM system (based on two architectures *CloudIDEA* [Fis+15] and DINGfest [Men+18]), which uses VMI-based tracing to augment the standard log files.

The following components are required for accomplishing the goals of a SIEM system that uses VMI for the data acquisition: data acquisition, intrusion detection, incident analysis, visual analysis, reconfiguration, and a central data storage.

Data Acquisition

The data acquisition is responsible for collecting and sending system logging information of different systems in a central data storage. Multiple data sources can be used for detecting and analyzing incidents. Each of them has different properties that qualify it for specific use cases, and the data sources can be combined to achieve better detection results. The most common data sources can be classified into three categories based on their origin: in-guest agents, infrastructure-based logs, VMI/Hypervisor-based. To compare them, we use the following criteria

- Level of detail
- Forensic soundness
- Costs
- Stealthiness

The following properties describe the costs:

- Monitoring cost (CPU, Memory)
- Overhead to production system performance
- Overhead to network communication
- Storage/Amount of logs
- Implementation cost

The first category are logs acquired by *in-guest agents*, e.g., logs from applications or the operating system. The forensic soundness of these traces can be considered low as they can be compromised by an attacker who has full control over the system. In order to improve the reliability of these logs, they should be sent to an external entity so that an attacker cannot modify past traces after the system is subverted. The implementation and performance cost of this approach are low because it can be usually easily implemented since many security-relevant applications already provide a logging mechanism. The cost of storing the data can be high because a lot of data can be generated. The stealthiness of this approach is low since an attacker with full control can identify logging mechanisms.

Logs generated by the *network infrastructure*, e.g., from switches and firewalls, can provide detailed information, depending on the capabilities of the device. Powerful network devices can be used, for example, to capture the network traffic or to provide meta-data concerning connections. Those logs can be considered as trustworthy as long as an attacker does not have control over it. The cost of storing the information can be high, depending on which data should be stored. The cost for implementation and overhead depends on the devices that are used. The stealthiness of this approach is high since an attacker can hardly find out what data the infrastructure is logging.

The third category covers traces that are generated by leveraging the *virtualization layer*. The traces can be divided in meta-information from virtual machines (CPU, network and I/O load) and VMI-based tracing and memory analysis. The forensic soundness of this data depends on the security of the separation between a guest virtual machine and the hypervisor, which can be compromised either by hardware or software flaw. The stealthiness of these tracing mechanisms varies and depends on the implementation and the required data. By just extracting meta information, the stealthiness is high. VMI-based methods can be considered as medium or highly stealthy because they usually affect the performance of the monitored system [Tuz+18].

Besides the measurements in Section 3.6, Rakotondravony et al. [RKR17] analyzed the costs of *Libvmtrace*-based tracing. They showed that the overhead for tracing system calls depends on two factors: the frequency, i.e., the number of system calls in a specific time and the time it takes to intercept one system call. The frequency of each system call depends on the application that invokes them. The time for intercepting a system call depends on the complexity of the extraction routine of the passed arguments. The tracing affects the performance of the monitored system since for each intercepted system call it must be paused and the computation time increases. However, VMI-based monitoring is not always expensive. The extraction of the process list can be considered as light-weight analysis and does not cause a large performance overhead.

Intrusion Detection

Since attacks can occur at various locations, all logs of a system should be considered to detect incidents. The detection can be done in several ways. However, due to the high amount of different log files it is not possible for a human operator to read all of them. Thus, the detection must be either automatic or visually prepared so that a human analyst can spot anomalies such as high network load. One way to aggregate log files for human operators is to search for known patterns/fingerprints of events that represent high-level interaction with a system, for example, the login attempt of a user, which can be derived from hundreds of system calls executed during the login process. Some automatic detection mechanisms are described in the next section. In any case, a large amount of data must be processed to detect attacks or anomalies and the processing needs to be fast enough to process all incoming events.

Incident Analysis

After a possible incident has been reported, a human analyst needs to analyze it and decide how to proceed. In some situations, he needs to request more information, e.g., by taking a memory snapshot of the affected system. Thus, there needs to be a way to send commands from the forensic workstation to the VMI data acquisition. The process of investigating an incident can be supported with forensic tools and visual analysis, but in any case, a human analyst needs to use them and interpret the results. Thus, an analysis can be time intensive and is not always in real-time even so it should be as fast as possible to prevent further damage.

Visual Analysis

The visual analysis in SIEM has two main tasks: detect anomalies and investigate potential attacks. Since not all attacks can be detected by memory signatures or a set of pre-defined actions such as system calls, it is essential that a human analyst can monitor a system with visual means to detect anomalies that are not (yet) defined in rules. VMI-based tracing and system and applications logs, in general, contain a high amount of information and most of it is usually not necessary to the forensic analysis. Thus, the task of visual analysis is to provide all log entries in a way that they can be linked together, filtered so that the required information, e.g., the executed commands during an attack can be extracted.

Reconfiguration

As discussed earlier, it is often not possible, not required or allowed to activate all tracing components. Hence, administrators, who use SIEM systems, need to configure which data acquisition mechanisms are effective at run-time. In most of the cases, this is triggered by the intrusion analysis. However, a human analyst might also want to enable more detailed tracing to analyze suspicious behavior.

Central Data Storage

All gathered data should be collected at central storage so that events can be easily accessed and processed. Since there is a high amount of different log entries, the storage service needs to be fast and scalable. Additionally, it needs to have a strategy for deleting old logs regularly to free space for more recent entries. Thus, a stream processing architecture such as Apache Kafka is feasible for storing data and processing them for incident detection. However, the data needs to be formatted in a way that supports automatic processing to detect incidents.

An essential aspect of the visualization and analysis is the ability to link events from different sources and to search for specific patterns. Thus, the data store needs to provide these functionalities. Apache Kafka cannot provide this since it only allows to query data from a specified offset, but it does not allow filtering or query for specific strings. In this case, a database that is optimized on indexing and querying data such as ElasticSearch is more appropriate. However, such databases are slow at inserting data since they also index them to improve the speed for later search operations.

6.4 Implementation

This section discusses aspects of the implementation of the system design of a SIEM system that uses VMI.

Data Acquisition

The access to virtual machines used by VMI can be granted by using the *CloudPhylactor* architecture (see Section 4.4). For the VMI-based monitoring tools such as *Libvmtrace*, Drakvuf or others can be used. For the use in a SIEM system, those tools should provide an interface to reconfigure the tracing mechanisms dynamically or to execute specific forensic operations, e.g., take a memory snapshot.

System and application logs can be acquired and collected from different systems, for example, by using traditional solutions such as syslog or with the ELK-stack[1].

Reconfiguration Approaches

For the reconfiguration of VMI-based tracing mechanisms the *CloudIDEA* and *DINGfest* architecture discuss two different approaches. The main idea of the *CloudIDEA* architecture is to have a dedicated analysis environment where in case of incidents affected virtual machines can be migrated. It is isolated from the production environment and has more CPU performance to run in-depth analysis so that service level agreements with tenants can be met.

The focus of the DINGfest architecture is the interaction between the different components of a SIEM system. It uses Kafka [KNR+11] as central storage for the logs and for sending commands between the entities. In contrast to the *CloudIDEA* architecture, it does not have an isolated analysis environment. However, it can dynamically re-configure the tracing in case of an incident so that it provides in-depth traces that can be used for forensic reports by sending commands to the acquisition if suspicious behavior is detected.

VMI-based Detection Mechanisms

Since VMI has performance drawbacks, it should only be used when it provides additional benefits compared to traditional in-guest agents, which is the case when

- The operating system is compromised, e.g., by a rootkit that hides itself from in-guest agents.
- The malware can disable/compromise the in-guest agent or virus scanner.
- To preserve data/traces of attacks or malware that help to reconstruct the operations, e.g., by extracting the key from memory that was used to encrypt files on the hard disk.

In the following paragraphs, detection mechanisms are discussed that benefit most from VMI-based tracing.

Signature Based

Signature-based approaches scan the contents in main memory for known malware signatures. It can be implemented very efficiently since the virtual machine does not need to be paused for accessing the contents in memory. Additionally, the VMI-based analysis can access the whole memory of a virtual machine and malware is not able to hide at protected areas. For example, the yarascan plug-in of volatility uses Yara signatures to search for malware signature in memory[2].

[1] `https://www.elastic.co/elk-stack`, Accessed 2019-06-19
[2] `https://github.com/volatilityfoundation/volatility/wiki/Command-Reference-Mal#yarascan`, Accessed 2019-06-19

Integrity Measurement

Bridging the semantic guest is a complex and costly task and requires knowledge about the monitored applications. However, in-guest agents can be compromised by attackers that take over a system. By combining both approaches, security can be improved, e.g., by doing security analysis with an in-guest agent and check its integrity with VMI to make sure that it is still working as intended and that an attacker did not turn off or manipulate the monitoring of the in-guest agent.

One way to check it is to read and compute a hash over the memory of the process that is mapped read-only and executable, e.g., the text segment that contains all the CPU instructions of a program. That is a very basic approach. However, it can detect changes to instructions that differ from the known (valid) state. It does not work on self-modifying binaries or buffer-overflow based attacks. Knowing the source code of the in-guest agent helps to define the valid state and helps to implement more sophisticated tools that check the control flow of the application.

Another challenge of this approach is the access of swapped pages. They can be either ignored under the assumption that when they are not loaded to memory they are also not used, or they somehow need to be loaded to memory. We already presented a solution that injects page faults to processes in Section 3.5.1. However, this approach introduces an additional overhead to the monitored system.

