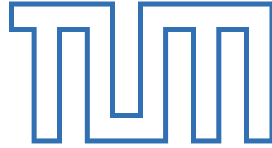


Fakultät für Informatik
der Technischen Universität München

Lehrstuhl für Sicherheit in der Informatik

Hardware-based Integrity Protection combined with Continuous User Verification in Virtualized Systems

Michael Velten

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Jörg Ott
Prüfer der Dissertation:	1. Prof. Dr. Claudia Eckert
	2. Prof. Dr. Georg Sigl

Die Dissertation wurde am 04.05.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.09.2017 angenommen.

Abstract

The ubiquity of computer systems, their application in security and privacy critical contexts like electronic voting, online shopping, or online banking, and the fact that these systems contain sensitive data pose security and privacy risks. There is a threat of malicious users and software, like viruses, worms, and trojans, attacking and exploiting vulnerabilities in order to compromise the computer systems and to spy on users or to manipulate data. Because mobile devices are often used to interact with these systems, another attack vector can be lost, stolen, or compromised devices used for gaining illegitimate access to the systems and to sensitive data hosted on the systems like account or credit card information. Such attacks constitute a threat to system integrity and data confidentiality and therefore ultimately to the confidence in the trustworthiness of the systems. Therefore, suitable protection and defense mechanisms are required.

In this work, we develop concepts and methods to ensure the continuous integrity of computer systems through a combination of virtualization techniques and a Trusted Platform Module (TPM). This includes the development of secure mechanisms for taking integrity measurements of software running inside virtual machines and for securely storing the obtained integrity measurements in a TPM. Furthermore, we explore how only legitimate users will be allowed to access the computer systems and the hosted user data by continuously authenticating all users interacting with the system.

We first show how to store and multiplex integrity measurements of arbitrarily many virtual machines in the TPM without lowering the security level provided by the TPM. A specially tailored attestation protocol enables us to prove the trustworthiness of individual virtual machines and to protect the privacy of other virtual machines by concealing sensitive data inherently required to be disclosed within the attestation process. We propose a virtualization-based architecture for securely taking integrity measurements from outside of the monitored virtual machines that effectively prevents attackers located in the virtual machines from manipulating the measurement process and the obtained measurements. The development of a flexible rule-based access control mechanism for protecting the integrity of virtual machines enables users to autonomously and securely install and update software packages while still preventing the installation of malicious or vulnerable software, and without requiring the intervention of the system administrator. Furthermore, we show how to continuously authenticate users of mobile devices—like smartphones and tablets—accessing and in-

teracting with the system by comparing the touchscreen interactions of the current user with the individual behavior patterns of the legitimate user. The proposed concept enables us to detect and disable compromised user accounts without requiring any special software, permissions, or provisioning on the device. We develop and evaluate prototypes to demonstrate the practicality of the proposed concepts and mechanisms.

Kurzfassung

Die Allgegenwärtigkeit von Computersystemen, deren Einsatz in sicherheits- und privatsphäre-kritischen Kontexten wie etwa Electronic Voting, Online Shopping oder Online Banking und die damit einhergehende Existenz sensibler Daten bergen besondere Risiken bezüglich der Sicherheit dieser Systeme und des Datenschutzes. Zum einen besteht die Gefahr, dass bösartige Nutzer und Malware, wie beispielsweise Viren, Würmer und Trojaner, durch die Ausnutzung von Schwachstellen der Computersysteme diese kompromittieren und so etwa Benutzer ausspionieren oder Daten manipulieren. Da der Zugriff auf die Systeme zudem oft über mobile Endgeräte erfolgt, stellen verlorene, gestohlene und kompromittierte Endgeräte einen weiteren potentiellen Angriffsvektor dar, um unberechtigten Zugriff auf die Systeme und auf dort vorhandene sensible Daten wie etwa Konto- und Kreditkarteninformationen zu erlangen. Solche Angriffe bedrohen somit die Integrität der Computersysteme, die Vertraulichkeit der dort befindlichen Nutzerdaten und damit letztlich das allgemeine Vertrauen der Benutzer in solche Systeme. Dementsprechend werden geeignete Absicherungs- und Verteidigungsmaßnahmen benötigt.

Gegenstand dieser Arbeit ist zum einen die Erforschung von Konzepten und Mechanismen zur kontinuierlichen Sicherstellung der Datenintegrität eines Systems unter dem Einsatz von Virtualisierungstechniken in Kombination mit einem Trusted Platform Module (TPM). Dies umfasst insbesondere die sichere Durchführung von Integritätsmessungen der sich in den virtuellen Maschinen befindlichen Komponenten und die sichere Verwahrung der entnommenen Integritätsmesswerte im TPM. Zum anderen wird untersucht, wie sichergestellt werden kann, dass nur legitime Benutzer Zugriff auf das System und auf die dort vorhandenen Nutzerdaten erhalten, indem kontinuierlich die Authentizität der mit dem System interagierenden Benutzer verifiziert wird.

Zunächst wird dazu gezeigt, wie man die aus beliebig vielen virtuellen Maschinen entnommenen Integritätsmesswerte so im TPM speichern und multiplexen kann, dass dies nicht zu einer Beeinträchtigung des Sicherheitsniveaus führt. Darauf aufbauend wird ein Attestationsprotokoll vorgestellt, das es erlaubt, die Vertrauenswürdigkeit einzelner virtueller Maschinen nachzuweisen, wobei inhärent offenzulegende Daten anderer virtueller Maschinen so verschleiert werden, dass keine kritischen Informationen preisgegeben werden. Zur sicheren Durchführung der Integritätsmessungen wird eine virtualisierungsbasierte Architektur entworfen, mit deren Hilfe die Integritätsmessungen von außerhalb der überwachten virtuellen Maschinen erfolgen

können, so dass sowohl Manipulationen des Messvorgangs als auch der Messwerte durch einen in den virtuellen Maschinen befindlichen Angreifer effektiv verhindert werden. Zusätzlich wird eine flexible regelbasierte Zugriffskontrolle zur Gewährleistung der Datenintegrität erarbeitet, die Benutzern von virtuellen Maschinen die autonome Installation und Aktualisierung von Software-Paketen ohne die Notwendigkeit der Intervention durch den Systemadministrator ermöglicht, dabei jedoch die Installation bösartiger und verwundbarer Software unterbindet. Des Weiteren wird untersucht, wie die Authentizität der mit dem System über mobile Endgeräte – wie Smartphones und Tablets – interagierenden Benutzer kontinuierlich verifiziert werden kann, indem die Interaktionen des aktuellen Benutzers mit dem Touchscreen des verwendeten mobilen Endgerätes mit den individuellen verhaltensbasierten Charakteristika des legitimen Benutzers abgeglichen werden. Das vorgestellte Konzept erlaubt die Erkennung und Deaktivierung kompromittierter Benutzerkonten, ohne dass es dazu spezieller Software, besonderer Berechtigungen oder einer vorherigen Provisionierung des Endgeräts bedarf. Die Praktikabilität und Effektivität der erarbeiteten Konzepte und Mechanismen werden durch prototypische Umsetzungen und deren Evaluation demonstriert.

Acknowledgments

First of all, I would like to thank Prof. Dr. Claudia Eckert for giving me the opportunity to pursue a PhD in the field of IT Security and for providing me with her supervision, support, and encouragement.

I would also like to thank Prof. Dr. Georg Sigl for his interest in my work and his contribution as a second examiner.

I am thankful to all the members of the departments Embedded Security and Trusted OS and Secure Operating Systems at the Fraunhofer Institute for Applied and Integrated Security. I would like to especially thank the following colleagues—in alphabetical order—for fruitful discussions, valuable feedback, and their support and companionship: Julian Horsch, Manuel Huber, Nisha Jacob, Prof. Dr. Dominik Merli, Dieter Schuster, Dr. Michael Weiß, Philipp Zieris.

I would also like to thank the former students Matthias Fischer and Peter Schneider who assisted me in my research.

Finally, I am grateful to my parents and the rest of my family for their endless support. This dissertation would not have been possible without them.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement 4
1.3 Contributions 6
1.4 Thesis Outline 9

2 Background **11**
2.1 Trusted Computing 11
2.1.1 Integrity Measurement 13
2.1.2 Remote Attestation 14
2.2 Virtualization 15
2.2.1 Full Virtualization 15
2.2.2 Paravirtualization 16
2.2.3 Operating-System-Level Virtualization 17
2.3 Distributed Filesystems 18
2.3.1 9P Filesystem 18
2.4 Machine Learning 20
2.4.1 Classification 20

3 Multiplexing TPM Integrity Measurements **23**
3.1 TPM Virtualization Challenges 24
3.2 Requirements Analysis 26
3.3 Attacker Model and Assumptions 27
3.4 System Overview 28
3.5 Multiplexed Storage and Attestation 30
3.6 Measurement Concealment 31
3.6.1 Multiplexed Measurement List 33
3.7 Integrity Reporting 34
3.8 Integrity Validation 35
3.9 Security Analysis 38

3.9.1 Discarding Measurements 39
3.9.2 Substituting Measurements 39
3.9.3 Substituting VM-IDs 39
3.9.4 Blinding Measurements and VM-IDs 39
3.10 Prototype Implementation 42
3.11 Performance Evaluation 43
3.12 Related Work 45
3.13 Summary 46

4 Integrity Monitoring using Paravirtualized Filesystems 49
4.1 Virtualization-based Integrity Monitoring 50
4.2 Attacker Model and Assumptions 52
4.3 Monitoring of Guest VMs 52
4.3.1 Filesystem Relocation Mechanism 53
4.3.2 Relocation Scenarios 53
4.4 System Overview 55
4.4.1 File Operation Monitor 56
4.4.2 Execution Detection Engine 56
4.4.3 Package Maintenance Engine 57
4.4.4 File Protection Enforcer 57
4.5 Monitoring and Analyzing File Operation Requests 57
4.5.1 Shadow Copy Write 58
4.6 Secure Storage of Integrity Measurements 59
4.7 Enforcing File Protection 61
4.7.1 Policy Predicates and Request Mapping 61
4.7.2 Package Policy Rules 63
4.7.3 Policy Example 63
4.8 Detecting Program Execution 65
4.9 Autonomous Software Package Installation and Upgrade . . . 66
4.9.1 Signaling of Package Maintenance Request 67
4.9.2 Checking Package Integrity and Permissions 67
4.9.3 Executing Package Maintenance Request 68
4.9.4 CPVM Rationale 68
4.10 Prototype Implementation 69
4.10.1 Installation and Upgrading of Packages 70
4.11 Performance Evaluation 70
4.11.1 Network-based Filesystem Relocation 73
4.12 Security Analysis 75
4.12.1 Persistent Malware 75
4.12.2 Fileless Malware 76

4.12.3 Persistent File Manipulations 76
4.12.4 Software Package Manipulations 76
4.12.5 Non-Persistent Manipulations 77
4.13 Related Work 77
4.14 Summary 79

5 Continuous Authentication using Touchscreen Dynamics 81
5.1 Behavioral Biometrics for Authentication 83
5.2 Attacker Model and Assumptions 85
5.3 Touch Interaction Selection in Web Contexts 85
5.3.1 Touchscreen Gestures 86
5.3.2 Device Sensor Data 87
5.4 System Overview 88
5.5 Touch Behavior Model Training 88
5.6 User Identity Verification 89
5.7 Feature Extraction 91
5.7.1 Path Offsets 91
5.7.2 Bounding Box 91
5.7.3 Raster 92
5.7.4 Velocity 92
5.7.5 Curvature 92
5.7.6 Acceleration 92
5.8 Verification Strategy 93
5.8.1 Subsequence Processing 93
5.8.2 Confidence Value Calculation 93
5.9 Framework Implementation 94
5.10 Classification Evaluation 96
5.10.1 Feature Suitability 97
5.10.2 Classification Accuracy 98
5.11 Performance Evaluation 101
5.11.1 CPU Usage 102
5.11.2 Battery Consumption 105
5.11.3 Network Traffic Generation 108
5.12 Security Analysis 111
5.12.1 Blocking Attack 111
5.12.2 Imitation Attack 112
5.12.3 Replay Attack 112
5.13 Related Work 114
5.14 Summary 116

6 Conclusion and Future Work **119**
6.1 Contributions 120
6.2 Future Research 122

Bibliography **125**

List of Acronyms **145**

List of Figures **150**

List of Tables **151**

Chapter 1

Introduction

In this chapter, we introduce and motivate the topic of this thesis. We give the problem statement, describe the main challenges, and list our contributions. Furthermore, we give an outline of the thesis.

1.1 Motivation

Computer systems have become ubiquitous and are used in a wide variety of scenarios like cloud computing, online shopping, social media, electronic voting, and online banking. The fact that these systems perform security and privacy critical operations and often host large numbers of user accounts containing sensitive data makes them a prime target for computer criminals trying to compromise and exploit them [103, 187, 49, 140, 20]. Malicious users and malware, like viruses, worms, and trojans, located on these systems will try to attack and exploit vulnerabilities in order to compromise the systems, gain illegitimate access to user accounts, and spy on users or manipulate data [66, 26, 21]. Because computer systems often act as servers providing functionality to clients over communication networks—most notably the internet—they are also exposed to network attacks trying to exploit vulnerabilities of provided services [75, 114]. To this end, attackers may also try to launch indirect attacks in order to compromise the accounts hosted on the server. In this context, attackers first exploit lost, stolen, and compromised mobile devices of users having legitimate access to the server's user accounts [37, 60, 7]. After successfully hijacking those user accounts, the attacker may then, for example, manipulate or steal data of cloud computing users [72], transfer money to the attacker's banking account [126], or manipulate electronic votes [88]. Depending on the privilege level of the hijacked account, the attacker may even be able to compromise

other user accounts and other parts of the system. For example, a hacked system administrator account is likely to entail a fully compromised system.

Therefore, it is crucial to ensure both the system integrity of the computer systems and the authenticity of users interacting with them. There exist various mechanisms and techniques to protect computer systems and to defend against attacks [43, 155, 184, 78, 35]. A well-known approach for improving system security is the utilization of a Trusted Platform Module (TPM) [173, 174] as specified by the Trusted Computing Group (TCG) [172]. A TPM is an implementation of a secure cryptoprocessor acting as a cornerstone for securely performing crucial cryptographic operations. This allows securing computer systems and proving their integrity as part of a *remote attestation* to third parties. Nowadays, a large majority of server systems, personal computers, and notebooks ship with a TPM [3]. Prominent use cases include Microsoft's BitLocker for full disk encryption [55] and the versatile utilization in Google Chromebooks [48]. In particular, the TPM can be used to securely store integrity measurements in special hardware-protected registers called Platform Configuration Registers (PCRs) that reflect a system's configuration. This can be utilized to keep track of the system's integrity starting with the initial startup of the system by measuring each component in the system's booting sequence, before the component is actually being executed, and storing the corresponding measurements in PCRs. The integrity measurements can then be used in the course of a remote attestation to prove to a remote party that the system platform is in a trusted state. Even in the event of an attacker compromising the computer system, it is impossible to manipulate integrity measurements already stored in the TPM.

The popular Integrity Measurement Architecture (IMA) [148] extends this *chain of trust* to the application layer by measuring programs executed in the operating system and storing the measurements in a PCR of the TPM. A common disadvantage of IMA and similar measurement agents is that they have no strong isolation from the measurement target (i.e., the measured operating system). Therefore, an attacker gaining sufficient privileges (e.g., through runtime attacks targeting the operating system kernel) may be able to tamper with the measurement agent such that measurements will not be taken correctly anymore. Consequently, attestations issued by the system about its integrity cannot be trusted.

A concept to mitigate the aforementioned problems and to isolate the measurement agent (e.g., IMA) from the measurement target (the operating system) is virtualization [97, 61]. In this case, the measurement target is placed inside a Virtual Machine (VM) and the measurement agent is located

outside of the VM, thus being able to monitor the target from "outside of the box". Using this approach, it is possible to securely take integrity measurements and store them in the TPM even in the event of a completely compromised VM. Virtualization also allows running and monitoring multiple measurement targets on a single physical computer system. This is especially relevant in scenarios like cloud computing where there usually exists a multitude of VMs. Furthermore, virtualization allows supervising guest operating systems and interposing and preventing critical operations (e.g., write operations on crucial files) [117, 61]. These mechanisms can be used to better defend against malware—like viruses, worms, and trojans—located in the VMs [79, 80, 41, 81].

However, a problem with using both virtualization and TPM functionality is that currently it is not possible to combine them without lowering the TPM security level. In particular, the TPM was not designed to store integrity measurements of multiple VMs. There exist approaches to solve this problem by introducing virtual TPMs [9, 89, 101, 46]. However, these approaches do not provide the same level of security as a hardware TPM because they emulate PCRs in software. On a compromised system, these PCRs can be manipulated by an attacker, allowing him to forge remote attestations. Therefore, in order to improve the security of such systems, a mechanism is required to securely store and multiplex integrity measurements from multiple VMs directly in the hardware TPM. Another difficulty arises from the fact that the virtualization layer introduces a semantic gap [61] resulting in external monitoring components having a rather abstract and less detailed view of the supervised guest operating system. Hooks placed inside the monitored VMs can help to better cope with the complexity of interpreting guest operating system specific structures and events and to gain a more detailed view. However, this poses the risk of attackers tampering with the hooks [4]. Hence, it is also necessary to have a mechanism that allows for proper and sufficiently detailed monitoring of VMs where an attacker cannot circumvent the monitoring process by tampering with the hooks.

The proper utilization of both virtualization and TPM functionality, as described above, allows us to secure and protect the integrity of guest operating systems and to defend against attackers and malware located in VMs. However, such virtualized systems typically contain a user class with special privileges—system administrators—who are not confined to VMs but are able to make system-wide modifications. Because these systems usually allow remote access, a system administrator can take advantage of portable devices like smartphones to access and maintain the system

from virtually everywhere. However, carrying around these devices amplifies the risk of loss or theft, thus increasing the threat of attackers hijacking critical system administrator accounts and potentially compromising the entire system [37, 7]. Therefore, in order to further improve the overall system security, it is necessary to continuously verify the identities of all users accessing and interacting with the system and to detect and disable hacked accounts [144].

In this thesis, we will explore novel ways of improving system security based on a combination of virtualization, the utilization of a TPM, and continuous user authentication. Virtualization allows us to monitor critical operations and to take integrity measurements of multiple, isolated VMs from outside of the VMs. By leveraging a hardware TPM to store and multiplex the obtained measurements, we can take full advantage of the TPM's hardware-based security guarantees. In addition, continuous user authentication enables us to detect and disable compromised accounts, thus further enhancing the overall system security.

1.2 Problem Statement

In the following, we describe the main challenges we have to solve in order to realize a system that successfully combines virtualization, the utilization of a TPM, and continuous user authentication in order to improve a system's overall security.

Securely storing measurements of multiple VMs. The utilization of a TPM can improve system security. However, the TPM was not designed to be used directly in virtualized environments. In particular, it is not possible to store integrity measurements on a per-VM basis because there is only a rather small number of registers (PCR) for storing measurements but a potentially large number of VMs. Virtual TPMs hold measurements in software-emulated PCRs to solve this problem. However, this makes them more susceptible to attacks and therefore does not provide the same level of security as a hardware TPM. Consequently, to improve the security of such systems, a mechanism is required that allows us to securely store integrity measurements from multiple VMs in the hardware TPM. The challenge is to share and multiplex measurements from n VMs in $m < n$ available PCRs such that there exists an integrity-protected mapping between each measurement and its respective VM that cannot be manipulated by an attacker (e.g., hiding malicious programs by mapping their measurements to other VMs).

Attesting the integrity of individual VMs. The integrity measurements can be used in the course of a remote attestation to prove to a remote party that the system platform is in a trusted state. However, the TPM inherently requires the disclosure of all measurements of all VMs that share PCRs with the attested VM. The reason is that the measurements are stored as a hash chain in the PCRs, and all elements of the hash chain are required to verify and guarantee the integrity of the measurement values. Therefore, to protect the privacy of other VMs, a novel mechanism is required that allows for attesting individual VMs without disclosing measurements of other VMs that share the same PCRs. Furthermore, it must not be possible for an attacker to hide measurements or map them to a different VM.

Taking reliable measurements. A measurement agent takes integrity measurements of crucial system components and stores them in the TPM. Virtualization can prevent attackers from tampering with the measurement agent. In this case, the measurement target is isolated in a VM and the measurement agent is located outside of the VM, thus being able to supervise the target from "outside of the box". However, the external measurement agent now only has a rather coarse and abstract view of the supervised operating system's internal structures. The challenge is to bridge this semantic gap [61] in such a way that the measurement agent is provided with sufficient information about all relevant operations occurring within the guest operating system in order to take adequate integrity measurements. An attacker must not be able to tamper with hooks placed inside of a VM in order to hide critical file operations.

Monitoring file operations using flexible policies. The described virtualized monitoring techniques not only allow to take measurements from outside of VMs. They also enable external monitoring components to interpose on critical file operations and to prevent them. Policy-based access control mechanisms can prohibit all filesystem operations within a VM unless an operation is explicitly granted by a policy rule. While such approaches can significantly increase the security of the monitored system, they may also heavily restrict what a legitimate user is allowed to do. In particular, they may result in an inflexible system where it is impossible for legitimate users to autonomously install or upgrade software without the need of manual intervention by the administrator of the physical system. The challenge is to develop a mechanism that allows us to monitor and prevent illegal file operations based on a policy-based approach. An attacker must not be able to evade the monitoring and manipulate files without the monitoring com-

ponent noticing it. However, at the same time, regular users of VMs should be able to autonomously install, remove, upgrade, and downgrade software packages in a secure and controlled manner.

Detecting compromised accounts. System administrator accounts and other accounts on a system can be accessed through portable devices like smartphones. However, this allows attackers to hijack accounts through the device, sometimes even without having to enter a password or PIN because having physical access to the device often entails direct access to several accounts where the user is still logged into. A possible countermeasure is to take advantage of behavioral biometrics. In this context, the user identity can be continuously verified based on individual interaction behavior (e.g., interactions with the touchscreen of a smartphone) caused by physical differences between users, varying habits, and personal preferences. The challenge is to find ways to enable the server to deploy such continuous user authentication on the fly without requiring any special software, prior setup, or special privileges on the user's device as this simplifies deployment significantly. Furthermore, the user verification process should be as unobtrusive and transparent for the user as possible. However, the above constraints can result in limited access to behavioral biometrics data and a lower degree of precision. Therefore, another challenge is to find a selection of features that still allow for successful user classification under these conditions.

1.3 Contributions

In this thesis, we explore novel mechanisms to multiplex integrity measurements originating from arbitrarily many VMs in a secure and privacy-aware manner in a hardware TPM, thus achieving a higher level of security compared to existing approaches emulating PCRs in software. We take advantage of virtualization to develop a system that enables us to monitor multiple operating systems from "outside of the box", take integrity measurements on a per-VM basis and securely store them in a hardware TPM, and to detect and prevent critical file operations through policy-based access control mechanisms. Finally, we show new methods to utilize continuous user authentication in order to better protect user accounts such that lost, stolen, or compromised user devices do not lead to compromised user accounts and systems.

Figure 1.1 gives a high-level view of the main components explored in this thesis and how they relate to each other. The three main components

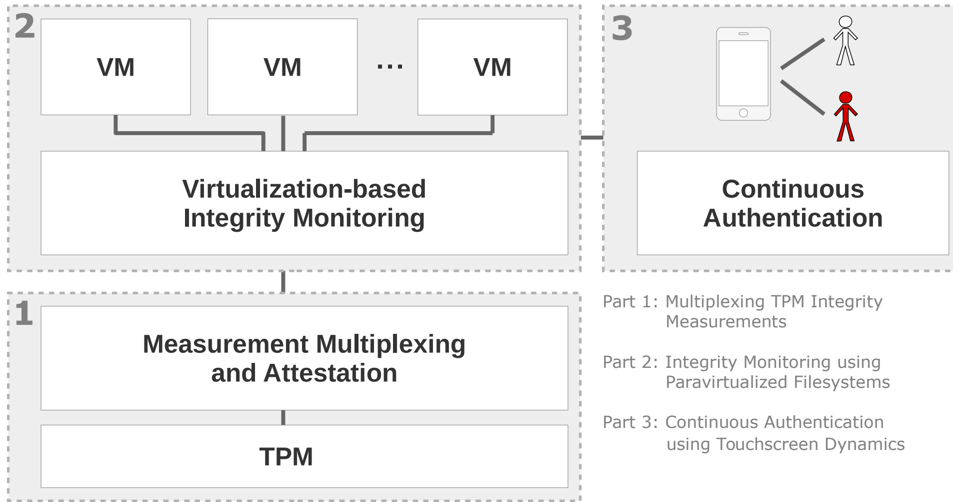


Figure 1.1: High-level view of the main components explored in this thesis.

correspond to the work done in Chapters 3 to 5. In each chapter, we contribute to the existing research. The conceptual work is complemented by implementing and evaluating proof of concepts to demonstrate the practicality of our proposed approaches.

In the following, we list the main contributions of this thesis.

Contribution 1. We show how to multiplex integrity measurements originating from arbitrarily many VMs with just a single standard TPM and only requiring one PCR. In contrast to previous work emulating PCRs in software, our approach achieves a higher level of security since measurements are always stored in the hardware-protected PCRs of the TPM. We implement a proof of concept to take and multiplex integrity measurements and evaluate its performance for different numbers of virtual machines.

Contribution 2. We develop a secure and privacy-aware remote attestation protocol for attesting the integrity of individual VMs. In this context, we present a novel approach for handling concealed integrity measurements that enables us to disclose only selected measurements of the hash chain stored in a shared PCR without degrading the TPM's security guarantees. However, this approach poses the risk of an attacker manipulating the attestation protocol through exploitation of the newly introduced concealment capabilities in order to hide mea-

surements. We give an exhaustive list of possible attacks and show how our remote attestation protocol protects against them.

Contribution 3. We present a virtualization-based architecture that allows for "outside of the box" integrity monitoring by relocating a supervised VM's entire filesystem into the isolated realm of the host such that all file operations must necessarily be routed through the hypervisor-level. This efficiently bridges the semantic gap and—in contrast to existing monitoring approaches—has the advantage that hooks placed inside the VMs are not prone to manipulation by malware because disabling hooks inevitably renders the VM incapable of accessing or manipulating its own filesystem. We implement and evaluate a proof of concept using a minimalist and lightweight virtualization solution.

Contribution 4. We take advantage of the developed virtualization-based architecture and present a policy-based access control mechanism for enforcing file protection. We solve the problem of too restricting and inflexible policy rules by enabling regular VM users to autonomously install, remove, upgrade, and downgrade software packages. Our approach contributes to the security of the VMs by preventing both regular users and malicious users from installing or downgrading software versions known to contain vulnerabilities.

Contribution 5. We show how to continuously authenticate users based on user interaction behavior with smartphone touchscreens. Our solution is widely applicable on everyday smartphones and, in contrast to other work, does not require any special software, prior setup, or special privileges on the user's smartphone. We rather take advantage of standard mobile web browser capabilities to remotely capture and analyze touchscreen interactions in order to continuously verify user identities. In contrast to existing work, we do not have direct access to the API of the touch device's operating system. This means that touch interaction data, proven to be beneficial for user classification, has a lower degree of precision and some data cannot be obtained at all. We provide a selection of features that still allow for successful user classification under these conditions. We implement a proof of concept and evaluate the user classification accuracy as well as its overhead to assess the practicality of our approach.

1.4 Thesis Outline

The remainder of this thesis is organized as follows.

Background. Chapter 2 provides the necessary background on Trusted Computing, virtualization, distributed filesystems, and machine learning. Furthermore, it introduces the terminology used throughout this thesis.

Multiplexing TPM Integrity Measurements. In Chapter 3, we introduce our approach for secure and privacy-aware multiplexing of hardware-protected TPM integrity measurements within virtualized environments. In particular, we show how to multiplex integrity measurements of arbitrarily many virtual machines with just a single TPM. Furthermore, we develop a privacy-aware remote attestation protocol for proving the integrity of individual virtual machines. We construct the protocol in such a way that no integrity measurements and no other sensitive information of other virtual machines is disclosed.

Integrity Monitoring using Paravirtualized Filesystems. In Chapter 4, we build upon the above work and develop a system for monitoring the filesystem of multiple virtual machines from "outside of the box" and store the so-obtained integrity measurements in the multiplexed TPM. In particular, we present our approach of relocating a supervised virtual machine's entire filesystem into the isolated realm of the host. In this way, we can enforce that all file operations on the virtual machine's filesystem must necessarily be routed through the hypervisor-level, and thus can be tracked and even be prevented. This enables us to secure and protect the integrity of virtual machines and to defend against attackers and malware—like viruses, worms, and trojans—located in the virtual machines.

Continuous Authentication using Touchscreen Dynamics. In Chapter 5, we develop a framework that allows for secure interaction with the above system. In particular, we protect the system's user accounts by continuously verifying user identities based on user interaction behavior with smartphone touchscreens. This enables us to disable critical functionality and to enforce a reauthentication in case of suspicious behavior. Our approach is completely transparent for the user and works on everyday smartphones without requiring any special software or privileges on the user's device. We show how to

successfully classify users even on the basis of limited and imprecise touch interaction data.

Conclusion and Future Work. Chapter 6 concludes this thesis and provides directions for future research.

Chapter 2

Background

In this chapter, we present the necessary background on Trusted Computing, virtualization, distributed filesystems, and machine learning. Furthermore, we introduce the terminology used throughout this thesis. The background information on Trusted Computing and virtualization is primarily required for Chapters 3 and 4, the background information on distributed file systems for Chapter 4, and the background information on machine learning for Chapter 5.

2.1 Trusted Computing

Trusted Computing is a technology developed by the Trusted Computing Group (TCG) allowing to protect computing infrastructure and end points based on a hardware root of trust [172]. The core component is the Trusted Platform Module (TPM) [173, 174]. A TPM is a cryptographic coprocessor that is present on most commercial PCs and servers [3]. A TPM is similar to a smart card, however, an important difference between a smart card and a TPM is that the latter is bound to a specific platform [47]. The TPM provides several security functionalities like random number generation, hashing, secure key generation, signing, and encryption. The provided functionality and the supported cryptographic algorithms differ between a TPM 1.2 [173] and the latest TPM 2.0 [174], the latter of which is sometimes called next generation Trusted Platform Module [132]. Table 2.1 gives an overview of some key differences between a TPM 1.2 and a TPM 2.0.

An essential cryptographic key of both TPM 1.2 and TPM 2.0 is the Endorsement Key (EK). The EK makes it possible to identify a particular TPM by utilizing asymmetric cryptography. In particular, the public portion of the EK key pair can be used to verify that a given TPM is in possession of

	TPM 1.2	**TPM 2.0**
Endorsement Key	*fixed:* single RSA 2048	*flexible:* multiple
Asymmetric Algs	*fixed:* RSA	*flexible:* RSA, ECC
Symmetric Algs	*optional:* AES	AES
Authorization	HMAC, PCR, PP, locality	PW, HMAC, EA
Hash Functions	*fixed:* SHA-1	*flexible:* SHA-1|256
Number of PCRs	minimum of 16	depends on platform

Table 2.1: Comparison of some key features of a TPM 1.2 and a TPM 2.0. PP=Physical Presence, PW=Password, EA=Enhanced Authorization.

the corresponding, matching private portion. The private portion of the EK key pair never leaves the TPM. The TPM 1.2 supports only an RSA 2048-bit EK key pair, whereas the flexible *algorithm agility* approach of the TPM 2.0 allows for various asymmetric key types. Furthermore, with TPM 2.0 it is possible to have multiple EKs instead of only one. In this case, all TPM 2.0 EKs can be created based on a single persistent, randomly generated Endorsement Primary Seed (EPS). This prevents the necessity of having to store all key pairs in the rather scarce amount of available TPM non-volatile memory.

TPM 1.2 supports only RSA as its asymmetric cryptographic algorithm, whereas TPM 2.0 supports both RSA and Elliptic Curve Cryptography (ECC). Regarding symmetric cryptographic algorithms, the TPM 1.2 specification states that the Advanced Encryption Standard (AES) may be supported but that TPM 1.2 does not expose any of the symmetric operations for general message encryption [173]. In contrast, TPM 2.0 does expose AES for general message encryption.

TPM 1.2 offers several means for authorization based on Hashed Message Authentication Code (HMAC), the state of PCRs (cf. Section 2.1.1), the assertion of physical presence, and proof of locality by indicating a command is coming from a trusted process. The TPM 2.0 offers password-based authorization, HMAC, and a powerful policy-based authorization mechanism called Enhanced Authorization (EA) which includes TPM 1.2 features: 1.2 HMAC, PCR, physical presence, and locality.

The TPM 1.2 specification is restricted to the hash function SHA-1 (160-bit). This restriction is getting increasingly critical. Theoretical attacks on SHA-1 have been known since 2005, and SHA-1 was officially deprecated by NIST in 2011 [44]. In 2017, security researchers at CWI Amsterdam and Google have announced the first practical collision attack on SHA-1 [165]. In contrast, the algorithm agility approach of the TPM 2.0 specification

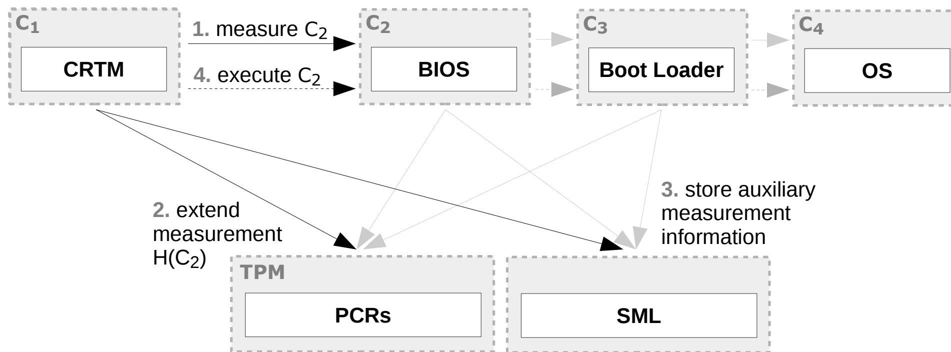


Figure 2.1: Authenticated boot sequence with components C_1 to C_4 involved in establishing a transitive chain of trust. Measurements are extended in the TPM and auxiliary information is stored outside of the TPM.

allows virtually any hash function (e.g., SHA-256), thus protecting against the above collision attack.

Finally, the TPM 1.2 specification [173] guarantees a minimum of 16 PCRs, whereas the platform-specific specifications of the TPM 2.0 range from a minimum of one PCR for TCG TPM 2.0 Automotive Thin Profile [175] to a minimum of 24 PCRs[1] for the TPM 2.0 TCG PC Client Platform TPM Profile (PTP) specification [176].

2.1.1 Integrity Measurement

An *authenticated boot* is used to establish a transitive *chain of trust* by measuring each component in the booting sequence, starting with an inherently trusted component called the Core Root of Trust for Measurement (CRTM). The sequence of steps involved in an authenticated boot is depicted in Figure 2.1. Each component C_i first *measures* the next component C_{i+1} in the booting sequence by applying a cryptographic hash function H to the executable code of C_{i+1} before control is eventually passed to C_{i+1}. The obtained *integrity measurement* $m := H(C_{i+1})$ will be stored in a shielded location of the TPM's volatile memory[2] called Platform Configuration Register (PCR). A PCR may hold arbitrarily many measurements by storing the measurements as a hash chain. The TPM only allows adding measurements to a hash chain through a special *extend* operation. The ex-

[1] Optionally, a TPM 2.0 may also support additional banks of PCRs.

[2] The TPM 2.0 Library Specification [174] also allows TPM vendors to implement PCRs in non-volatile memory.

tend operation for a measurement m to a PCR with index i is defined as: $PCR[i] \leftarrow H(PCR[i] \mid\mid m)$ (where $\mid\mid$ denotes concatenation).[3] Note that the extend operation does not change the size of the hash chain value stored in the PCR. Furthermore, the extend operation preserves the chronological ordering of added measurements. The TPM does not allow to remove a once extended measurement from a PCR, nor does it allow to reset a PCR without restarting the entire system (platform power-up). In addition to the extend operations, all measurements—along with auxiliary information—will be stored in an ordered list (outside of the TPM) called Stored Measurement Log (SML) or TCG event log. The log keeps track of all measurements used for the extend operations and allows recalculating the hash chain value of a PCR. Eventually, control is given to the next component C_{i+1} and the steps will be repeated. The established chain of trust may extend up to the operating system and application layer by taking and extending integrity measurements of operating system components, executables and other files as well as configuration data.

2.1.2 Remote Attestation

A *remote attestation* allows a system to prove its trustworthiness to another (remote) party by leveraging the attesting system's TPM. The attesting platform is called *prover*. The party requesting the attestation is called *verifier*. The remote attestation procedure can be divided in two phases: *integrity reporting* and *integrity validation*. In the integrity reporting phase, the prover sends the SML to the verifier. This allows the verifier to inspect a list of all components running on the attesting platform. In addition, the prover transmits the contents of the relevant PCRs by requesting a corresponding *quote* from the TPM. This means that the TPM uses a special signing key called Attestation Identity Key (AIK) (TPM 1.2) or Attestation Key (AK) (TPM 2.0) to vouch for the authenticity and integrity of the transmitted PCR contents. The AIK and AK, respectively, are used instead of the TPM's EK for reasons of privacy. Further privacy-aware techniques include Direct Anonymous Attestation (DAA) [18] and property-based attestation [69, 24, 146]. In the integrity validation phase, the verifier verifies the signature and calculates the hash chain values based on the SML. If the calculated hash chain values match the (signed) PCR values, the verifier can be sure that the SML is untampered. Finally, the verifier determines whether the components contained in the SML represent a trustworthy system.

[3] On platform power-up, all PCRs are initially set to zero.

2.2 Virtualization

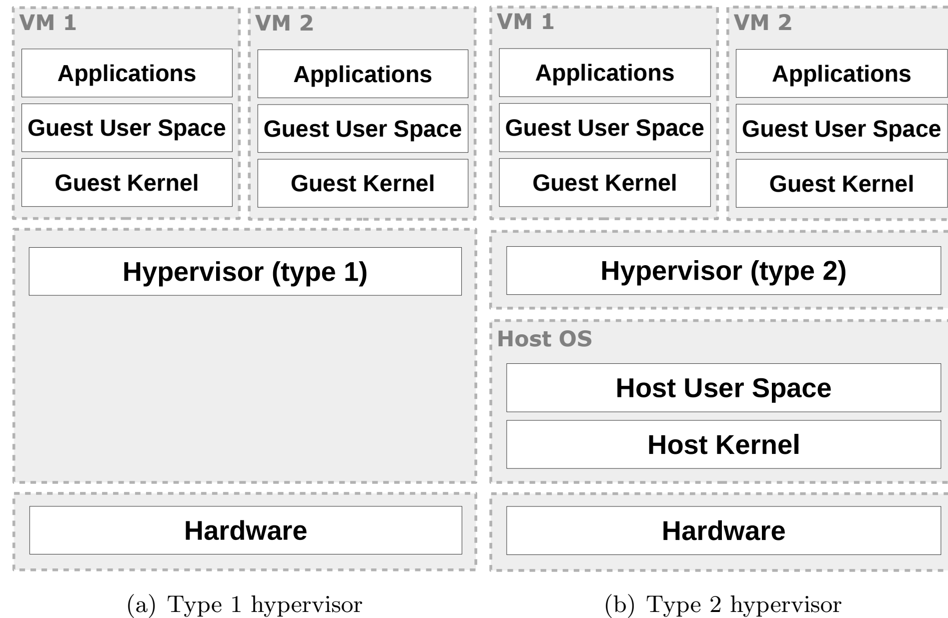
Virtualization allows creating special environments called Virtual Machines (VMs) on a physical machine and maintaining hardware resources for these VMs such that it is possible to run (different) operating systems inside of them [129, 164]. The virtualization is realized by a piece of software called *hypervisor* or Virtual Machine Monitor (VMM). The hypervisor has complete control of the machine's hardware resources—in particular, the CPU, memory, and I/O devices—and is able to regulate which VM is allowed to access which hardware resources. In this way, the hypervisor decouples the software from the hardware by forming a level of indirection between the software running in a VM and the hardware [142]. The physical machine running the hypervisor is called the *host machine* and the VMs are called the *guest machines*. An operating system running inside of a VM is called a *guest operating system*.

The hypervisor provides strong *isolation* by preventing a VM from accessing or manipulating the software running in the hypervisor or in a separate VM [61]. Isolation in combination with the hypervisor's ability to inspect the state of VMs and to interpose on certain operations makes virtualization an interesting technique w.r.t. system security. In this context, inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it is called Virtual Machine Introspection (VMI) [61]. The inherent loss of certain semantic information due to the rather coarse and abstract view of a virtual machine's state from the outside is called the *semantic gap* [25].

2.2.1 Full Virtualization

Full virtualization allows running unmodified operating systems by having the hypervisor simulate the underlying hardware. This is realized by the hypervisor's ability to *trap* instructions executed by the guest operating system and to emulate the desired behavior. The efficiency of full virtualization can be improved by leveraging hardware-assisted virtualization.

Hypervisors are often classified into *type 1* and *type 2* hypervisors. Both hypervisor types are shown in Figure 2.2. A type 1 hypervisor or *bare-metal* hypervisor runs directly on the host machine's hardware to manage guest operating systems. In this case, the hypervisor runs in the CPU's *kernel mode* and the guest operating system runs in the CPU's *user mode*. A type 2 hypervisor or *hosted* hypervisor runs on a regular operating system called *host operating system* and, in turn, spawns guest operating systems



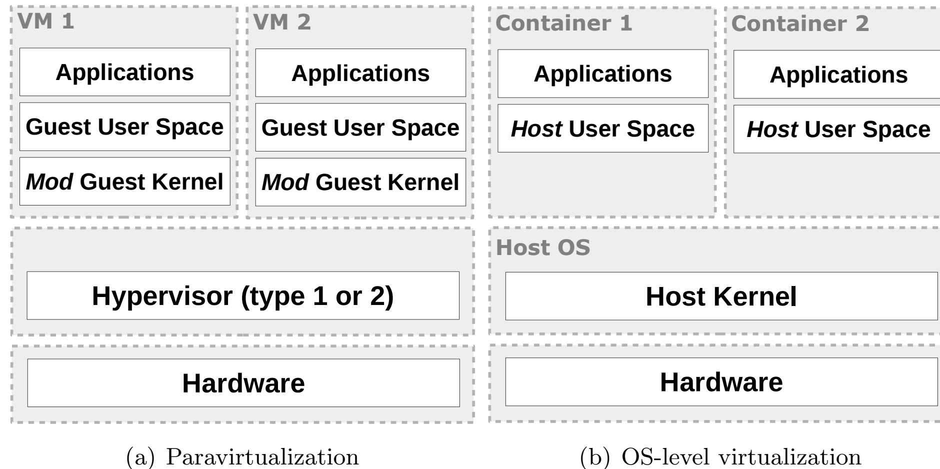
(a) Type 1 hypervisor (b) Type 2 hypervisor

Figure 2.2: Comparison of type 1 and type 2 hypervisors. A type 1 hypervisor runs directly on the machine's hardware, whereas a type 2 hypervisor runs on top of a native operating system.

as processes of the host operating system. Prominent examples of type 1 and type 2 hypervisors are Xen [5] and QEMU [8], respectively.

2.2.2 Paravirtualization

In contrast to full virtualization, *paravirtualization* provides virtual machines with a software interface that differs from that of the underlying hardware. Therefore, guest operating systems must be modified to take advantage of this special interface by including hypervisor-specific code. This is depicted in Figure 2.3a. In particular, sensitive instructions like processor mode changes or hardware accesses will be replaced with *hypercalls*. A hypercall is used by the paravirtualized guest operating system to make calls to the hypervisor, similarly to a user space program making system calls to a (regular) operating system kernel. Furthermore, paravirtualization provides special *hooks* that will be triggered on certain guest state changes. One of the advantages of paravirtualization is its possibly improved performance over full virtualization (especially in the absence of hardware-assisted virtualization) since expensive instructions—that is, instructions that are difficult and time-consuming to simulate—will be replaced with more efficient hyper-



(a) Paravirtualization (b) OS-level virtualization

Figure 2.3: Comparison of paravirtualization and OS-level virtualization. Paravirtualization takes advantage of modified guest kernels. OS-level virtualization uses one host kernel which may run multiple isolated user space instances (all of which use the same host kernel).

calls. Furthermore, paravirtualization explicitly allows requesting particular functionality from the hypervisor upon certain conditions. The concept of paravirtualization is complementary to the concept of type 1 and type 2 hypervisors and may be used with either approach (cf. Figure 2.3a).

2.2.3 Operating-System-Level Virtualization

Another way of achieving virtualization is called *operating-system-level virtualization*, or *OS-level virtualization*. This approach differs from the above virtualization mechanisms in that it utilizes multiple isolated user space instances, called *containers*, which all share the same (host) kernel. This is shown in Figure 2.3b. OS-level virtualization can be thought of as a more sophisticated version of Unix-like `chroot` environments.

One advantage of OS-level virtualization is that it is relatively easy to set up, and to employ and run different containers. Furthermore, OS-level virtualization, in general, achieves high performance. This is especially true in case of machines lacking hardware-assisted virtualization support. In this context, OS-level virtualization is particularly well suited for deploying virtualization on mobile devices [73]. Another advantage is the flexible and efficient use of resources (and avoidance of redundancy) by different containers. However, this flexibility may result in one container negatively impacting the available resources, and performance, of other containers—

intentionally or by accident. Therefore, in addition to isolating containers from one another, the kernel also often allows limitation and prioritization of resources like CPU, memory, and I/O. One drawback of OS-level virtualization is the fact that all containers must use the same (host) kernel. This excludes the possibility of running a different operating system kernel within a container. For example, a Linux (host) kernel implementing OS-level virtualization is not capable of running Windows within a container. A prominent example of software utilizing OS-level virtualization is the open-source project Docker [108].

2.3 Distributed Filesystems

A *filesystem* usually acts as an interface to a connected non-volatile memory storage device – typically a Hard Disk Drive (HDD) using magnetic storage or a Solid State Drive (SSD) using flash memory. The storage for HDDs and SSDs is organized as blocks and the filesystem maps files to the block-level storage. A *distributed filesystem*, sometimes also called *network filesystem*, allows accessing remote files over a computer network without requiring direct block-level access to the storage of the physical device where the files reside on [162]. In this way, distributed filesystems may even be used to implement computing devices having no attached physical storage device at all. Distributed filesystems are based on a client-server model where the client is able to access and operate on remote files by communicating with one or more servers over the network. Prominent examples of distributed filesystems include Network File System (NFS) and Server Message Block (SMB), the latter of which is also known as Common Internet File System (CIFS). The aforementioned filesystems along with other common filesystems are supported by the major operating systems like Microsoft Windows, Apple's macOS, and Linux.

2.3.1 9P Filesystem

Plan 9 is a distributed operating system developed by Bell Labs and has been available as open source software since 2003. The objective of Plan 9 is to represent all resources as files without distinguishing between local and non-local objects. The *9P filesystem* has been developed as part of Plan 9 in order to achieve this. In particular, 9P has been designed as a distributed filesystem protocol that may be used over a computer network and which operates on a file-based granularity. The client-server protocol uses messages that reflect ordinary file operations like reading or writing a

class	**op-code**	**description**
session	version	parameter negotiation
	auth	security authentication
	attach	establish a connection
	flush	abort a request
	error	return an error
file	walk	lookup pathname
	open	access a file
	create	create and access a file
	read	transfer data from a file
	write	transfer data to a file
	clunk	release a file
metadata	stat	read file attributes
	wstat	modify file attributes

Table 2.2: Set of 9P2000 operations [178]

file. Each request message, called T-message, results in a single response message, called R-message, or—in case of an error—an R-error message.

There exist three important versions of the 9P protocol: 9P2000, 9P2000.U, and 9P2000.L. The 9P2000 protocol [178] is the last version of the Plan 9 protocol that has been developed by Bell Labs. It only contains a small set of basic operations all of which are initiated by the client. The operations are shown in Table 2.2. 9P2000.U extends the 9P protocol by incorporating better Unix support and providing full POSIX semantics. 9P2000.L [62] is another extension of the protocol in order to make it better suitable for Linux; it is feature complete as of 2.6.38. v9fs [91] is a Linux client implementation of the 9P protocol supporting the versions 9P2000, 9P2000.U, and 9P2000.L.

An important feature and advantage of 9P is that it is protocol independent. In fact, it can be used over any reliable, in-order transport [178]. In particular, the 9P client may communicate with a remote 9P server over a computer network (e.g., using a TCP connection) but it may also communicate with a 9P server located on the same machine (e.g., using named pipes).

2.4 Machine Learning

Machine learning is widely used in computer science and other fields and can be applied to figure out how to perform important tasks by generalizing from examples [38]. This allows using experience to improve performance or making accurate predictions [111].

First, a model is learned in the *training* or *learning* phase. In this context, the *examples* contained in the *training set* are used to adjust the parameters of the adaptive model. After the training phase, the model can be used to make predictions for unseen data contained in a *test set*. The ability of correctly handling unseen data in a reasonable manner is known as *generalization*. For most practical applications, the original input variables are typically preprocessed to transform them into some new space of variables where, it is hoped, the pattern recognition problem will be easier to solve [11]. In this context, it is common to reduce the dimensionality of complex data. To this end, certain properties of the data, called *features*, will be considered that are known or presumed to be discriminative and characteristic for the observed inputs. Multiple features are often grouped in a *feature vector*. The described preprocessing procedure itself is called *feature extraction*.

Two important categories of machine learning are *supervised learning* and *unsupervised learning*. In supervised learning, the examples of the training set are labeled. This means that the correct output of the model for a given input of the training set is known. In unsupervised learning, the examples of the training set are unlabeled and the objective is to group the data based on similarities. Supervised learning is more developed in the literature than unsupervised learning [58].

2.4.1 Classification

Classification is an important instance of supervised learning. In fact, it is the most mature and widely used machine learning type [38]. The corresponding unsupervised method is called *clustering*. In classification, the training data is labeled and a discrete output is produced that belongs to one out of two or more *classes*. A common example of classification is spam filtering where emails are processed as input and assigned to the classes “spam” or “no spam”, that is, the emails will be labeled accordingly. The described case of assigning a given input to one out of two possible classes is called *binary classification*. Figure 2.4 shows an exemplary visual representation of binary classification for the use case of spam filtering. In contrast,

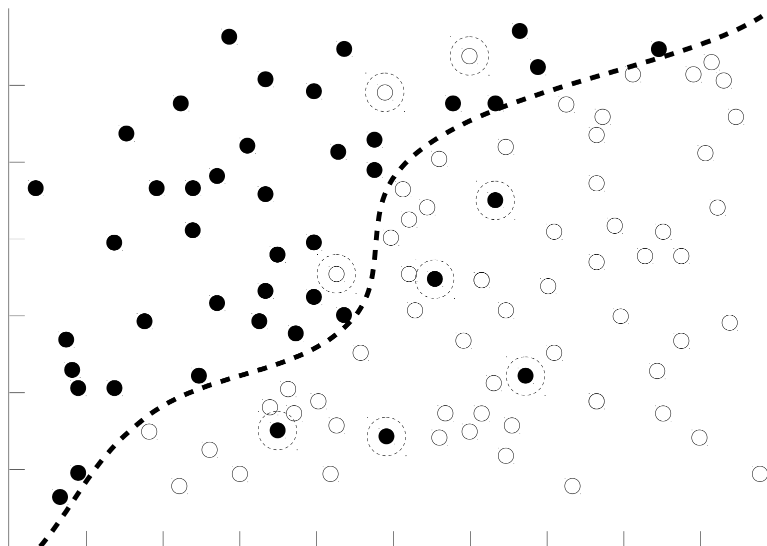


Figure 2.4: Visual representation of binary classification. The white dots represent “no spam” emails (first class). The black dots represent “spam” emails (second class). The dashed black line shows the class boundary approximated by the utilized classifier. The encircled white dots represent type I errors (false positives). The encircled black dots represent type II errors (false negatives).

multiclass classification[4] or *multinomial classification* assigns a given input to one out of more than two classes, e.g., the newspaper categories “politics”, “science”, and “business”. A *classifier* is an algorithm implementing classification. Some well-known classifiers include Naïve Bayes [59], k-Nearest Neighbor (k-NN) [30], Support Vector Machines (SVMs) [29], and Random Decision Forests [70], in particular, Breiman’s Random Forests [17]. *Model selection* uses available labeled data to train a range of models and compare them on independent labeled data, called a *validation set*, to select the one having the best predictive performance [11]. However, in practice, the amount of labeled data is often too small to set aside a validation set since that would leave an insufficient amount of training data. Therefore, a common technique known as *n-fold cross validation* can be applied where the labeled data is used for both model selection and training [111]. Classifier performance can be measured by the percentage of new patterns that are assigned to the wrong category [40]. This is called the classification *error rate*. A classifier may exhibit two types of errors. A *type I error*, also known

[4]Not to be confused with multi-label classification where multiple labels are assigned to a given input.

as a *false positive*, is an output indicating a tested condition is true (e.g., email is spam) even though the condition is not fulfilled (email is, in fact, not spam). In Figure 2.4, type I errors are represented by encircled white dots. A *type II error*, also known as a *false negative*, is an output indicating a tested condition is false (e.g., email is not spam) even though the condition is fulfilled (email is, in fact, spam). In Figure 2.4, type II errors are represented by encircled black dots.

Chapter 3

Multiplexing TPM Integrity Measurements

Measuring the integrity of critical operating system components and securely storing these measurements in a hardware-protected TPM is a well-known approach for improving system security. However, currently it is not possible to securely extend this approach to TPMs used in virtualized environments while maintaining the same level of security.

In this chapter, we show how to multiplex integrity measurements originating from arbitrarily many Virtual Machines (VMs) with just a single standard TPM. In contrast to existing approaches such as Virtual Trusted Platform Module (vTPM), our approach achieves a higher level of security since measurements, once stored, will never be held in software but are fully hardware-protected by the TPM at all times. We develop a remote attestation protocol enabling the integrity reporting of individual VMs. We establish an integrity-protected mapping between each measurement and its respective VM such that it is not possible for an attacker to alter this mapping during remote attestation without being detected. Furthermore, all measurements will be stored in the TPM in a concealed manner in order to prevent information leakage of other VMs during remote attestation. The experimental results of our proof of concept implementation show the feasibility of our approach.

In Chapter 4, we will explore how to take integrity measurements of multiple VMs by monitoring the VMs from "outside of the box". We will build upon and take advantage of the work developed in this chapter in order to securely store the obtained measurements in a single, multiplexed TPM.

Parts of this chapter have been published in *Secure and Privacy-Aware Multiplexing of Hardware-Protected TPM Integrity Measurements among Virtual Machines* at the 15th International Conference on Information Security and Cryptology (ICISC) in 2012 [179].

The rest of this chapter is organized as follows. In Section 3.1, we describe the challenges in using TPMs in virtualized environments and present our contributions to solve the challenges. In Section 3.2, we state the requirements and central points of our concept. Section 3.3 describes the attacker model and our assumptions. Section 3.4 gives an overview of how we take integrity measurements, store and multiplex them, and provide privacy-aware attestation information to third parties. The multiplexing technique used for storing the measurements and attesting individual VMs is explained in Section 3.5. We take special precautions to guarantee the privacy of VMs by concealing the measurements; the details of these concealment transformations are described in Section 3.6. In Section 3.7, we show our adapted remote attestation protocol enabling the integrity reporting of individual VMs. Section 3.8 describes how to verify the measurements of an attested VM. In the security analysis in Section 3.9, we discuss how we prevent an attacker from manipulating measurements and the mapping to VMs. Section 3.10 describes our proof of concept implementation. Section 3.11 presents the performance evaluation results. Section 3.12 discusses related work. Section 3.13 concludes this chapter.

3.1 TPM Virtualization Challenges

Virtualization and the utilization of secure cryptoprocessors are two well-known approaches for improving system security. Virtualization can be used to partition a system into several VMs such that critical system components are isolated from one another, thus being able to reduce the Trusted Computing Base (TCB) of the overall system. Virtualization is also heavily used in the context of cloud computing [105] where multiple VMs of different customers run concurrently on the same system platform. In this context, it is crucial that one VM cannot access or manipulate data of another VM.

A secure cryptoprocessor is a microprocessor or System on Chip (SoC) usually capable of securely managing cryptographic keys and storing data such that it is not possible for an attacker to extract or manipulate these keys and data. A very prominent and widespread implementation of a secure cryptoprocessor is the Trusted Platform Module (TPM), most notably the TPM 1.2 [173] and the TPM 2.0 [174] as specified by the TCG [172]. In particular, the TPM can be used to securely store integrity measurements

in special PCRs that reflect a system's configuration. An *authenticated boot* is used to establish a *chain of trust* by measuring each component in the booting sequence, starting with an inherently trusted component called the CRTM. Developments such as the IMA [148] extend this chain of trust to the application layer by measuring programs executed in the Operating System (OS) and storing the measurements in a PCR of the TPM. Finally, the integrity measurements are used in the course of a *remote attestation* to prove to a remote party that the system platform is in a trusted state.

Unfortunately, the TPM was not designed to be used directly in virtualized environments and thus the advantages of virtualization and TPMs cannot be easily combined. In particular, both TPM 1.2 and TPM 2.0 were not designed to store integrity measurements on a per-VM basis within the TPM. Furthermore, it is impossible to perform remote attestations only for particular VMs. Researchers have proposed several ideas to tackle these problems. The emulation of PCRs in software for each VM was proposed in [9, 89, 101, 46]. However, on a compromised system these PCRs can be manipulated by an attacker, allowing him to forge remote attestations. There also exist proposals that describe TPM adaptations with hardware-based virtualization support that do not suffer from the aforementioned security vulnerability [166, 51]. However, these proposals suffer from other limitations which will be described in detail in Section 3.12.

In this chapter, we make the following contributions:

- We show how to multiplex integrity measurements originating from arbitrarily many VMs with just a single standard TPM and only requiring one PCR. In contrast to [9, 46], which emulate PCRs in software, our approach achieves a higher level of security since measurements are always stored in the hardware-protected PCRs of the TPM.

- We show how to establish an integrity-protected mapping between each measurement and its respective VM such that it is not possible for an attacker to alter this mapping (e.g., hiding malicious programs by mapping their measurements to other VMs) without being detected.

- We develop a remote attestation protocol for attesting the integrity of individual VMs. A crucial problem we have to solve in the context of remote attestation is that our approach of sharing PCRs among VMs, inherently requires the disclosure of all measurements of all VMs. This entails security and privacy issues as even a legitimate challenger in the remote attestation protocol is then able to determine exactly which

software is running in all other VMs. This information might be used to exploit (known) vulnerabilities of that software. We overcome this problem by storing all measurements in the multiplexed PCR in a concealed manner. This enables us to fully disclose the (concealed) contents of the PCR and to selectively reveal non-concealed measurements on a per-VM basis.

3.2 Requirements Analysis

The concept for secure and privacy-aware multiplexing of integrity measurements among VMs as developed in this chapter shall satisfy the following requirements. A rationale is given for each requirement to illustrate as to why the requirement is important.

(R1) Hardware Protection. Integrity measurements must be stored on a per-VM basis in the hardware-protected secure storage area of a TPM such that stored measurements cannot be manipulated by an attacker in order to forge remote attestations of VMs, even on an entirely compromised system w.r.t. software running on the system, which includes the hypervisor in virtualized environments. The rationale is to achieve a higher level of security compared to approaches storing the measurements of VMs only in software.

(R2) Measurement Multiplexing and Isolation. The secure storage must be shared by all VMs. Storing integrity measurements of a VM in the shared secure storage must not influence previous or future measurements of other VMs. This is an important requirement because of the limited amount of available registers due to specification and hardware constraints. In particular, the TPM 1.2 specification [173] guarantees only a minimum of 16 PCRs, whereas the platform-specific specifications of the TPM 2.0 range from a minimum of one PCR for TCG TPM 2.0 Automotive Thin Profile [175] to a minimum of 24 PCRs for the TPM 2.0 TCG PC Client Platform TPM Profile (PTP) specification [176].

(R3) Integrity-Protected Measurement Mapping. Integrity measurements must be annotated in such a way that there exists a one-to-one mapping between each measurement and its associated VM. It must not be possible to change a mapping afterwards without being detected. Therefore, we require the mapping to be part of the measurement data which will be securely stored in the utilized PCR of

the TPM. Furthermore, we require the remote attestation protocol (cf. Requirement R5) to be based on this hardware-protected mapping such that the mapping cannot be manipulated in software within a remote attestation protocol run.

(R4) Unlimited Measurements. There is no (conceptual) upper bound on the number of integrity measurements that may be stored for a VM. Additionally, there is no (conceptual) upper bound on the total number of VMs. The rationale is to avoid potential restrictions likely to be entailed by naive multiplexing approaches and to stay as flexible as possible such that the concept can even be applied to machines and use cases that depend on a very large number of VMs, for example, servers used in cloud computing.

(R5) Privacy-Aware Remote Attestation. The remote attestation protocol must allow for the attestation of individual VMs in such a way that no integrity measurements and no other sensitive information of other VMs is disclosed. This is important as such information could otherwise not only be gained and misused by an outside adversary attacking the protocol but also by legitimate participants of the protocol extracting sensitive information like knowledge of software used by competitors, which can result in a market advantage or can even be used to launch targeted attacks.

3.3 Attacker Model and Assumptions

Our objective is to multiplex integrity measurements originating from arbitrarily many VMs in a secure and privacy-aware manner with just a single standard TPM and only requiring one PCR.

In this context, we develop a remote attestation protocol for attesting the integrity of individual VMs. We focus on the measurement data of the multiplexed PCR as we assume the rest of the system can be attested with general remote attestation techniques. We consider all Man-in-the-Middle (MITM) attacks on the remote attestation protocol where an attacker tries to manipulate measurements of one or more attested VMs in order to hide certain software (e.g., malware) and configurations present in the respective VMs. The MITM is located between the prover and the verifier and is able to intercept and manipulate the transmitted data, e.g., the attacker may discard or forge measurements.

We assume that the Attestation Agent (AA), which is responsible for attesting the trustworthiness of individual VMs, utilizes some form of VM-

based (one-way) authentication of parties requesting a remote attestation for a particular VM. Note that even for authenticated parties, we still need to privacy-protect the measurement data of other (non-authenticated) VMs as done by our approach.

We also consider attacks on the attesting platform in which an attacker tampers with the platform's software (e.g., by forging remote code updates intended for the attesting platform). The malicious software (or the attacker) may try to hide its execution by removing or manipulating its respective integrity measurement.

We do not consider direct physical attacks on the TPM.

3.4 System Overview

Our concept is based on a virtualized platform consisting of a single hardware TPM[1], a hypervisor, and arbitrarily many VMs. We take integrity measurements of supervised files located in the VMs and securely store these measurements in the TPM such that individual VMs can be attested in a privacy-aware manner. In general, there mainly exist two approaches within the described scenario for obtaining integrity measurements. In the first approach, a Measurement Agent (MA) (e.g., IMA [148]) runs in each VM and monitors (the execution of) supervised files, calculates according integrity measurements, and propagates them to the hypervisor-level. In the second approach, the MAs are not located within the VMs themselves but the monitoring of supervised files of *all* VMs is rather conducted by an MA located at the hypervisor-level. This "outside of the box" monitoring is used in our concept and has the advantage of only requiring a single MA. Thus, it is more scalable than the first approach where as many MAs are required as there exist supervised VMs. Additionally, it is more secure than the first approach where at the hypervisor-level one has to trust the measurement information originated from the VMs themselves—this information is untrustworthy and might be manipulated by an attacker located in the VM. In Chapter 4, we show the details of how to realize this integrity monitoring approach by taking advantage of paravirtualized filesystems in order to take measurements from VMs in a secure way.

Our system architecture is shown in Figure 3.1. The Measurement Agent (MA) is responsible for taking measurements of all VMs as described above (step 1). A new measurement m along with a VM identifier vm uniquely

[1]For the sake of readability, the descriptions in this chapter are geared towards TPM 1.2. However, the developed concepts can also be applied to TPM 2.0 (e.g., using SHA-256 instead of SHA-1, `PCR2_Quote` instead of `PCR_Quote`, or AK instead of AIK).

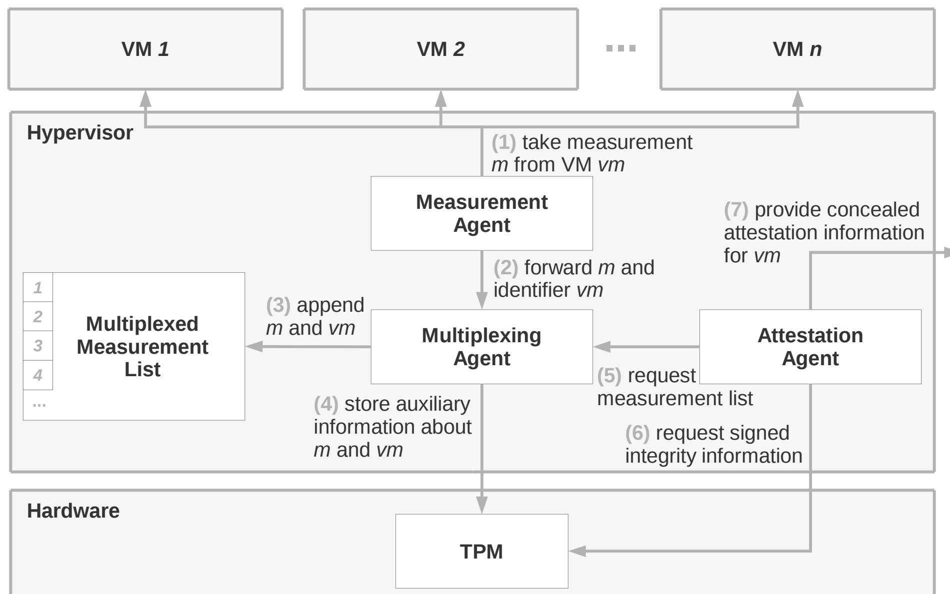


Figure 3.1: System architecture showing the main components and the workflow for virtualizing integrity measurements and reporting them.

representing the measurement's associated VM (i.e., the VM containing the supervised file of which the measurement was taken) will be forwarded to the Multiplexing Agent (MPA) (step 2). The MPA appends m and vm to the so-called Multiplexed Measurement List (MML) (step 3). This allows the MPA to keep track of all measured files together with their associated VMs. Furthermore, the MPA stores auxiliary information about m and vm in the TPM in order to hardware-protect this information (step 4). The details will be described in Sections 3.5 and 3.6. Note that the integrity of the MML must not be specially protected as manipulations of the measurements or VM identifiers will be detected in the course of a remote attestation through the auxiliary information stored in the TPM. The Attestation Agent (AA) is responsible for attesting the trustworthiness of individual VMs. In this context, the AA takes advantage of the information provided by the MML (step 5) as well as of the auxiliary information provided by the TPM (step 6). Note that our approach of sharing and multiplexing measurement information of multiple VMs in the TPM inherently requires the disclosure of all measurements of all VMs in the course of a remote attestation. In order to preserve the privacy of all other (non-attested) VMs, the AA conceals all measurements of the other VMs. This is done in such a way that a proper

remote attestation is still possible. The concealed data is then transmitted to the verifier (step 7). The details will be described in Section 3.7. We note that the AA is not required to be part of the TCB as even a completely compromised AA cannot be misused by an attacker to induce a forged remote attestation. The reason is that the auxiliary measurement information stored in the TPM is still protected from manipulation. Finally, the verifier inspects the received concealed attestation information in order to attest the VM. The verifier possesses knowledge of the concealing algorithm allowing him to perform the necessary transformations of the received data in order to carry out a successful validation. The details will be described in Section 3.8.

3.5 Multiplexed Storage and Attestation

The MPA stores integrity measurements in a single shared PCR of the TPM. This ensures that all measurements are hardware-protected by the TPM (cf. Requirement R1). Furthermore, by only requiring one PCR, we are able to efficiently deal with the limited resources offered by the TPM (cf. Requirement R2). We define an *integrity measurement* (often just *measurement*) m as the output of a hash function H applied to (the contents of) a file f, i.e., $m := H(f)$. A PCR of a TPM may hold arbitrarily many measurements by *extending* the measurements as a hash chain, i.e., $PCR[i] \leftarrow H(PCR[i]||m)$, for a measurement m and PCR with index i (where $||$ denotes concatenation). We take advantage of this property in order to store and multiplex a conceptually unlimited number of integrity measurements of arbitrarily many VMs (cf. Requirement R4). However, to retain the integrity-protected information in which VM a measurement m took place (cf. Requirement R3), the MPA not only extends m but also the corresponding VM's unique Virtual Machine Identifier (VM-ID) in the PCR (cf. Section 3.6).