Lie Detector

Garfinkel [GR03] introduced the concept of a VMI-based lie detector. The idea of it is to compare the system state, e.g., the process list or open network connections, from in-guest agents and compare it with the results gained with VMI. If malware hides itself or a malicious process by removing entries in the `proc` file system the in-guest agent does not see it [HW12]. However, the VMI-based process list extraction that parses the data structures of the kernel can detect the hidden process. By comparing the two process lists, the hidden process should be the difference. One problem of this approach is that it is necessary to do the two measurements at the same time so that always the same system state is measured to avoid false positives. Another approach to circumvent that problem is to do more measurements and only generate an alert if the difference is visible in a set of measurements.

This approach does not work with kernel module-based rootkits, which hide themselves by removing their entry from the kernel data structure that stores the list of kernel modules. In that case, both, the in-guest agent and the VMI-based approach would not work if they are only considering the kernel data structure. One way to detect this rootkit is by dynamically reconstructing the system state by monitoring system calls that modify it and compare it with the system state gathered via VMI or the in-guest agent. In that case, if a kernel module is inserted but cannot be detected later because it was removed from the list, the discrepancy can be used to detect malicious behavior.

This approach of cross-checking logs and events from different data sources can be extended for various other use cases.

TLS and SSH Sessions Monitoring

As described in Chapter 5, it is possible to use VMI to monitor SSH sessions and to decrypt TLS communication channels, e.g., to analyze decrypted malware communication or to monitor the activities of an attacker in an SSH session. Both approaches can be very useful in SIEM systems and implemented in such a way that the monitoring costs can be reduced to a level where it is possible to use them production environments.

System Call Monitoring

System call tracing is an often discussed mechanism to detect malicious software [Din+08; PSE11]. The drawback of this approach, especially by using VMI-based tracing mechanisms, is that the monitoring overhead is too high for SIEM systems. One way to reduce the overhead is to trace

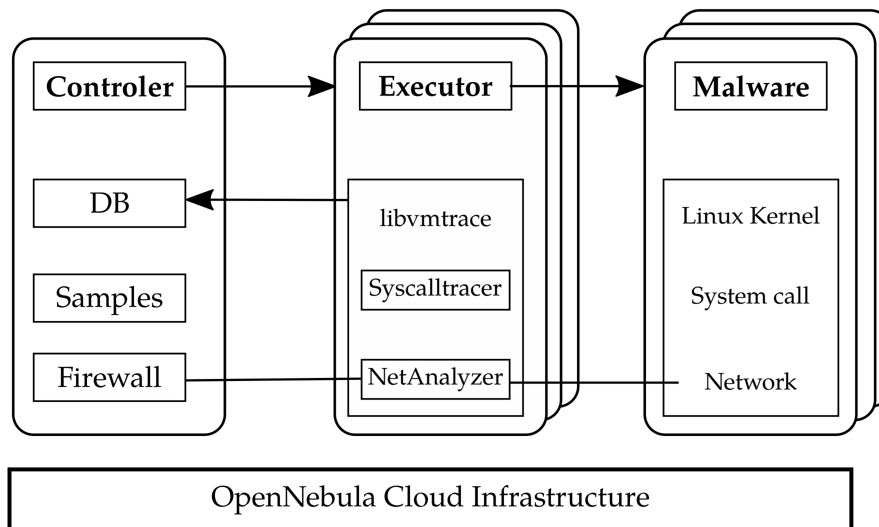


Figure 6.2: Architecture for running and analyzing malware samples in a cloud environment [TK17]

only a subset of all system calls. The selection of system calls should consider their tracing cost, i.e., how often they are called, and how well they can be used to detect malicious behavior. For example, a good system call to trace could be `init_module`, which is used to load kernel modules. This system call is not used very often (low tracing cost) and might be used to load by malware to load rootkit kernel modules.

In [TK17] and [TR16] we propose a system design for dynamic malware analysis. It leverages the *CloudPhylactor* to start a virtual machine for a malware sample and monitors it executing using VMI-based system call tracing (see Figure 6.2). After running the malware samples in this architecture, we train a machine learning-based classifier that aims to detect malicious behavior based on systems calls. For the feature selection of this classifier, we discuss how to find a trade-off between detection accuracy and tracing overhead.

6.5 Evaluation

The evaluation of a fully fledged SIEM system is complicated and out of the scope of this thesis. Thus, we only want to evaluate how VMI can contribute to a SIEM system.

Attack Surface

One important aspect of introducing a new component to a SIEM system is that it does not introduce a new attack surface. By using VMI to monitor the execution of a VM, the only interaction an attacker has with the analysis systems is the contents in memory. By placing crafted data in memory, an attacker could subvert the interpretation routine to inject instructions to the monitoring machine. If the permissions are configured in such a way that each MVM has only access to one other PVM, an attacker can not cause harm to other virtual machines running on the same node. This attack scenario assumes that the attacker has permissions to the memory regions that are analyzed by VMI, which is often only parts of the kernel space. Thus, we can consider that the attacker already entirely took over the system.

Lie Detector

We have tested the lie detector implementation with an adapted version of the Linux kernel module rootkit available at [HW12]. It hooks the file system operations of the `proc` file system to remove the process data structures of dynamically configured processes. Additionally, it removes itself from the list of kernel modules.

By using VMI, it is possible to detect hidden processes, since the Linux Kernel still maintains the data structures. In-guest agents that use the `proc` file system to build the process list cannot detect the hidden process anymore. Thus, the discrepancy between the results of the in-guest agent and the VMI-based analysis can be used to raise an alarm for a potential incident.

The second method of removing the module from the module list makes the rootkit also invisible for VMI-based approach that parse the kernel data structures. However, the volatility plug-in that searches hidden kernel modules can spot it. It uses a heuristic based approach and searches for the signatures of kernel modules in memory instead of traversing the list of kernel modules. Thus, this approach is slower, but it can detect the rootkit. By using this approach of searching for hidden kernel modules, the lie detector approach can be used to detect the discrepancy between the view of the in-guest agent and the output of volatility.

6.6 Summary

In this chapter, we discussed how virtual machine introspection can be used in SIEM systems. The main problem of using VMI-based tracing mechanisms in SIEM systems, is the performance. To tackle this problem we present two solutions. The *CloudIDEA* architecture distinguishes between light-weight and heavy-weight tracing. In normal mode, the virtual machines are monitored using a light-weight mechanism to detect incidents. Only in case of a potential incident, heavy-weight tracing is activated, and the potentially affected virtual machine is migrated to an isolated analysis environment with more resources. The *DINGfest* architecture uses a similar approach and provides analysts means to activate and configure on demand the appropriate tracing mechanism.

Besides, we discussed VMI-based monitoring techniques that are particularly suitable for intrusion detection. The key aspect here is to leverage the untampered view on the system state. It can be used to build signature-based malware scanning in main memory and to measure the code integrity, for example, to monitor in-guest agents. Lie detectors are another approach, which compares the in-guest view on the system with the system view obtained with VMI.

7 CONCLUSIONS

This thesis tackles problems of digital forensics on main memory in production environments: the lack of tools for VMI analysis; technical problems of the data acquisition process; fast information extraction; and the adaption of VMI-based methods to different application areas of digital forensics.

For addressing the problem of missing tool support, this thesis summarizes the requirements of virtual machine introspection and presents the *Libvmtrace* framework that can be used for multiple main memory forensics related purposes and supports dynamic re-configuration at run-time. Additionally, we present the implementation and evaluate its performance.

For data acquisition, this thesis shows how the main memory of mobile devices and virtual machines in cloud environments can be accessed. For mobile devices, we improve the practicability of cold boot attacks and show how to perform memory acquisition from the secure world of the TrustZone. For cloud environments, we present an architecture that enables cloud customers to perform virtual machine introspection on their production virtual machines and allows live migration of the monitored system in addition.

For fast information extraction, this thesis evaluates different strategies for efficiently extracting TLS sessions keys from the address space of applications. To improve the performance, we propose a mechanism that derives a path that follows the pointers in data structures from the stack of an application to the master secret of TLS connection. With that approach we achieve a monitoring overhead that is feasible for extracting TLS session keys from common Android applications at run-time.

For SIEM systems, we show that VMI-based tracing can be used to augment the common data acquisition process with additional information. One important aspect of using VMI-based tracing in SIEM systems of production environments is to maintain low performance overhead. To achieve that, we propose to use light-weight monitoring mechanisms to detect incidents and heavy-weight monitoring mechanisms to analyze incidents.

7.1 Contributions

VMI Architecture There are only a few VMI-based tools that can be used to trace virtual machines. However, those tools are often complicated to use or outdated and cannot be used with currently available virtualization solutions. Our contribution concerning this problem is that we discuss the design and implementation of our *Libvmtrace* framework, which is the basis for *TLSKex*,

DroidKex, *Sarracenia* and the *DINGfest* architecture in this thesis. By using *Libvmtrace* for these applications, we show that our framework is feasible for different use cases and platforms.

Data Acquisition In this thesis, we analyze how cold boot attacks and the ARM TrustZone can be used for data acquisition. Previous cold boot attacks typically use a fully-fledged Linux kernel to access the remaining contents in memory after a restart. However, this approach overwrites the data structures of the prior running system. To tackle this problem, we present the design and implementation of a minimal operating system for accessing the data that minimizes the modifications in the memory of the analyzed system. The retrieved data is transferred via the serial interface to a forensic workstation where it gets analyzed. With that approach, we ensure that only very little data is overwritten by the minimal operating system and that we can retain more data for the analysis.

Cold boot attacks cannot be used for live analysis of a running system because they require to restart the system. For that purpose, we port the interface of LibVMI to the secure world of the ARM TrustZone. With that approach, we can analyze the system state of the normal world from the secure world at run-time. By providing the same interface as LibVMI, developers can port already available VMI-applications to mobile devices.

The contribution of the *CloudPhylactor* architecture is the concept of using dedicated virtual machines for running VMI-based analysis on production virtual machines. The separation into these two categories has the advantage of minimizing the attack vector against VMI-based applications and makes it feasible for cloud environments where regular users do not have access to the most privileged domain. Hence, the *CloudPhylactor* architecture enables cloud tenants to perform VMI-based analysis on their own virtual machines.

The *TwinPorter* architecture extends the *CloudPhylactor* architecture to monitor virtual machines that are live migrated during their run-time. It ensures that both the monitoring and the monitored virtual machine are migrated in parallel to the same cloud node. To keep the downtime of the monitored virtual machine low and to ensure that the monitored system is never running without monitoring, we extend the pre-copy migration mechanism of Xen.

Information Retrieval To tackle the problem of efficient information retrieval, we discuss two real-world use cases. The first one is the extraction of the cryptographic key material from the address space of applications to decrypt their TLS communication. We point out that the extraction mechanism must be fast so that the normal behavior of an application is not disturbed and executed at the right time when the required information is in memory. The contribution of the *TLSKex* architecture is to use a brute-force based approach to extract the cryptographic key material. To speed up the process *TLSKex* minimizes the search space in advance by applying multiple heuristics that do not require any semantic knowledge about the application.

The contribution of the *DroidKex* architecture is to optimize the approach of *TLSKex* by directly accessing the master secret in memory using semantic knowledge about the data structures obtained from snapshots in advance. To obtain the semantic knowledge, it presents an approach for deriving the data structure layout from several snapshots in a training phase. To trigger the extraction process, *TLSKex* monitors the network traffic and starts the extraction process when a TLS session is established. *DroidKex* intercepts the control flow of applications and extracts the key when network related send and receive functions are invoked.

The second example is the reconstruction of SSH sessions obtained by monitoring the SSH daemon in a virtual machine using VMI. To minimize the overhead caused by the monitoring, we intercept function calls and extract the session information using the semantic knowledge obtained from the debugging information. The contribution of the *Sarracenia* honeypot is the implementation of VMI-based honeypot that reconstructs the activities of a user in an SSH session. In order to be stealthy to the attackers, we show how to optimize the performance of the tracing methods by selecting the function calls where the information can be extracted with minimal overhead.

VMI in SIEM Systems In this thesis, we discuss how VMI-based tracing mechanisms can be used to improve the data acquisition in SIEM systems. The contribution of the SIEM system use case is the discussion on how VMI-based tracing mechanisms can be used efficiently for the data acquisition. We propose two approaches that tackle the problem of the overhead caused by the tracing to use VMI for data acquisition. The contribution of the *CloudIDEA* architecture is to address the performance problem by only using light-weight detection mechanisms in production systems to detect anomalies. In the case of an incident, the corresponding virtual machine is migrated to an isolated analysis environment with more resources. There, heavy-weight monitoring can be used to gain more insights about an incident. The contribution of the *DINGfest* architecture is that a forensic investigator can dynamically interact with VMI-based tracing and select tracing methods that are appropriate for the current situation.

7.2 Future Work

The approaches presented in this thesis improve the security of systems. However, they can be also misused to compromise the privacy and confidentially of user-related data in cloud environments without the user's knowledge. In particular, the stealthiness of the information extraction leaves cloud tenants unaware of the ex-filtration of data. This raises two questions: Do cloud customers have any means to detect VMI-based tracing on their machines and can VMI-based tracing techniques be so stealthy that they can not be detected by malware or legitimate cloud users? Both questions have to be addressed in the future.

Currently, there is no VMI-based tracing method that is completely stealthy, especially due to the overhead that can be measured by attackers. Thus, future work should analyze how the performance of VMI-based tracing can be further improved. One way to do that is to minimize the performance impact caused by handling VMI events. This can for example achieved by integrating the VMI-based analysis directly into the hypervisor so that no additional switches to the analyzing VM (Dom0 or MVM) are required to handle VMI events at run-time.

Additionally, the ITZ library that ports the LibVMI interface to the secure world of the TrustZone should be extended in the future. One step in this direction is to use the approach of SPROBES [GVJ14] to support dynamic tracing of the normal world from the secure world together with the ITZ library.

Furthermore, the concept of monitoring a system from outside should be extended to other types of virtualization. For example, Linux containers are a light-weight virtualization approach for devices with limited resources or for cloud computing. Hence, this technique could be used to build new honeypot systems that run in cloud or internet-of-things environments.

List of Abbreviations

AEAD	Authenticated Encryption with Associated Data
BMA	Bare Metal Application
CCS	Change Cipher Spec
CH	Client Hello
CR	Client Random
DAC	Discretionary Access Control
DH	Diffie-Hellman
Dom0	Domain 0
DTB	Directory Table Base
ECDH	Elliptic-curve Diffie-Hellman
ELF	Executable and Linking Format
Flask	Flux Advanced Security Kernel
IaaS	Infrastructure-as-a-Service
MAC	Mandatory Access Control
MCS	Multi-category Security
MLS	Multi-level Security
MVM	Monitoring Virtual Machine
PFS	Perfect Forward Secrecy
PRF	Pseudo Random Function
PVM	Production Virtual Machine
SH	Server Hello
SIEM	Security Information and Event Management
SR	Server Random
TCI	TwinCommunicationInterface
TLS	Transport Layer Security
TR	TwinReceiver
TS	TwinSender

VM .... Virtual Machine
VMI .... Virtual Machine Introspection
VMM .... Virtual Machine Monitor

XSM .... Xen Security Modules

List of Figures

1.1 Contributions of the thesis 3

2.1 LibVMI architecture with Xen 18
2.2 TLS handshake 22

3.1 Generic design of a forensic framework 34
3.2 *Libvmtrace*: design 36
3.3 *Libvmtrace*: timing of breakpoint handling mechanism 38
3.4 *Libvmtrace*: steps required for the injection of the read access 39
3.5 *Libvmtrace*: steps required for injecting code into the context of a running process 40
3.6 *Libvmtrace*: network connection monitoring overhead 48

4.1 Architecture for cold boot via serial interface 55
4.2 ARM secure boot procedure 56
4.3 ITZ-library design 62
4.4 *CloudPhylactor*: design 66
4.5 *CloudPhylactor*: VMI performance evaluation when PVM is idle 70
4.6 *CloudPhylactor*: evaluation of read-page operation 70
4.7 *CloudPhylactor*: VMI performance evaluation when PVM is under load 71
4.8 *CloudPhylactor*: VMI read-page performance evaluation when PVM is under load . 71
4.9 *CloudPhylactor*: VMI performance evaluation policy size 10,000 73
4.10 *CloudPhylactor*: VMI performance evaluation policy size 50,000 73
4.11 *CloudPhylactor*: VMI read-page performance evaluation policy size 10,000 73
4.12 *CloudPhylactor*: VMI read-page performance evaluation policy size 50,000 73
4.13 *TwinPorter*: migration approach 76
4.14 *TwinPorter*: migration steps 78
4.15 *TwinPorter*: communication channels 79

5.1 *TLSKex*: interaction of the components 93
5.2 *TLSKex*: monitoring actions during TLS session negotiation 94
5.3 *DroidKex*: interaction of the components 100
5.4 *DroidKex*: monitoring actions during TLS session negotiation 102
5.5 *DroidKex*: OpenSSL data structures 103
5.6 *DroidKex*: Firefox data structures 104
5.7 *DroidKex*: path expansion 105
5.8 *DroidKex*: memory size of applications 106
5.9 *DroidKex*: snapshot time 107
5.10 *DroidKex*: extraction time 107

5.11 *DroidKex*: path computation time 108
5.12 *DroidKex*: path computation iterations 108
5.13 *DroidKex*: overhead per connection 108
5.14 *Sarracenia*: architecture 111
5.15 *Sarracenia*: overhead evaluation 114
5.16 Comparison of information extraction approaches 116

6.1 Data flow and interactions in SIEM systems 121
6.2 Cloud-based architecture for malware analysis 126

List of Tables

3.1 Comparison of different virtual machine introspection frameworks 26
3.2 *Libvmtrace*: evaluation of system call tracing overhead in different use cases 45
3.3 *Libvmtrace*: evaluation of CR3 monitoring overhead 46

4.1 Coldboot evaluation of remanence effect 59
4.2 Coldboot evaluation of volatility plug-ins 60
4.3 ITZ-library: code base size 63
4.4 *CloudPhylactor*: evaluation policy compile time 72
4.5 *TwinPorter*: evaluation time required for the migration steps 81
4.6 *TwinPorter*: total time required for the migration and the corresponding downtime 82

5.1 Comparison of different TLS decryption solutions 86
5.2 Comparison of SSH honeypots 88
5.3 *TLSKex*: modified memory pages during negotiation 96
5.4 *TLSKex*: evaluation of key identification heuristics 97
5.5 *Sarracenia*: intercepted function calls of the SSH daemon 112

List of Listings

2.1 LibVMI functions 20

3.1 *Libvmtrace*: instructions used to trigger a page-fault 39
3.2 *Libvmtrace*: implanted instructions required to inject a command 41

BIBLIOGRAPHY

[AA06] Keith Adams and Ole Agesen. “A Comparison of Software and Hardware Techniques for x86 Virtualization.” In: *SIGARCH Comput. Archit. News* 34.5 (Oct. 2006), pp. 2–13. ISSN: 0163-5964. DOI: 10.1145/1168919.1168860. URL: http://doi.acm.org/10.1145/1168919.1168860.