In our concept, the AA is able to attest the integrity of individual VMs to a verifier (cf. Section 3.7) by utilizing the information that has previously been stored by the MPA. However, without further precautions, this requires the disclosure of *all* measurements of *all* VMs sharing the PCR. This entails security and privacy related problems as described in the introduction. Therefore, before extending the PCR, the MPA first conceals each measurement with a special value called *concealment*. A concealment is a non-predictable random or pseudorandom value that is at least the size of the output of the hash function H in which we will use it (cf. Section 3.6). The reason for this size is to adequately protect against dictionary attacks, brute force attacks, and lookup attacks trying to extract the plain measure-

ments [110, 106]. The concept of a concealment is related to the concept of a salt. However, in contrast, a concealment is unknown to a verifier and will only be disclosed to him when attesting a particular VM. The MPA maintains one *base concealment* for each VM and derives further concealments from it. In addition to concealing measurements, we also conceal the measurement's associated VM-ID to prevent a verifier from gathering information about how many measurements have been taken in other VMs. This information might otherwise be misused to detect usage patterns (e.g., activity level of VMs of competitors).[2]

Finally, this enables the AA to disclose all measurements of all VMs in a concealed manner to a verifier (cf. Requirement R5). For the attested VM, the non-concealed measurements, along with the attested VM's base concealment, are additionally revealed. The base concealment is used by the verifier to derive the same concealments as the MPA, which are then used to link the non-concealed measurements to their corresponding concealed measurements. This, in turn, allows the verifier to recalculate the proper hash chain (consisting of concealed measurements only) and to match it against the (signed) PCR value, thus ensuring the measurements' integrity and authenticity (cf. Section 3.8).

3.6 Measurement Concealment

In the following, we describe the details of the aforementioned concealment transformations of the integrity measurements. We let $id_{vm_1}, \ldots, id_{vm_n}$ denote unique and publicly known VM-IDs w.r.t. the set of all n VMs on a particular system. The MPA maintains for each VM id_{vm} one non-predictable *base concealment* $c_{vm} \in \{0,1\}^k$, with $k \geq 160$ (i.e., at least the size of the output of SHA-1). For the i'th measurement transformation (counting from zero) of a VM id_{vm}, the MPA derives a new *concealment* c^i_{vm} by incrementing c_{vm} i times, that is, $c^i_{vm} := c_{vm} + i,\ i \geq 0$. Note that c^0_{vm} denotes the base concealment c_{vm}. For brevity, we define $H := SHA1$ for the remainder of this chapter. We note that in the case of TPM 1.2, the specification is restricted to the hash function SHA-1. This restriction is getting increasingly critical. Theoretical attacks on SHA-1 have been known since 2005,

[2]Note that even though the employed technique efficiently reduces information leakage of other VMs, an attacker might still be able to infer vague (and rather unreliable) information based on the presence, absence, or number of concealed VM-IDs. For example, a large number of concealed VM-IDs may imply the presence of some other very active VM (many measurements stored) but it may also imply the presence of several VMs, with each VM being relatively inactive.

and SHA-1 was officially deprecated by NIST in 2011 [44]. In 2017, security researchers at CWI Amsterdam and Google have announced the first practical collision attack on SHA-1 [165]. In contrast, the algorithm agility approach of the TPM 2.0 specification allows virtually any hash function (e.g., SHA-256), thus protecting against the above collision attack.

Each time MA measures (the content of) a monitored file f executed in VM id_{vm} by calculating $m := H(f)$ and forwards it to the MPA, the MPA associates m with id_{vm}, conceals both m and id_{vm}, and extends the result to the shared PCR p. In particular, for the i'th measurement of VM id_{vm}, the MPA does the following five steps (called a *round* in the following):

1. Derive new VM-specific concealment c_{vm}^{i} from base concealment c_{vm}
2. Conceal measurement m by hashing it with c_{vm}^{i}, i.e., $\mu := H(m||c_{vm}^{i})$
3. Conceal VM-ID id_{vm} with same concealment c_{vm}^{i}, that is, $\delta_{vm} := H(id_{vm}||c_{vm}^{i})$
4. Hash over the concealed measurement value μ combined with the concealed VM-ID δ_{vm}, i.e., $\varphi := H(\mu||\delta_{vm})$
5. Extend the TPM's shared PCR p with φ

Note that it is not possible to defer this measurement transformation (e.g., to the point in time where a remote attestation is requested) because the measurement must immediately be stored in the TPM in order to prevent an attacker from removing or manipulating previous integrity measurements once the system gets compromised.

Step one guarantees that we use a new concealment for each round. It is important that a verifier is able to produce the exact same sequence of concealments $c_{vm}^{1}, c_{vm}^{2}, \ldots$ from the base concealment $c_{vm} = c_{vm}^{0}$ (cf. Section 3.8). Note that simple incrementation is sufficient for deriving the concealments (in terms of confidentiality of the concealed values in steps two and three) since two consecutive (and thus similar) concealments c_{vm}^{i} and c_{vm}^{i+1} result in two completely different hash values $H(c_{vm}^{i})$ and $H(c_{vm}^{i+1})$ due to the avalanche effect [177].

Step two makes sure that it is sufficient to only disclose concealed measurements to a verifier V in order to reconstruct the hash chain represented

by the shared PCR p. Note that if the measurements were extended to the PCR without concealing them first, a verifier V would be required access to all non-concealed measurements in order to recalculate the hash chain (cf. Section 3.8). This would violate the privacy-aware remote attestation requirement (cf. R5) as in this case the measurements of all other (non-attested) VMs had to be revealed. With our approach, V can easily verify that a measurement m of the attested VM corresponds to the concealed hash value μ by checking whether $\mu = H(m||c^i_{vm})$ holds. Note that it is infeasible to find some other preimage $x \neq m||c^i_{vm}$ such that $H(x) = H(m||c^i_{vm})$ because of the second-preimage resistance property of H.

In step three, we conceal the VM-ID to prevent V from gathering usage patterns of other VMs. Note that the usage of a static (VM-based) concealment c_{vm} would always map a VM-ID id_{vm} to the same concealed VM-ID $\delta_{vm} = H(id_{vm}||c_{vm})$, thus allowing to link (concealed) VM-IDs and measurements. We use different concealments for each round in order to prevent this.

Step four establishes the mapping between μ and δ_{vm} and thus implicitly also between m and id_{vm}.

In step five, the concealed hash value φ gets finally extended to the PCR p by using φ as the incoming operand `TPM_DIGEST` of the `TPM_Extend` command [173]. Note that it is sufficient to use the standard, non-modified `TPM_Extend` operation. Also note that storing the just described mapping between measurement and VM-ID directly in the integrity protected PCR (`PCR_Quote` may be used to sign the value of the PCR) makes it redundant to maintain an external integrity protected mapping.

3.6.1 Multiplexed Measurement List

In the course of a remote attestation, the verifier needs to inspect all integrity measurements of an attested VM in order to determine the VM's trustworthiness. However, the final hash chain value contained in PCR p is not sufficient to reconstruct the actual measurement data. Therefore, the MPA additionally stores all measurement data in chronological order w.r.t. their corresponding `TPM_Extend` operations in the Multiplexed Measurement List (MML) as depicted in the system architecture in Figure 3.1. The MML is an ordered list of pairs of the form (m, id_{vm}), where m is a (non-concealed) measurement and id_{vm} the corresponding (non-concealed) VM-ID id_{vm}, denoting the VM in which the measurement took place, that is:

$$MML := \Big\langle (m_0, id_{vm_{i_0}}), (m_1, id_{vm_{i_1}}), \ldots, (m_n, id_{vm_{i_n}}) \Big\rangle$$

The AA takes advantage of the information provided by the MML and transforms it in such a way that only the information required for the successful attestation of a specific VM will be disclosed. This will be described in the following as part of the remote attestation's integrity reporting phase.

3.7 Integrity Reporting

The TPM specification defines *integrity reporting* as "the process of attesting to integrity measurements recorded in a PCR. The philosophy behind integrity measurement, logging, and reporting is that a platform may enter any state possible—including undesirable or insecure states—but is required to accurately report those states" [174].

Figure 3.2 shows our adapted remote attestation protocol enabling the integrity reporting of individual VMs. Note that the remote attestation process actually consists of the integrity reporting phase as explained in the following as well as of the integrity validation phase as explained in Section 3.8.

In the integrity reporting phase, the verifier V first requests integrity measurement data for a particular VM and PCR p by providing the VM's unique and publicly known VM-ID id_{vm}. The prover P (represented by AA, in our case) then triggers the TPM to sign the content pcr_p of the requested PCR p—together with a supplied nonce in order to guarantee the freshness of pcr_p—with a special key of the TPM, called an AIK. This proves to V the content of the requested PCR. Note that P is required to disclose all information involved in the `TPM_Extend` operations (cf. Section 3.6, step five) in order to allow V to validate the measurements by recalculating the final hash chain value as described in Section 3.8. As explained above, this is the reason for extending only concealed measurements and VM-IDs in the PCR because it is sufficient then to only disclose concealed measurements and VM-IDs for non-attested VMs instead of revealing all plain measurements and VM-IDs. Therefore, in the multiplexed remote attestation protocol, P does not disclose the MML (containing all non-concealed measurements) directly to V as this would violate the privacy of all other (non-attested) VMs. Instead, the VM-specific Concealed Multiplexed Measurement List (CMML) $CMML_{vm}$ is constructed from the MML for attesting the VM id_{vm}. The construction is done by sequentially processing all pairs of the MML from left to right. Pairs not belonging to the attested VM id_{vm} are substi-

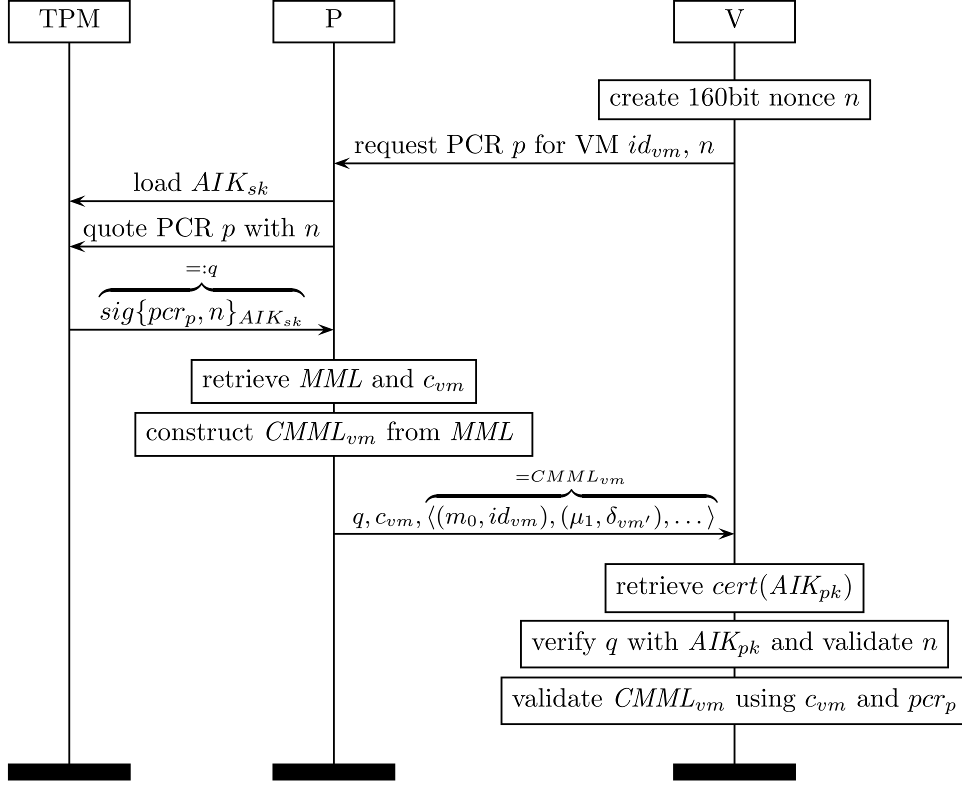


Figure 3.2: Multiplexing remote attestation protocol enabling the integrity reporting and validation of individual VMs.

tuted with their concealed counterparts (cf. Requirement R5). In particular, the i'th occurrence (counting from zero) of a pair $(m, id_{vm'}) \in MML$, for some measurement m and some VM-ID $id_{vm'} \neq id_{vm}$, gets substituted with $(H(m||c^i_{vm'}), H(id_{vm'}||c^i_{vm'}))$. In this context, the AA requests the required concealment values from the MPA. Pairs belonging to the attested VM remain non-concealed. Finally, P sends $CMML_{vm}$, the base concealment c_{vm}, and the signature data q of content pcr_p to V. Note that AA may cache the concealed pairs to avoid recalculating them for each remote attestation.

3.8 Integrity Validation

In the following, we describe how to validate the CMML and its contained measurements transmitted in an integrity reporting run as described before. We will show that a validation always fails if a MITM manipulates com-

binations of $CMML_{vm}$, c_{vm}, and q. The verification process done by V is twofold.

In the first phase, the CMML is validated to make sure that a MITM did not tamper with the data and that consequently all contained measurements are correct. The validation process will be explained in detail in the following.

In the second phase, V inspects these measurements to determine the trustworthiness of the attested VM. This might be done by a whitelist or blacklist approach that checks for good measurements (e.g., legitimate programs) or bad measurements (e.g., known malware), respectively. The TPM specification states that "an independent process may evaluate the integrity states and determine an appropriate response" and "log entries may be evaluated individually to determine if the change in system state indicated by the event is acceptable" [174]. The descriptions are vague because determining the trustworthiness of an attested VM (or, in general, an attested machine) and determining the appropriate responses by the verifier strongly depend on a multitude of factors and must be determined based on the actual use case. In this work, we also focus on the first phase of the integrity validation to protect against attacks trying to manipulate the reported integrity measurements. This builds the necessary foundation for the correct execution of the second phase of the integrity validation.

The validation process for the first phase of the integrity validation is shown in Algorithm 1. V first uses AIK_{pk} on q to verify the authenticity and integrity of content pcr_p of the requested PCR p. This detects all manipulations of q by a MITM as well as replay attacks due to the included nonce n. V then validates the CMML with the help of c_{vm} and pcr_p. The validation process is shown in Algorithm 1. It simulates all `PCR_Extend` operations that have (allegedly) been done by P and compares the result with the signed PCR value $pcr_p =: pcr$. This is done by inspecting each pair (a, b) of the CMML. Each pair (a, b) with $b = id_{vm}$ contains non-concealed measurement data a for the attested VM id_{vm}. However, since all measurements have been extended to the PCR by P in a concealed manner (cf. Section 3.6), V needs to reconstruct the corresponding concealed value $\varphi := H(\mu||\delta)$, where $\mu := H(a||c^i_{vm})$ and $\delta := H(b||c^i_{vm})$ (for round i), in order to correctly simulate all `PCR_Extend` operations. All other pairs (a, b) with $b \neq id_{vm}$ do not belong to the attested VM id_{vm} and have already been concealed by P, that is, $a = \mu$ and $b = \delta$. Thus, the concealed values μ and δ can be directly used to construct $\varphi := H(\mu||\delta)$. Finally, φ is used to simulate the `PCR_Extend` operation. These steps are repeated for each pair of the CMML. If the final simulated PCR value pcr' matches the signed

Algorithm 1 Validation of CMML

```
1: procedure VALIDATE_CMML(id_vm, cmml_vm, c_vm, pcr)
2:     pcr' := 0
3:     used_c := false
4:     for (a, b) in cmml_vm do
5:         if b = id_vm then                 ▷ does pair belong to attested VM?
6:             μ ← H(a||c_vm)                ▷ construct concealed measurement value
7:             δ ← H(b||c_vm)                ▷ construct concealed VM-ID
8:             c_vm ← c_vm + 1               ▷ set concealment for next round
9:             used_c ← true                 ▷ exhaustive blinding attack now impossible
10:        else
11:            μ ← a                         ▷ measurement already concealed
12:            δ ← b                         ▷ VM-ID already concealed
13:            if δ = H(id_vm||c_vm) ∨ δ = H(id_vm||c_vm − 1) then
14:                return false              ▷ blinding attack
15:            end if
16:        end if
17:        φ ← H(μ||δ)
18:        pcr' ← H(pcr'||φ)                 ▷ simulate PCR_Extend
19:    end for
20:    if pcr' = pcr ∧ used_c = true then
21:        return true                       ▷ confirm integrity of cmml_vm
22:    else
23:        return false                      ▷ integrity violation detected
24:    end if
25: end procedure
```

PCR value pcr, the measurements of the CMML correctly reflect the actual measurements of the attested VM.

The check in line 13 of Algorithm 1 detects *blinding attacks* where a MITM tries to hide non-concealed pairs $(a, b) = (a, id_{vm})$ belonging to the attested VM id_{vm}. The *blinding* is done by substituting pairs (a, id_{vm}) with their corresponding concealed pairs $(\mu, \delta) := (H(a||c_{vm}^i), H(id_{vm}||c_{vm}^i))$, with the intention of misleading V into thinking that the concealed pairs (μ, δ) do not belong to VM id_{vm}. Note that in this case the recalculated pcr' would still match pcr since (μ, δ) has indeed been extended to the TPM. Note also that in our concept we intentionally conceal a measurement m and its corresponding VM-ID id_{vm} separately instead of concealing m and id_{vm} combined, e.g., $\varphi := H(m||id_{vm}||c_{vm}^i)$. The reason is that in the latter

case, it would be impossible for V to check whether a non-concealed pair (m, id_{vm}) has been blinded (i.e., checking whether $\varphi = H(m||id_{vm}||c^i_{vm})$ holds) because the measurement m is unknown to V. We will come back to blinding attacks in the Security Analysis in Section 3.9.

A special case of the described blinding attack is to blind *all* non-concealed pairs and to additionally substitute the base concealment $c_{vm} = c^0_{vm}$ with some $c'_{vm} \neq c^0_{vm} \wedge c'_{vm} \neq c^1_{vm}$. Note that in this case the check for blinding attacks in line 13 fails since the original base concealment c_{vm} used for the (first) blinding operation now differs from the concealment c'_{vm} used in the check. Furthermore, the substitution of c_{vm} with c'_{vm} will not be detected since c'_{vm} is never used to calculate a concealed pair out of a non-concealed one (because there are no non-concealed pairs left) and thus pcr' matches pcr. Therefore, in order to detect such *exhaustive blinding attacks*, we explicitly check in lines 9 and 20 of Algorithm 1 that V used the base concealment in the calculation of pcr'.

3.9 Security Analysis

In this chapter, we presented our approach for multiplexing integrity measurements originating from arbitrarily many VMs within a TPM in a secure and privacy-aware manner. In the following, we evaluate the security of this approach. In general, we assume that an attacker tries to manipulate measurements of one or more attested VMs in order to hide certain software (e.g., malware) and configurations present in the respective VMs. To achieve this, the attacker might try to compromise the attesting system in an attempt to either remove or manipulate the corresponding integrity measurements. However, it is not possible to remove or manipulate the corresponding measurement data stored by the MPA in order to forge a valid remote attestation because of our utilized hardware-protection of measurements and integrity-protected measurement mapping realized by our approach (cf. Requirements 1 and 3).[3] However, on a compromised system, the attacker is still able to manipulate all data located outside of the hardware-protected realm of the TPM, in particular, the VM-specific concealments and the VM-specific CMMLs. In the following, we focus on an adversary attacking our remote attestation protocol by manipulating this data.

[3]We note that an attacker is able to *add* (fake) measurements after the system has been compromised but this is considered non-critical as it still does not allow the attacker to manipulate already taken measurements.

3.9.1 Discarding Measurements

The attacker might try to hide measurements—such as a measurement m of a malicious program—of the attested VM by simply discarding the element $(m, id) \in \mathit{CMML}$. In this case, however, the hash chain value pcr' calculated by the verifier from the CMML (cf. Algorithm 1) will not match the TPM's quoted PCR value pcr anymore and the attestation will fail.

3.9.2 Substituting Measurements

Another attack consists of substituting a measurement m with some other (fake) measurement $m' \neq m$ such that the recalculated hash chain value pcr' still matches pcr. This implies that for the hash function H it must hold that $H(m) = H(m')$. However, finding such an m' is infeasible because of the second-preimage resistance property of the hash function H. Note that this argument still holds in the case of an attacker substituting multiple measurements (such that the final hash chain value pcr' still matches pcr, even though intermediate values eventually leading to the value pcr' might differ) as such an attack can be reduced to the above case of manipulating a single measurement.

3.9.3 Substituting VM-IDs

Instead of manipulating a measurement m itself, the attacker might try to change a measurement's associated VM-ID id with the intention of hiding the measurement in the attested VM. However, the naive approach of just substituting the id of a pair $(m, id) \in \mathit{CMML}$ with some other $id' \neq id$ does not work since the pair (m, id') will be incorrectly treated as an already concealed one (cf. Algorithm 1) and thus the validation will fail. The case of an attacker not only substituting the VM-ID id but also the corresponding measurement m of a pair $(m, id) \in \mathit{CMML}$ is called a blinding attack and will be discussed in the following.

3.9.4 Blinding Measurements and VM-IDs

In a *blinding attack*, a MITM substitutes both the measurements *and* the VM-IDs with their corresponding concealed pairs. The attacker's goal is to hide a measurement m belonging to an attested VM id_{vm}. In order to blind a measurement m and a VM-ID id_{vm}, the attacker substitutes the pair $(m, id_{vm}) \in \mathit{CMML}$ with the corresponding concealed pair $(\mu, \delta) := (H(m||c^i_{vm}), H(id_{vm}||c^i_{vm}))$, with the intention of misleading the verifier into thinking that the concealed pair (μ, δ) does not belong to the attested VM

id_{vm}. The rationale is that—according to the integrity reporting protocol in Section 3.7—a pair will be transmitted in its concealed form only if the pair belongs to another (non-attested) VM. Note that in the case of a blinded pair, the recalculated pcr' in Algorithm 1 would still match pcr since (μ, δ) has indeed been extended to the TPM as described in Section 3.6.

There exist four types of blinding attacks (and combinations thereof) w.r.t. the position of the blinded pairs within the CMML: intermediate, trailing, leading, and exhaustive. In the following, we will analyze each type and show that our concept protects against all of them.

Note that since concealed pairs in the CMML do not influence the state of the concealment c_{vm} in Algorithm 1, we consider, w.l.o.g., only pairs of the attested VM. In particular, we assume the following CMML (along with signature data q and base concealment c_{vm}) is sent from the prover P to the verifier V in the course of a remote attestation protocol run as described in Section 3.7:

$$CMML_{vm} = \langle (m_0, id_{vm}), (m_1, id_{vm}), (m_2, id_{vm}) \rangle$$

Intermediate Blinded Pairs

In this attack, a MITM blinds a pair (or several consecutive pairs) which is neither the first pair nor the last pair of the CMML. In other words, there exists at least one pair both before and after the blinded pair:

$$q, c^0_{vm}, \Big\langle (m_0, id_{vm}), \underline{(H(m_1 || c^1_{vm}), H(id_{vm} || c^1_{vm}))}, (m_2, id_{vm}) \Big\rangle$$

In this case, the attestation fails because the wrong concealment c^1_{vm} is used by the algorithm to conceal the third pair. The reason is that the concealment will not be incremented when processing the intermediate blinded pair (cf. Algorithm 1, lines 11 to 15).

Trailing Blinded Pairs

In this attack, a MITM blinds one or more consecutive trailing pairs:

$$q, c^0_{vm}, \Big\langle (m_0, id_{vm}), \underline{(H(m_1 || c^1_{vm}), H(id_{vm} || c^1_{vm}))}, \underline{(H(m_2 || c^2_{vm}), H(id_{vm} || c^2_{vm}))} \Big\rangle$$

In contrast to the previous scenario, in this case the attestation would actually succeed if there was not the explicit check for blinded pairs in line 13 of

Algorithm 1. The reason is that the "out of sync" concealment will not be used anymore after concealing the first pair (as was the case above). With the explicit check, the algorithm detects $H(id_{vm}||c^1_{vm})$ in the second pair and the attestation fails. In general, the check always matches the leftmost trailing blinded pair.

Note that in our concept we intentionally conceal a measurement m and its corresponding VM-ID id_{vm} separately instead of concealing m and id_{vm} combined like $H(m||id_{vm}||c^i_{vm})$. The reason is that in the latter case, it would be impossible for the verifier V to check whether a non-concealed pair (m, id_{vm}) has been blinded with c^i_{vm} because the measurement m is unknown to V.

Leading Blinded Pairs

In this attack, a MITM blinds one or more consecutive leading pairs:

$$q, c^2_{vm}, \left\langle \underline{(H(m_0||c^0_{vm}), H(id_{vm}||c^0_{vm}))}, \underline{(H(m_1||c^1_{vm}), H(id_{vm}||c^1_{vm}))}, (m_2, id_{vm}) \right\rangle$$

Note that in this type of attack, the MITM needs to manipulate the transmitted base concealment such that it correctly blinds the first non-concealed pair in the CMML. In particular, since the base concealment is now c^2_{vm}, the explicit check for the first pair on whether δ equals $H(id_{vm}||c^2_{vm}) \vee H(id_{vm}||c^1_{vm})$ fails because c^0_{vm} was used for the blinding by the MITM. However, the check matches the second pair and the attestation fails. In general, the check always matches the rightmost leading blinded pair.

Exhaustively Blinded Pairs

In this attack, a MITM blinds *all* pairs and substitutes the base concealment c^0_{vm} with some $c'_{vm} \neq c^0_{vm} \wedge c'_{vm} \neq c^1_{vm}$. In this case, the above checks for detecting intermediate, trailing, and leading blinded pairs attacks fail. The reason is that the proper base concealment c^0_{vm} is never used by the verifier to calculate a concealed pair out of a non-concealed one—as explained in Section 3.8. Therefore, we enforce the utilization of the base concealment c^0_{vm} in the recalculation of the measurement hash chain (lines 9 and 20 of Algorithm 1) in order to detect the base concealment's manipulation, thus preventing such attacks.

3.10 Prototype Implementation

We have implemented a proof of concept based on the Integrity Measurement Architecture (IMA) [148]. The implementation realizes the first approach for obtaining integrity measurements (measuring from within the VMs) as described in Section 3.4. The proof of concept is described in the following and is also used for the performance evaluation in Section 3.11. In Chapter 4, we show how to improve this concept, enabling us to monitor the VMs from outside by taking advantage of paravirtualized filesystems.

The proof of concept utilizes the QEMU emulator [8] (version 1.0.50) with enabled Kernel-based Virtual Machine (KVM) [84] full virtualization support. The host system runs the Ubuntu OS (version 11.04). Each guest VM runs Ubuntu 10.04 with an IMA-enabled Linux kernel (2.6.35). We patched the IMA kernel code so that measurements are not directly extended to the TPM but instead are forwarded to the MA running in the host system. The communication between the MA and the VMs is done over Virtual Local Area Network (VLAN). The MA listens on a dedicated range of ports for incoming connections. Whenever a new VM is started, QEMU connects the VM to a free port in that range using guest forward (`guestfwd`) rules. The so established socket is then used by our patched IMA to forward measurements to the MA; all communication over other ports is blocked.

The MPA has exclusive write access to the TPM (using TrouSerS [171], version 0.3.5-2) and implements the multiplexing concept as described in Section 3.4. The MPA requests the port number of the connected VM from the MA and uses it to derive the VM's unique VM-ID. Furthermore, the MPA dynamically generates and maintains a fresh concealment for each VM. Note that the mapping of VMs to VM-IDs and the VMs' associated concealments cannot be changed from within a VM in an attempt to forge the VM-ID or concealment since this data is maintained solely by QEMU, the MA, and the MPA, respectively, all of which are isolated from the VM.

Note that our patched IMA does not block until the measurements have actually been extended to the TPM. It rather just forwards them to the MA and is immediately ready for further tasks. The MA asynchronously processes and forwards the measurements to the MPA which, in turn, extends the measurements in a round-robin fashion to the TPM. This asynchronous approach significantly increases response times and overall performance in the VMs.

VMs	No IMA	Patched IMA		Ratio VM	Ratio total
		VM only	Total		
1	48.87	86.73	200.84	1.77	4.11
2	50.91	104.96	400.29	2.06	7.86
3	79.61	171.85	601.02	2.16	7.55
4	108.73	229.32	825.06	2.11	7.59
5	146.32	295.27	1318.17	2.02	9.01

Table 3.1: Average processing time for 10,000 measured files in each VM (in seconds)

3.11 Performance Evaluation

We analyze our proof of concept implementation to determine whether the MA and MPA might constitute possible performance bottlenecks since they represent the centralized location where all measurements from all VMs are collected, processed, and extended to the TPM.

The testing hardware consists of

- PC with an Intel Core2 Duo 3 GHz CPU,
- 4,096 MB RAM,
- TPM 1.2.7.40 from STM.

Each VM gets assigned 512 MB RAM and contains 10,000 distinct testing binaries which, on execution, just return. Furthermore, each VM runs our patched IMA that we additionally modified for the evaluation such that only the testing binaries get processed. To start the testing, we simultaneously trigger in all VMs the execution of the testing binaries in successive order.

Section 3.11 shows the testing results. Column four lists the total time needed from measuring all files to extending the measurements in the TPM. Note that the TPM requires most of the computation time. It takes about $200ms$ for 10,000 operations. Column three shows the fraction spent in a VM (on average) for measuring and forwarding. Column two lists the time needed by a VM (on average) running no IMA at all. The latter allows us to better compare how the parallel execution of multiple VMs naturally slows down program execution time in the VMs because of shared hardware resources. In fact, the parallel execution of three or more VMs

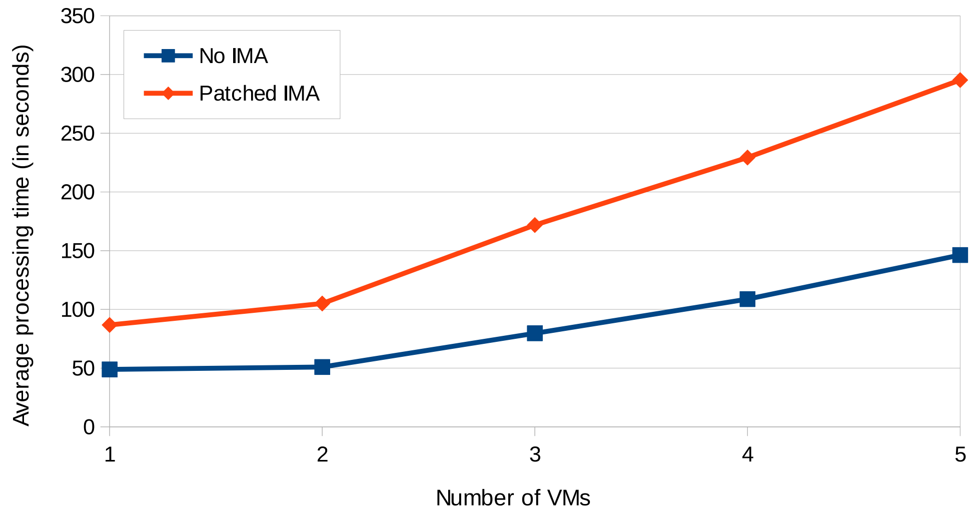


Figure 3.3: The average processing time of three or more VMs executed in parallel results in a slowdown for both our patched IMA and the system running no IMA at all.

exhibits such a slowdown for both our approach and the system running no IMA at all, as visualized in Figure 3.3. The time ratio in column five of Section 3.11 indicates an overhead of approximately a factor of two for our approach considering the time spent in the VMs. This is due to forwarding the measurements over VLAN to the MPA. Techniques like shared memory may be used to further reduce this overhead. The total ratio in column six reflects mainly the time needed for the `TPM_Extend` operations as noted above. We can see that our approach scales roughly linearly with the number of VMs. The increased slowdown with more than three VMs is mainly due to the rather limited hardware resources of our testing system as it occurs also with the system running no IMA at all. Consequently, our results indicate that the MA and MPA do not constitute performance bottlenecks.

We also note that for a remote attestation, in order to attest a single VM, we need to send the measurement data (CMML) of all VMs (cf. Section 3.7). Thus, in general, the size of the transmitted data increases linearly with the number of VMs. This is a disadvantage compared to other approaches that emulate a set of PCRs for each VM in software [9, 46] or maintain them in hardware [166, 51], where it is sufficient to only send measurement data of the attested VM. However, these approaches suffer from other limitations as described in the following Section 3.12.

3.12 Related Work

There exist various approaches that try to tackle the challenges of using TPMs in virtualized environments. In particular, the field of cloud computing [105] lead to various security-related research [23, 151, 94, 183, 33, 10]. In our work, we focus on one of the core security-related problematics which appears both in the context of cloud computing and in other more general virtualization scenarios; namely, how we can securely store integrity measurements originating from multiple virtual machines in a TPM and maintain the same level of security as provided by the TPM in a non-virtualized environment. In the following, we provide an overview of related research.

Berger et al. describe a virtualized TPM emulating TPM functionality in software, called vTPM [9]. In particular, each VM is provided its own vTPM with its own instance of (upper) PCRs. All upper PCRs are held in software and their contents may be signed by the hardware TPM. However, this does not provide the same level of security as storing measurements in hardware-protected PCRs since measurements held in software can be manipulated by an attacker once the system is compromised.

The available vTPM implementations [89, 101] of the Xen and QEMU virtualization solutions, respectively, are based on the above work and suffer from the same problems. In fact, Cucurull et al. [31] review the existing vTPM implementations for Xen and QEMU and conclude that they do not provide a level of security comparable to a non-virtualized solution.

In [46], England et al. try to reduce the complexity of approaches such as vTPM by not emulating the entire TPM interface in software. They utilize a para-virtualized approach that will pass through most of the functionality of a real TPM, but changes some aspects of the device interface. However, this approach suffers from the same problem as vTPM since (upper) PCRs are emulated in software and thus can be manipulated on a compromised system.

Feller et al. propose dcTPM [51], an architecture to multiplex several software-based TPMs, hardware TPMs, or a combination thereof. By multiplexing only hardware TPMs, the above issue of software-emulated PCRs can be solved. However, their approach does not scale very well. In fact, multiplexing cloud systems consisting of hundreds of VMs becomes infeasible in terms of technology (e.g., limited number of Field-Programmable Gate Array (FPGA) pins needed for multiplexing TPMs) and in terms of economy (e.g., hardware must be especially built with as many hardware TPMs as the (maximum) number of associated VMs).