[AL13] Ferrol Aderholdt and Stephen L. Scott. “The Secure Migration of a Virtual Machine Introspection Intrusion Detection System.” In: *IASTED Multiconferences, (793) Artificial Intelligence and Applications / 794: Modelling, Identification and Control / 795: Parallel and Distributed Computing and Networks / 796: Software Engineering / 792: Web-based Education*. Mar. 2013. DOI: 10.2316/P.2013.795-046.

[Aga+18] Mayank Agarwal, Rami Puzis, Jawad Haj-Yahya, Polina Zilberman, and Yuval Elovici. “Anti-forensic = Suspicious: Detection of Stealthy Malware that Hides Its Network Traffic.” In: *IFIP International Conference on ICT Systems Security and Privacy Protection*. Springer. 2018, pp. 216–230.

[ARM09] ARM Limited. *ARM Security Technology – Building a Secure System Using TrustZone Technology*. 2009. URL: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf.

[Aza+14] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. “Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World.” In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 90–102. ISBN: 978-1-4503-2957-6. DOI: 10.1145/2660267.2660350. URL: http://doi.acm.org/10.1145/2660267.2660350.

[BSM14] Hyun wook Baek, Abhinav Srivastava, and Jacobus Van der Merwe. “CloudVMI: Virtual Machine Introspection as a Cloud Service.” In: *2014 IEEE International Conference on Cloud Engineering*. Mar. 2014, pp. 153–158. DOI: 10.1109/IC2E.2014.82.

[Bah+10] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. “DKSM: Subverting Virtual Machine Introspection for Fun and Profit.” In: *2010 29th IEEE Symposium on Reliable Distributed Systems*. Oct. 2010, pp. 82–91. DOI: 10.1109/SRDS.2010.39.

[Bal+10] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. “Efficient Detection of Split Personalities in Malware.” In: *Proceedings of the Symposium on Networked Systems Design & Implementation*. 2010.

[Bar00] Moshe Bar. *The Linux Signals Handling Model.* https://www.linuxjournal.com/article/3985, Accessed: 2019-07-03. 2000.

[Bar+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. “Xen and the Art of Virtualization.” In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: http://doi.acm.org/10.1145/1165389.945462.

[BVR13] Michael Beham, Marius Vlad, and Hans P. Reiser. “Intrusion detection and honeypots in nested virtualization environments.” In: *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* June 2013, pp. 1–6. DOI: 10.1109/DSN.2013.6575329.

[BL73] D. Elliott Bell and Leonard J. LaPadula. *Secure Computer Systems: Mathematical Foundations.* Tech. rep. MTR-2547, Vol. 1. Bedford, MA: MITRE Corp., 1973.

[Ben+13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. “SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge.” In: *Advances in Cryptology – CRYPTO 2013.* 2013, pp. 90–108. ISBN: 978-3-642-40084-1.

[Beu18] Patrick Beuth. *Wie Apple und Google der Polizei das Leben erschweren.* http://www.spiegel.de/netzwelt/gadgets/android-9-und-ios-12-wie-apple-und-google-der-polizei-die-arbeit-erschweren-a-1233738.html. Accessed: 2018-10-23. 2018.

[BMZ14] Sandeep Bhatt, Pratyusa K. Manadhata, and Pratyusa K. Zomlot. “The Operational Role of Security Information and Event Management Systems.” In: *IEEE Security and Privacy* 12.5 (Sept. 2014), pp. 35–41. ISSN: 1540-7993. DOI: 10.1109/MSP.2014.103. URL: doi.ieeecomputersociety.org/10.1109/MSP.2014.103.

[BP19] Bhavesh Borisaniya and Dhiren Patel. “Towards virtual machine introspection based security framework for cloud.” In: *Sādhanā* 44.2 (Jan. 2019), p. 34. ISSN: 0973-7677. DOI: 10.1007/s12046-018-1016-6. URL: https://doi.org/10.1007/s12046-018-1016-6.

[BY17] Alexei Bulazel and Bülent Yener. “A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web.” In: *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium.* ROOTS. Vienna, Austria: ACM, 2017, 2:1–2:21. ISBN: 978-1-4503-5321-2. DOI: 10.1145/3150376.3150378. URL: http://doi.acm.org/10.1145/3150376.3150378.

[BR18a] Micah Bushouse and Douglas Reeves. “Furnace: Self-service Tenant VMI for the Cloud.” In: *Research in Attacks, Intrusions, and Defenses.* Springer International Publishing, 2018, pp. 647–669. ISBN: 978-3-030-00470-5.

[BR18b] Micah Bushouse and Douglas Reeves. “Hyperagents: Migrating Host Agents to the Hypervisor.” In: *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy.* CODASPY ’18. Tempe, AZ, USA: ACM, 2018, pp. 212–223. ISBN: 978-1-4503-5632-9. DOI: 10.1145/3176258.3176317. URL: http://doi.acm.org/10.1145/3176258.3176317.

[CJ06] Sean Campbell and Michael Jeronimo. “An introduction to virtualization.” In: *Published in “Applied Virtualization”, Intel* (2006).

[Car16] Radu Caragea. “TeLeScope - real-time peering into the depths of TLS traffic from the hypervisor.” In: *HITBSECCONF.* 2016. URL: http://conference.hitb.org/hitbsecconf2016ams/wp-content/uploads/2015/11/D1T1-Radu-Caragea-Peering-into-the-Depths-of-TLS-Traffic-in-Real-Time.pdf.

[CG04] Brian D. Carrier and Joe Grand. "A hardware-based memory acquisition procedure for digital investigations." In: *Digital Investigation* 1.1 (2004), pp. 50–60. ISSN: 1742-2876. DOI: 10.1016/j.diin.2003.12.001.

[Cas11] Eoghan Casey. *Digital evidence and computer crime*. Academic Press, 2011.

[CER03] CERT. *Multiple vulnerabilities in snort preprocessors*. https://www.cert.org/historical/advisories/CA-2003-13.cfm, Accessed: 2019-01-31. Apr. 2003.

[Cer17] US Cert. *Alert (TA17-075A) HTTPS Interception Weakens TLS Security*. https://www.us-cert.gov/ncas/alerts/TA17-075A, Accessed: 2019-01-31. 2017.

[CXZ08] Kang Chen, Jun Xin, and Weimin Zheng. "Empirical Performance Evaluation of Message Passing Programs Running in Virtual Machines." In: *Grid and Cooperative Computing, 2008. GCC '08. Seventh International Conference on*. Oct. 2008, pp. 620–627. DOI: 10.1109/GCC.2008.98.

[Che+08] Xu Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario. "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware." In: *IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. June 2008, pp. 177–186. DOI: 10.1109/DSN.2008.4630086.

[Cla+05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Hansen, Gorm Jacob, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. "Live migration of virtual machines." In: *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation*. USENIX Association. 2005, pp. 273–286.

[CNB17] CNBC. *Senator reveals that the FBI paid $900,000 to hack into San Bernardino killer's iPhone*. https://www.cnbc.com/2017/05/05/dianne-feinstein-reveals-fbi-paid-900000-to-hack-into-killers-iphone.html, Accessed: 2019-01-31. 2017.

[Coh14] Michael Cohen. "Rekall memory forensics framework." In: *Sans DFIR Summit Prague* (2014). URL: https://digital-forensics.sans.org/summit-archives/dfirprague14/Rekall_Memory_Forensics_Michael_Cohen.pdf.

[CBC11] Michael I. Cohen, D. Bilby, and Germano Caronni. "Distributed forensics and incident response in the enterprise." In: *Digital Investigation* 8 (2011). The Proceedings of the Eleventh Annual DFRWS Conference, S101–S110. ISSN: 1742-2876. DOI: 10.1016/j.diin.2011.05.012. URL: http://www.sciencedirect.com/science/article/pii/S1742287611000363.

[Coj15] Razvan Cojocaru. *The Bitdefender virtual machine introspection library is now on GitHub*. https://blog.xenproject.org/2015/08/04/the-bitdefender-virtual-machine-introspection-library-is-now-on-github/, Accessed: 2019-01-31. 2015.

[Cok08] George Coker. *Xen Security Modules (XSM)*. http://pdub.net/proj/usenix08boston/xen_drive/resources/xensummit/slides/coker-xsm-summit-090706.pdf, Accessed: 2019-01-31. 2008.

[Com+10] DWARF Debugging Information Format Committee et al. "DWARF debugging information format, version 4." In: *Free Standards Group* (2010).

[Com04] Common Vulnerabilities and Exposures. *CVE-2004-0536*. Available from MITRE, CVE-ID CVE-2004-0536. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-0536, Accessed: 2019-01-31. 2004.

[Com15a] Common Vulnerabilities and Exposures. *CVE-2015-2752*. Available from MITRE, CVE-ID CVE-2015-2752. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2752, Accessed: 2019-01-31. 2015.