In [166], Stumpf et al. propose a concept for enhancing a TPM to support hardware-based virtualization without the above scaling issues. This is achieved by employing a root-data structure that is only accessible at the hypervisor-level and a TPM-control structure that is used to dynamically swap TPM-context information of each VM in and out in an encrypted manner. Unfortunately, such a TPM is not available for production use.

Proskurin et al. propose seTPM [131] and Raj et al. propose fTPM [135]. Both approaches provide software-only TPM emulations, but they utilize supplemental hardware-based features to increase their level of security. With seTPM, the TPM functionality is implemented as an applet on a Java Card. The authors mention that the isolation capabilities of Java Card applets within secure elements may be used to manage multiple instances of seTPM in order to provide TPM functionality to multiple virtual machines, given sufficient memory resources of the secure element. The implementation of the firmware-based fTPM leverages ARM TrustZone. The authors show how fTPM can be used to build software systems with security guarantees similar to those of dedicated trusted hardware. However, the authors do not cover how to take advantage of fTPM in virtualized environments but we suspect the concept could be extended to support multiple virtual machines.

The approach presented in this thesis allows us to store and multiplex integrity measurements from an unlimited number of virtual machines in the actual hardware TPM. By leveraging the hardware-based capabilities of the TPM, we attain a high level of security while also making our approach relatively easy to deploy.

3.13 Summary

We have shown that it is possible to multiplex integrity measurements originating from arbitrarily many VMs with just a single standard TPM and only requiring one PCR. In contrast to existing approaches that emulate PCRs in software, our approach achieves a higher level of security since measurements, along with the mapping to their respective VMs, will always be stored in the hardware-protected PCRs of the TPM. We presented a remote attestation protocol for attesting the integrity of individual VMs. A crucial problem we had to solve in this context, was that our approach of sharing PCRs among VMs, inherently requires the disclosure of all measurements of all VMs. We overcame this by storing measurements in the PCR in a concealed manner. We additionally conceal a measurement's associated VM-ID, so as to prevent the collection of usage patterns. This enables us to

fully disclose the (concealed) contents of the PCR and to selectively reveal non-concealed measurements of individual VMs. However, this approach poses the risk of a MITM launching blinding attacks where the measurements and VM-IDs will be substituted with their corresponding concealed pairs in order to hide certain measurements of the attested VM. We presented an exhaustive list of blinding attacks and showed that our integrity validation algorithm protects against all of them. Finally, the experimental results of our proof of concept implementation show the practicality of our approach.

Chapter 4

Integrity Monitoring using Paravirtualized Filesystems

In Chapter 3, we demonstrated how to store and multiplex integrity measurements of arbitrarily many VMs with just a single standard TPM, thus building a secure foundation for a variety of virtualized environment use cases. However, in all scenarios, it is crucial to make sure that integrity measurements will be obtained in a reliable and secure way and that it is not possible for an attacker to manipulate the measurements before they reach the TPM. In particular, having a Measurement Agent (MA) reside within the realm of a supervised VM makes the MA susceptible to attacks carried out in this VM and thus to measurement manipulations.

In this chapter, we show how to solve this problem by developing a system that monitors and takes integrity measurements of multiple VMs from outside of the VMs and only requires a single MA located at the hypervisor-level. We present our approach of relocating a supervised VM's entire filesystem into the isolated realm of the host. In this way, we can enforce that all file operations (e.g., modifying an existing file or creating a new file) on the VM's filesystem must necessarily be routed through the hypervisor-level, and thus can be tracked and even be prevented. This guarantees secure and complete file integrity monitoring of VMs. The experimental results of our proof of concept implementation show the feasibility of our approach.

In Chapter 5, we will build upon Chapters 3 and 4 and explore how to enable system administrators and other users to remotely interact with the system developed in this chapter in a secure manner to protect the system's data integrity. We will achieve this by continuously authenticating users of the system based on their interaction behavior with touchscreen devices.

Parts of this chapter have been published in *Active File Integrity Monitoring Using Paravirtualized Filesystems* at the 5th International Conference on Trusted Systems (INTRUST) in 2013 [180].

The rest of this chapter is organized as follows. In Section 4.1, we motivate the need for virtualization-based integrity monitoring, potential weaknesses of other approaches, and present our contributions. Section 4.2 states the attacker model and our assumptions. Section 4.3 sketches our monitoring approach of relocating a supervised VM's entire filesystem into the isolated realm of the host. In Section 4.4, we introduce our system architecture. Section 4.5 explains how we monitor and analyze file operation requests. In Sections 4.7 and 4.8, we describe the techniques for enforcing file protection and detecting program execution, respectively. Section 4.9 details the autonomous software package installation and upgrading mechanism. Section 4.10 describes our proof of concept implementation. Section 4.11 gives the performance evaluation results. In Section 4.12, we evaluate the security of our approach. Section 4.13 discusses related work. Section 4.14 concludes this chapter.

4.1 Virtualization-based Integrity Monitoring

Protecting the integrity of file objects is a fundamental security objective for building trustworthy systems and for counteracting malware threats. A prominent example of achieving file integrity protection is the Host-based Intrusion Detection System (HIDS) Tripwire [82], which detects manipulations to filesystem objects by comparing their hash values to reference hash values in periodic intervals. However, the problem with Tripwire, OSSEC [16], and similar systems—including approaches based on Linux Security Modules (LSM), in particular, the four standard LSM modules SELinux [163], AppArmor [22], Smack [152], and TOMOYO [120]—is that critical security components (e.g., the monitoring components) are not isolated from the supervised system. This allows malware to attack and disable the monitoring components in order to conceal attack traces or to hide their presence altogether. Researchers have proposed architectures leveraging virtualization in order to isolate the critical security components from the supervised system. The supervised system is moved into a separate VM while the monitoring components are isolated and placed outside of the VM [61, 117]. The monitoring components employ techniques like Virtual Machine Introspection (VMI) [61] to inspect the supervised system and to potentially interpose on certain operations (e.g., write operations on critical files). This approach

prevents malware—like viruses, worms, and trojans—located in the VM from attacking and disabling the monitoring components.

A challenge in this context is that the external monitoring components now only have a rather coarse and abstract view of the supervised operating system's internal structures and high-level OS abstractions. This limitation introduced by the virtualization layer is called the *semantic gap* [25]. Operating system level semantics may be deduced from hardware-level state (e.g., physical memory pages and registers) and events (e.g., interrupts and memory accesses) in order to bridge the semantic gap [61]. Furthermore, researchers have proposed monitoring techniques where hooks are placed inside the monitored VMs in order to better cope with the complexity of interpreting guest OS specific structures and events and to gain a more detailed view [124, 134, 79, 182, 127, 128, 189]. However, malware can often circumvent such monitoring by tampering with the hooks and components placed inside the VMs, thus resulting in critical operations (e.g., critical file operations) not being noticed on the hypervisor-level.

In this chapter, we make the following contributions:

- We present an architecture where we relocate a supervised VM's entire filesystem into the isolated realm of the host. The only way of accessing and manipulating a VM's filesystem is by communicating with a privileged component located at the hypervisor-level which has exclusive access to the VM's filesystem. Therefore, it is guaranteed that all file operations on the VM's filesystem are necessarily routed through the hypervisor-level.

- We build upon this architecture to monitor all file I/O operations within a VM in real-time from "outside of the box" and to interpose and prevent them from happening. Our approach solves the aforementioned problem of having malware disable hooks in the VM as this would render the VM (and as such the malware itself) incapable of accessing or manipulating the VM's filesystem.

- We efficiently bridge the semantic gap and preserve all relevant file operation information by leveraging the paravirtualized Plan 9 filesystem protocol [178] for communicating between guest VMs and the hypervisor.

- We build upon and improve the work done in Chapter 3 to measure all executed binaries of all VMs and to store these measurements in

a single, multiplexed TPM. This allows for attesting the integrity of individual VMs in the course of a remote attestation.

- We enable regular users of VMs to autonomously install and upgrade software packages in a secure and controlled manner without the need of requiring the intervention of the administrator of the physical system. Additionally, and complementary to the autonomous approach, our proposed concept allows system administrators to actively enforce the upgrading of (outdated) packages running in VMs from the hypervisor-level.

4.2 Attacker Model and Assumptions

Our objective is to monitor the integrity of files on the VM's filesystem and to prevent critical file events and illegal file modifications by leveraging a paravirtualized filesystem. We assume a virtualized platform where attackers, particularly in the form of malware, have full access to their respective guest VMs.

We consider local and remote attacks against guest VMs as well as legitimate users of guest VMs that may compromise the guest VM and gain control of the guest user space and kernel.

We assume a correct implementation of the utilized 9P server such that a guest VM cannot compromise the 9P server by sending specially crafted requests, for example, requests causing a malfunctioning of the 9P server through exploitation of buffer overflows.

We do not consider direct remote attacks against the host, nor do we consider direct physical hardware attacks.

4.3 Monitoring of Guest VMs

The key aspect of our concept is that we relocate a guest VM's entire filesystem from the guest VM to the isolated realm of the host. We then grant only a privileged component, located at the hypervisor-level, exclusive access to the guest filesystems. This means that for all guest VMs, the only way of accessing and manipulating their own filesystems is by communicating with this privileged component located at the hypervisor-level. Therefore, it is guaranteed that all file operations on a VM's filesystem (e.g., write operations on critical files) are necessarily routed through the hypervisor-level. This allows us to monitor all such file operations and to prevent them before they actually happen. Consequently, this enables us to protect against

malware attacks trying to infect the system. We protect against malware manipulations of a guest VM's filesystem where the malware enforces its reactivation even after a restart of the guest VM. By being able to prevent such threats, we can guarantee that the guest VM starts in a trustworthy state, free from malware. Our approach also makes sure that it is impossible for an attacker to bypass the hypervisor (and as such circumvent the monitoring), even in the event of a completely compromised VM—since otherwise there is no way of accessing the VM's filesystem. This is an advantage over other approaches (e.g., [124] and [134]) where disabling hooks in the VM still allows for manipulation of filesystem objects.

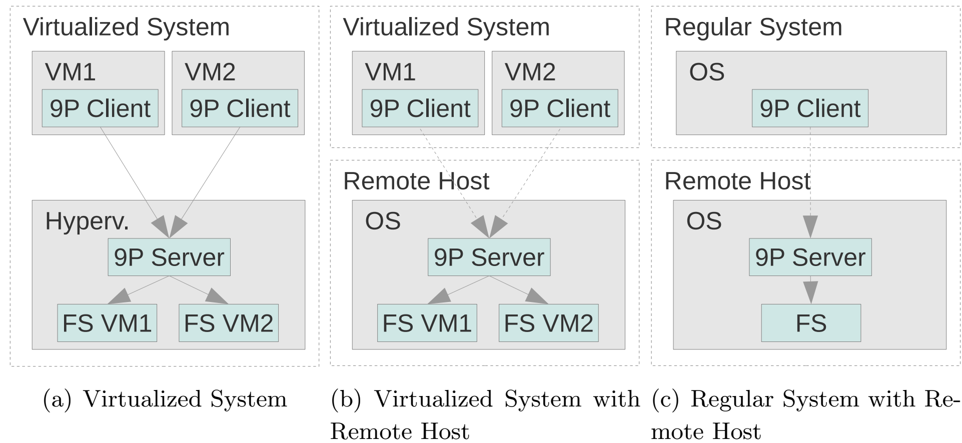
4.3.1 Filesystem Relocation Mechanism

For our concept, we make use of the Plan 9 filesystem protocol *9P* in order to relocate a guest VM's filesystem to the host. The 9P protocol is designed as a distributed filesystem protocol that may be used over the network and which operates on a file-based granularity. The client-server protocol uses messages that reflect ordinary file operations, for example, messages originating from read or write system calls. In the following, we give the rationale for choosing 9P as a means of achieving the described filesystem relocation of guest VM filesystems.

9P is a minimalist message-oriented filesystem protocol which has been designed with simplicity in mind. The core protocol consists of only twelve basic operations which can be initiated by a 9P client [178]. Therefore, 9P's complexity is easier to handle than more full-fledged protocols like NFS [150] or CIFS [28]. This also results in a relatively small code base of the 9P implementation, thereby reducing the likelihood of bugs which may be exploited by an attacker. Another reason why 9P is well suited for our concept is its protocol independence. In fact, it can be used over any reliable, in-order transport [178]. This allows us to use it in a flexible manner both on the same machine and over the network.

4.3.2 Relocation Scenarios

Figure 4.1 depicts three filesystem relocation scenarios which we will describe in the following. Figure 4.1a shows the main scenario we consider in this chapter and is also the scenario described above. In this virtualized system, the filesystems of the VMs are not directly available to the VMs but relocated to the hypervisor-level such that the hypervisor is able to monitor and analyze all file operation requests. A 9P client resides in each guest VM and cooperates with the 9P server located at the hypervisor-level. The actual



(a) Virtualized System (b) Virtualized System with Remote Host (c) Regular System with Remote Host

Figure 4.1: Filesystem relocation scenarios. In the first case, the filesystems of the virtual machines are located on the same machine. In the other cases, the filesystems are located on a remote machine.

communication between the 9P clients and the 9P server is done over virtio [145], which is the de-facto standard of a paravirtualizing framework for Linux. This allows us to efficiently bridge the semantic gap and to preserve all relevant file operation information. The concept and architecture will be explained in more detail in this chapter. The detailed system architecture is shown in Figure 4.2.

An advantage of our filesystem relocation approach using the Plan 9 filesystem protocol 9P (cf. Section 4.3.1 above) is that the filesystems of the monitored VMs do not necessarily have to be located on the same machine but may be used in a distributed fashion and moved to another physical machine—or even multiple machines. This is depicted in Figure 4.1b and Figure 4.1c. In Figure 4.1b, we consider a virtualized system with one or more virtual machines. Each virtual machine contains a 9P client that communicates with a 9P server—located on a remote host—over the network. In Figure 4.1c, we consider a regular (non-virtualized) system containing a 9P client. Similar to the previous scenario, the filesystem of the regular system is moved to a remote host and the 9P client communicates with the remote host's 9P server in order to access and operate on the filesystem.

The architectural arrangements in Figure 4.1b and Figure 4.1c can be particularly interesting for use cases in the context of the Internet of Things (IoT) and Automotive. In particular, relocating the filesystem to a remote host not only allows monitoring systems but also to support devices with no or only a very limited amount of storage. This is often the case with

minimalist IoT devices. Furthermore, it can also be applicable in the case of a vehicle's Electronic Control Units (ECUs). For example, in order to monitor certain security-critical ECUs within a vehicle, their filesystems could be relocated to the central gateway (GW) of the vehicle. The GW is (among others) often responsible for security-related tasks like monitoring certain components within the vehicle or controlling network traffic. Thus, the GW could be extended to realize a similar kind of monitoring of ECUs as is the case with the virtualized system in Figure 4.1a, where the hypervisor monitors the virtual machines. In addition, the GW can be equipped with a TPM which can be utilized for securely storing integrity measurements. This mechanism will be described in Section 4.6.

In order to decide whether a 9P request originating from an IoT device or an ECU will be granted or denied, a special agent located on the remote host implements a policy-based access control approach. The details of our policy-based approach will be explained in Section 4.7. In the case of IoT devices, the remote host may be a backend system located within the secure realm of the IoT operating company. In the automotive case, the remote host may be the GW (as described above) or also a backend system located at the OEM or Tier1. However, implementing the policy-based access control directly on the GW has the advantage of not requiring an online connection to the backend as well as a significantly lower latency (which can be critical for certain ECUs). Finally, the policy rules on the GW can be updated and extended by having the backend push the policy updates to the vehicle's GW. The performance aspects of relocating the filesystems to a remote host as described in this section, will be evaluated and discussed in Section 4.11.1.

4.4 System Overview

Our paravirtualized monitoring architecture is shown in Figure 4.2. In general, the hypervisor runs one or more guest VMs, which are subject to monitoring. Each guest VM contains a 9P client that translates ordinary file operation requests originating from within the VM to 9P request messages. These messages will be forwarded by the respective 9P client to the 9P server located in the realm of the hypervisor. The 9P server has exclusive access to the filesystems of the guest VMs. The guest filesystems are located on the filesystem of the host. The 9P server processes the 9P requests accordingly, for example, by reading a file (and providing it to the 9P client) or by writing to the filesystem.

There are four components responsible for monitoring and enforcing file integrity of the guests. These monitoring components are isolated from the

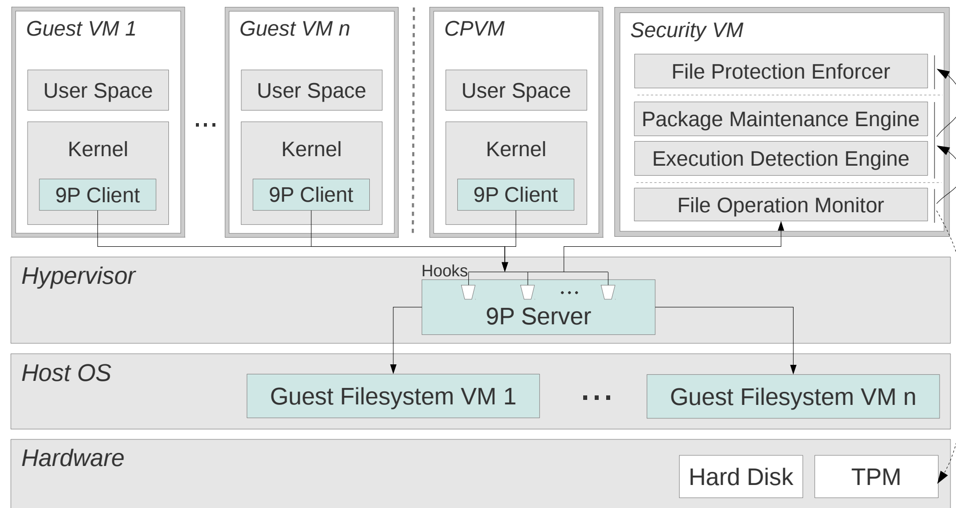


Figure 4.2: Paravirtualized monitoring architecture with externalized guest filesystems. Each VM's 9P client communicates with the 9P server, which has exclusive access to the VM's filesystems. The components responsible for monitoring and enforcing file integrity of the guests are isolated in a special security VM.

guest VMs (and the hypervisor) in a special security VM (cf. Figure 4.2). We place hooks in all relevant parts of the request handlers of the 9P server in order to inform the monitoring components of all relevant file operations. This enables the security VM to monitor all requests originating from a VM's 9P client trying to access its guest filesystem. The components process the 9P requests and decide—based on an access control policy—whether a request will be granted or denied.

4.4.1 File Operation Monitor

The File Operation Monitor (FOM) receives and analyzes all hooked 9P request messages from the 9P server. Relevant requests will be forwarded to EDE and PME (see below). The details will be described in Section 4.5.

4.4.2 Execution Detection Engine

The Execution Detection Engine (EDE) detects when a guest VM is trying to execute a file based on a heuristic approach which is based on recognizing distinct sequences of 9P requests. Execution of files (possibly along with certain write operations, see Section 4.6 for the details) will be securely

recorded by storing a corresponding integrity measurement (hash value) in a TPM [173]. We take advantage of the work done in Chapter 3 to store the integrity measurements in a single, multiplexed TPM. The details will be described in Section 4.8.

4.4.3 Package Maintenance Engine

The Package Maintenance Engine (PME) detects when a guest VM is trying to install, remove, upgrade, or downgrade software packages, and handles it by utilizing a special VM, called the Complementary Privileged Virtual Machine (CPVM). The details will be described in Section 4.9.

4.4.4 File Protection Enforcer

The File Protection Enforcer (FPE) decides whether a 9P request will finally be granted or denied based on policy rules. The details will be described in Section 4.7.

4.5 Monitoring and Analyzing File Operation Requests

The File Operation Monitor (FOM) is responsible for analyzing 9P request messages forwarded by the 9P server. In particular, FOM scans for all *critical requests* of the utilized 9P2000.L[1] protocol [62]. A request is considered critical if it has the potential to impact the integrity of the guest's filesystem. Table 4.1 lists all critical 9P requests that are handled by FOM along with a description of their potential impacts.

Note that Table 4.1 does not list the 9P `read` request since it cannot be used to affect a file's integrity. However, `read` requests still play an important part in our concept as they occur as a distinct sequence of 9P requests whenever a file in the guest VM is going to be executed. Since the 9P filesystem protocol does not incorporate a dedicated `execute` request itself, we take advantage of this sequence of `read` signature requests in order to come up with a heuristic to detect the execution of files. The details will be described in Section 4.8.

[1] 9P2000.L includes the core 9P2000 requests as a subset

9P Request	Potential Impact
`write`	Writing new files or modifying the content of existing files, e.g., altering configuration files or executables
`rename`, `renameat`	Moving files, thus effectively deleting them from one location within the filesystem and possibly replacing other files with the content of the renamed file
`remove`, `unlinkat`	Removing files or directories, e.g., changing the behavior of programs by deleting their configuration files or hiding traces by deleting log files
`lcreate`, `mkdir`	Creating new files or directories; may be misused to truncate existing files
`link`, `symlink`	Creating a hardlink or symbolic link, e.g., creating a link in a directory like `/bin` to a malicious executable in `/tmp` (where the creation of arbitrary files may be allowed)

Table 4.1: Critical requests of the Plan 9 9P2000.L protocol

4.5.1 Shadow Copy Write

The 9P `write` request requires further consideration. A special case of the `write` request is that it may exceed the message size of the 9P client or the 9P server implementation (or both). The reason is that the entire (to be written) payload data has to be sent from the 9P client to the 9P server. In such cases, the 9P client splits up a `write` request $w[f, d]$ (containing the payload data d for a file f) into several sub-requests $w_1[f, d_1], \ldots, w_n[f, d_n]$ [62]. This poses a problem for monitoring `write` requests because FPE may not be able to decide upon the partial information of a sub-request $w_i[f, d_i]$ (in particular, the first sub-request $w_1[f, d_1]$) on whether the overall request $w[f, d]$ should be granted or not. In particular, if FPE only allows a file f to be written if its future content (i.e., the content of f after applying the `write` operation $w[f, d]$) matches a certain hash value (cf. Section 4.7), knowledge of the entire future content of f is required in order to be able to calculate the hash value of f. Note that in this regard, it is not sufficient to only consider the payload data d. The reason is that a `write` request may only partially write a file f. In this case, the payload data d differs from the content of the resulting file f.

We solve this problem by introducing a technique called *shadowing*, which works in three phases:

1. FOM detects a `write` sub-request $w_1[f, d_1]$ by inspecting the request's header data. If f already exists on the guest's filesystem, FOM creates a *shadow copy* f' with the same content as f. If f does not exist, FOM creates an empty file f'. The shadow copy f' is located outside of the guest's filesystem and only accessible by FOM. Depending on the size of f, and possibly other factors (e.g., hardware and performance constraints), the shadow copy may be kept entirely in RAM.

2. FOM applies the sub-request $w_1[f, d_1]$ along with all other corresponding sub-requests $w_2[f, d_2], \ldots, w_n[f, d_n]$ exclusively to the shadow copy f'. When all sub-requests $w_1[f, d_1], \ldots, w_n[f, d_n]$ have been processed (which is detected by a terminal `clunk` or `fsync` operation [178]), FOM signals to FPE that there is a new `write` request $w[f, d]$ and passes a pointer to f'.

3. FPE is then able to calculate the hash value of f', which resembles the potential future content of f, and to finally decide whether the overall `write` request $w[f, d]$ should be granted or denied. If it is granted, the 9P server eventually gets signaled to allow and process all sub-requests $w_1[f, d_1], \ldots, w_n[f, d_n]$. Finally, the shadow copy f' gets discarded.

4.6 Secure Storage of Integrity Measurements

We build upon and improve the work done in Chapter 3 to measure all executed binaries of all VMs and store these measurements in a single, multiplexed TPM. In this way, we are able to extend the chain of trust in virtualized environments up to the respective application layers of the guest operating systems and to effectively extend important concepts like the TCG-based Integrity Measurement Architecture (IMA) [148], while at the same time protecting the integrity measurements through the high security properties of a (hardware) TPM. In Section 3.4, we described two approaches for obtaining integrity measurements. In the first approach, a Measurement Agent (MA) runs in each VM and monitors the execution of supervised files, calculates according integrity measurements, and propagates them to the hypervisor-level. In the second approach, the MAs are not located within the VMs themselves but the monitoring of supervised files of *all* VMs is rather conducted by an MA located at the hypervisor-level. The second approach has the advantage that only a single MA is

required for monitoring the execution of files of all VMs from “outside of the box” and for storing integrity measurements in the TPM. Additionally, this prevents attackers from tampering with the monitoring and measuring components, respectively, since they are out of the reach of the guest VMs. Because of these advantages, we conceptually took advantage of a Measurement Agent (MA) implementing the second approach for obtaining integrity measurements in Chapter 3 as depicted in the system architecture in Figure 3.1. However, there we just assumed the existence of such an MA as the focus of Chapter 3 was on the secure and privacy-aware multiplexing of integrity measurements. We are now able to fill this gap by leveraging the paravirtualized integrity monitoring technique developed in this chapter.

In addition to measuring *execution operations* on files as described above, our concept is also capable of monitoring file modifications (e.g., changes to configuration files) by supervising *write operations* to these files. This allows us to take integrity measurements of the modified file contents and to securely store the measurements in the multiplexed PCR of the TPM, so as to proof that the respective files have been altered in a certain way. In the context of a remote attestation, this integrity measurement information— along with the measurement information about executed files— can then be provided to a verifier who uses it to assess the trustworthiness of the attested virtual machine. For example, a verifier may only accept particular manifestations of file contents for a configuration file responsible for specifying the cryptographic algorithm that is used for performing certain encryption operations, and reject all other file contents. This may allow the verifier to determine whether the attested system uses, for example, one of the strong ciphers required by the verifier for deeming the system trustworthy. The file contents are uniquely identified[2] by their respective hash values, and the verifier is assumed to be in knowledge of the respective hash values because the expected potential file contents (the strong ciphers, in the example) are known to him and thus he is able to calculate the hash values himself. The verifier is then able to compare the hash values (integrity measurements) provided as part of the remote attestation with his own list of reference hash values. If a provided hash value is not in the list of reference hash values, the corresponding file is considered untrustworthy. We note that, in principle, the attesting system could additionally provide information about the corresponding contents of hash values that are not known to the verifier (e.g., a new strong cipher, in the example). This would allow the verifier to confirm that the file content indeed matches the hash value submitted

[2] With very high probability.

in the remote attestation by recalculating the hash value (based on the file content). However, this may involve manual intervention of the verifier in order to understand the semantics of the received file content and to assess whether it is considered trustworthy, and as such may not be feasible for many use cases.

Finally, we note that the work developed in Chapters 3 and 4 of this thesis provides the conceptual and technical groundwork for realizing the described attestation scenarios. However, in this thesis we do not focus on developing concrete policy rules for verifiers in order to assess the trustworthiness of attested VMs (along with the derivation of appropriate actions and responses) as they strongly depend on a multitude of factors and must be determined based on the actual use cases.

4.7 Enforcing File Protection

The File Protection Enforcer (FPE) is responsible for deciding whether a 9P request will be granted or denied. Our implemented mechanism resembles a Rule Based Access Control (RBAC) approach[3] [27]. In particular, the decision making is based on Access Control Lists (ACLs) that define which filesystem operations are allowed within guests and which ones are prohibited. FPE implements a default closed policy [77], also known as whitelisting [156], that prohibits all filesystem operations within a VM unless an operation is explicitly granted by an ACL.

For each guest VM there exist zero or more ACLs. An ACL defines for a given file f whether certain operations on f are allowed or denied. This, in turn, is realized through one or more Access Control Entries (ACEs) associated with a given ACL. ACLs may be realized through various approaches such as using extended file attributes or utilizing a file path based approach. In our prototype implementation (cf. Section 4.10), we take advantage of a file path-based lightweight database approach for administering ACLs.

4.7.1 Policy Predicates and Request Mapping

We define a minimal set of four predicates that may be used to construct an ACE. All predicates evaluate to either true or false. The predicates are:

[3] Not to be confused with Role Based Access Control.

9P Request	Predicate Policy Check
`write(f,d)`	$f' \leftarrow w[f,d] : W(f) \wedge H(f')$
rename(f_1,f_2), renameat(f_1,f_2)	$W(f_2) \wedge D(f_1) \wedge H(f_1)$
`remove(f)`, `unlinkat(f)`	$D(f)$
`lcreate(f)`	$W(f)$
link(f_1,f_2), symlink(f_1,f_2)	$W(f_2) \wedge H(f_1)$
`exec(f)` (*)	$E(f) \wedge H(f)$

(*) virtual request

Table 4.2: Critical requests mapped to policy checks using only predicates.

$W(f)$: (partial) writing of file f allowed?
$D(f)$: deletion of file f allowed?
$E(f)$: execution of file f allowed?
$H(f)$: hash of the content of file f matches a reference hash value?

The objective of exclusively using this minimal set of predicates in the ACEs, is to abstract from the actual 9P requests and to come up with simpler, more generic ACEs. This has the advantage that one does not have to create ACEs for each specific 9P request. Instead, it is only required to define for a file f whether writing $W(f)$, deletion $D(f)$, and execution $E(f)$ is allowed or denied (the latter of which is the default)—possibly in conjunction with reference hash values that have to be matched ($H(f)$). In particular, a file may be associated with a list of one or more *reference hash values* $\langle h_1, \ldots, h_n \rangle$. The predicate $H(f)$ evaluates to true if and only if the content of f matches one of the hash values h_i or if the list of reference hash values contains the wildcard character "$*$". Otherwise, $H(f)$ always evaluates to false.

For example, an ACE for a file f may define that writing of f is allowed (W predicate) if the resulting content matches one of several reference hash values (H predicate). Such an ACE may then evaluate to true not only for 9P `write` requests but also for `rename`, `renameat`, `lcreate`, `link`, and `symlink` requests, respectively, as will be explained in the following.

For the actual policy enforcement, the FPE internally maps all critical 9P requests (cf. Table 4.1) to corresponding policy checks using only these predicates. The mapping is shown in Table 4.2 (for clarity, we only illustrate the policy checks for files and omit the checks for directories). If the overall expression of such a policy check evaluates to true, the respective 9P request will be granted by FPE. Otherwise, it will be denied. Note that a `write` request $w[f, d]$ (which might be a partial write) will first be applied to a

temporary file f' (denoted by $f' \leftarrow w[f, d]$ in Table 4.2). This is similar to shadowing as described in Section 4.5.1. If the content of the resulting file f' matches a valid reference hash value, $H(f')$ evaluates to *true*. Also note that `exec` is not an actual 9P request but a *virtual request* which is propagated by EDE. This is explained in Section 4.8. Finally, note that for the predicates we do not use any supplemental information (e.g., user ID) originating from within a VM. The reason is that this information is not trustworthy as it may be manipulated.

4.7.2 Package Policy Rules

In addition to the predicates described above, we also define predicates to determine which software package maintenance operations may be autonomously issued by legitimate guest VM users (cf. Section 4.9). The predicates are:

$P_i(p)$: installing, upgrading, or downgrading package p allowed?
$P_r(p)$: removing package p allowed?
$P_h(p)$: hash of the package p matches a reference hash value?

Whenever PME detects an installation, upgrading, or downgrading attempt of a package p (cf. Section 4.9), respectively, it is propagated to FPE which, in turn, will check whether the predicate $P_i(p)$ evaluates to true. Furthermore, the predicate $P_h(p)$ may be used—analogously to $H(f)$ as described above—to restrict the installation, upgrading, and downgrading of a package p to packages that match a reference hash value. This allows us to selectively permit only certain packages (and package versions) while prohibiting others, e.g., older versions with known vulnerabilities. For removing attempts of p, FPE will check whether the predicate $P_r(p)$ evaluates to true.

4.7.3 Policy Example

Table 4.3 shows a simplified example for a machine with three guest VMs to illustrate how the predicates may actually be utilized. In the table, each row represents a policy rule. The first column defines the virtual machine the policy rule is associated with. The label should uniquely identify the virtual machine. The second column lists the policy rule's directory path (row 1), file path (rows 2 to 4), or package name (row 5). The third column defines the allowed operations on the directory, file, or package. This is done by stating the respective predicates as described in Sections 4.7.1 and 4.7.2. The fourth column defines a list of reference hash values (or the wildcard

VM	Path or Package	Predicates	Reference Hash Values
1	`/tmp/*`	W, D	$\langle * \rangle$
1	`/etc/ssh/cipher`	W	$\langle a3b1\ldots,\ 4c17\ldots \rangle$
2	`/etc/crontab`	W	$\langle * \rangle$
3	`/bin/web-server`	E	$\langle * \rangle$
3	`ssh-server`	P_i	$\langle 018f\ldots,\ c668\ldots,\ d4b2\ldots \rangle$

Table 4.3: Policy rules example utilizing the described predicates and reference hash values for a machine running three virtual machines.

character "$*$") that have to be matched in order for the respective predicates to evaluate to true such that the respective operation will be allowed.

The first policy rule defines that in VM 1, write operations (as well as create, rename, and link operations, cf. Table 4.2) and delete operations in directory `/tmp` are permitted. Furthermore, since the list of reference hash values uses the wildcard character "$*$", *all* write operations in `/tmp` are allowed. Such policy rules can be used (by the system administrator) to exclude directories and files where operations such as writing and deleting directories and files are considered to not be crucial. In practice, this may depend on several factors, for example, the actual use case, the required security level, or the virtual machine. The second policy rule states that the (config) file `/etc/ssh/cipher` may be written but only if the hash value of the resulting file content matches the reference hash value "$a3b1\ldots$" or "$4c17\ldots$". In this example, the configuration file `/etc/ssh/cipher` contains the cryptographic algorithm that is used for performing the encryption (and decryption) of traffic when using the Secure Shell (SSH) protocol. If the system administrator, for example, only wants to allow the cipher `aes` or `3des`, he can use the respective reference hash values matching the file contents when using these ciphers (in this example, "$a3b1\ldots$" and "$4c17\ldots$", respectively). In this way, the system administrator is able to enforce the usage of strong ciphers and to effectively prohibit the usage of weak ciphers. The third policy rule allows arbitrary write operations to the file `/etc/crontab` in VM 2. The fourth policy rule allows the execution of `/bin/web-server` in VM 3 (but does not allow writing or deleting it). The fifth policy rule allows installing, upgrading, and downgrading the package `ssh-server`. Additionally, the system administrator is able to restrict the `ssh-server` package resulting from these operations to the ones matching the reference hash values "$018f\ldots$", "$c668\ldots$", or "$d4b4\ldots$". In this way, it is possible for the administrator to enforce the usage of certain `ssh-server`

package versions (matching the reference hash values) and to effectively prohibit obsolete versions or versions with known vulnerabilities.