[Com15b] Common Vulnerabilities and Exposures. *CVE-2015-4163*. Available from MITRE, CVE-ID CVE-2015-4163. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4163, Accessed: 2019-01-31. 2015.

[Com15c] Common Vulnerabilities and Exposures. *CVE-2015-4163*. Available from MITRE, CVE-ID CVE-2015-4164. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-4164, Accessed: 2019-01-31. 2015.

[Com15d] Common Vulnerabilities and Exposures. *CVE-2015-7812*. Available from MITRE, CVE-ID CVE-2004-0536. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-7812, Accessed: 2019-01-31. 2015.

[Com18a] Common Vulnerabilities and Exposures. *CVE-2018-12126*. Available from MITRE, CVE-ID CVE-2018-12126. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2018-12126, Accessed: 2019-01-31. 2018.

[Com18b] Common Vulnerabilities and Exposures. *CVE-2018-12127*. Available from MITRE, CVE-ID CVE-2018-12127. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2018-12127, Accessed: 2019-01-31. 2018.

[Com18c] Common Vulnerabilities and Exposures. *CVE-2018-12130*. Available from MITRE, CVE-ID CVE-2018-12130. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12130, Accessed: 2019-01-31. 2018.

[Com18d] Common Vulnerabilities and Exposures. *CVE-2018-3665*. Available from MITRE, CVE-ID CVE-2018-3665. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3665, Accessed: 2019-01-31. 2018.

[Com18e] Common Vulnerabilities and Exposures. *CVE-2018-3690*. Available from MITRE, CVE-ID CVE-2018-3690. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3690, Accessed: 2019-01-31. 2018.

[Com19] Common Vulnerabilities and Exposures. *CVE-2019-11091*. Available from MITRE, CVE-ID CVE-2019-11091. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2019-11091, Accessed: 2019-01-31. 2019.

[Com85] Computer Security Center (U.S.) *Computer Security Requirements: Guidance for Applying the Department of Defense Trusted Computer System Evaluation Criteria in Specific Environments*. CSC-STD. DOD Computer Security Center, 1985. URL: https://web.archive.org/web/20070715134110/http://csrc.nist.gov:80/secpubs/rainbow/std004.txt.

[Cor06] Jose Antonio Coret. *Kojoney - A Honeypot For The SSH Service*. http://kojoney.sourceforge.net/, Accessed: 2019-01-31. 2006.

[Cox18] Joseph Cox. *Cops Told 'Don't Look' at New iPhones to Avoid Face ID Lock-Out*. https://motherboard.vice.com/en_us/article/5984jq/cops-dont-look-iphonex-face-id-unlock-elcomsoft, Accessed: 2019-01-31. 2018.

[DA99] Tim Dierks and Christopher Allen. *The TLS Protocol Version 1.0*. https://tools.ietf.org/html/rfc2246. 1999.

[Din+08] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. "Ether: Malware Analysis via Hardware Virtualization Extensions." In: *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*. Alexandria, Virginia, USA: ACM, 2008, pp. 51–62. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455779.

[Dol+11] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. "Virtuoso: Narrowing the semantic gap in virtual machine introspection." In: *IEEE Symposium on Security and Privacy (SP)*. 2011, pp. 297–312. ISBN: 978-0-7695-4402-1.

[Dua+18] Nuno O. Duarte, Sileshi Demesie Yalew, Nuno Santos, and Miguel Correia. "Leveraging ARM TrustZone and Verifiable Computing to Provide Auditable Mobile Functions." In: *Proceedings of the 15th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*. MobiQuitous '18. New York, NY, USA: ACM, 2018, pp. 302–311. ISBN: 978-1-4503-6093-7. DOI: 10.1145/3286978.3287015. URL: http://doi.acm.org/10.1145/3286978.3287015.

[Dur+17] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. "The Security Impact of HTTPS Interception." In: *Network and Distributed Systems Symposium (NDSS).* 2017.

[DS13] Josiah Dykstra and Alan T Sherman. "Design and implementation of FROST: Digital forensic tools for the OpenStack cloud computing platform." In: *Digital Investigation* 10 (2013), S87–S95.

[Edg15] Jake Edge. *TLS in the kernel.* `https://lwn.net/Articles/666509/`, Accessed: 2019-01-31. 2015.

[Ett17] Wolfgang Ettlinger. *Chainsaw of Custody: Manipulating forensic evidence the easy way.* `https://www.sec-consult.com/en/blog/2017/05/chainsaw-of-custody-manipulating/`. 2017.

[Eur16] European Union Agency for Network and Information Security (ENISA). *Exploring Cloud Incidents.* `https://www.enisa.europa.eu/publications/exploring-cloud-incidents/at_download/fullReport`, Accessed: 2019-01-31. 2016.

[Fis+15] Andreas Fischer, Thomas Kittel, Bojan Kolosnjaji, Tamas K Lengyel, Waseem Mandarawi, Hans P. Reiser, Benjamin Taubmann, Eva Weishäupl, Hermann de Meer, Tilo Müller, and Mykola Protsenko. "CloudIDEA: A Malware Defense Architecture for Cloud Data Centers." In: *Cloud and Trusted Computing.* 2015. ISBN: 978-3-319-26148-5. DOI: `10.1007/978-3-319-26148-5_40`.

[Fou] The Volatility Foundation. *Volatility framework.* `https://github.com/volatilityfoundation`, Accessed: 2019-01-31.

[Fre12] Freescale Semiconductor, Inc. *i.MX53 Multimedia Applications Processor Reference Manual.* `https://cache.freescale.com/files/32bit/doc/ref_manual/iMX53RM.pdf`, Accessed: 2019-06-13. 2012.

[FL13] Yangchun Fu and Zhiqiang Lin. "EXTERIOR: Using a dual-VM Based External Shell for guest-OS Introspection, Configuration, and Recovery." In: *SIGPLAN Not.* 48.7 (Mar. 2013), pp. 97–110. ISSN: 0362-1340. DOI: `10.1145/2517326.2451534`. URL: `http://doi.acm.org/10.1145/2517326.2451534`.

[FLH13] Yangchun Fu, Zhiqiang Lin, and Kevin W Hamlen. "Subverting system authentication with context-aware, reactive virtual machine introspection." In: *Proceedings of the 29th Annual Computer Security Applications Conference.* ACM. 2013, pp. 229–238.

[FZL14] Yangchun Fu, Junyuan Zeng, and Zhiqiang Lin. "HYPERSHELL: A Practical Hypervisor Layer Guest OS Shell for Automated In-VM Management." In: *USENIX Annual Technical Conference.* 2014, pp. 85–96.

[Gar10] Simson L. Garfinkel. "Digital forensics research: The next 10 years." In: *Digital Investigation* 7 (2010), S64–S73. DOI: `10.1016/j.diin.2010.05.009`.

[GR03] Tal Garfinkel and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *Proc. of the Network and Distributed Systems Security Symposium (NDSS).* 2003, pp. 191–206.

[GVJ14] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. "SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture." In: *CoRR* abs/1410.7747 (2014). arXiv: `1410.7747`. URL: `http://arxiv.org/abs/1410.7747`.

[Gen] Genode Labs. *An Exploration of ARM TrustZone Technology.* `https://genode.org/documentation/articles/trustzone`, Accessed: 2019-01-31.

[Gol73] Robert P Goldberg. *Architectural principles for virtual computer systems.* Tech. rep. Harvard Univ Cambridge MA Div of Engeneering and Applied Physics, 1973.

[GRR] GRR Project. *GRR Rapid Response is an incident response framework focused on remote live forensics*. https://github.com/google/grr, Accessed: 2019-03-26.

[GL16] Yufei Gu and Zhiqiang Lin. "Derandomizing Kernel Address Space Layout for Memory Introspection and Forensics." In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. CODASPY '16. New Orleans, Louisiana, USA: ACM, 2016, pp. 62–72. ISBN: 978-1-4503-3935-3. DOI: 10.1145/2857705.2857707. URL: http://doi.acm.org/10.1145/2857705.2857707.

[Gue+17] Miguel Guerra, Miguel Correia, Benjamin Taubmann, and Hans P. Reiser. "ITZ: An Introspection Library for ARM TrustZone." In: *Proceedings of INFORUM*. 2017.

[Gue+18] Miguel Guerra, Benjamin Taubmann, Hans P. Reiser, Sileshi Yalew, and Miguel Correia. "Introspection for ARM TrustZone with the ITZ Library." In: *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. July 2018, pp. 123–134. DOI: 10.1109/QRS.2018.00026.

[Gur+15] Mordechai Guri, Yuri Poliak, Bracha Shapira, and Yuval Elovici. "JoKER: Trusted Detection of Kernel Rootkits in Android Devices via JTAG Interface." In: *IEEE Trustcom/BigDataSE/ISPA*. Vol. 1. Aug. 2015, pp. 65–73. DOI: 10.1109/Trustcom.2015.358.

[Hal+09] J. Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. "Lest we remember: cold-boot attacks on encryption keys." In: *Communications of the ACM* 52.5 (2009), pp. 91–98.

[Han06] Chad Hanson. "SELinux and MLS: Putting the pieces together." In: *Proceedings of the 2nd Annual SELinux Symposium*. 2006.

[HLM15] Yacine Hebbal, Sylvie Laniepce, and Jean-Marc Menaud. "Virtual Machine Introspection: Techniques and Applications." In: *10th International Conference on Availability, Reliability and Security*. Aug. 2015, pp. 676–685. DOI: 10.1109/ARES.2015.43.

[Hen+14] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. "Make It Work, Make It Right, Make It Fast: Building a Platform-neutral Whole-system Dynamic Binary Analysis Platform." In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: ACM, 2014, pp. 248–258. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610407. URL: http://doi.acm.org/10.1145/2610384.2610407.

[HW12] Arkadiusz Hiler and Michał Winiarski. *Sample Rootkit for Linux*. https://github.com/ivyl/rootkit, Accessed: 2019-01-31. 2012.

[HR05] Thorsten Holz and Frederic Raynal. "Detecting honeypots and other suspicious environments." In: *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC*. IEEE. 2005, pp. 29–36.

[Hom13] Josh Homan. *How to Decrypt OpenSSL Sessions using Wireshark and SSL Session Identifiers*. http://www.cloudshield.com/blog/advanced-malware/how-to-decrypt-openssl-sessions-using-wireshark-and-ssl-session-identifiers/, Accessed: 2015-10-02. 2013.

[Hoo09] John Hoopes. *Virtualization for security: including sandboxing, disaster recovery, high availability, forensic analysis, and honeypotting*. Syngress, 2009.

[Hub+16] Manuel Huber, Benjamin Taubmann, Sascha Wessel, Hans P. Reiser, and Georg Sigl. "A flexible framework for mobile device forensics based on cold boot attacks." In: *EURASIP Journal on Information Security* 1 (2016), p. 17. DOI: 10.1186/s13635-016-0041-4.

[Int17] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf. 2017.

[Int] International Secure Systems Lab (iSecLab). *Anubis.* http://anubis.iseclab.org/. The webservice is not available anymore.

[Ive14] Steven Iveson. *Using ssldump to Decode/Decrypt SSL/TLS Packets.* http://packetpushers.net/using-ssldump-decode-ssltls-packets/, Accessed: 2019-01-31. 2014.

[Jai+14] Bhushan Jain, Mirza Basim Baig, Dongli Zhang, Donald E Porter, and Radu Sion. "Sok: Introspections on trust and the semantic gap." In: *Security and Privacy (SP), 2014 IEEE Symposium on.* IEEE. 2014, pp. 605–620.

[JHE14] David Johnson, Mike Hibler, and Eric Eide. "Composable Multi-Level Debugging with Stackdb." In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments.* VEE '14. Mar. 2014, pp. 213–225.

[JAA06] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Antfarm: Tracking Processes in a Virtual Machine Environment." In: *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference.* ATEC '06. Boston, MA: USENIX Association, 2006, pp. 1–1. URL: http://dl.acm.org/citation.cfm?id=1267359.1267360.

[JS11] R.C. Joshi and Anjali Sardana. *Honeypots: a new paradigm to information security.* CRC Press, 2011.

[Jur15] Jurriaan Bremer. *Transparent MITM with Cuckoo Sandbox.* http://jbremer.org/mitm/, Accessed: 2019-01-31. 2015.

[KII10] Y. Kawakoya, M. Iwamura, and M. Itoh. "Memory behavior-based automatic malware unpacking in stealth debugging environment." In: *5th International Conference on Malicious and Unwanted Software.* Oct. 2010, pp. 39–46. DOI: 10.1109/MALWARE.2010.5665794.

[Ken18] Troy Kensinger. *Google and Android have your back by protecting your backups.* https://security.googleblog.com/2018/10/google-and-android-have-your-back-by.html, Accessed: 2019-01-31. 2018.

[KS06] Karen Kent and Murugiah Souppaya. *Guide to Computer Security Log Management.* https://csrc.nist.gov/publications/detail/sp/800-92/final, Accessed: 2019-07-03. National Institute of Standards and Technology (NIST), 2006.

[KVK14] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. "Barecloud: Bare-metal Analysis-based Evasive Malware Detection." In: *Proc. of the 23rd USENIX Conference on Security Symposium.* SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 287–301. ISBN: 978-1-931971-15-7.

[Kle06] Tobias Klein. *All your private keys are belong to us – extracting RSA private keys and certificates from process memory.* http://www.trapkit.de/papers/keyfinder_v1.0_20060205.pdf, Accessed: 2019-01-31. 2006.

[KBC97] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-Hashing for Message Authentication.* https://tools.ietf.org/html/rfc2104. 1997.

[KNR+11] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing." In: *Proceedings of the NetDB.* 2011, pp. 1–7.

[Law96] Kevin P. Lawton. "Bochs: A portable PC emulator for unix." In: *Linux Journal* 1996.29 (1996), p. 7.

[LW18] Jaeho Lee and Dan S Wallach. "Removing Secrets from Android's TLS." In: *Network and Distributed Systems Security (NDSS).* 2018.

[Lei11] John Leitch. *Process Hollowing*. https://github.com/m0n0ph1/Process-Hollowing/raw/master/pdf/process-hollowing.pdf, Accessed: 2019-03-26. 2011.

[Len16] Tamas K. Lengyel. *Stealthy monitoring with Xen altp2m*. https://blog.xenproject.org/2016/04/13/stealthy-monitoring-with-xen-altp2m/, Accessed: 2019-01-31. 2016.

[Len+14] Tamas K. Lengyel, Steve Maresca, Bryan D. Payne, George D. Webster, Sebastian Vogl, and Aggelos Kiayias. "Scalability, Fidelity and Stealth in the DRAKVUF Dynamic Malware Analysis System." In: *Proceedings of the 30th Annual Computer Security Applications Conference*. 2014.

[Len+12] Tamas K. Lengyel, Justin Neumann, Steve Maresca, Bryan D Payne, and Aggelos Kiayias. "Virtual Machine Introspection in a Hybrid Honeypot Architecture." In: *CSET*. 2012.

[LL09] Nan Li and Yiming Li. "A study of Inter-Domain communication mechanisms on Xen-Based hosting platforms." In: *Advanced Topics in Operating Systems* (2009).

[LK17] Tal Liberman and Eugene Kogan. *Lost in Transaction: Process Doppelgänging*. https://www.blackhat.com/docs/eu-17/materials/eu-17-Liberman-Lost-In-Transaction-Process-Doppelganging.pdf, Accessed: 2019-03-26. 2017.

[LRX09] Zhiqiang Lin, Ryan D. Riley, and Dongyan Xu. "Polymorphing Software by Randomizing Data Structure Layout." In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 107–126. ISBN: 978-3-642-02918-9.

[Lin] Linaro Limited. *Open Portable Trusted Execution Environment*. https://www.op-tee.org/, Accessed: 2019-01-31.

[LS01] Peter Loscocco and Stephen Smalley. "Integrating Flexible Support for Security Policies into the Linux Operating System." In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 29–42. ISBN: 1-880446-10-3. URL: http://dl.acm.org/citation.cfm?id=647054.715771.

[Ma18] Lele Ma. *TinyVMI In Progress*. https://tinyvmi.github.io/gsoc-blog/about/, Accessed: 2019-01-31. 2018.

[Mar07] Antonio Martin. "FireWire memory dump of a windows XP computer: a forensic approach." In: *Black Hat DC* (2007), pp. 1–13.

[MC12] Ben Martini and Kim-Kwang Raymond Choo. "An integrated conceptual digital forensic framework for cloud computing." In: *Digital Investigation* 9.2 (2012), pp. 71–80.

[MJ93] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." In: *USENIX winter*. Vol. 46. 1993.

[McM16] Stuart McMurray. *High-interaction MitM SSH honeypot*. https://github.com/magisterquis/sshhipot, Accessed: 2019-01-31. 2016.

[Men+18] Florian Menges, Fabian Böhm, Manfred Vielberth, Alexander Puchta, Benjamin Taubmann, Noëlle Rakotondravony, and Tobias Latzo. "Introducing DINGfest: An architecture for next generation SIEM systems." In: *Short Paper, GI Sicherheit*. 2018, pp. 257–260. ISBN: 978-3-88579-675-6. DOI: 10.18420/sicherheit2018_21.

[Mic] Michael Schierl. *jSSLKeyLog - Java Agent Library to log SSL session keys to a file for Wireshark*. http://jsslkeylog.sourceforge.net/, Accessed: 2019-01-31.

[Mil+11] David Miller, Shon Harris, Allen Harper, Stephen VanDyke, and Chris Blask. *Security information and event management (SIEM) implementation*. Network pro library. New York, NY: McGraw-Hill, 2011. ISBN: 9780071701099.

[Mir+17] Najmeh Miramirkhani, Mahathi Priya Appini, Nick Nikiforakis, and Michalis Polychronakis. “Spotless Sandboxes: Evading Malware Analysis Systems Using Wear-and-Tear Artifacts.” In: *IEEE Symposium on Security and Privacy (SP)*. May 2017, pp. 1009–1024. DOI: 10.1109/SP.2017.42.

[Mit] Mitmproxy Project. *mitmproxy*. https://mitmproxy.org/, Accessed: 2019-01-31.

[MT14] Asit More and Shashikala Tapaswi. “Virtual machine introspection: towards bridging the semantic gap.” In: *Journal of Cloud Computing* 3.1 (Oct. 2014), p. 16. ISSN: 2192-113X. DOI: 10.1186/s13677-014-0016-2. URL: https://doi.org/10.1186/s13677-014-0016-2.

[MFD11] Tilo Müller, Felix C. Freiling, and Andreas Dewald. “TRESOR Runs Encryption Securely Outside RAM.” In: *USENIX Security Symposium*. Vol. 17. 2011.