Finally, note that in the context of a remote attestation, a verifier is able to additionally check reference hash values in order to assess the trustworthiness of the attested virtual machine (cf. Section 4.6). For example, a verifier may additionally require that the executed `/bin/web-server` in VM 3 matches a particular reference hash value (e.g., a specific version of the web server the verifier only trusts), whereas the policy rule in Table 4.3 (row 4) per se does not impose any restrictions on the executed version of `/bin/web-server`. This allows the verifier to implement flexible, supplemental policies on top of the policy rules used on the attested machine.

4.8 Detecting Program Execution

The detection of executed programs from outside of the guest VMs through EDE is not straight forward due to the fact that 9P does not distinguish between reading a file and executing a file. Instead, in both cases a read request is sent by the 9P client and only the VM decides afterwards whether the read file will be executed. Note that we cannot just extend the 9P clients (and server) such that they distinguish between read and execute requests (e.g., an executable-bit). The reason is that this information would not be trustworthy since an attacker may tamper with it (e.g., setting the executable-bit from 1 to 0) once the VM is compromised. Therefore, we incorporate EDE which is able to detect the execution of a file within a VM by utilizing a heuristic approach. EDE is protected from the aforementioned attacks since it is located in the security VM (cf. Figure 4.2) and monitors the VMs from “outside of the box”, without relying on auxiliary (untrustworthy) information sent from the VM.

Whenever a program is going to be executed within a VM, there will be a distinct sequence of preceding Plan 9 requests in a defined chronological order, as described in the following. EDE recognizes this sequence of *signature requests* and deduces which file is intended to be executed. FPE may then grant or deny the execution based on policy rules as described in Section 4.7.

For the execution detection, we consider the Executable and Linking Format (ELF) [170], which is the standard binary format for executables on many Unix-like operating systems, including Linux. The heuristic for detecting the execution of ELF files under Linux, consists of the following signature requests (in their chronological order of occurrence):

1. The `execve` system call first reads 128 bytes to determine the binary type of a file f. Consequently, EDE scans for the corresponding 9P read requests.

2. The ELF loader of the Linux kernel invokes the function `kernel_read`, which reads 224 bytes from f, starting from offset 52.

3. A subsequent invocation of `kernel_read` reads 19 bytes from f, starting from offset 276, which gets treated as the path to an interpreter [170].

The above signature requests are usually followed by multiple read requests that attempt to map the entire file f into memory. Note that EDE is also able to detect the loading of ELF libraries, which generate signature requests similar to that of executed binaries.

4.9 Autonomous Software Package Installation and Upgrade

Another key feature of our approach is that it is possible for legitimate users of guest VMs to autonomously install, remove, upgrade, and downgrade software packages without the need of any manual intervention by the administrator of the physical system. However, these *package maintenance operations* are not allowed to be done in an arbitrary manner, but all such operations have to adhere to the policy rules enforced by FPE as described in Section 4.7. Also note that it is not possible for a guest VM user to directly manipulate the package contents as they are write protected. This prevents illegal modifications of the guest VM by attackers—which includes legitimate but maliciously acting VM users.

Another advantage of our software package installation and upgrading mechanism is that in addition to the above autonomous approach, PME is also able to actively enforce the upgrading of (outdated) packages running in VMs from the hypervisor-level. This allows the system administrator to keep critical software running in the VMs up to date, thus increasing the overall system security by reducing the likelihood of an attacker exploiting software packages with known vulnerabilities.

The work flow for installing, removing, upgrading, and downgrading software packages is depicted in Figure 4.3 and will be described in the following.

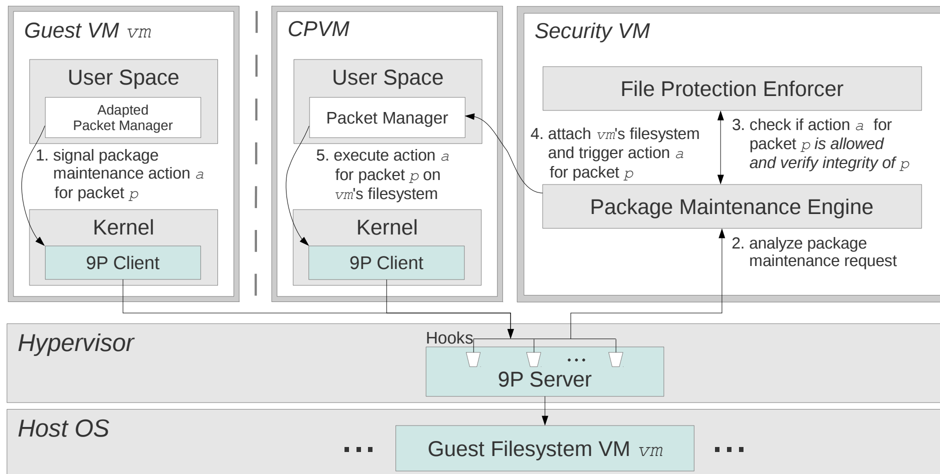


Figure 4.3: Installation and upgrading of packages via CPVM. First, a VM signals a package maintenance request. PME and FPE analyze the request by checking the package integrity and permissions. Finally, the CPVM may execute the request by operating directly on the VM's filesystem.

4.9.1 Signaling of Package Maintenance Request

First, a legitimate user of the guest VM executes the package manager within the VM with the corresponding maintenance action a (and parameters) for a package p (step 1 of Figure 4.3). The request is forwarded by the 9P client to the 9P server. The Package Maintenance Engine (PME) located in the security VM catches and analyzes the package maintenance request (step 2). In this regard, it is important to note that PME considers all information gathered from the guest VM as untrustworthy. This means that even if an attacker compromised the guest VM, it would still not be possible for him to use the package manager to send fake information in a way that would allow the circumvention of the policy rules or the malfunctioning of any other security-critical component outside of the guest VM.

4.9.2 Checking Package Integrity and Permissions

PME sends a query to FPE in order to determine whether p is a known and valid package on which the requested action a may be applied (step 3). Hence, FPE first checks if the action a on package p is allowed for the respective VM by evaluating the P_i and P_r predicates of the corresponding ACE. Afterwards, FPE verifies the integrity of the package p by evaluating

the $P_h(p)$ predicate of the corresponding ACE as described in Section 4.7. The usage of reference hash values allows us to selectively permit only certain packages—and package versions—while prohibiting others (like older versions with known vulnerabilities) that may otherwise be exploited by an attacker to compromise the system. If the hash value is not valid, the maintenance process is aborted and an error is signaled to the package manager of the guest VM.

4.9.3 Executing Package Maintenance Request

The package maintenance process for all guest VMs is executed in a special VM, called the Complementary Privileged Virtual Machine (CPVM). The CPVM runs in parallel to the guest VMs and has exclusive permission to install, remove, upgrade, or downgrade packages of all VMs. A key feature of the CPVM is that it operates—via the 9P protocol—on the same filesystem (located in the host) as the guest VM *vm* that triggered the respective package maintenance request. This is achieved by attaching *vm*'s filesystem (on the fly) to CPVM, for the duration of the package management process (step 4). In this way, all package management operations done by CPVM (step 5) are immediately visible to *vm*, and vice versa. This prevents synchronization problems and guarantees that both VMs always operate on the same state of the VM, e.g., consistent information on which packages are installed, their package versions, accompanying configuration file settings, and so on. Note that the guest VMs only require minimal and non-security critical user space modifications of the package management tools (cf. Section 4.10.1) and no kernel modifications.

4.9.4 CPVM Rationale

In the following, we justify the execution of the package maintenance operations within CPVM as opposed to executing them in the guest VM itself. The latter case could be achieved by having FPE properly adjust the policy rules such that file operations like creating, deleting, or modifying files belonging to a certain package would be temporarily permitted for a certain VM. However, many modern package managing tools also allow packages (e.g., Debian packages, as used in our prototype implementation in Section 4.10.1) to contain script files that will be executed before and after a package maintenance operation, respectively. Parsing these script files (which may contain arbitrarily complex commands) and extracting their complete semantics in order to be able to have FPE temporarily grant the corresponding file operations is a highly complex task. Possible workarounds

include disallowing such scripts or imposing certain constraints on their contents. However, this would prevent taking advantage of real-life packages as shipped with modern Unix-like operating systems. Our CPVM approach solves the aforementioned problems, yet it is fully compatible with full-fledged Unix-like operating systems, e.g., Linux distributions such as Debian.

Note that our approach does not require to suspend or pause a guest VM *vm* while CPVM is executing its software management operations on *vm*'s filesystem but both VMs can run in parallel. This is due to the fact that both VMs communicate with the same 9P server—which deals with the correct synchronization of 9P requests. As such, the functioning of *vm* and CPVM is comparable to two (especially isolated) processes operating on the same filesystem within the realm of an ordinary operating system.

4.10 Prototype Implementation

We have implemented a proof of concept using the Native Linux KVM Tool (NLKVM) [45], version 3.1.rc7, with enabled KVM full virtualization support. In contrast to QEMU-KVM [84, 8], NLKVM has the goal to provide a clean, from-scratch, lightweight KVM host tool with only the minimal amount of legacy device emulation. NLKVM ships with a 9P file server utilizing the virtio framework [145] for communicating with the 9P clients residing in the guest VMs. The 9P client functionality is provided by the v9fs client of the Linux kernel [91], which supports both the standard 9P2000 protocol and the extended 9P2000.L protocol, the latter of which we use.

Our host system runs Ubuntu 12.10. Each guest VM runs Debian 6.0 with Linux kernel 3.5.0 and enabled virtio and 9P support. The Linux guest kernel images reside on the host filesystem and will be passed as a parameter to NLKVM whenever a new VM is started. The security VM and CPVM also run Debian 6.0 with Linux kernel 3.5.0. The attached guest filesystem of CPVM is passed to NLKVM as a reference to a symbolic link. PME redirects this symbolic link dynamically to other guest filesystems as required by package maintenance requests.

The 9P server hooking functionality is realized by patching all relevant request handlers of the 9P virtio implementation so that FOM gets signaled and forwarded all required information. FOM and EDE are implemented in C. PME and FPE are implemented using a combination of Python scripts and shell scripts. Furthermore, FPE utilizes SQLite3 for efficiently managing the policy rules.

4.10.1 Installation and Upgrading of Packages

As mentioned, the guest VMs run Debian, which ships with the package management tool `dpkg` [116]. Since we do not allow guests to directly install, remove, upgrade, or downgrade packages on their own (cf. Section 4.9), we replace the user space tool `dpkg` with our own version `dpkg`^{R} which forwards all package maintenance requests to PME via the 9P protocol. We take advantage of regular 9P requests and have PME treat them specially in order to pass the information of package management action, parameters, and package name. This approach allows us to use regular 9P protocol implementations without the need of modifying them. In particular, we utilize the 9P `mkdir` request (cf. Table 4.1) because it allows us to transfer all required information. PME parses the request and queries FPE on whether the action for package p is allowed and whether p is a valid package. If the request gets granted by FPE, PME creates a new corresponding *job* by placing a file in a special directory which is only accessible by CPVM. The job file contains the respective command that will be executed by the privileged CPVM as soon as CPVM gets scheduled by PME. PME attaches the filesystem of the respective guest VM to CPVM and schedules CPVM which, in turn, detects the new job and executes it. The return value of the utilized 9P `mkdir` request can be used to report the status of the package maintenance request to the requesting VM. In particular, upon successful completion of the job, PME grants the 9P `mkdir` request in order to signal to `dpkg`^{R} that the package maintenance request has been successfully executed.

Finally, we note that it is sufficient to only replace `dpkg` as described above in order to additionally yield 9P-protocol-aware versions of other package management tools like `apt`, `aptitude`, and `synaptic`. The reason is that all of these tools utilize the shared library `libapt-pkg` [186] which provides the common functionality for managing packages such as installation and removal of packages. The library `libapt-pkg`, in turn, takes advantage of `dpkg` itself for executing low-level operations. Therefore, by replacing `dpkg` with `dpkg`^{R} the 9P-protocol-awareness propagates to all of the aforementioned package management tools.

4.11 Performance Evaluation

We assess the performance of our prototype implementation by measuring its write and read performance, and by comparing the results to three other environments. The testing hardware consists of

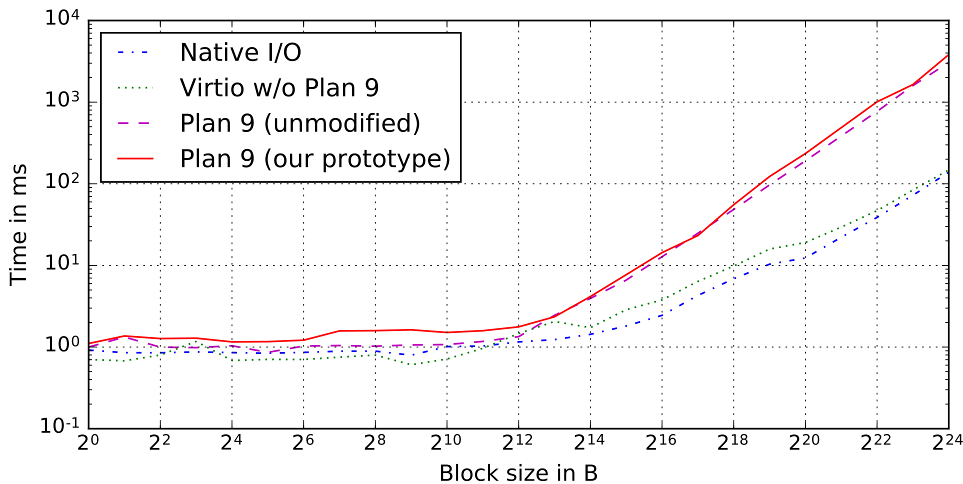


Figure 4.4: Write performance for different environments. The time to write data is given as a function of the block size. Our prototype performs similarly to the other examined environments for block sizes of up to approximately 4kB (2^{12}B).

- PC with an Intel Core i7-2640M 2.8GHz CPU,
- 4 GB RAM,
- Intel SSDSA2BW160G3L solid-state drive,
- ext4 filesystem with a block size of 4kB.

Figure 4.4 and Figure 4.5 show our testing results of the write and read performance benchmarks, respectively. We conducted the write and read operations with block sizes from 1B of up to 16MB (2^{24}B). For the experiments, we disabled caching.

The write performance is depicted in Figure 4.4. The time (in ms) to write data is given as a function of the block size (in bytes). All four examined environments—native I/O, virtio block without 9P, virtio block with unmodified plain 9P, and virtio block with modified 9P (our prototype)—perform similarly up to block sizes of approx. 4kB (2^{12}B). For larger block sizes, the Plan 9 environments perform worse than native I/O and virtio block. However, in our usage scenario such larger block sizes are negligible since I/O operations are usually done in block sizes of typical filesystems. These block sizes normally lie in the range of 512B to 4kB—the latter of

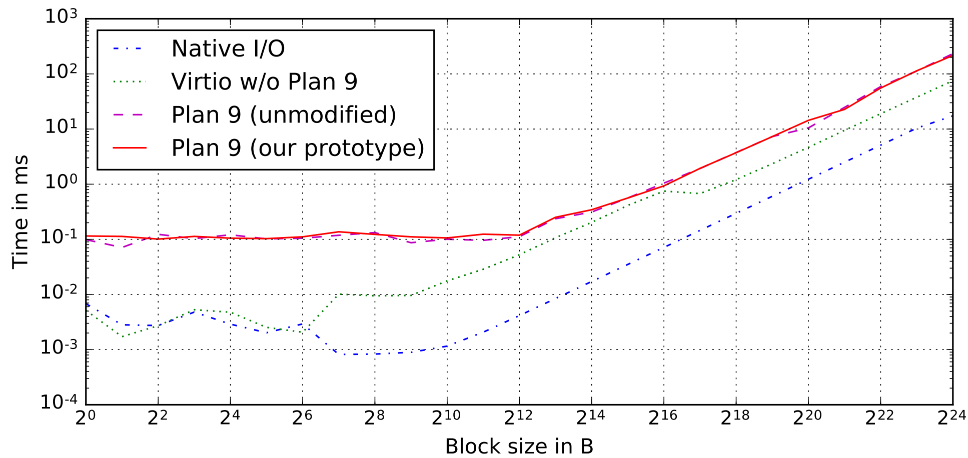


Figure 4.5: Read performance for different environments. The time to read data is given as a function of the block size. Similarly to the unmodified Plan 9 environment, the read performance of our prototype stays relatively constant for block sizes of up to approximately 4kB (2^{12}B).

which is also the maximum block size for ext4 on most architectures. Furthermore, we note that 4kB is also the preferred block size for efficient filesystem I/O of our Linux testing environment as determined by the `stat` system call.[4] Finally, note that there is no significant performance difference between plain 9P and our prototype.

The read performance is depicted in Figure 4.5. Analogously to the write performance, the time (in ms) to read data is given as a function of the block size (in bytes). As might be expected, native I/O takes the least time to read blocks, followed by virtio block, followed by the 9P environments—which inherently have the biggest performance overhead. Furthermore, the read performance of all four examined environments stays relatively constant for block sizes of up to approx. 4kB (2^{12}B) and only then starts to decline for larger block sizes—similarly to the write performance above. Note that, analogously to the write performance, there is no significant performance difference between plain 9P and our prototype.

[4] The `stat(2)` man page states that "the `st_blksize` field of the `stat` structure gives the 'preferred' blocksize for efficient filesystem I/O".

4.11.1 Network-based Filesystem Relocation

In Section 4.3.2, we noted that an advantage of our filesystem relocation approach using the Plan 9 filesystem protocol 9P is that the filesystems of the monitored VMs do not necessarily have to be located on the same machine but may be used in a distributed fashion and moved to another physical machine. This incurs an additional network overhead for the write and read performance evaluation results in Figure 4.4 and Figure 4.5. This overhead consists of several factors which may differ depending on the actual scenario, use case, and setup. In the following, we describe the most important factors that influence the network overhead and latency:

- The given system must establish a connection to the remote host. This is usually done over TCP/IP. In particular, for v9fs [91] this may be achieved with the option `trans=tcp`.
- The payload for executing read and write requests over the network must be transferred to or received from the remote host. In general, the network overhead grows poportionally with the payload size. In this regard, an important factor is the connection bandwidth (the maximum possible transfer rate) as well as the throughput (actual achieved transfer rate). For write requests the payload is transferred to the remote host which means that the upstream speed is most relevant. For read requests the payload is received from the remote host which means that the downstream speed is most relevant.
- The latency increases with the length and the network hops the data packets must travel as well as the utilized network telecommunications protocol. For the latter, the latency can be as low as 10 ms (T1 line), around 20 ms (DSL) or up to about 500-700 ms (satellite internet access).

Because of the multitude of factors as mentioned above as well as varying architectures and infrastructures used for various use cases, it is difficult to determine a generally valid network overhead through practical experiments. Therefore, in the following, we estimate the network overhead for different bandwidths and payload sizes. Figure 4.6 shows the required time in ms to transfer a certain amount of bytes over a network with a bandwidth of 1 Mbit/s (blue curve), 10 Mbit/s (red curve), and 100 Mbit/s (yellow curve), respectively. The stated bandwidths may refer to the upstream for write requests or the downstream for read requests.

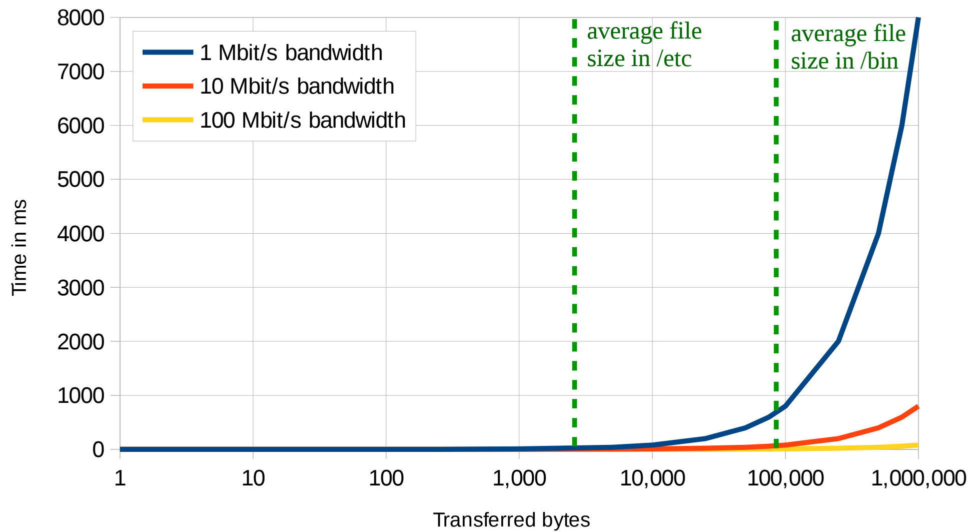


Figure 4.6: Required time to read and write files over the network for various bandwidths.

We note that for many use cases the average amount of transferred bytes will presumably be relatively small due to the average size of relevant files. In particular, the average file size of configuration files in `/etc` on the author's Debian GNU/Linux system is 3,640 bytes—as indicated by the dashed green line on the left-hand side in Figure 4.6. The files in `/etc` may be read and written. The average file size of executables in `/bin` on the author's system is 94,691 bytes—as indicated by the dashed green line on the right-hand side in Figure 4.6. The files in `/bin` will primarily be read (and executed). The time for transferring the payload of files in `/etc` is relatively low. The time for transferring the payload of files in `/bin` increases for a small bandwidth of 1 Mbit/s but is still relatively low for bandwidths of 10 Mbit/s and 100 Mbit/s. The available bandwidths as well as the actual achieved transfer rate depend on multiple factors and the use case. For example, in an IoT use case the devices may be connected via (V)DSL and in an automotive use case LTE may be utilized. For VDSL, the typical data transfer rate ranges from 50 Mbit/s[5] (profile 8a) up to 400 Mbit/s (profile 35b). For LTE, the downstream will be around 300 Mbit/s and approximately 50 Mbit/s upstream. Therefore, in both mentioned sample use case scenarios VDSL

[5]Bidirectional, i.e., upstream plus downstream.

and LTE, respectively, provide sufficient bandwidth to keep the network overhead relatively low.

In conclusion, relocating the filesystem over the network incurs a significant (but expected) overhead compared to a local setup—the latter of which is the primary focus of this chapter. Finally, we note that the overhead in general may be reduced by maintaining a single TCP/IP connection instead of establishing separate connections for each 9P request. Furthermore, the overhead for write requests can be reduced by not transferring the entire written file but only the changed bytes along with the offsets.

4.12 Security Analysis

In this chapter, we presented our approach for monitoring the integrity of guest VMs by relocating a supervised VM's entire filesystem into the isolated realm of the host. Our objective is to monitor the integrity of files on the VM's filesystem and to prevent critical file events and illegal file modifications by leveraging a paravirtualized filesystem. This enables us to protect against malware attacks trying to infect the system. In the following, we evaluate the security of our approach.

4.12.1 Persistent Malware

An integral part of most malware is a mechanism that enables it to enforce its reactivation after a restart of the infected system in order to regain control of the system. In general, this requires the malware to make persistent modifications to the filesystem. The malware could, for example, exploit the operating system's mechanism for automatically starting programs at boot time (e.g., manipulating Windows Autostart or System V init scripts) or by installing a kernel module that gets loaded at boot time (e.g., manipulating `/etc/modules` and similar files). To hide such file manipulations from the monitoring agent in the host (in particular, FOM), the malware could try to compromise the guest OS kernel and tamper with the 9P kernel components such that the 9P messages corresponding to the manipulations will be blocked from being propagated to the 9P server. However, as explained in Section 4.3, all such file operation requests must necessarily be routed through the 9P server. Otherwise, it is impossible for a guest VM to access the VM's filesystem. It follows that it is impossible for a guest VM to stealthily make modifications to the VM's filesystem as would be required above.

4.12.2 Fileless Malware

Fileless malware describes a relatively new type of malware that is capable of surviving restarts of the infected system without direct manipulation of the filesystem. Examples of such fileless malware include POWELIKS [137], Kovter [167, 39], and Phase Bot [98, 99]. These all have in common that they run under the Windows operating system and exploit the autostart functionality of the Windows Registry. As the Windows Registry is held in Random Access Memory (RAM), this technique does not require the direct manipulation of the filesystem.[6] However, in our case, the guest VMs run a paravirtualized Linux—or conceptually some other Unix-like OS—which adheres to the "everything is a file" principle. This reduces malware to the case of persistent malware as described above, which can be detected and prevented by our approach.

4.12.3 Persistent File Manipulations

Certain types of malware may only execute a payload once but do not get reactivated after a restart of the system. However, the payload may result in persistent file manipulations impacting the system even after a restart. For example, the payload may weaken the system's security by configuring weak encryption algorithms used for SSL/TLS connections or by generating new remote login accounts as a backdoor. However, in all of these cases we are able to detect and prevent the corresponding file manipulations as explained in Section 4.12.1, thus protecting against such attacks.

4.12.4 Software Package Manipulations

A special case of file manipulations is the manipulation of software packages. As explained in Section 4.9, it is possible for a guest VM to install, remove, upgrade, and downgrade software packages. This fact could be misused by malware, for example, to install malicious software or to downgrade already installed software to a vulnerable version such that an attacker could exploit this software even after a restart of the system. However, we enforce a mechanism that allows such software package maintenance operations only in a well-defined manner. This guarantees that all software package maintenance operations must necessarily be routed through the host (cf. Section 4.9.1)

[6] We note that the Windows Registry eventually also gets stored on the filesystem by the OS in order for changes to survive reboots. In theory, this makes it possible to detect changes of the corresponding files. However, in practice, maintaining a list of reference hash values of these files is likely to prove difficult due to the frequent legitimate changes of the Windows Registry.

as the guest VM does not possess sufficient privileges for executing these operations itself. This, in turn, allows the host to check permissions and to verify the package integrity (cf. Section 4.9.2). Therefore, we are able to detect and prevent illegitimate software package manipulations.

4.12.5 Non-Persistent Manipulations

Some attacks may also consist of temporary, non-persistent manipulations that do not survive a restart of the VM. For example, suppose that a legitimate user of the guest VM downloads an executable from the internet and wants to run it, without knowing that the downloaded file contains malware. In this case, the host (in particular, EDE) detects and measures the execution of the file as described in Section 4.8. However, a security flaw like inadequate policy rules may cause EDE to allow the execution. In this case, we would still be able to detect and protect against persistent manipulations by the malware as described above and to guarantee that a guest VM (re)starts in a trustworthy state, free from malware. However, the malware may be able to perform non-persistent manipulations such as altering the behavior of running processes in RAM or trying to execute a program directly from RAM without accessing the filesystem. This could not be detected by our filesystem-based approach, where we focus on manipulations of the filesystem, and should be protected against by complementary work (e.g., [93, 185, 71, 4]).

4.13 Related Work

Tripwire [82] is a commonly known HIDS, which detects changes to filesystem objects by checking the filesystem in periodic intervals. However, there is no support for real-time checking. Hence, Tripwire cannot *prevent* attacks but just detect them after they have happened. Furthermore, Tripwire is not isolated from the monitored system and as such is susceptible to attacks. I^{3}FS [122] tries to improve Tripwire by adding real-time integrity checks. However, since the supervising agent and the relevant databases are located within the realm of the monitored system, I^{3}FS is also vulnerable to attacks.

Livewire [61] tries to solve the aforementioned drawbacks by introducing a virtualization layer in order to place the monitored system into an isolated VM and moves the supervising agent outside of the VM. By using a suite of different intrusion detection policies Livewire is able to detect attacks within the VM. However, Livewire is only capable of reporting an attack as opposed

to interfering and preventing it. In contrast, our paravirtualized approach not only allows the detection of attacks but also enables us to prevent them.

Lares [124] and Xenprobe [134] place hooks in the guest VMs in order to trace syscalls. However, these hooks can be attacked and disabled from within the VM. Hence, the hypervisor is not able to reliably monitor the VMs. Our approach of relocating the guest VM's filesystem from the realm of the guest VM to the host guarantees that all file operations originating from a VM are necessarily routed through the hypervisor-level in order to realize reliable monitoring.

In [189], Zhao et al. implement monitoring in a virtualized environment. They try to bridge the semantic gap between disk blocks and logic files with the help of the block tap library *blktap* [123]. However, they still allow the modification of files in security-critical directories (e.g., `/etc`) while only logging these modifications, thus being incapable of *preventing* potential attacks. Our monitoring approach allows VMs to autonomously upgrade software packages in a controlled manner, thus enabling the secure and restricted modification of files in security-critical directories.

In [79], Xuxian et al. utilize virtualization to detect stealthy malware that uses hidden files within the guest VM. In our case, we relocate the filesystem of the guest VM and allow the creation of files only through the host over a well-defined interface. In this way, we prevent stealthy malware from exploiting hidden files because without involving the host, it is impossible for the malware to create any files on the guest filesystem.

In [80], Jones et al. describe Lycosid which enables the detection of malware by utilizing a hidden process detection and identification service. Their approach does not require knowledge of specific guest OS details. While Lycosid is able to detect malware, it is not possible to prevent malware in the first place. In contrast, our approach allows us to inspect all operations on the guest VM's filesystem before they actually happen and to prevent them.

ReVirt [41] and IntroVirt [81] (which builds upon ReVirt) use VMI to monitor and log the execution of application and operation system software within a VM in order to allow for replaying events starting from a previous VM state. This may be used to trace the cause of a vulnerability in a compromised VM. The disadvantage is that they rely on information obtained from the VM even though this information may be manipulated by malware. In contrast, we treat information originating from a VM as untrustworthy, for example, whether a write request (including forged requests) is granted or not depends only on the policy rules of the respective VM and not on supplemental context information supplied by the VM.

Patagonix [93], Manitou [92], and in [185] Wessel et al. realize hypervisor-based integrity monitoring and take advantage of the Memory Management Unit (MMU) to realize guest memory protection. Running programs will be identified by comparing the hash values of individual memory pages. The disadvantage is that these approaches require a large database of trusted hash values of valid code pages.

Similar to Patagonix, HIMA [4] provides hypervisor-based monitoring of critical guest events and guest memory protection. A notable feature is that HIMA provides Time of Check to Time of Use (TOCTTOU) consistency, i.e., changes of a process over its lifetime are reflected in the measurement of the corresponding program. However, both HIMA and Patagonix require considerable effort for bridging the semantic gap. In contrast, our approach is very efficient in preserving the semantic knowledge of file operation events within VMs on a high-level abstraction by utilizing the 9P filesystem protocol.

4.14 Summary

In this chapter, we presented our virtualized architecture that allows for secure file integrity monitoring. The key idea of our approach is to relocate a supervised VM's entire filesystem into the isolated realm of the host such that all file operations must necessarily be routed through the hypervisor-level. This allows for complete monitoring and the prevention of critical filesystem events. In contrast to existing monitoring approaches, our technique has the advantage that hooks placed inside the VMs are not prone to manipulation by malware. The reason is that disabling hooks in a VM inevitably renders the VM incapable of accessing or manipulating its own filesystem (provided by the respective hook). Another key feature of our approach is that we enable regular users of VMs to autonomously install and upgrade software packages in a secure and controlled manner, without the need of requiring the intervention of the administrator of the physical system. Finally, we measure all executed binaries of all VMs and store these measurements in a single, multiplexed TPM by building on the work developed in the previous chapter. The experimental results of our prototype implementation show the practicality of our approach.

Chapter 5

Continuous Authentication using Touchscreen Dynamics

In Chapter 3, we demonstrated how to securely store and multiplex integrity measurements of arbitrarily many VMs with just a single standard TPM. We then built upon this work in Chapter 4 and developed a system for monitoring the filesystems of multiple VMs from outside of the VMs and stored the so-obtained integrity measurements in the TPM. This enables us to secure and protect the integrity of VMs and to defend against attackers and malware—like viruses, worms, and trojans—located in the VMs.

However, such virtualized systems typically contain a user class with high privileges (system administrators) who are not confined to VMs but are able to make system-wide modifications. For example, system administrators may use a web frontend offered by the system in order to create and configure VMs or to change VM policy rules. Because these systems usually allow remote access (one reason being that users are often located at geographically different locations than the server), system administrators can take advantage of portable devices like smartphones to access and maintain the system from virtually everywhere. However, carrying around these devices amplifies the risk of loss or theft, thus increasing the threat of attackers hijacking critical system administrator accounts and compromising VMs and potentially the entire system.

In this chapter, we present a framework to secure accounts by continuously verifying user identities based on user interaction behavior with smartphone touchscreens. This enables us to protect user accounts by disabling critical functionality and enforcing a reauthentication in case of suspicious behavior. Furthermore, additional actions may be triggered such as sending an alarm to a (different) system administrator to investigate the situation.

The developed techniques could also be used in other scenarios such as online banking, where the user experience can be improved by increasing or even removing the timeout until the user gets automatically logged out of an online banking session while retaining the same or even gaining a higher level of security.

We take advantage of standard mobile web browser capabilities to remotely capture and analyze touchscreen interactions. This approach is completely transparent for the user and works on everyday smartphones without requiring any special software or privileges on the user's device. We show how to successfully classify users even on the basis of limited and imprecise touch interaction data as is prevalent in web contexts. We evaluate the user classification of our framework and show that the user identity verification accuracy is higher than 99% after collecting about a dozen touch interactions. Finally, we evaluate the CPU overhead, battery life, and generated network traffic to assess the practicality of our approach.

Parts of this chapter have been published in *User Identity Verification Based on Touchscreen Interaction Analysis in Web Contexts* at the 11th International Conference on Information Security Practice and Experience (ISPEC) in 2015 [181].

The rest of this chapter is organized as follows. In Section 5.1, we give an overview of how behavioral biometrics can be used for authenticating users, sketch our approach and highlight the differences to existing research, and present our contributions. Section 5.2 states the attacker model and our assumptions. In Section 5.3, we evaluate suitable touch interactions for user classification in web contexts. Section 5.4 outlines our system architecture and the main components. In Sections 5.5 and 5.6, we explain the touch behavior model training of our framework and detail the user identity verification procedure, respectively. Section 5.7 describes the features we extract from scroll gestures and the device's acceleration sensor data in order to realize the user identity verification by our framework. Regarding the verification strategy, we identify the requirements for a reliable and accurate confidence value calculation in Section 5.8. In Section 5.9, we describe our proof of concept implementation. Sections 5.10 and 5.11 present the classification and performance evaluation results, respectively. In Section 5.12, we evaluate the security of our continuous user identity verification approach. Section 5.13 discusses related work. Section 5.14 concludes this chapter.