[MS13] Tilo Müller and Michael Spreitzenbarth. “Frost.” In: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, pp. 373–388.

[MTF12] Tilo Müller, Benjamin Taubmann, and Felix C. Freiling. “TreVisor.” In: *Applied Cryptography and Network Security*. Ed. by Feng Bao, Pierangela Samarati, and Jianying Zhou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 66–83. ISBN: 978-3-642-31284-7.

[OM13] Digit Oktavianto and Iqbal Muhardianto. *Cuckoo Malware Analysis*. Packt Publishing, 2013. ISBN: 1782169237, 9781782169239.

[Oos14] Michel Oosterhof. *Cowrie SSH/Telnet Honeypot*. https://github.com/micheloosterhof/cowrie, Accessed: 2019-01-31. 2014.

[Osb13] Grant Osbourne. *Memory Forensics: Review of Acquisition and Analysis Techniques*. Tech. rep. Defence science, technology organisation Edinburgh (Australia) Cyber, and electronic warfare div, 2013.

[Pal+01] Gary Palmer et al. “A road map for digital forensic research.” In: *First Digital Forensic Research Workshop, Utica, New York*. 2001, pp. 27–30.

[Pay12] Bryan D. Payne. *Simplifying virtual machine introspection using libvmi*. Sandia report, https://prod-ng.sandia.gov/techlib-noauth/access-control.cgi/2012/127818.pdf, Accessed: 2019-05-14. 2012.

[PAL07] Bryan D. Payne, Martim D. P. de A. Carbone, and Wenke Lee. “Secure and Flexible Monitoring of Virtual Machines.” In: *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*. Dec. 2007, pp. 385–397. DOI: 10.1109/ACSAC.2007.10.

[Pay+08] Bryan D. Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. “Lares: An architecture for secure active monitoring using virtualization.” In: *IEEE Symposium on Security and Privacy*. IEEE. 2008, pp. 233–247. DOI: 10.1109/SP.2008.24.

[Pen+13] Niels Penneman, Danielius Kudinskas, Alasdair Rawsthorne, Bjorn De Sutter, and Koen De Bosschere. “Formal Virtualization Requirements for the ARM Architecture.” In: *J. Syst. Archit.* 59.3 (Mar. 2013), pp. 144–154. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2013.02.003. URL: http://dx.doi.org/10.1016/j.sysarc.2013.02.003.

[PSL13] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. “Characterizing Hypervisor Vulnerabilities in Cloud Computing Servers.” In: *Proc. of the 2013 Int. Workshop on Security in Cloud Computing*. Cloud Computing ’13. Hangzhou, China: ACM, 2013, pp. 3–10. ISBN: 978-1-4503-2067-2. DOI: 10.1145/2484402.2484406.

[PSE11] Jonas Pfoh, Christian Schneider, and Claudia Eckert. “Nitro: Hardware-based system call tracing for virtual machines.” In: *International Workshop on Security*. Springer. 2011, pp. 96–112.

[PV17] Jonas Pfoh and Sebastian Vogl. *rVMI*. https://github.com/fireeye/rvmi, Accessed: 2019-01-31. 2017.

[Pie17] Piergiovanni Cipolloni. *Universal Android SSL Pinning bypass with Frida*. https://techblog.mediaservice.net/2017/07/universal-android-ssl-pinning-bypass-with-frida/, Accessed: 2019-01-31. 2017.

[PS19] Sandro Pinto and Nuno Santos. "Demystifying Arm TrustZone: A Comprehensive Survey." In: *ACM Computing Surveys (CSUR)* 51.6 (2019), p. 130.

[PMT13] Rainer Poisel, Erich Malzer, and Simon Tjoa. "Evidence and Cloud Computing: The Virtual Machine Introspection Approach." In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 4.1 (Mar. 2013), pp. 135–152.

[PG74] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures." In: *Communications of the ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: http://doi.acm.org/10.1145/361011.361073.

[RKK07] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. "Detecting System Emulators." English. In: *Information Security*. Vol. 4779. LNCS. Springer, 2007, pp. 1–18. ISBN: 978-3-540-75495-4.

[Raj+18] Sundaresan Rajasekaran, Harpreet Singh Chawla, Zhen Ni, Neel Shah, Emery Berger, and Timothy Wood. "CRIMES: Using Evidence to Secure the Cloud." In: *Proceedings of the 19th International Middleware Conference*. Middleware '18. Rennes, France: ACM, 2018, pp. 40–52. ISBN: 978-1-4503-5702-9. DOI: 10.1145/3274808.3274812. URL: http://doi.acm.org/10.1145/3274808.3274812.

[RKR17] Noëlle Rakotondravony, Johannes Köstler, and Hans P. Reiser. "Towards a Generic Architecture for Interactive Cost-Aware Visualization of Monitoring Data in Distributed Systems." In: *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*. SHCIS '17. Neuchâtel, Switzerland: ACM, 2017, pp. 25–30. ISBN: 978-1-4503-5271-0. DOI: 10.1145/3099012.3099017. URL: http://doi.acm.org/10.1145/3099012.3099017.

[Rak+17] Noëlle Rakotondravony, Benjamin Taubmann, Waseem Mandarawi, Eva Weishäupl, Peng Xu, Bojan Kolosnjaji, Mykolai Protsenko, Hermann de Meer, and Hans P. Reiser. "Classifying malware attacks in IaaS cloud environments." In: *Journal of Cloud Computing* 6.1 (Dec. 2017), p. 26. DOI: 10.1186/s13677-017-0098-8.

[Rav] Ole Ravnås. *Frida - A world-class dynamic instrumentation framework*. https://www.frida.re/, Accessed: 2019-01-31.

[Raz+16] Abbas Razaghpanah, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Christian Kreibich, Phillipa Gill, Mark Allman, and Vern Paxson. "Haystack: A Multi-Purpose Mobile Vantage Point in User Space." In: (2016). URL: http://arxiv.org/abs/1510.01419v3.

[RRK17] Hans P. Reiser, Noëlle Rakotondravony, and Johannes Köstler. *Mikromodul 8003: Grundlagen von Cloud-Forensik*. https://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/reiser/openc3s/CloudSecFor-MM-8003.pdf, Accessed: 2019-01-31. 2017.

[RI00] John Scott Robin and Cynthia E. Irvine. "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor." In: *Proceedings of the 9th USENIX Security Symposium*. 2000.

[RVJ09] Sandra Rueda, Hayawardh Vijayakumar, and Trent Jaeger. "Analysis of Virtual Machine System Policies." In: *Proceedings of the 14th ACM Symposium on Access Control Models and Technologies*. SACMAT '09. Stresa, Italy: ACM, 2009, pp. 227–236. ISBN: 978-1-60558-537-6. DOI: 10.1145/1542207.1542243. URL: http://doi.acm.org/10.1145/1542207.1542243.

[Sal+08] Joseph Salowey, Hao Zhou, Pasi Eronen, and Hannes Tschofenig. *Transport Layer Security (TLS) Session Resumption without Server-Side State*. https://tools.ietf.org/html/rfc5077. 2008.

[SBH15] John T. Saxon, Behzad Bordbar, and Keith Harrison. "Efficient Retrieval of Key Material for Inspecting Potentially Malicious Traffic in the Cloud." In: *2015 IEEE International Conference on Cloud Engineering*. Mar. 2015, pp. 155–164. DOI: 10.1109/IC2E.2015.26.

[STR18] Stewart Sentanoe, Benjamin Taubmann, and Hans P. Reiser. "Sarracenia: Enhancing the Performance and Stealthiness of SSH Honeypots Using Virtual Machine Introspection." In: *Nordic Conference on Secure IT Systems*. Springer. 2018, pp. 255–271. ISBN: 978-3-030-03638-6. DOI: 10.1007/978-3-030-03638-6_16.

[STR17] Stewart Sentanoe, Benjamin Taubmann, and Hans P. Reiser. "Virtual Machine Introspection Based SSH Honeypot." In: *Proceedings of the Workshop on Security in Highly Connected IT Systems*. Neuchâtel, Switzerland, 2017, pp. 13–18. ISBN: 978-1-4503-5271-0. DOI: 10.1145/3099012.3099016.

[SS99] Adi Shamir and Nicko van Someren. "Playing "Hide and Seek" with Stored Keys." In: *Proc. of the 3rd Int. Conf. on Financial Cryptography*. FC '99. London, UK, UK: Springer-Verlag, 1999, pp. 118–124. ISBN: 3-540-66362-2. URL: http://dl.acm.org/citation.cfm?id=647503.728464.

[Sha+09] Monirul I Sharif, Wenke Lee, Weidong Cui, and Andrea Lanzi. "Secure In-VM Monitoring Using Hardware Virtualization." In: *Proceedings of the 16th ACM conference on Computer and communications security*. ACM. 2009, pp. 477–487.

[Sha+14] Adrian L. Shaw, Behzad Bordbar, John Saxon, Keith Harrison, and Chris I. Dalton. "Forensic Virtual Machines: Dynamic Defence in the Cloud via Introspection." In: *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*. IEEE Computer Society, 2014, pp. 303–310. ISBN: 978-1-4799-3766-0. DOI: 10.1109/IC2E.2014.59. URL: http://dx.doi.org/10.1109/IC2E.2014.59.

[Shi+07] Hyun-Sup Shin, Kang-Ho Kim, Chei-Yol Kim, and Sung-In Jung. "The new approach for inter-communication between guest domains on Virtual Machine Monitor." In: *International Symposium on Computer and information sciences (ISCIS), 2007*. Nov. 2007, pp. 1–6. DOI: 10.1109/ISCIS.2007.4456875.

[Sic11] Bundesamt für Sicherheit in der Informationstechnik. *Leitfaden „IT-Forensik", Version 1.0.1 (März 2011)*. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Cyber-Sicherheit/Themen/Leitfaden_IT-Forensik.pdf?__blob=publicationFile&v=2, Accessed: 2019-01-31. 2011.

[Som+11] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. "All Your Clouds Are Belong to Us: Security Analysis of Cloud Management Interfaces." In: *ACM Workshop on Cloud Computing Security*. ACM, 2011, pp. 3–14. ISBN: 978-1-4503-1004-8. DOI: 10.1145/2046660.2046664.

[Sun+14] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. "TrustDump: Reliable Memory Acquisition on Smartphones." In: *Computer Security - ESORICS 2014*. Cham: Springer International Publishing, 2014, pp. 202–218. ISBN: 978-3-319-11203-9.

[Suo19] NetBSD/xen Kernel Interfaces Manual (Kimmo Suominen). *Xenbus – Xen bus abstraction for paravirtualized drivers*. http://netbsd.gw.com/cgi-bin/man-cgi?xenbus+4.i386+NetBSD-8.0, Accessed: 2019-04-30. 2019.

[Syl12] Joe Sylve. "Lime-linux memory extractor." In: *Proceedings of the 7th ShmooCon conference*. 2012.

[Syl+12] Joe Sylve, Andrew Case, Lodovico Marziale, and Golden G. Richard. "Acquisition and analysis of volatile memory from android devices." In: *Digital Investigation* 8.3 (2012), pp. 175–184. ISSN: 1742-2876. DOI: 10.1016/j.diin.2011.10.003. URL: http://www.sciencedirect.com/science/article/pii/S1742287611000879.

[TAR18] Benjamin Taubmann, Omar Alabduljaleel, and Hans P. Reiser. "DroidKex: Fast extraction of ephemeral TLS keys from the memory of Android apps." In: *Digital Investigation* 26 (2018), S67–S76. DOI: 10.1016/j.diin.2018.04.013.

[TBR19] Benjamin Taubmann, Alexander Böhm, and Hans P. Reiser. "TwinPorter – An Architecture For Enabling the Live Migration of VMI-based Monitored Virtual Machines." In: *Accepted for publication at TrustCom'19*. 2019.

[Tau+15a] Benjamin Taubmann, Dominik Dusold, Christoph Frädrich, and Hans P. Reiser. "Analysing malware attacks in the cloud: A use case for the TLSInspector toolkit." In: *Proceedings of the Workshop on Security in Highly Connected IT Systems*. Sept. 2015.

[Tau+16] Benjamin Taubmann, Christoph Frädrich, Dominik Dusold, and Hans P. Reiser. "TLSkex: Harnessing virtual machine introspection for decrypting TLS communication." In: *Digital Investigation* 16 (2016), pp. 114–123. DOI: 10.1016/j.diin.2016.01.014.

[Tau+15b] Benjamin Taubmann, Manuel Huber, Sascha Wessel, Lukas Heim, Hans P. Reiser, and Georg Sigl. "A lightweight framework for cold boot based forensics on mobile devices." In: *International Conference on Availability, Reliability and Security (ARES)*. Aug. 2015, pp. 120–128. DOI: 10.1109/ARES.2015.47.

[TK17] Benjamin Taubmann and Bojan Kolosnjaji. "Architecture for Resource-Aware VMI-based Cloud Malware Analysis." In: *Proceedings of the Workshop on Security in Highly Connected IT Systems*. Neuchâtel, Switzerland, 2017, pp. 43–48. ISBN: 978-1-4503-5271-0. DOI: 10.1145/3099012.3099015.

[TRR16] Benjamin Taubmann, Noëlle Rakotondravony, and Hans P. Reiser. "CloudPhylactor: Harnessing Mandatory Access Control for Virtual Machine Introspection in Cloud Data Centers." In: *IEEE Trustcom/BigDataSE/ISPA*. Aug. 2016, pp. 957–964. DOI: 10.1109/TrustCom.2016.0162.

[TR16] Benjamin Taubmann and Hans P. Reiser. "Secure Architecture for VMI-based Dynamic Malware Analysis in the Cloud." In: *DSN fast abstract*. 2016. URL: https://hal.archives-ouvertes.fr/hal-01316519.

[Tau+15c] Benjamin Taubmann, Hans P. Reiser, Thomas Kittel, Andreas Fischer, Waseem Mandarawi, and Hermann de Meer. "CloudIDEA: Cloud Intrusion Detection, Evidence preservation and Analysis." In: *EuroSys poster*. Apr. 2015. URL: http://eurosys2015.labri.fr/posters/p37.pdf.

[TCP] TCPDUMP. *TCPDUMP/LIBPCAP public repository*. http://www.tcpdump.org/, Accessed: 2019-01-31.

[Tes17] Joe Testa. *SSH man-in-the-middle tool*. https://github.com/jtesta/ssh-mitm, Accessed: 2019-01-31. 2017.

[Tuz+18] Tomasz Tuzel, Mark Bridgman, Joshua Zepf, Tamas K. Lengyel, and KJ Temkin. "Who watches the watcher? Detecting hypervisor introspection from unprivileged guests." In: *Digital Investigation* 26 (2018), S98–S106.

[Uhl+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C.M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. "Intel virtualization technology." In: *Computer* 38.5 (May 2005), pp. 48–56. ISSN: 0018-9162. DOI: 10.1109/MC.2005.163.

[Urb+14] David Urbina, Yufei Gu, Juan Caballero, and Zhiqiang Lin. "SigPath: A Memory Graph Based Approach for Program Data Introspection and Modification." In: *Computer Security - ESORICS 2014: 19th European Symposium on Research in Computer Security, Wroclaw, Poland, September 7-11, 2014. Proceedings, Part II.* Cham: Springer International Publishing, 2014, pp. 237–256. ISBN: 978-3-319-11212-1. DOI: 10.1007/978-3-319-11212-1_14.

[Van+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution." In: *Proceedings of the 27th USENIX Security Symposium.* USENIX Association, Aug. 2018.

[VMw10] Inc. VMware. *VProbes Programming Reference.* https://labs.vmware.com/vmtj/vprobes-deep-observability-into-the-esxi-hypervisor, Accessed: 2019-03-26. 2010.

[WG15] P. Wächter and M. Gruhn. "Practicability study of android volatile memory forensic research." In: *2015 IEEE International Workshop on Information Forensics and Security (WIFS).* Nov. 2015, pp. 1–6. DOI: 10.1109/WIFS.2015.7368601.

[Wei+18] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution." In: *Technical report* (2018).

[Win08] Johannes Winter. "Trusted Computing Building Blocks for Embedded Linux-based ARM Trustzone Platforms." In: *Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing.* STC '08. Alexandria, Virginia, USA: ACM, 2008, pp. 21–30. ISBN: 978-1-60558-295-5. DOI: 10.1145/1456455.1456460. URL: http://doi.acm.org/10.1145/1456455.1456460.

[Wir15] Wireshark contributors. *Wireshark Wiki about Secure Socket Layer (SSL).* https://wiki.wireshark.org/SSL, Accessed: 2019-01-31. 2015.

[Wu16] Peter Wu. *sslkeylog.* https://git.lekensteyn.nl/peter/wireshark-notes/tree/src/, Accessed: 2019-01-31. 2016.

[Xav16] Xavier de Carné de Carnavalet, Mohammad Mannan. "Killed by Proxy: Analyzing Client-end TLS Interception Software." In: *Network and Distributed Systems Symposium (NDSS'16).* 2016.

[Xu+17] Xiaoyan Xu, Bo Zhao, Xiaorui Wang, and Rongcai Zhao. "Research on Semantic Gap Problem of Virtual Machine." In: *Wireless Personal Communications* 97.4 (Dec. 2017), pp. 5983–6004. ISSN: 1572-834X. DOI: 10.1007/s11277-017-4823-x. URL: https://doi.org/10.1007/s11277-017-4823-x.

[Yal+17] S. D. Yalew, G. Q. Maguire, S. Haridi, and M. Correia. "T2Droid: A TrustZone-Based Dynamic Analyser for Android Applications." In: *2017 IEEE Trustcom/BigDataSE/ICESS.* Aug. 2017, pp. 240–247. DOI: 10.1109/Trustcom/BigDataSE/ICESS.2017.243.

[YC14] Fangzhou Yao and Roy H. Campbell. "CryptVMI: Encrypted Virtual Machine Introspection in the Cloud." In: *IEEE 7th International Conference on Cloud Computing.* June 2014, pp. 977–978. DOI: 10.1109/cloud.2014.149.

[ZR15] Julian Zach and Hans P. Reiser. "LiveCloudInspector: Towards Integrated IaaS Forensics in the Cloud." In: *Proceedings of Distributed Applications and Interoperable Systems: 15th IFIP WG 6.1 International Conference, DAIS 2015.* Cham: Springer International Publishing, 2015, pp. 207–220. ISBN: 978-3-319-19129-4. DOI: 10.1007/978-3-319-19129-4_17. URL: http://dx.doi.org/10.1007/978-3-319-19129-4_17.