5.1 Behavioral Biometrics for Authentication

Modern touch devices like smartphones and tablets have become ubiquitous in everyday life. Consequently, they are used for an increasing number of security-sensitive tasks ranging from reading email to more critical tasks like online banking. The risk of loss or theft of such mobile devices is especially prevalent because users carry them around and use them in unprotected environments. Additionally, users often choose simple and weak secrets, increase the screen lock timeouts of their devices, or completely disable unlock [56]. This allows attackers to hijack accounts, sometimes even without having to enter a password or PIN because having physical access to the device, often entails direct access to several accounts where the user is still logged into. A technique to protect against such threats is continuous user authentication. This can be realized by continuously verifying the user identity based on individual interaction behavior with the touch device caused by physical differences between users, varying habits, and personal preferences. In case of suspicious user behavior, the user account may be temporarily locked and a reauthentication enforced. Furthermore, additional actions may be triggered such as sending an alarm to a system administrator to investigate the situation. This provides an additional protection layer which can be used in combination with existing, complementary techniques.

In contrast to other authentication mechanisms such as entering a PIN or fingerprint recognition, where it is possible for an attacker to enter the PIN himself or to trick the fingerprint sensor with a mold of the legitimate user's fingerprint, individual behavior patterns are difficult to imitate precisely [14]. This fact has been utilized to verify the identity of users in normal desktop computer scenarios based on their keystroke dynamics [113] and mouse movements [190, 50]. Recent research advances these techniques by exploring ways to verify users based on their individual interaction patterns with touch devices. There exists diverse research ranging from identifying and verifying users based on their behavioral patterns when tapping on the touchscreens [85, 191] to work that analyzes interactions like scroll gestures [56, 14, 192]. Other work tries to infer keystrokes based on touch events [32] or device sensor information [121]. Some research utilizes dedicated hardware, e.g., modified touch displays [52, 54] or even specially prepared gloves [53].

However, all these approaches require dedicated software on the user's device. Furthermore, computationally intensive user verification algorithms executed on the smartphone can negatively affect the user experience. A more generally applicable solution would be to verify user identities from

remote servers, without the need to install, set up, and run any special software on the device itself. Continuous user identity verification is then performed by a remote entity. On the one hand, such a solution does not suffer from the above problems and users can use their unmodified everyday devices. On the other hand, users can still benefit from increased security by having their identities continuously verified. In particular, this enables us to improve the security of the system we developed in the previous chapters by continuously authenticating users and system administrators. Furthermore, other fields like online banking may also benefit from our work by enabling them to improve the user experience (e.g., no automatic log out after a period of inactivity) while retaining the same or even gaining a higher level of security.

In this chapter, we make the following contributions:

- We show how to continuously verify a user's identity by remotely analyzing the user's touch behavior using machine learning classification techniques. Depending on a calculated confidence value indicating whether the active user is indeed the legitimate user, we either provide the full service functionality or disable critical functionality and enforce a reauthentication.
- In contrast to previous work [85, 191, 109, 56, 107, 14, 159, 130, 192], our proposed method only requires a web browser running on the user's device which is used by the user to access web pages of remote sites that utilize the techniques described in this chapter. Our approach does not require any special privileges on the device and is completely transparent for the user.
- A major challenge is that in contrast to existing work, we do not have direct access to the API of the touch device's operating system. This means that touch interaction data, proven to be beneficial for user classification [147], has a lower degree of precision and some data cannot be obtained at all. We provide a selection of features that still allow for successful user classification under these conditions and implement a framework to continuously verify user identities.
- We evaluate the user classification accuracy of our framework by analyzing touch interaction data sets of 45 users. The results indicate the feasibility of our approach with both False Acceptance Rate (FAR) and False Rejection Rate (FRR) potentially being as low as $< 1\%$ after collecting about 14 touch interactions.

- We assess the practicality of our approach by conducting several experiments to evaluate the CPU overhead on the device, how this affects battery life, and we investigate the network traffic overhead generated by a typical smartphone with enabled user verification.

5.2 Attacker Model and Assumptions

Our objective is to protect user accounts hosted by a remote server against account abuse. The attacker is assumed to have physical access to a legitimate user's device (e.g., stolen device) and may use it for accessing the user's remote accounts. In this case, we assume the user to still be logged into all of his accounts (active sessions). Additionally, the attacker is assumed to be capable of reaching the legitimate user's device over the network. We consider all MITM attacks on the communication between client and server. We do not consider direct attacks nor inside attacks (e.g., corrupt system administrator) on the server infrastructure: web server, Touch Behavior Verifier (TBV), and Continuous Authentication Monitor (CAM).

5.3 Touch Interaction Selection in Web Contexts

In contrast to existing work [85, 191, 109, 56, 107, 14, 159, 130, 192], we do not have direct access to the API of the touch device's operating system. Instead, we capture touch interaction data from the user's web browser. As a consequence, this data has a lower degree of precision, access is restricted to low-rate streams which provide data with slower frequencies as compared to those provided in-app [104], and some data, proven to be beneficial for user classification [147], cannot be obtained at all. Another difficulty arises from the fact that different users are likely to use different kinds of touch devices, all with potentially different properties. At first, it may seem that under these circumstances it is actually easier to classify users and verify their identities based on distinct properties of their associated devices—along with other (software) properties like the utilized browser, browser-version, browser-plugins and their configuration, language and time-zone. In fact, (web-based) *device fingerprinting* has been the subject of various research [42, 115, 102, 19] and even accommodates for changing fingerprints (e.g., changing browser plugin configurations) using heuristics. However, even though these fingerprinting techniques can be used to identify devices, they cannot reliably be used to verify the identity of the user operating (the

touchscreen of) such a device. This is especially true, for example, in the case of a stolen device where device properties remain the same.

Therefore, we focus on user characteristics as opposed to device characteristics. We require that touch interactions used for user classification should be sufficiently precise. Furthermore, they should be available on all touch devices, operating systems, and web browsers. Otherwise, it might not be possible to classify certain users whose devices, operating systems, or web browsers lack properties we use for classification. This requires us to carefully select suitable touch interactions that can be successfully used for user classification under these constraints.

In the following, we evaluate and select several touch interactions for user classification in web contexts and give the rationale for our choice. We distinguish two classes of information: details about gestures executed by the user on the touchscreen, and complementary device sensor data obtained from the device while executing those gestures.

5.3.1 Touchscreen Gestures

Scroll or *Swipe* (actions: press, move, lift) is one of the most common gestures used in web browser contexts as it constitutes the primary means of touch-based navigation on web pages. Scroll gestures between different users are quite distinctive, yet we discovered that scroll gestures of a single user are relatively similar and consistent (cf. Section 5.10.1). These properties make scroll gestures well suited for user classification in web contexts. However, as mentioned above, we have to deal with less precise data and only know a few, varying number of points along the scroll gesture's path. We will show how to solve this problem in Section 5.7.

Tap (actions: press, lift) is primarily used to follow hyperlinks on web pages. In general, behavioral tapping patterns can be used for user classification [85, 191, 109]. However, in web contexts, discriminative tap features like the point in time where the maximal pressure occurs or the area covered by the finger touching the screen cannot be obtained at all or only with low precision. Furthermore, in our experiments, tap gestures occurred about 90% less than scroll gestures. Therefore, we exclude tap gestures for user classification in this work.

Zoom (e.g., pinch, double tapping, double touch drag) allows increasing or decreasing the web page's content. Even though zooming may be recognized by interpreting primitive touch events [168], we do not consider it for user classification as the (internal) coordinate resolution and page offsets of captured data changes, thus resulting in data that is difficult to compare.

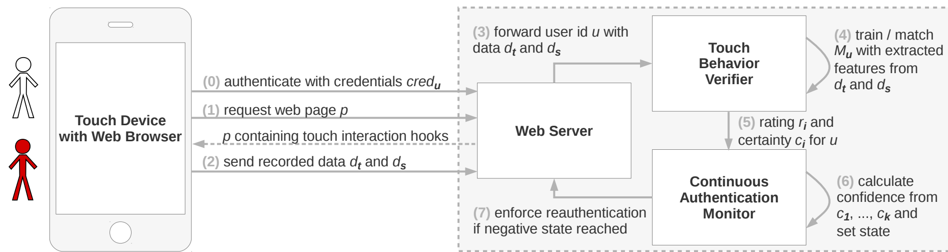


Figure 5.1: System architecture. The touch device on the left is used to access web pages hosted by the web server. Our framework on the right consists of the web server, Touch Behavior Verifier (TBV), and Continuous Authentication Monitor (CAM).

Other gestures like *Drag* or *Rotate* normally do not occur in web contexts and will therefore not be considered in this work.

5.3.2 Device Sensor Data

Acceleration data represents the acceleration force along the device's x, y, and z axes. We use it to evaluate the device's feedback to scroll gestures, thus increasing the classification accuracy. Capturing the acceleration data as done in this chapter does not require any special privileges (as opposed to, for example, capturing GPS location data) and is completely transparent for the user.

Force indicates how much pressure the user applied to the touchscreen. According to [147], the finger pressure gives discriminative information for user classification. However, this data cannot be retrieved with all devices and browsers (see also Table 5.1 for a list of our tested devices, operating system versions, and web browsers). Furthermore, the update rate of the force attribute is rather slow and the precision may be low—as described in Section 5.11.1. We compensate for lack of this information by analyzing the device's acceleration feedback to touch interactions.

Gyroscope data gives the rate of rotation around the x-axis (compass direction), y-axis (front-to-back tilt), and z-axis (left-to-right tilt). We do not incorporate this information for classification as it caused overfitting in our experiments, for example, front-to-back tilt differed depending on whether the user was standing or sitting, thus hindering generalization.

5.4 System Overview

The system architecture is shown in Figure 5.1. The smartphone on the left is used to access web pages hosted by the web server on the right. Our framework consists of the web server, the Touch Behavior Verifier (TBV), and the Continuous Authentication Monitor (CAM). The web server is responsible for initially authenticating users based on traditional authentication schemes (e.g., password authentication) in order to provide access to user accounts. TBV trains and maintains touch behavior models in order to classify users based on their touch interactions. CAM continuously assesses whether the current user is the legitimate user and enforces a reauthentication on suspicious behavior.

In general, there exist two phases: *training* and *verification*. In the training phase, a touch behavior model of a given user will be created based on the user's touch interactions and supplemental device sensor data. In the verification phase, this touch behavior model is used to verify the identity of the user. We point out that in this thesis we pursue *user (identity) verification* as opposed to *user identification*. In the first case, for a given user a certain a priori identity is assumed or determined in some way (in our case, the web server initially determines the identity based on traditional authentication schemes) and subsequently, within the verification process, either confirmed or disproved with a certain probability. In the second case, for a given user the identity will be determined with a certain probability within the identification process, without having any a priori knowledge of the user's identity. Note that in our approach, the training and maintenance of the touch behavior models as well as the user classification task are not performed directly on the user device. We rather just forward the relevant touch interaction data from the device to the remote server which then executes the necessary tasks. This has the advantage of not requiring any special software on the user's device, and this approach works out of the box on everyday smartphones. Furthermore, this outsources all machine learning tasks of the training and verification phases to the (more powerful) backend in order to save battery life on the user's device.

5.5 Touch Behavior Model Training

In the following, we describe the steps involved in training a user's touch behavior model. First, a user u authenticates himself to the web server through an SSL/TLS secured channel [57, 36] with credentials $cred_u$ (step 0), for example, by entering a combination of username and password on

a page hosted by the web server. For each requested web page p (step 1), the web server provides a modified version of p containing *touch interaction hooks*. The hooks cause the user's web browser to record touch interaction data d_t along with supplemental device sensor data d_s. Both d_t and d_s will be periodically sent to the web server (step 2) which, in turn, forwards the data to the TBV (step 3). In step 4, the TBV parses the recorded (raw) touch interaction data d_t and identifies high-level gestures. The recognized gestures will be augmented with device sensor data d_s that occurred during the time the respective gesture was executed. The combined result is called an *observation*. The TBV extracts relevant features from an observation and uses them to train a touch behavior model $\mathcal{M}_u$. A detailed description of the features used by our framework is given in Section 5.7. Note that the just-described training phase inherently assumes that the recorded data indeed belongs to the legitimate user u and that no illegitimate user compromised the account during this time. The trained model $\mathcal{M}_u$ is eventually used in the verification phase (which additionally includes steps 5-7) to verify user identities.

Our utilized hooking technique (cf. Section 5.9) works out of the box with all common mobile operating systems and browsers and does not require any special permissions from the user (neither for recording touch interactions nor for recording sensor data) and there is no indication provided to the user that interactions are being recorded. Hence, it is completely transparent for the user.

5.6 User Identity Verification

After the model $\mathcal{M}_u$ has been trained, subsequently recorded interaction data of the active user u' (steps 0-3) will be used by the TBV to verify that u' is indeed the legitimate user u (as claimed by the credentials $cred_u$ of the currently active user u'). We use the touch behavior model $\mathcal{M}_u$ and utilize machine learning classification techniques to verify the user (step 4). In step 5, the TBV calculates for an observation o_i a *binary rating* r_i (*true* or *false*) indicating whether the framework considers the active user to be the legitimate user (*true*) or an illegitimate user (*false*). The TBV also calculates an associated *certainty score* c_i that represents the probability that the rating r_i is correct. Since the accuracy to verify a user based on a single observation is not very high (as shown in Section 5.10.2 as well as in [14]), we consider sequences of (consecutive) observations in order to improve the overall accuracy. This is achieved by aggregating the certainty scores of all observations of a sequence (as explained in Section 5.8) in order to calculate

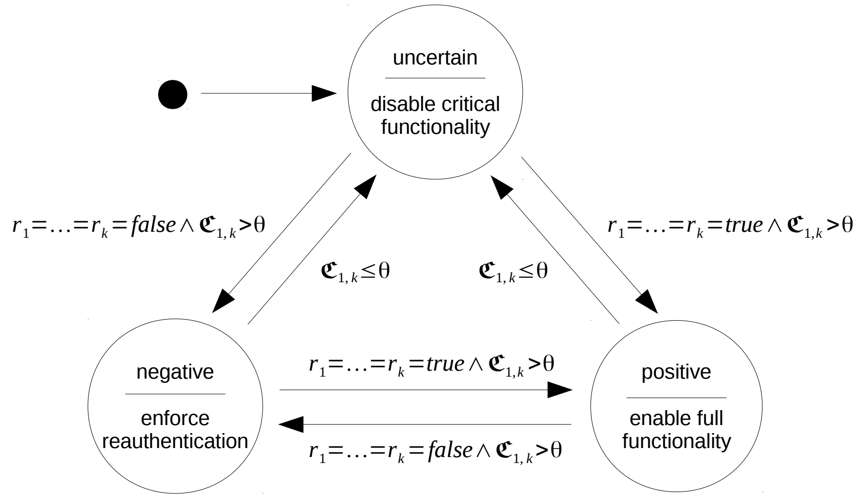


Figure 5.2: State diagram maintained by CAM for enforcing actions based on the ratings $r_1, \ldots, r_k$ and the confidence value $\mathfrak{C}_{1,k}$.

an aggregated certainty score called *confidence*. The CAM calculates and uses this confidence value to decide if the active user is still the legitimate user (step 6). The details will be explained in Section 5.8.

CAM distinguishes three states depending on the ratings and the confidence: *uncertain*, *positive*, and *negative*. The state diagram is shown in Figure 5.2. The uncertain state is the initial state and is also entered if the ratings of consecutive observations are too inconsistent or the confidence is lower than a specified threshold θ. In this case, critical functionality may be temporarily disabled until the positive state is entered (again). The positive state is entered if the ratings are consistently true *and* the confidence exceeds θ. This state indicates that the framework deems the active user legitimate and the full functionality is provided. The negative state is entered if the ratings are consistently false *and* the confidence exceeds θ. In this case, the framework considers the active user to be an illegitimate user and enforces a reauthentication (step 7)—possibly combined with other actions, e.g., notifying the system administrator. Note that CAM's described behavior is optimized for security-critical applications where protecting against attackers is prioritized over the risk of (temporarily) restricting legitimate users.

Finally, the observations used for user verification will also be used as training data to continuously improve the model $\mathcal{M}_u$. However, since illegitimate users may only be recognized after inspecting multiple observations, the inclusion is deferred until the TBV is sufficiently sure that the observations actually belong to the legitimate user.

5.7 Feature Extraction

For user identity verification, we extract various features from scroll gestures and the device's acceleration sensor data. A scroll gesture is composed of multiple touch events. A *touch event* t is a triplet $t = \langle t^{\tau}, t^{x}, t^{y} \rangle$ representing a touched point $\langle t^{x}, t^{y} \rangle$ at time t^{τ} on the two-dimensional Euclidean plane represented by a device's touch screen. Each touch event t is one of three *basic touch events*: t_s (*touch start*, i.e., finger pressed down), t_m (*touch move*, i.e., finger is moving while pressed down), or t_e (*touch end*, i.e., finger lifted), that is, $t \in \{t_s, t_m, t_e\}$. A *scroll gesture* S_k refers to a sequence of basic touch events starting with a *touch start* event, followed by one or more *touch move* events, and terminated with a *touch end* event. Consequently, a scroll gesture S_k is represented by an ordered list of touch events $S_k := \langle t_s, t_{m_1}, \ldots, t_{m_n}, t_e \rangle$ with $n \geq 1$.

The following features will be used for user identity verification by our framework.

5.7.1 Path Offsets

In web contexts, only a few unevenly distributed points along the path described by a scroll gesture are known (cf. Section 5.3). However, it is necessary to have the same number of points for all scroll gestures in order to make their feature sets comparable. We solve this by dividing the overall duration of a scroll gesture into a globally fixed number I of equal-sized time intervals (where the interval size is fixed w.r.t. a single scroll gesture). At each time interval boundary, we approximate the touch coordinates along the path. These touch coordinates are called *intermediate points* $t_1, \ldots, t_I$. The respective x-components of the intermediate points are enumerated with $t_1^x, \ldots, t_I^x$ (analogously, the following descriptions apply to the y-components). The x-component t_i^x of an intermediate point t_i is interpolated based on the x-components of the two (chronologically) adjacent touch events $t_i^{\leftarrow}, t_i^{\rightarrow} \in S_k$ of t_i. In order to facilitate the following calculations, we further define $t_0^x := t_s^x$ and $t_{I+1}^x := t_e^x$. Finally, *intermediate offsets* are defined by calculating the distance of each intermediate point to the touch start event $t_s \in S_k$, that is, $t_i^{\triangle x} := t_i^x - t_s^x$ where $0 \leq i \leq I + 1$.

5.7.2 Bounding Box

A bounding box is constructed around a scroll gesture shape in order to obtain a coarse representation of the gesture. The bounding box's width and height reflect individual user behavior and scroll preferences on an abstract

level (cf. Figure 5.5 and Figure 5.6). The width of a scroll gesture S_k is defined as $S_k^w := \max_{t \in S_k} \pi_2(t) - \min_{t \in S_k} \pi_2(t)$, where π_2 is the second projection t^x of a triplet $t = \langle t^\tau, t^x, t^y \rangle$. Similarly, the height of a scroll gesture S_k is defined as $S_k^h := \max_{t \in S_k} \pi_3(t) - \min_{t \in S_k} \pi_3(t)$.

5.7.3 Raster

A coarse grid is fit over a scroll gesture's bounding box to obtain an abstract scroll gesture representation. The intention is to allow for better recognition of gestures at different sizes as well as across different devices with varying screen sizes. If the path of a scroll gesture S_k crosses the cell $\langle i, j \rangle \in \{1, \ldots, R\}^2$ of an $R \times R$ raster, then $S_k^{r_{i \times j}} = 1$. Otherwise, $S_k^{r_{i \times j}} = 0$.

5.7.4 Velocity

The velocity of the finger's motion is analyzed at different times when executing a scroll gesture. The velocity varies for different users and different sections of a scroll gesture (cf. Figure 5.7). We utilize intermediate offsets as defined above in order to calculate the velocity over the horizontal distance intervals $[t_{i-1}^x, t_i^x]$, where $1 \leq i \leq I + 1$, and analogously over the vertical distance intervals. We define $t_i^{v_x} := \left(t_i^{\triangle x} - t_{i-1}^{\triangle x} \right) \cdot \frac{I+1}{t_e^\tau - t_s^\tau}$, where $1 \leq i \leq I+1$, and analogously $t_i^{v_y}$.

5.7.5 Curvature

The curvature of a scroll gesture is inspected at each intermediate offset in order to recognize the same gesture shape executed at different sizes. This approach is similar to the techniques used by gesture-recognition applications like easystroke [76]. We define $t_i^{\measuredangle} := \arctan \left(\frac{t_i^{\triangle y}}{t_i^{\triangle x}} \right)$ where $1 \leq i \leq I+1$.

5.7.6 Acceleration

A scroll gesture's intrinsic features are augmented with the device's acceleration sensor data obtained at each intermediate point.[1] This allows us to evaluate the device's feedback to touch interactions and is similar to the approach used in [14]. For an intermediate point t_i, we combine the x, y, and z accelerations $t_i^{a_x}$, $t_i^{a_y}$, and $t_i^{a_z}$, respectively, and define

[1] To be precise, the device's acceleration will be recorded at the points in time when the touch events $t_1, \ldots, t_n$ of a scroll gesture S_k occurred and will then be used for the interpolated intermediate points calculated from these touch events.

$t_i^a := \sqrt{(t_i^{a_x})^2 + (t_i^{a_y})^2 + (t_i^{a_z})^2}$ where $0 \leq i \leq I+1$.

Finally, for a scroll gesture S_k with intermediate points $t_0, \ldots, t_{I+1}$ the feature vector F_{S_k} is defined as $F_{S_k} := \langle T^{\triangle x}, T^{\triangle y}, S_k^w, S_k^h, S_k^r, T^{v_x}, T^{v_y}, T^{\measuredangle}, T^a \rangle$ where (for reasons of readability T^X is an abbreviation for $t_0^X, \ldots, t_{I+1}^X$ and S_k^r for the raster $S_k^{r_{1\times1}}, \ldots, S_k^{r_{R\times R}}$.

5.8 Verification Strategy

We use the touch behavior model $\mathcal{M}_u$ and utilize random forests [17] classification to verify the identity of a user u. Random forests have proven to be fast and effective classifiers [160] with good results in the context of touch-screen interaction classification [53, 1] similar to our scenario. As explained above, the overall user verification accuracy is improved by considering subsequences $\langle o_i, \ldots, o_j \rangle$ of the sequence of all observations $\mathcal{O} := \{o_1, \ldots, o_n\}$, with $1 \leq i \leq j \leq n$, instead of only considering single observations. Such a subsequence is denoted w.l.o.g. with $\mathcal{O}_{1,k} := \langle o_1, \ldots, o_k \rangle$.

5.8.1 Subsequence Processing

For a subsequence $\mathcal{O}_{1,k}$, we calculate the corresponding binary ratings $R_{1,k} := \langle r_1, \ldots, r_k \rangle$ with each $r_i \in \{true, false\}$ indicating whether the features of the corresponding observation o_i match (*true*) or do not match (*false*) the model $\mathcal{M}_u$. All ratings of $\mathcal{O}_{1,k}$ are required to be consistent, i.e., $r_1 = \ldots = r_k = true$ or $r_1 = \ldots = r_k = false$, in order to serve as a basis for a meaningful confidence value calculation as described below. We always consider the longest possible subsequences with consistent ratings. For binary ratings $R_{1,k}$, we derive associated certainty scores (probability values) $C_{1,k} := \langle c_1, \ldots, c_k \rangle$ based on the class probabilities calculated from the random forests. Finally, for certainty scores $C_{1,k}$, the confidence value $\mathfrak{C}_{1,k}$ is calculated as an aggregated certainty score.

5.8.2 Confidence Value Calculation

We identified three requirements for a reliable and accurate confidence value. First, a subsequence $\mathcal{O}_{1,k}$ should have a minimum length $\mathcal{L}$ in order to compensate for the impact of unusually high certainty scores of single observations. Otherwise, the framework might, for example, enforce a reauthentication solely based on one outlier. Second, the confidence should increase proportionally to the sequence length because, intuitively, the more consis-

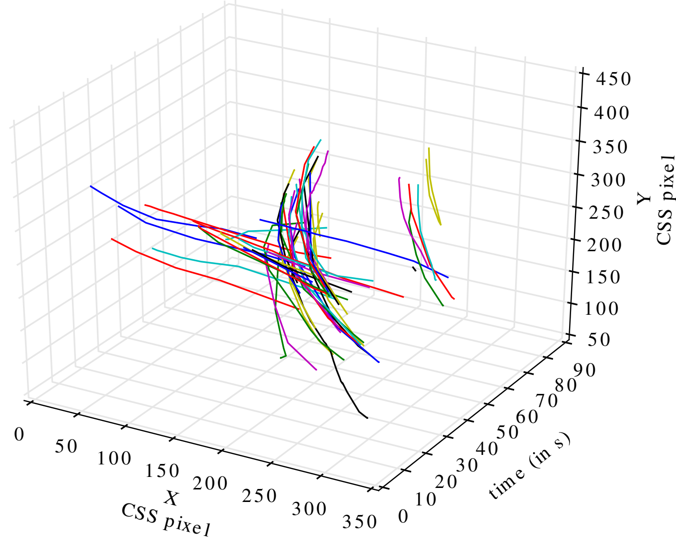


Figure 5.3: Plotted scroll gestures of a user who utilizes both vertical and horizontal scroll gestures, where vertical gestures have pronounced curvatures.

tent ratings exist the more likely it is that the rating value is correct. Third, the chronological order of observations is significant: more recent observations should be considered more important than older ones. This is achieved by assigning different weights $w_1 \leq \ldots \leq w_k$ to the certainty scores $C_{1,k}$.

The following formula satisfies the above requirements and is used by CAM to calculate the confidence for a subsequence $\mathcal{O}_{1,k}$ with certainty scores $c_1, \ldots, c_k$:

$$\mathfrak{C}_{1,k} := \mathfrak{C}(c_1, \ldots, c_k) := f(k) \cdot \frac{1}{\sum_{i=1}^{k} w_i} \cdot \sum_{i=1}^{k} w_i c_i, \quad f(k) := \begin{cases} 0 & : k < \mathcal{L} \\ g(k) & : k \geq \mathcal{L} \end{cases}$$

The function g with $0 \leq g(k) \leq 1$ should be monotonically increasing in order to satisfy the second requirement. The actual definition of g as well as the values of the minimum length $\mathcal{L}$ and the weights $w_1, \ldots, w_k$ may be flexibly adjusted depending on the use case scenario and policy rules.

5.9 Framework Implementation

We have implemented our framework (as shown in Figure 5.1) as a proof of concept. We run an Apache web server and utilize a Python script responsible for authentication and for injecting JavaScript code for the touch interaction hooks into all web pages hosted by the web server. The hooks

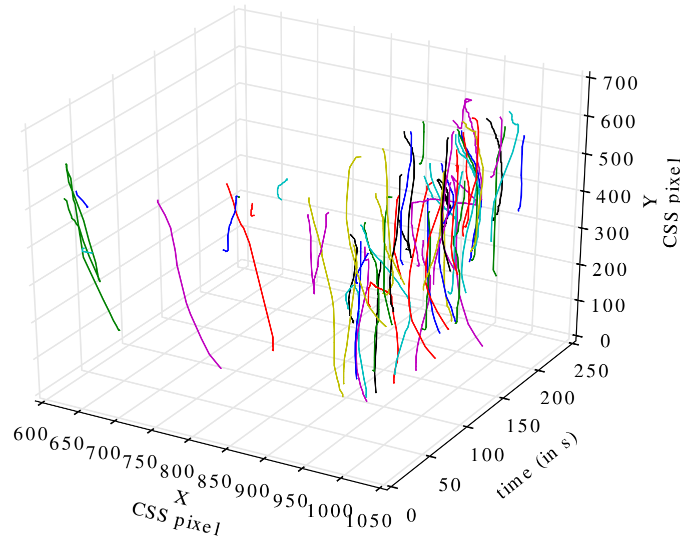


Figure 5.4: Plotted scroll gestures of a user where the vertical scroll gestures are rather straight and quite distinct from the vertical scroll gestures depicted in Figure 5.3.

will be executed on the client side within the user's web browser. To realize the hooking functionality, we leverage the JavaScript Touch Event API [153, 13, 154] in order to capture the required touch events.

We add event listeners for the following events on the `document` DOM object: `touchstart` (t_s), `touchmove` (t_m), and `touchend` (t_e). The user agent (browser) dispatches these events to indicate when the user places a touch point on the touch surface, moves a touch point along the touch surface, and removes a touch point from the touch surface, respectively [153]. Furthermore, we add an event listener for the `devicemotion` event (t^a) on the `window` DOM object. This causes the browser to record the device's acceleration. We continuously record the acceleration and use the gathered data to augment the recorded `touchstart`, `touchmove`, and `touchend` events. However, the acceleration data recorded when no scroll gesture is active (i.e., all acceleration data recorded outside of a `touchstart` – `touchend` interval) is not required for user classification. Therefore, to improve efficiency a possible approach could be to only register the `devicemotion` event listener whenever a `touchstart` event fires and to remove the `devicemotion` event listener again when a subsequent `touchend` event is detected. However, in our experiments we discovered that once the `devicemotion` event listener gets registered the actual recording of the acceleration suffers from a delay. This may cause the initial `touchstart` and (one or more) subsequent

`touchmove` events to have no or only inaccurate supplemental acceleration data. This may in turn result in a degraded classification performance. To solve this, we leave the `devicemotion` event listener registered indefinitely. In this case the incurred CPU and battery overhead is negligible (cf. also Section 5.11) as the continuous recording takes only place if the web page (containing the hooks) is in the foreground browser window. The recording is suspended if another app—or even only another browser tab—is in the foreground.

We note that capturing the touch events and device sensor data does not require any special privileges and is completely transparent for the user—as tested on Android 4.4.4 with Android Browser 4.4.4, Chrome 39.0.2171.59, and Firefox 33.1 as well as on iOS 7.1 with Safari 7.0 and Chrome 39.0.2171.45.

The browser periodically transmits the captured data in the background to the web server using AJAX via an `XMLHttpRequest`. Both TBV and CAM are implemented in Python and run within the web server. The TBV utilizes the *scikit-learn* open source machine learning library [125] to invoke Breiman's random forests algorithm [17]. The CAM continuously calculates the confidence value as explained in Section 5.8. If the ratings are consistently false and the confidence exceeds a certain threshold, CAM triggers a user reauthentication by the web server.

5.10 Classification Evaluation

We use the implementation described in Section 5.9 and adapt it within an experimental setup in order to evaluate the suitability of the extracted features (cf. Section 5.7) and to evaluate the overall user classification accuracy of our framework. The test web server hosts several hyperlinked web pages. Note that the web pages were intentionally not fabricated in such a way that the layout would likely be the most beneficial for the user classification accuracy of our framework. The reason is that we strived to obtain more universal real-world data and evaluation results. Therefore, the classification accuracy may improve for specially tailored web pages and may decline for less suitable web pages. In this regard, based on our findings in this chapter, such specially tailored web pages could consist of pronounced vertically distributed content as this would result in many vertical scroll gestures and consequently in a higher number of samples (cf. Figure 5.8). Furthermore, web pages with relatively large vertical dimensions—where the content is also largely spread out across the pages—presumably cause the user to perform larger scroll gestures which in turn may contain more usable information about features like acceleration, velocity or curvature (cf. Sec-

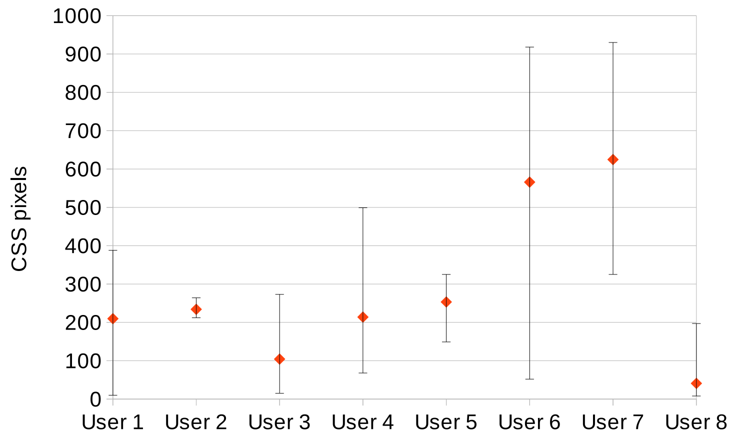


Figure 5.5: Minimum, mean, and maximum values calculated (on a per-user basis) over the heights S^h_k of all bounding boxes of the scroll gestures of eight random users.

tion 5.7). Finally, such specially crafted web pages increase the number of total (consecutive) scroll gestures and as such allow to improve the classification accuracy by examining sequences of scroll gestures instead of only single scroll gestures. This will be evaluated in the following in Section 5.10.2.

We recorded and evaluated touch interaction data sets of 45 users. The users were intentionally not informed about the experiment's objective of verifying user identities based on their touch interaction patterns in order for them to be unbiased. To evaluate our framework under real-world conditions, the users used their normal everyday devices. This entails that the data sets were obtained from a heterogeneous group of devices, operating systems, and web browsers.

5.10.1 Feature Suitability

We demonstrate the suitability of scroll gestures for user classification as used by our framework and show the rationale for selecting certain classification features. Figure 5.3 and Figure 5.4 depict the scrolling behaviors of two users. The visualization is based on the calculated intermediate points of each scroll gesture (cf. Section 5.7). Note that scroll gestures between these two users are quite distinctive (e.g., the curvature $t^{\measuredangle}_i$), yet the scroll gestures of a single user are relatively consistent. This property makes scroll gestures well suited for user classification.

Figure 5.5 and Figure 5.6 exemplarily show that the bounding boxes vary between different users. In particular, different mean values are char-

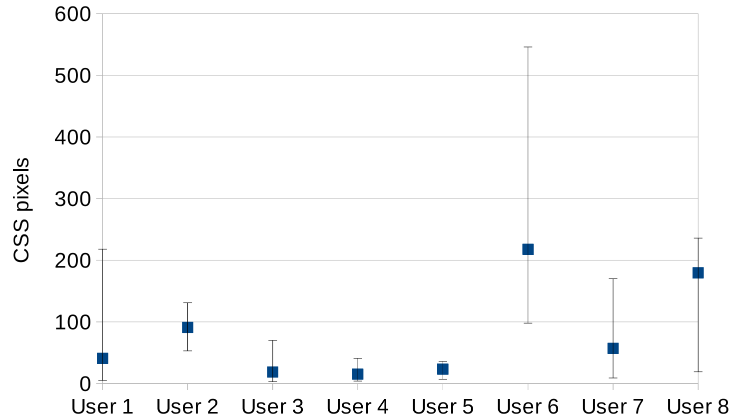


Figure 5.6: Minimum, mean, and maximum values calculated (on a per-user basis) over the widths S_k^w of all bounding boxes of the scroll gestures of eight random users. The users are the same as in Figure 5.5.

acteristic for different users. Note that in both Figure 5.5 and Figure 5.6 very low minimum values for both S_k^h and S_k^w (e.g., user 3) may also be caused by accidental screen touches interpreted as scroll gestures.

Figure 5.7 shows various features of a horizontal scroll gesture from right to left (user 1) and of a vertical scroll gesture from the bottom up (user 2), respectively. We use $I := 12$ for the evaluation which results in twelve intermediate points for a scroll gesture (plus start and end points both of which are omitted in the diagram). The *delta x* bars represent the intermediate offsets $t_1^{\triangle x}, \ldots, t_{12}^{\triangle x}$ of the respective scroll gesture in CSS pixels (analogously for *delta y*). The *curvature* bars show the angles $t_1^{\measuredangle}, \ldots, t_{12}^{\measuredangle}$ in degrees for each section of the scroll gesture. The *velocity x* bars show the (scaled) velocities $t_1^{vx}, \ldots, t_{12}^{vx}$ for each section of the scroll gesture (analogously for *velocity y*). These features reflect subtle but distinct individual user behavior, for example, by indicating where the curvature's peak values are located or by representing the velocity fluctuations within scroll gestures.

5.10.2 Classification Accuracy

We evaluate the user classification accuracy by using the de facto performance evaluation method for touch-based authentication systems on smartphones and other touch input devices (cf. [157, 34, 53, 138, 67]). In particular, the touch interaction data samples of a given user are tested against the touch behavior model of another user. The scores are obtained by cross-validation with a 70% (training set) to 30% (validation set) partitioning of the gathered data. We inspect the False Acceptance Rate (FAR) and False

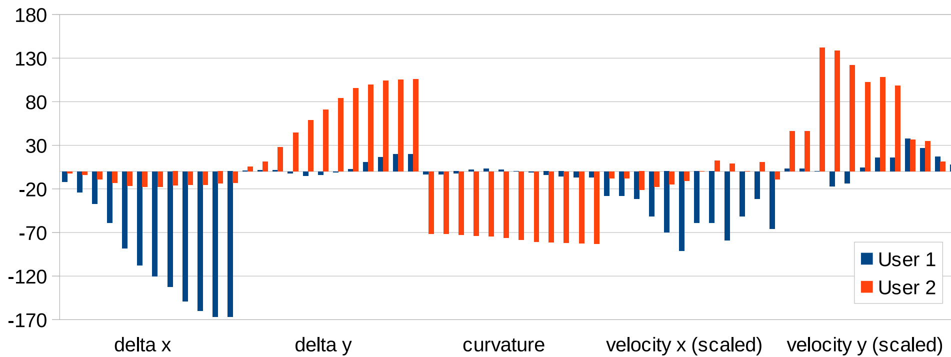


Figure 5.7: Feature value comparison of scroll gestures by different users. User 1 executes a horizontal scroll gesture from right to left. User 2 executes a vertical scroll gesture from the bottom up. We use $I := 12$ for the evaluation which results in twelve intermediate points for a scroll gesture.

Rejection Rate (FRR) based on different numbers of observations. The FAR is the probability that an illegitimate user is accepted. The FRR is the probability that a legitimate user is rejected.

Single Gesture

We analyze the classification accuracy based on only a single scroll gesture. In this case, the FAR and FRR are 23% and 22%, respectively. Figure 5.8 shows how the FAR and FRR develop for different users over an increasing amount of training data. Each pair of blue and red bars represents a single user with the specified number of samples. The reason the FAR and FRR fluctuate is because some users may be classified more easily than others (e.g., some user's features may be rather unique among the set of users and thus classification works well even with fewer samples). Nonetheless, for 70 or more samples the FAR and FRR development, on average, stabilizes and starts to improve noticeably. These results indicate that it is advisable to trigger actions (e.g., a forced reauthentication) not before this minimum number of samples has been acquired. The above results indicate, however, that user classification on the basis of only one gesture is not accurate enough to reliably verify users. This has even been shown to be the case when more precise interaction data can be gathered directly on the device [188, 14]. Therefore, we consider sequences of consecutive scroll gestures (as explained in Section 5.8). This enables us to significantly improve the classification accuracy.

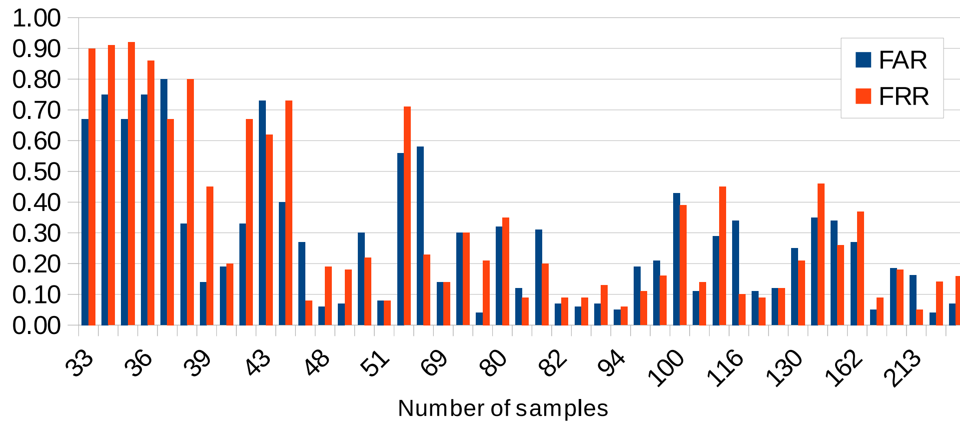


Figure 5.8: FAR and FRR based on only a single scroll gesture for (different) users with an increasing number of samples (non-linear x-axis). Each pair of blue and red bars represents a different user, thus FAR and FRR fluctuate.

Multiple Gestures

We evaluate how the classification accuracy changes within our experimental setup if we increase the number of considered scroll gestures. The graph in Figure 5.9 shows the FAR and FRR for different subsequence lengths. For subsequences containing just four scroll gestures, we are already able to achieve FAR and FRR of less than 10%. Further significant improvement requires more than ten scroll gestures, resulting in FAR and FRR of less than 5%. When increasing the subsequence length to 14 scroll gestures, both FAR and FRR can further be reduced to $< 1\%$ within the experimental setup. These results can be utilized to adjust the parameters (e.g., the minimum subsequence length) used for calculating the confidence values.

Note that the FAR and FRR may deteriorate even with an increasing subsequence length. In Figure 5.9, this is, for example, the case for the FAR between the subsequence length of six and seven. The reason is that the certainty score of the last observation of the subsequence may influence the confidence value in such a way that it causes a misbehavior that would not have occurred without the last observation. For example, without the last observation the system may (correctly) reject an illegitimate user based on the confidence value, whereas a (slightly) different confidence value (caused by the certainty score of the last observation) may lead the system to (incorrectly) accept the user. Finally note that in general the classification accuracy ultimately depends on the actual set of users: the more the users' gestures are alike (e.g., an attacker performs scroll gestures in a similar

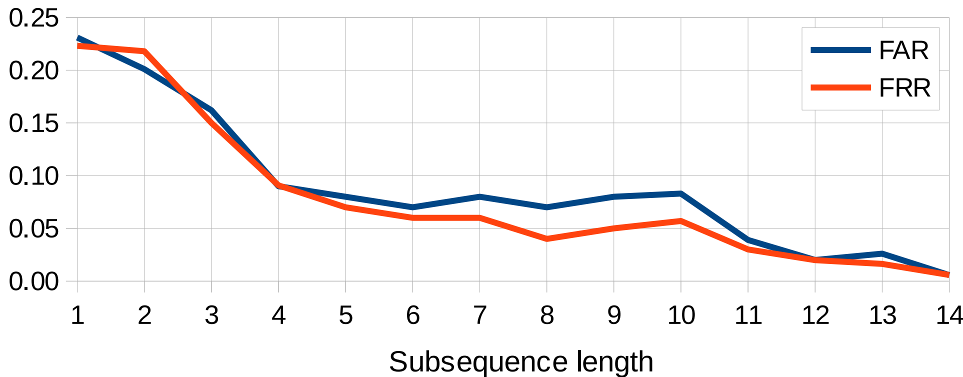


Figure 5.9: FAR and FRR for different subsequence lengths. Significant improvement is achieved after only considering four scroll gestures with FAR and FRR $< 10\%$. After ten scroll gestures, both FAR and FRR are $< 5\%$, and after 14 scroll gestures $< 1\%$.

fashion as the legitimate user) the more difficult it gets to correctly distinguish and classify them. Consequently, the FAR and FRR may fluctuate accordingly.

5.11 Performance Evaluation

We assess the practicality of our approach by conducting several experiments to evaluate the performance of the implementation. In the following, we first give a description of the experimental setup. In Section 5.11.1, we then evaluate the CPU overhead on the device, i.e., the additional CPU load required for the user identity verification. If this additional CPU load is too high, it could negatively influence the performance of other services and apps running on the device and thus it may have a negative impact on the overall user experience. In Section 5.11.2, we analyze how this affects battery life; this is important because if battery life is affected too negatively, users may be unwilling to accept the user verification. Furthermore, we investigate the network traffic overhead generated by a typical smartphone with enabled user verification in Section 5.11.3. If there is too much traffic overhead, users with a limited mobile data plan might be inclined to disable the user verification in order to save bandwidth.

Experimental Setup

For the practical performance evaluation in this section, we use a Nexus 6 equipped with a Snapdragon 805 (2.7 GHz quad-core Krait 450) as the test device with the following setup:

- Stock Android (Lollipop, version 5.1.1, with kernel 3.4.0)
- Enabled WiFi
- Disabled Bluetooth, GPS, and NFC
- Screen always on (in order to prevent Android from going into deep sleep mode or utilizing other energy-saving features); fixed minimum brightness with adaptive brightness feature disabled; disabled automatic screen timeout (using the app *Stay Alive* [119])
- Disabled Android's *app auto-update* functionality to prevent undesired CPU usage while running the experiment
- Fresh reboot of completely charged phone
- Chrome browser version 43.0.2357.93 (unless stated otherwise) with only 1 tab open containing our test website
- No additionally installed apps aside from the apps that were already installed on stock Android and *Stay Alive*
- No explicitly started apps except Chrome browser
- Device is put onto a table face-up and is not moved during the whole experiment

5.11.1 CPU Usage

We evaluate the CPU overhead on the device caused by our approach. If the CPU load caused by the user identity verification is too high, other services and apps might suffer from bad performance (e.g., deteriorating responsiveness). This could have a negative impact on the overall user experience and may cause users to disable the user verification. Furthermore, a high CPU load decreases battery life; we investigate this in Section 5.11.2.

We conduct an experiment where we simulate touchscreen interactions in a well-defined and reproducible manner. This is realized with the help of the monkeyrunner tool [112]. Within the experimental setup, our web server hosts an HTML test page containing sufficient content to allow for

```
1  #!/usr/bin/env monkeyrunner
2
3  import time
4  from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice
5
6  device = MonkeyRunner.waitForConnection()
7
8  a=(500,1700)
9  b=(500,700)
10 duration=1.0
11
12 for i in range(0, 900): # execute for 60 min
13     device.drag(a, b, duration, 30) # scroll down
14     time.sleep(1)
15     device.drag(b, a, duration, 30) # scroll up
16     time.sleep(1)
```

Listing 5.1: Monkeyrunner script used for the evaluation of the CPU usage and the generated network traffic (cf. Section 5.11.3)

vertical scrolling. We open this page in the test device's web browser prior to starting the experiment. A laptop is connected to the test device with a USB cable in order to execute the monkeyrunner instructions on the test device and to gather CPU load measurements using the Android Debug Bridge (ADB) [63]. The monkeyrunner script is shown in Listing 5.1. It causes alternating executions of scroll down and scroll up gestures on the web page.

The CPU load over a period of one hour is shown in Figure 5.10. The blue curve shows the CPU load with enabled user identity verification. The mean is 10.45%. The red curve shows the CPU load with disabled verification (scientific control). The mean is 7.88%. Thus, the additional CPU load required for the verification is about 2.57%.[2]

In the experiment, we discovered that measuring just the CPU overhead for the registered event listeners for `touchstart`, `touchmove`, and `touchend` (cf. Section 5.9) caused virtually no overhead, i.e., we obtained no statistically significant deviation from the scientific control. The same is true for determining the CPU overhead for sending the recorded touch event

[2] Note that in a real world scenario (i.e., no simulated touch interactions) the CPU load will be higher than in our experiment because of the processing of the user's touchscreen interactions, Android's gesture recognition, and so on. However, as this additional CPU load is caused in both cases—with and without enabled touchscreen interaction hooks—and since we are only interested in the relative difference between the two cases, we can neglect this fact.

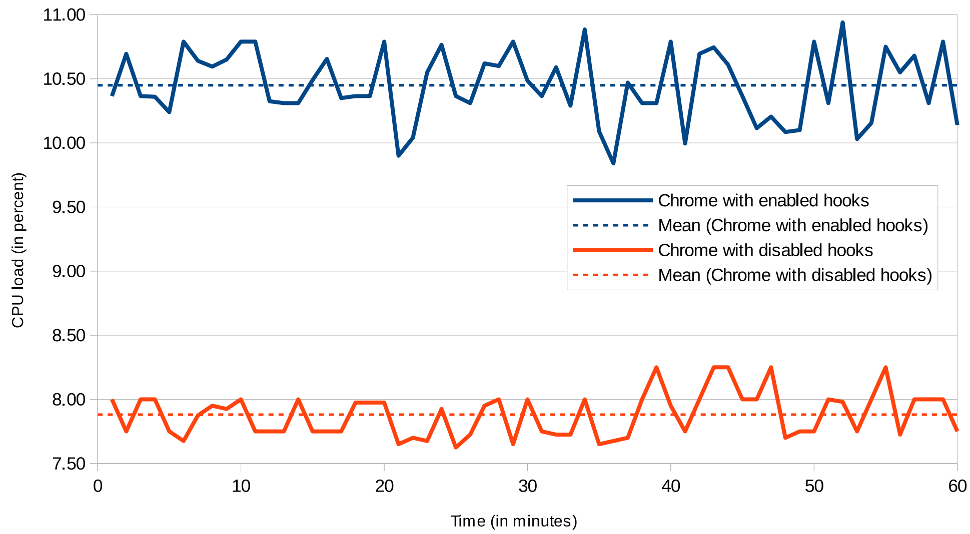


Figure 5.10: Comparison of CPU load of Chrome browser with enabled and disabled user identity verification. The mean for enabled verification is 10.45%, for disabled verification it is 7.88%.

data. However, registering the `devicemotion` event listener for recording the devicemotion while executing a scroll gesture has a significant impact on the increased CPU load shown in Figure 5.10. This knowledge could be used to reduce the CPU load of our approach by avoiding the utilization of the acceleration data (`devicemotion`) for user classification. Instead, the information about how much pressure the user applied to the touchscreen (`force`) could be used. The latter causes less CPU load, is supported by current browsers (cf. Table 5.1) and gives discriminative information for user classification [147]. However, the precision of the force attribute is rather low as the applied finger pressure is normally derived by inspecting auxiliary information like the finger's covered surface area. This problem is likely to be solved by emerging technologies like Apple's Force Touch [83] and 3D Touch [133], or Synaptics' ClearForce [74]. In our experiments, we discovered another problem: the update rate of the force attribute is rather slow causing multiple touchmove events to have the same force value even when the finger pressure is steadily changing. In this case, the force update rate may be too low to be utilized for reliable user verification.

	Chrome 43.0.2357.93	Opera 30.0.1856.93524	Firefox 39.0	Dolphin 11.4.17
Galaxy S4	✗	✗	✗	✗
Nexus 5	✓	✓	✓	✗
Nexus 6	✓	✓	✓	✗

Table 5.1: Tested devices and browsers supporting `force` attribute. The Galaxy S4 runs Android KitKat 4.4.4. The Nexus 5 and Nexus 6 both run Android Lollipop 5.1.1.

5.11.2 Battery Consumption

We evaluate how much battery overhead is caused by the user identity verification on the device. Having a low battery overhead is likely to be of high importance to users. There should be a reasonable trade-off between the increased security and the lower smartphone battery life. If battery life is too negatively affected, users may be unwilling to accept the user verification.

For measuring the battery overhead, we consider the recording of the scroll gestures and device sensor data (acceleration) as well as the transmission of the recorded data to the backend. For the latter, we distinguish whether the test device is connected to a WiFi network or is utilizing a mobile data connection (4G in our experiments). We do not use the monkeyrunner tool as in Section 5.11.1 because it would require us to connect a laptop to the phone via USB cable, thus causing the phone to be charged. This would interfere with our objective of measuring the battery consumption. Therefore, we only simulate the recording of touch interactions; this is possible without having a USB cable plugged into the phone. For this purpose, we patch the JavaScript code of the test web page such that the event listeners for `touchstart`, `touchmove`, `touchend`, and `devicemotion` will be periodically triggered without requiring any real (or monkeyrunner-simulated) touchscreen interaction. We take advantage of the knowledge gathered in Section 5.11.3 in order to trigger the event listeners as often as would be the case under real circumstances when executing scroll gestures with a duration of one second each. Thus, we are able to construct data records of the proper size and to transmit them to the backend.

We take advantage of the Linux power supply subsystem [90] and monitor the current charge counter[3] giving us relative, time-based measurements in μAh. This allows us to continuously track the battery usage over time.

[3] `/sys/class/power_supply/battery/charge_counter`

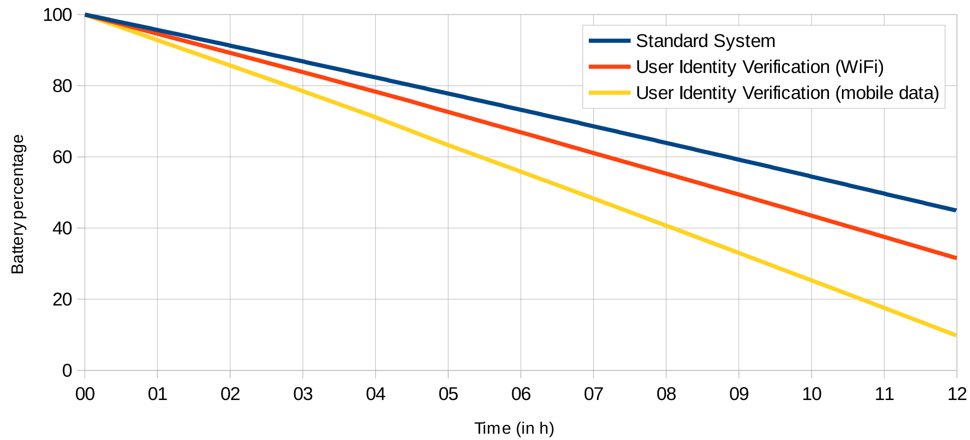


Figure 5.11: Comparison of battery usage (in percent) of a standard system without user identity verification, enabled verification connected via WiFi, and enabled verification connected via mobile data.

For this purpose, we deploy a shell script on the phone using adb [63] and execute it with the `nohup` command such that it ignores the hangup signal generated when the USB cable gets removed and continues to run.

The battery usage over a period of twelve hours is shown in Figure 5.11. The measurements are based on the recording of simulated scroll gestures executed successively—with a duration of one second each—and the transmission of the recorded data, as described above. The blue curve shows the battery percentage of the test device of a standard system without user identity verification (scientific control). The red and yellow curves show the battery percentage with enabled user verification where the test device is connected via WiFi and mobile data, respectively. Figure 5.12 shows the corresponding battery usage in μA over a period of 60 minutes.[4]

In the following, we calculate the average battery overhead of a more realistic scenario based on the measurement data of Figure 5.11 and Figure 5.12.

Overhead Calculation

In the following, we calculate the battery overhead for WiFi and mobile data. We first give an estimation of how much time an average user typically

[4]Note that the differences among the measurements are relative—not absolute—as we do not consider the battery usage caused by touchscreen interactions, Android's gesture recognition, and so on.

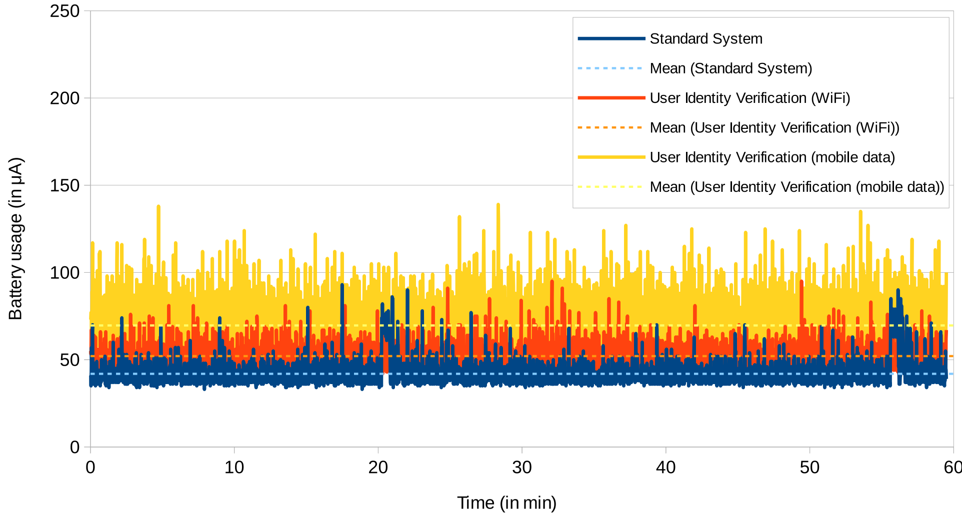


Figure 5.12: Comparison of battery usage (in μA) of a standard system without user identity verification, enabled verification connected via WiFi, and enabled verification connected via mobile data. The means are 41.90 μA (standard system), 52.06 μA (user verification via WiFi), and 69.63 μA (user verification via mobile data).

utilizes the user identity verification per day. Note that, in this regard, the verification technique is only used on a few websites during a typical day (e.g., online banking websites or a cloud configuration web frontend) where the web server makes use of our approach and injects the appropriate hooks. We estimate the *average time on site* for such a website to be 20 min = 1200 s. Furthermore, we estimate that in 10% of the time (120 s) there occur scroll gestures and that each scroll gesture has a duration of 1.0s, resulting in 120 scroll gestures. In the remaining time the user may be reading, typing, idling and so on. Assuming that there occur 3 sessions per day, this results in a total of $3 \cdot 120 = 360$ scroll gestures a day and entails a total recording time (caused by the injected hooks) of $t_r := 3 \cdot 120\ s = 360\ s$ a day.

Furthermore, we give a rough estimation on the total time an average user spends on his smartphone in order to compare it to the fraction of time actually spent doing user integrity verification. Based on the arithmetic mean of two recent surveys results [141, 96], we assume this to be 6 hours.

Of course, the actual user behavior strongly depends on multiple factors such as user preferences (e.g., slow vs. fast scrolling), website content (e.g., large vs. little amount of content), and use cases (e.g., changing only one

configuration option on the website vs. complex tasks). The actual user behavior may therefore deviate (significantly) from the above estimations.

Based on the measurements in Figure 5.12, the battery usage overhead using WiFi is $52.06\ \mu A - 41.90\ \mu A = 10.16\ \mu A$ per second. Given our estimated recording time above, we obtain an overhead of $360 \cdot 10.16\ \mu A = 3,657.60\ \mu A$ per day. The battery usage without enabled user identity verification over a duration of 6 hours (21600 seconds) is $21,600 \cdot 41.90\ \mu A = 905,040.00\ \mu A$. Consequently, the total battery overhead per day in percent is $3,657.60\ \mu A\ /\ 905,040.00\ \mu A \cdot 100 \approx 0.4\ \%$ for WiFi.

We calculate the overhead when using mobile data instead of WiFi in the same way. The battery usage overhead using mobile data is $69.63\ \mu A - 41.90\ \mu A = 27.73\ \mu A$ per second. Hence, we obtain an overhead of $360 \cdot 27.73\ \mu A = 9,982.80\ \mu A$ per day. Consequently, the total battery overhead per day in percent is $9,982.80\ \mu A\ /\ 905,040.00\ \mu A \cdot 100 \approx 1.1\ \%$ for mobile data.

The calculations show that the battery usage overhead is low.

5.11.3 Network Traffic Generation

We evaluate the network traffic overhead generated by a typical smartphone with enabled user identity verification. This information is especially interesting for users with a limited mobile data plan. If there is too much traffic overhead, users might be inclined to disable the user verification in order to save bandwidth.

In the following, we consider all recorded scroll gesture data and device sensor data sent over the network to the backend in order to calculate the traffic overhead. We estimate the amount of generated network traffic by first determining the number of fired JavaScript events (caused by the registered event listeners for `touchstart`, `touchmove`, and `touchend` as described in Section 5.9) for a typical scroll gesture. For this purpose, we use the monkeyrunner script shown in Listing 5.1 and count the number of fired events. The experiment is done on a Nexus 5 and Nexus 6 (both running Android Lollipop 5.1.1), and on a Samsung Galaxy S4 (running Android KitKat 4.4.4). On each device we use the browsers Chrome 43.0.2357.93, Opera 30.0.1856.93524, Firefox 39.0, and Dolphin 11.4.17. The results are shown in Figure 5.13.

The first observation is that in our experiment the number of fired events varies based on the utilized browser—but seems to be independent of the used device. In general, the rate at which a browser sends `touchmove` events is implementation-defined, and may depend on hardware capabilities

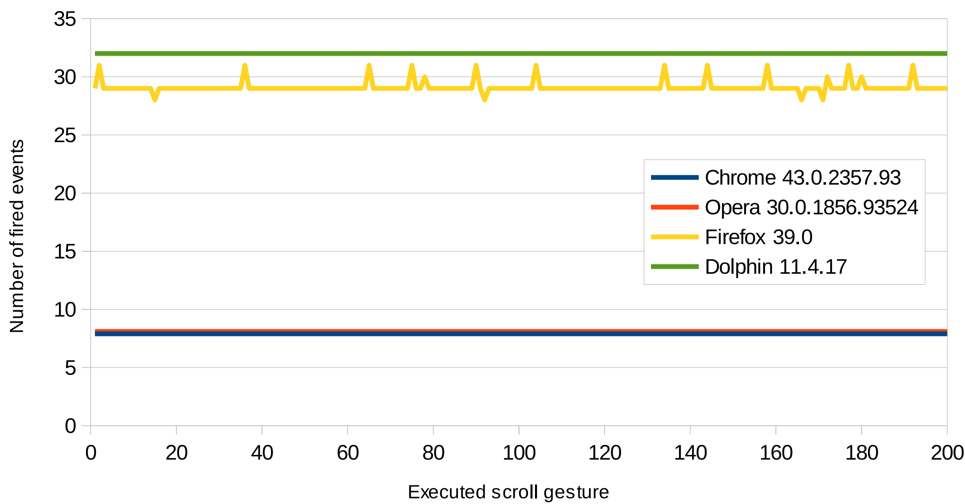


Figure 5.13: Comparison of the number of fired JavaScript events caused by the registered event listeners for `touchstart`, `touchmove`, and `touchend` for a typical scroll gesture using different web browsers. Chrome, Opera, and Dolphin all exhibit a constant number of fired events whereas Firefox generates a varying number of fired events.

and other implementation details [153]. In Figure 5.13, both Chrome and Opera exhibit a constant number of eight fired events (one `touchstart`, six `touchmove`, one `touchend`) per scroll gesture (both up and down). Dolphin generates a relatively high constant number of 32 fired events per scroll gesture. The only browser with a varying number of fired events—for both scroll down and scroll up gestures—is Firefox where the number lies in the interval $[28, 31]$ with an arithmetic mean of approximately 29 and a median of 29.[5]

We note that in our experiments, for both Firefox and Dolphin the actual number of fired events was also dependent on the content as well as the zoom-level of the website. With Chrome and Opera this was not the case, which, we suspect, is due to the fact that both Chrome and Opera use the web browser engine Blink [12], a fork of the WebCore component of WebKit [169], and therefore exhibit the same behavior.

[5]The relatively high number of fired events does not seem to be caused by Firefox's scroll animation as disabling it by using JavaScript's `preventDefault()` does not reduce the number of fired events.

	Fired events per day	Traffic/day in kB	Traffic/month in MB (% of quota)
Chrome 43.0.2357.93	2,880	126.56	3.71 (0.36%)
Opera 30.0.1856.93524	2,880	126.56	3.71 (0.36%)
Firefox 39.0	10,440	458.79	13.44 (1.31%)
Dolphin 11.4.17	11,520	506.25	14.83 (1.49%)

Table 5.2: Comparison of generated network traffic per browser. The obtained results are based on 360 scroll gestures a day with enabled user identity verification and a monthly quota of 1 GB.

In the next step, we determine the size of a typical data record generated as a result of the triggered events. A data record consists of the following fields:

Timestamp The number of milliseconds since 1970/01/01 as obtained by JavaScript's `getTime()` method, e.g., 1443688563854

Touch type One of the triggered events `touchstart`, `touchmove`, or `touchend` encoded as 1, 2, and 3, respectively

Touch coordinates The two-dimensional point on the device's touchscreen where the touch event occurred, e.g., (1621.53, 864.17)

Acceleration data The acceleration force along the device's x, y, and z axes at the time when the touch event fired, e.g., (4.21, 7.44, 0.51)

Hence, a typical data record may look like

1443688563854|3|1621.53|864.17|4.21|7.44|0.51

and has an (average) length of 45 bytes.[6]

Finally, we are able to calculate the amount of transmitted data for one scroll gesture by multiplying the determined number of fired events for a typical scroll gesture with the length of an average data record generated for each such fired event. With this knowledge, we utilize the user behavior model given in Section 5.11.2 to estimate the daily and monthly generated network traffic. The generated network traffic caused by different browsers is shown in Table 5.2.

[6] It is possible to reduce this length by using a more efficient encoding; however, this is out of the scope of this thesis.

Based on Figure 5.13, we now consider the mean over all browsers and obtain $\lceil (8+8+29+32)/4 \rceil = 20$ fired events for a single scroll gesture on average. Consequently, 360 scroll gestures a day cause $360 \cdot 20 = 7,200$ fired events. This results in $7,200 \cdot 45\ B = 324,000\ B \approx 316.41\ kB$ network traffic per day, or $30 \cdot 316.41\ kB = 9,492.30\ kB \approx 9.27\ MB$ per month. Assuming the user's limited mobile data plan has a quota of 1 GB, the user verification requires approximately 0.91%[7] on average. The calculations show that taking advantage of the presented user identity verification approach is feasible w.r.t. network traffic and only incurs a small overhead on the monthly quota.

5.12 Security Analysis

In this chapter, we deal with the scenario that a remote server tries to determine if a client really is who he claims to be by assessing whether the current user behavior fits the (previously learned) user behavior associated with the claimed credentials. Our objective is to protect the user accounts hosted by such a remote server against account abuse. However, we do not focus on protecting the user's touch device itself—this may be realized by complementary research [191, 34, 1, 192, 56, 14]. In the following, we evaluate the security of our continuous user identity verification approach. We point out that the majority of cases where our approach thwarts account abuse is presumably against ordinary attackers not being aware of the utilized user identity verification. However, in the following we deliberately consider a more sophisticated attacker with complete knowledge of the employed user identity verification mechanisms who tries to actively circumvent them. Finally, note that orthogonal attacks not targeting the user identity verification itself but, for example, targeting the payload data (e.g., a trojan manipulating the amount and target account of a bank transfer) are out of the scope of this discussion.

5.12.1 Blocking Attack

In a blocking attack, an attacker blocks or disables the transmission of the touch interaction data to the server in order to not get recognized as another (malicious) user accessing the legitimate user's account. This may be realized by tampering with the JavaScript touch interaction hooks responsible for recording and sending the touch interaction data. The attack may be launched on the attacker's device or on the legitimate user's device (e.g.,

[7] Less so when also taking advantage of WiFi connections.

the attacker steals the device). In the first case, the attacker first has to circumvent the usual security measures employed by the server (e.g., PIN entry) to access the user account. In the second case, this might not be necessary as the legitimate user might still be logged into some accounts (active sessions).

However, the server is able to detect that even though there occurs user interaction on the website (e.g., HTTP requests for a bank transfer), there is no expected corresponding touch interaction data available—which consequently means that this likely constitutes an attack.

5.12.2 Imitation Attack

In an imitation attack, an attacker tries to impersonate a legitimate user by physically imitating the user's touch interaction behavior. The attacker may use direct observation techniques such as shoulder surfing [95] and camera-based recording to examine the user behavior. As with a blocking attack, this may either be done on the user's device (e.g., stolen device—possibly with active sessions) or the attacker's device.

In both cases, the difficulty is that the attacker is required to imitate the legitimate user's individual behavior patterns with high precision. However, research has shown that this is very hard to achieve [14]. We note that a sophisticated attacker may still be able to carry out the imitation in an automated manner though. Such an attack could involve, say, a high-resolution camera recording the user's touchscreen interaction, then inferring the exact corresponding touch event data, and finally using this data in a replay attack as described in the following.

5.12.3 Replay Attack

A replay attack is similar to an imitation attack in that an attacker tries to impersonate a legitimate user by imitating the user's touch interaction behavior. However, in contrast to an imitation attack, the attacker does not try to physically imitate the behavior patterns but rather (re)uses touch interaction data intercepted from the legitimate user while he was accessing his account. In the following, we will discuss several attack vectors.

Network Attack

The attacker intercepts the network communication between the user's device and the server in order to extract the touch interaction data required for the replay attack. However, we require that the communication between

the client and server is secured (cf. Section 5.5) by complementary techniques such as Secure Sockets Layer (SSL) [57], Transport Layer Security (TLS) [36], and HTTPS [139], respectively. In particular, this provides a private and authenticated connection between the user's device and the server, which prevents the interception of the user's touch interaction data.

Malicious Website Attack

The attacker operates a malicious website containing JavaScript code to record the user's touch interaction required for the replay attack. The attacker may trap the user to visit the website by employing additional techniques such as phishing [136]. A possible countermeasure is to have the user's device only accept and execute JavaScript code—in particular, code containing critical event listeners for `touchstart`, `touchmove`, `touchend`, and `devicemotion` (cf. Section 5.9)—from trustworthy websites identified by a valid server certificate. The level of protection against such attacks also depends on the mechanisms provided by the mobile operating system and browser on the user's device. The fact that it is possible for an arbitrary website to employ JavaScript code capable of recording touch interactions and device sensor data from the user *without requesting permission first*[8] is regarded by us as a general security and privacy issue of the affected operating systems and browsers. In fact, recent research builds upon this issue and demonstrates how user security can be compromised using sensor data gathered through malicious JavaScript code [104].

Local Device Attack

The attacker gains physical access to the legitimate user's device and tries to acquire the user's touch interaction data required for the replay attack. However, we deliberately never store the recorded (and forwarded) touch interaction data from previous sessions on the device. Therefore, the attacker has to compromise the device before the touch interactions get recorded. This may be done by first installing malicious software capable of stealthily logging the user's touch interactions, then returning the compromised device, and finally receiving the logged data over the internet—possibly through a covert channel. This shows that the attack requires a relatively high effort as well as physical proximity to the user, but it still may be feasible. As a possible countermeasure, the server could check if the touch interaction data (being replayed by the attacker) has already been used before (by the

[8]Tested on Android and iOS with various web browsers (cf. Section 5.9).

legitimate user), possibly using some sort of heuristics in order to recognize slightly altered replayed data. Another possible countermeasure involves comparing the actions of the attacker on the website (e.g., actions for initiating a bank transfer) with the replayed data to determine whether they match up in a plausible way. However, realizing such countermeasures in a reliable manner is a non-trivial task and out of the scope of this work.

Remote Device Attack

This attack is similar to a local device attack but this time the attacker does not have physical access to the device but rather tries to remotely compromise the device to acquire the user's touch interaction data. This is usually done by exploiting a vulnerability in the device's software or by tricking the user into installing a malicious app.

In the first case, the attacker may control a malicious website containing code for exploiting a web browser vulnerability of the user's device. However, depending on the browser, the attacker may has to circumvent additional security measures such as Chrome's sandboxing mechanism [6] which prevents one browser process (one "browser tab") from eavesdropping what happens in another browser process or manipulating data of the other process. An attack launched through such a malicious website does therefore not necessarily entail that one compromised browser process allows access to touch interaction data generated in another browser process. In general, mobile operating systems often follow the *principle of least privilege* [149] in order to mitigate similar exploitation.

In the second case, the attacker may trick the user into installing a malicious app provided, say, through the Google Playstore [65] or F-Droid [68], which records touch interactions and forwards them to the attacker. However, the app sandboxing mechanisms employed by modern mobile operating systems (e.g., Android's Application Sandbox [64] and iOS's App Sandboxing [2]) prevent the malicious app from having access to the touch interaction data of the browser app.

5.13 Related Work

The field of biometric authentication is usually divided into two categories: physiological and behavioral biometrics. Physiological biometrics consider static physical attributes like human fingerprints, facial features, or DNA. Behavioral biometrics distinguish user behavior such as speaking, walking, or typing. We focus on behavioral biometrics.

Early work considers how users interact with computer peripherals like mice and keyboards. In [113], Monrose et al. combine keyboard typing patterns with the user's password to generate a hardened password. In more recent work [143], Roth et al. analyze the user's keyboard typing behavior as observed by a webcam pointing toward the keyboard. Zheng et al. [190] and Feher et al. [50] verify user identities according to characteristics of their interaction with a computer mouse.

The focus of recent research has shifted towards analyzing the interaction behavior with modern touch devices like smartphones and tablets. In [191], Zheng et al. build a user verification system based on tapping behaviors to increase the security when entering a PIN. Kolly et al. [85] programmed and published a quiz game to collect and analyze tap interaction data. The extracted features include pressure dynamics such as time of the pressure peak and gradients of the pressure. De Luca et al. [34] and Angulo et al. [1] analyze how patterns are drawn on a device's touchscreen in order to enhance smartphone lock patterns. We do not include tap dynamics for classification as in web contexts discriminative tap features (e.g., finger pressure) cannot be obtained at all or only with low precision.

In addition to touchscreens, modern smartphones are equipped with a wide variety of sensors including accelerometers and gyroscopes. While the data obtained from these sensors is a useful resource for inferring rather harmless information about users (e.g., recognizing user activities [86]), such sensor data can also be used to infer security and privacy critical user information. Owusu et al. [121] exploit accelerometer readings as a side channel to extract sequences of entered text on smartphone touchscreen keyboards, allowing them to break passwords. The recent work in [161] shows practical defense mechanisms against such kind of inertial sensor attacks by introducing artificial sensory noise. In [109], Miluzzo et al. identify tap locations on the screen to infer passwords based on accelerometer and gyroscope data. Neverova et al. [118], Mantyjarvi et al. [100], and Kumar et al. [87] explore how to identify users based on natural human kinetics, gait patterns, and arm movement patterns, respectively. While those scenarios are different from ours, the same hooking technique used in our work could be utilized by an attacker to capture accelerometer data over the web. In fact, in work developed parallel to ours, Bojinov et al. [15] exploit this technique in order to de-anonymize mobile devices as they connect to web sites.

The following work combines touch behavior with device sensor data to verify users and is closest to our research. Zhu et al. [192] construct a behavior model based on user gestures and device sensor data from accelerometers, gyroscopes, and magnetometers. Their prototype allows them

to identify smartphone owners and non-owners. In [56], Frank et al. examine basic navigation maneuvers such as scroll gestures and analyze the effectiveness of various features. Bo et al. [14] analyze user classification based on tap and scroll gestures along with the device's feedback (accelerometer and gyroscope) to these actions. Both [56] and [14] improve the classification accuracy by inspecting multiple, consecutive observations. This approach is similar to ours, however, we calculate a confidence value based on additional parameters like the minimum required length of observations and different weights for the certainty scores of individual gestures.

Google's recent research Project Abacus [118, 158] demonstrates that human kinematics convey necessary information about person identity and therefore can be useful for user authentication on mobile devices. They suggest to augment and improve the results by additionally analyzing user touch patterns (along with other relevant characteristics) as done in this chapter.

In this thesis, and in contrast to other work, we present a framework that does not require any special software or privileges on the user's smartphone. We rather take advantage of standard mobile web browser capabilities to remotely capture and analyze touchscreen interactions in order to continuously verify user identities.

5.14 Summary

We have presented a framework that allows us to continuously verify user identities from remote servers by analyzing user interaction behavior with smartphone touchscreens. This enables us to protect user accounts by disabling critical functionality and enforcing a reauthentication in case of suspicious behavior. In particular, this allows us to improve the security of the system we developed in the previous chapters by continuously authenticating users and system administrators. Our solution is widely applicable on everyday smartphones and does not require any special software or privileges on the device, and is completely transparent for the user. We have shown how to successfully classify users even on the basis of limited and imprecise touch interaction data. This is achieved by constructing a touch behavior model of the user and only selecting features that possess sufficient precision and are available on all touch devices, operating systems, and web browsers. For user classification based on only a single scroll gesture, we are able to achieve FAR and FRR of 23% and 22%, respectively. We show how to significantly improve the classification accuracy by considering sequences of observations instead of only single touch interactions. This technique is

used in the calculation of a confidence value that allows for a more stable and reliable assessment of whether the current smartphone user is indeed the legitimate user. The final performance evaluation of our framework implementation shows that both FAR and FRR can be as low as $< 1\%$ after collecting a sequence of about 14 scroll gestures.

Chapter 6

Conclusion and Future Work

In this thesis, we have developed methods for improving system security based on a combination of using a TPM, virtualization, and continuous user authentication. A TPM can be used to securely store integrity measurements and protect them from manipulation. However, currently it is not possible to adequately combine a TPM and virtualization without lowering the security level provided by the TPM. Virtualization can be used to monitor critical operations and to take integrity measurements of multiple, isolated VMs from outside of the VMs. However, attackers may be able to evade monitoring by attacking special hooks placed inside the VMs which are responsible for bridging the semantic gap introduced by the virtualization layer. Computer systems secured through both a TPM and virtualization, can often still be exploited through compromised user accounts, in particular, critical accounts like system administrator accounts. Therefore, it is crucial to continuously authenticate users in order to detect and disable compromised accounts, thus further enhancing the overall system security. The work in this thesis has attempted to solve these problems by storing and multiplexing integrity measurements directly in the hardware TPM, monitoring VMs through a filesystem-based monitoring approach with secured hooks, and protecting accounts through continuous user authentication based on behavioral biometrics.

We have explored novel mechanisms to multiplex integrity measurements originating from arbitrarily many VMs in a secure and privacy-aware manner in a hardware TPM, thus achieving a higher level of security compared to existing approaches emulating PCRs in software. We have taken advantage of virtualization to develop a system that enables us to monitor multiple operating systems from “outside of the box”, take integrity measurements on a per-VM basis and securely store them in a hardware TPM, and to detect and

prevent critical file operations through policy-based access control mechanisms. We have shown new ways of utilizing continuous user authentication based on user interaction behavior with smartphone touchscreens in order to better protect user accounts such that lost, stolen, or compromised user devices do not lead to compromised user accounts and systems. Finally, the proposed concepts have been implemented and evaluated as proof of concepts. This has enabled us to demonstrate the practicality of the concepts proposed in this thesis.

6.1 Contributions

In the following, we describe our contributions in detail. In Chapter 3, we have shown how to multiplex integrity measurements originating from arbitrarily many VMs with just a single standard TPM and only requiring one PCR. In contrast to existing work that emulate PCRs in software, our approach achieves a higher level of security since measurements, along with the mapping to their respective VMs, will always be stored in the hardware-protected PCRs of the TPM. We have presented a remote attestation protocol for attesting the integrity of individual VMs. A crucial problem we had to solve in this context, was that our approach of sharing PCRs among VMs, inherently requires the disclosure of all measurements of all VMs in order to retain the security guarantees of the TPM. This entails security and privacy issues as even a legitimate challenger in the remote attestation protocol is then able to determine exactly which software is running in all other VMs. This information might then, for example, be used to exploit (known) vulnerabilities of that software. We have solved the problem by a novel approach for concealing and storing measurements and their associated VM-IDs in the PCR that allows us to only reveal certain measurements of the hash chain stored in the PCR without degrading the security guarantees of the TPM. However, this approach inherently poses the risk of a MITM launching blinding attacks where the measurements and VM-IDs will be substituted with their corresponding concealed pairs in order to hide certain measurements of the attested VM. Therefore, we have conducted an exhaustive analysis of blinding attacks and have shown that our integrity validation algorithm protects against all of them. We have implemented a proof of concept to take and multiplex integrity measurements by utilizing the QEMU emulator and extending IMA, and evaluated the performance for different numbers of virtual machines.

In Chapter 4, we have explored a virtualization-based architecture that allows for “outside of the box” file integrity monitoring. In contrast to exist-

ing work, our approach has the advantage that hooks placed inside the VMs, for detecting and preventing file operations, are protected against manipulation by malware. We have achieved this through relocating a supervised VM's entire filesystem into the isolated realm of the host. The only way of accessing and manipulating the VM's filesystem is by communicating with a privileged component located at the hypervisor-level which has exclusive access to the VM's filesystem. We have demonstrated that this makes it impossible for attackers to manipulate files without being detected. Building upon this work, we have presented a policy-based access control mechanism for enforcing file protection. In this context, we have shown how to mitigate the problem of too restricting and inflexible policy rules by enabling VM users to autonomously install, remove, upgrade, and downgrade software packages in a secure and controlled manner, without the need of requiring the intervention of the administrator of the physical system. We have implemented and evaluated a proof of concept using a minimalist native Linux KVM virtualization solution, the virtio framework, and the paravirtualized Plan 9 filesystem protocol.

In Chapter 5, we have developed a framework that allows us to continuously verify a user's identity by remotely analyzing the user's touch behavior using machine learning classification techniques. This enables us to improve the security of computer systems and to protect user accounts by disabling critical functionality and enforcing a reauthentication in case of suspicious behavior. We have shown that our solution is widely applicable on everyday smartphones since, in contrast to existing work, we do not require any special software, prior setup, or special privileges on the user's smartphone. We have rather taken advantage of standard mobile web browser capabilities to remotely capture and analyze touchscreen interactions in order to continuously verify user identities. In contrast to existing work, we do not have direct access to the API of the touch device's operating system. In this context, we have shown how to successfully classify users even on the basis of limited and imprecise touch interaction data. This has been achieved by constructing a touch behavior model of the user and only selecting features that possess sufficient precision and are available on all touch devices, operating systems, and web browsers. We have demonstrated how to significantly improve the classification accuracy by considering sequences of observations instead of only single touch interactions. This technique has been used in the calculation of a confidence value that allows for a more stable and reliable assessment of whether the current smartphone user is indeed the legitimate user. We have implemented a proof of concept using the JavaScript Touch Event API, the scikit-learn open source machine learn-

ing library, and Breiman's random forest algorithm. We have evaluated the user classification accuracy and conducted several experiments to evaluate the CPU overhead, network traffic overhead, and battery life.

6.2 Future Research

The research in this thesis has led to new methods for improving system security based on a combination of using a TPM, virtualization, and continuous user authentication. In the following, we outline possible future research directions.

In Chapter 3, we have shown how to multiplex integrity measurements of VMs in a single PCR of a hardware TPM, thus achieving a higher level of security than existing approaches like vTPM. It would be interesting to examine whether additional functionality provided by a vTPM could be transferred and realized directly in a hardware TPM instead. However, we suspect this to predominantly be the case in rather special circumstances and use cases because of the TPM's hardware restrictions. Therefore, future research could explore how to combine the advantages of vTPMs, like cryptographic key handling per VM, with our approach of multiplexing integrity measurements directly in a hardware TPM in order to combine the advantages of both worlds.

In Chapter 4, we have taken advantage of the Plan 9 filesystem protocol 9P to relocate a guest VM's filesystem to the host. Because 9P is designed as a network protocol, future research could explore and evaluate the performance of various distributed setups where VMs and their respective filesystems are located at physically separated sites. Furthermore, we have described our heuristic approach of detecting program execution of ELF files under Linux based on a sequence of signature 9P requests. This approach could be extended to other file formats and could also include the execution of script files (e.g., shell scripts) to cover a broader range of executed files.

In Chapter 5, we have explained that a user's finger pressure on a touchscreen gives discriminative information for user classification but currently suffers from two problems in web contexts: lack of precision and slow update rate. Regarding the former, emerging technologies like Apple's Force Touch and 3D Touch, or Synaptics' ClearForce might yield improved user classification accuracy. Likewise, we assume improvements in user classification performance once the slow update rate of current web browsers will be increased. For our touch behavior model, we have assumed that a user utilizes an arbitrary but fixed touch device. In order to accommodate for multiple heterogeneous groups of devices, operating systems, and web

browsers for a given user, further research may investigate how to best approach this challenge, for example, how to distinguish different devices and how to maintain touch behavior sub-models. We have evaluated the user classification accuracy based on touch interaction data sets of 45 users. It would be interesting to examine how an increased number of users affects the FAR because of the increased probability of different users exhibiting similar gestures. Furthermore, we have evaluated the user classification accuracy by using the de facto performance evaluation method for touch-based authentication systems on smartphones and other touch input devices. Recent research utilizing robotic attacks on touch-based authentication [157] may act as a starting point for more sophisticated security benchmarking leading to improved and better protected touch-based authentication systems. We have argued that the fact that it is possible for an arbitrary website to employ JavaScript code capable of recording touch interactions and device sensor data from the user without requesting permission first, is a general security and privacy issue of current mobile web browsers as tested on Android and iOS. Future research may complement the research in this thesis by developing methods to better protect the user from such unintended and potentially harmful JavaScript code.

Bibliography

[1] Angulo J. and Wästlund E. Exploring Touch-Screen Biometrics for User Identification on Smart Phones. In *Privacy and Identity Management for Life*, pages 130–143. Springer, 2012. (Cited on pages 93, 111, 115)

[2] Apple Inc. App Sandboxing. Apple Developer Documentation. `https://developer.apple.com/app-sandboxing/`. *Last access: March 03, 2017*. (Cited on page 114)

[3] Arthur W. and Challener D. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, Berkely, CA, USA, 1st edition, 2015. ISBN 1430265833, 9781430265832. (Cited on pages 2, 11)

[4] Azab A. M., Ning P., Sezer E. C., and Zhang X. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *ACSAC*, pages 461–470. IEEE Computer Society, 2009. ISBN 978-0-7695-3919-5. (Cited on pages 3, 77, 79)

[5] Barham P., Dragovic B., Fraser K., Hand S., Harris T., Ho A., Neugebauer R., Pratt I., and Warfield A. Xen and the Art of Virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. (Cited on page 16)

[6] Barth, A. and Jackson, C. and Reis, C. and Google Chrome Team. The Security Architecture of the Chromium Browser. Journal Paper, 2008. (Cited on page 114)

[7] Behavioural Insights Team. Reducing Mobile Phone Theft and Improving Security. *International Crime and Policing Conference. Home Office Research report*, 2015. (Cited on pages 1, 4)

[8] Bellard F. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, Berkeley, CA, USA, 2005. USENIX Association. (Cited on pages 16, 42, 69)

[9] Berger S., Cáceres R., Goldman K. A., Perez R., Sailer R., and van Doorn L. vTPM: Virtualizing the Trusted Platform Module. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association. (Cited on pages 3, 25, 44, 45)

[10] Berger S., Cáceres R., Pendarakis D., Sailer R., Valdez E., Perez R., Schildhauer W., and Srinivasan D. TVDc: Managing Security in the Trusted Virtual Datacenter. *ACM SIGOPS Operating Systems Review*, 42(1):40–47, 2008. (Cited on page 45)

[11] Bishop C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 0387310738. (Cited on pages 20, 21)

[12] Blink Rendering Engine for the Chromium Project. Project Page. `http://blog.chromium.org/2013/04/blink-rendering-engine-for-chromium.html`. *Last access: March 03, 2017.* (Cited on page 109)

[13] Block S. and Popescu A. Device Orientation Event Specification. W3C Working Draft, W3C, 2011. (Cited on page 95)

[14] Bo C., Zhang L., Li X.-Y., Huang Q., and Wang Y. SilentSense: Silent User Identification via Touch and Movement Behavioral Biometrics. In *Proceedings of the 19th Annual International Conference on Mobile Computing & Networking*, pages 187–190. ACM, 2013. (Cited on pages 83, 84, 85, 89, 92, 99, 111, 112, 116)

[15] Bojinov H., Michalevsky Y., Nakibly G., and Boneh D. Mobile Device Identification via Sensor Fingerprinting. *arXiv preprint arXiv:1408.1416*, 2014. (Cited on page 115)

[16] Bray R., Cid D., and Hay A. *OSSEC Host-based Intrusion Detection Guide*. Syngress, 2008. (Cited on page 50)

[17] Breiman L. Random Forests. *Machine Learning*, 45(1):5–32, 2001. ISSN 0885-6125. doi: 10.1023/A:1010933404324. (Cited on pages 21, 93, 96)

[18] Brickell E., Camenisch J., and Chen L. Direct Anonymous Attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145. ACM, 2004. (Cited on page 14)

[19] Broenink R. Using Browser Properties for Fingerprinting Purposes. In *16th biennial Twente Student Conference on IT, Enschede, Holanda*, 2012. (Cited on page 85)

[20] Burg D. et al. US cybercrime: Rising risks, reduced readiness. Key findings from the 2014 US State of Cybercrime Survey. Technical Report, PricewaterhouseCoopers, 2014. (Cited on page 1)

[21] Cajucom E., Dacuno P., Aquino K., Aquilino B., Hilyati A., Jamaludin S., Pilkey A., and Michael M. Threat Report 2015. Technical Report, F-Secure Corporation, 2016. (Cited on page 1)

[22] Canonical Ltd. AppArmor (Application Armor) Security Project. Official AppArmor Website. http://apparmor.net. *Last access: March 03, 2017.* (Cited on page 50)

[23] Chen C., Raj H., Saroiu S., and Wolman A. cTPM: A Cloud TPM for Cross-Device Trusted Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014. (Cited on page 45)

[24] Chen L., Landfermann R., Löhr H., Rohe M., Sadeghi A.-R., and Stüble C. A Protocol for Property-based Attestation. In *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16. ACM, 2006. (Cited on page 14)

[25] Chen P. M. and Noble B. D. When Virtual is Better than Real. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 133–138. IEEE, 2001. (Cited on pages 15, 51)

[26] Cheng W., Dailey S., Frosst D., Greve P., Hux T., et al. 2016 Threats Report, June 2016. Technical Report, McAfee Labs and Intel Security, 2016. (Cited on page 1)

[27] Ciampa M. *Security+ Guide to Network Security Fundamentals*. Cengage Learning, 2012. (Cited on page 61)

[28] Common Internet File System (CIFS) Protocol. Microsoft TechNet Library. https://technet.microsoft.com/en-us/library/cc939973.aspx. *Last access: March 03, 2017.* (Cited on page 53)

[29] Cortes C. and Vapnik V. Support-Vector Networks. *Machine learning*, 20(3):273–297, 1995. (Cited on page 21)

[30] Cover T. and Hart P. Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1):21–27, 1967. (Cited on page 21)

[31] Cucurull J. and Guasch S. Virtual TPM for a Secure Cloud: Fallacy or Reality? *Universidad de Alicante*, 2014. (Cited on page 45)

[32] Damopoulos D., Kambourakis G., and Gritzalis S. From Keyloggers to Touchloggers: Take the Rough with the Smooth. *Computers & Security*, 32:102–114, 2013. (Cited on page 83)

[33] Danev B., Masti R. J., Karame G. O., and Capkun S. Enabling Secure VM-vTPM Migration in Private Clouds. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 187–196. ACM, 2011. (Cited on page 45)

[34] De Luca A., Hang A., Brudy F., Lindner C., and Hussmann H. Touch Me Once and I know it's You!: Implicit Authentication Based on Touch Screen Patterns. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 987–996. ACM, 2012. (Cited on pages 98, 111, 115)

[35] Denning D. E. An Intrusion-Detection Model. *IEEE Transactions on software engineering*, (2):222–232, 1987. (Cited on page 2)

[36] Dierks, T. and Rescorla, E. The Transport Layer Security (TLS) Protocol, Version 1.2. RFC 5246, 2008. (Cited on pages 88, 113)

[37] Dimensional Research. The Impact of Mobile Devices on Information Security: A Survey of IT Professionals. Technical Report, Dimensional Research, 2013. (Cited on pages 1, 4)

[38] Domingos P. A Few Useful Things to Know about Machine Learning. *Communications of the ACM*, 55(10):78–87, 2012. (Cited on page 20)

[39] Dove, A. Fileless Malware – A Behavioural Analysis Of Kovter Persistence. Reverse Engineering Blog Article, Airbus Defence & Space, 2016. http://blog.airbuscybersecurity.com/post/2016/03/FILELESS-MALWARE—A-BEHAVIOURAL-ANALYSIS-OF-KOVTER-PERSISTENCE. *Last access: March 03, 2017.* (Cited on page 76)

[40] Duda R. O., Hart P. E., and Stork D. G. *Pattern Classification*. John Wiley & Sons, 2012. (Cited on page 21)

[41] Dunlap G. W., King S. T., Cinar S., Basrai M. A., and Chen P. M. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *ACM SIGOPS Operating Systems Review*, 36(SI): 211–224, 2002. (Cited on pages 3, 78)

[42] Eckersley P. How Unique Is Your Web Browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010. (Cited on page 85)

[43] Eckert C. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Oldenbourg, 9th edition, 2014. (Cited on page 2)

[44] Elaine Barker and Allen Roginsky. Transitions: Recommendation for Transitioning the Use of Cryptographic Algorithms and Key Lengths. In *Archived NIST Technical Series Publication, SP 800-131A*. National Institute of Standards and Technology, 2011. (Cited on pages 12, 32)

[45] Enberg, P. Native Linux KVM Tool. Project Page. `https://github.com/penberg/linux-kvm`. *Last access: March 03, 2017*. (Cited on page 69)

[46] England P. and Loeser J. Para-Virtualized TPM Sharing. In Lipp P., Sadeghi A.-R., and Koch K.-M., editors, *Trusted Computing - Challenges and Applications*, volume 4968 of *Lecture Notes in Computer Science*, pages 119–132. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-68978-2. (Cited on pages 3, 25, 44, 45)

[47] Ezirim K., Khoo W., Koumantaris G., Law R., and Perera I. M. Trusted Platform Module–A Survey. *The Graduate Center of The City University of New York*, 2012. (Cited on page 11)

[48] Fang K., Hanus D., and Zheng Y. Security of Google Chromebook. *Massachutts Institute of Technology Cambridge, MA*, 2139, 2010. (Cited on page 2)

[49] Federal Bureau of Investigation (FBI). 2015 Internet Crime Report. Technical Report, Internet Crime Complaint Center, 2015. (Cited on page 1)

[50] Feher C., Elovici Y., Moskovitch R., Rokach L., and Schclar A. User Identity Verification via Mouse Dynamics. *Information Sciences*, 201: 19–36, 2012. (Cited on pages 83, 115)

[51] Feller T., Malipatlolla S., Kasper M., and Huss S. dcTPM: A Generic Architecture for Dynamic Context Management. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 211 –216, 30 2011-dec. 2 2011. (Cited on pages 25, 44, 45)

[52] Feng T., Liu Z., Carbunar B., Boumber D., and Shi W. Continuous remote mobile identity management using biometric integrated touch-display. In *Microarchitecture Workshops (MICROW), 2012 45th Annual IEEE/ACM International Symposium on*, pages 55–62. IEEE, 2012. (Cited on page 83)

[53] Feng T., Liu Z., Kwon K.-A., Shi W., Carbunar B., Jiang Y., and Nguyen N. Continuous Mobile Authentication using Touchscreen Gestures. In *2012 IEEE Conference on Technologies for Homeland Security (HST)*, pages 451–456. IEEE, 2012. (Cited on pages 83, 93, 98)

[54] Feng T., Prakash V., and Shi W. Touch Panel with Integrated Fingerprint Sensors Based User Identity Management. In *Technologies for Homeland Security (HST), 2013 IEEE International Conference on*, pages 154–160. IEEE, 2013. (Cited on page 83)

[55] Ferguson N. AES-CBC+ Elephant Diffuser: A Disk Encryption Algorithm for Windows Vista, 2006. (Cited on page 2)

[56] Frank M., Biedert R., Ma E., Martinovic I., and Song D. Touchalytics: On the Applicability of Touchscreen Input as a Behavioral Biometric for Continuous Authentication. *Information Forensics and Security, IEEE Transactions*, 8(1):136–148, 2013. (Cited on pages 83, 84, 85, 111, 116)

[57] Freier, A. and Karlton, P. and Kocher, P. The Secure Sockets Layer (SSL) Protocol, Version 3.0. RFC 6101, 2011. (Cited on pages 88, 113)

[58] Friedman J., Hastie T., and Tibshirani R. *The Elements of Statistical Learning*, volume 1. Springer series in statistics, Springer, Berlin, 2001. (Cited on page 20)

[59] Friedman N., Geiger D., and Goldszmidt M. Bayesian Network Classifiers. *Machine learning*, 29(2-3):131–163, 1997. (Cited on page 21)

[60] G Data. G Data Mobile Malware Report. Whitepaper, Threat Report Q4/2015, G Data, 2016. (Cited on page 1)

[61] Garfinkel T. and Rosenblum M. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium*, pages 191–206, 2003. (Cited on pages 2, 3, 5, 15, 50, 51, 77)

[62] Garlick, J. Plan 9 – 9P2000.L Protocol. Project Page of Distributed I/O Daemon 9P File Server. https://github.com/chaos/diod. *Last access: March 03, 2017.* (Cited on pages 19, 57, 58)

[63] Google Inc. Android Debug Bridge. Android Open Source Project Documentation, Google, 2016. http://developer.android.com/tools/help/adb.html. *Last access: March 03, 2017.* (Cited on pages 103, 106)

[64] Google Inc. Android Application Sandbox. Android Open Source Project Documentation, Google, 2016. https://source.android.com/security/. *Last access: March 03, 2017.* (Cited on page 114)

[65] Google Inc. Google Play Store. Project Page, Google, 2016. https://play.google.com/store. *Last access: March 03, 2017.* (Cited on page 114)

[66] Gostev A., Unuchek R., Garnaeva M., Makrushin D., and Ivanov A. IT Threat Evolution in Q1 2016. Technical Report, Kaspersky Lab, 2016. (Cited on page 1)

[67] Govindarajan S., Gasti P., and Balagani K. S. Secure Privacy-Preserving Protocols for Outsourcing Continuous Authentication of Smartphone Users with Touch Data. In *Biometrics: Theory, Applications and Systems (BTAS), 2013 IEEE Sixth International Conference on*, pages 1–8. IEEE, 2013. (Cited on page 98)

[68] Gultnieks C. et al. F-Droid Installable Catalogue of Free and Open Source Software Applications for the Android Platform. Project Page. https://f-droid.org. *Last access: March 03, 2017.* (Cited on page 114)

[69] Haldar V., Chandra D., and Franz M. Semantic Remote Attestation: A Virtual Machine Directed Approach to Trusted Computing. In *USENIX Virtual Machine Research and Technology Symposium*, volume 2004, 2004. (Cited on page 14)

[70] Ho T. K. Random Decision Forests. In *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, volume 1, pages 278–282. IEEE, 1995. (Cited on page 21)

[71] Horsch J. and Wessel S. Transparent Page-Based Kernel and User Space Execution Tracing from a Custom Minimal ARM Hypervisor. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 408–417. IEEE, 2015. (Cited on page 77)

[72] Hubbard D. and Sutton M. Top Threats to Cloud Computing. Technical Report, Cloud Security Alliance, 2010. (Cited on page 1)

[73] Huber M., Horsch J., Velten M., Weiss M., and Wessel S. A Secure Architecture for Operating System-Level Virtualization on Mobile Devices. In *International Conference on Information Security and Cryptology*, pages 430–450. Springer, 2015. (Cited on page 17)

[74] Hurd, D., Synaptics Inc. Synaptics ClearForce. Press Release, 2015. `http://synaptics.com/company/news/clearforce-for-smartphones`. *Last access: March 03, 2017.* (Cited on page 104)

[75] Intel Security. The Top Five Network Attack Methods – An overview of prevalent attack techniques and effective countermeasures. Technical Report, McAfee Labs and Intel Security, 2015. (Cited on page 1)

[76] Jaeger T. Easystroke Gesture Recognition Application. Project Page. `https://sourceforge.net/projects/easystroke/`. *Last access: March 03, 2017.* (Cited on page 92)

[77] Jajodia S., Samarati P., Sapino M. L., and Subrahmanian V. Flexible Support for Multiple Access Control Policies. *ACM Transactions on Database Systems (TODS)*, 26(2):214–260, 2001. (Cited on page 61)

[78] Jaquith A. *Security Metrics: Replacing Fear, Uncertainty, and Doubt.* Pearson Education, 2007. (Cited on page 2)

[79] Jiang X., Wang X., and Xu D. Stealthy Malware Detection through VMM-based Out-of-the-Box Semantic View Reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM, 2007. (Cited on pages 3, 51, 78)

[80] Jones S. T., Arpaci-Dusseau A. C., and Arpaci-Dusseau R. H. VMM-Based Hidden Process Detection and Identification using Lycosid. In

Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, pages 91–100. ACM, 2008. (Cited on pages 3, 78)

[81] Joshi A., King S. T., Dunlap G. W., and Chen P. M. Detecting Past and Present Intrusions Through Vulnerability-Specific Predicates. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 91–104. ACM, 2005. (Cited on pages 3, 78)

[82] Kim G. H. and Spafford E. H. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29. ACM, 1994. (Cited on pages 50, 77)

[83] King N., Kerr D., Herbst P., and Hotelling S. Electronic device having display and surrounding touch sensitive bezel for user interface and control. Google Patents, US Patent 9,047,009, 2015. (Cited on page 104)

[84] Kivity A., Kamay Y., Laor D., Lublin U., and Liguori A. KVM: The Linux Virtual Machine Monitor. In *OLS '07: Proceedings of the Linux Symposium*, volume 1, pages 225–230, June 2007. (Cited on pages 42, 69)

[85] Kolly S. M., Wattenhofer R., and Welten S. A Personal Touch: Recognizing Users based on Touch Screen Behavior. In *Proceedings of the Third International Workshop on Sensing Applications on Mobile Phones*, page 1. ACM, 2012. (Cited on pages 83, 84, 85, 86, 115)

[86] Kolosnjaji B. and Eckert C. Neural Network-Based User-Independent Physical Activity Recognition for Mobile Devices. In *International Conference on Intelligent Data Engineering and Automated Learning*, pages 378–386. Springer, 2015. (Cited on page 115)

[87] Kumar R., Phoha V. V., and Raina R. Authenticating Users through their Arm Movement Patterns. *arXiv preprint arXiv:1603.02211*, 2016. (Cited on page 115)

[88] Lauer T. The Risk of E-Voting. *Electronic Journal of E-government*, 2(3):177–186, 2004. (Cited on page 1)

[89] Linux Foundation. Xen Virtual Trusted Platform Module (vTPM) Support. Project Page. http://wiki.xenproject.org/wiki/

Virtual_Trusted_Platform_Module_(vTPM). *Last access: March 03, 2017.* (Cited on pages 3, 25, 45)

[90] Linux Kernel Organization. Linux Power Supply Class. Linux Kernel Documentation, . https://www.kernel.org/doc/Documentation/power/power_supply_class.txt. *Last access: March 03, 2017.* (Cited on page 105)

[91] Linux Kernel Organization. v9fs: Plan 9 Resource Sharing for Linux. Linux Kernel Documentation, . https://www.kernel.org/doc/Documentation/filesystems/9p.txt. *Last access: March 03, 2017.* (Cited on pages 19, 69, 73)

[92] Litty L. and Lie D. Manitou: A Layer-Below Approach to Fighting Malware. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 6–11. ACM, 2006. (Cited on page 79)

[93] Litty L., Lagar-Cavilla H. A., and Lie D. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proceedings of the 17th conference on Security symposium*, SS'08, pages 243–258, Berkeley, CA, USA, 2008. USENIX Association. (Cited on pages 77, 79)

[94] Liu D., Lee J., Jang J., Nepal S., and Zic J. A Cloud Architecture of Virtual Trusted Platform Modules. In *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*, pages 804–811. IEEE, 2010. (Cited on page 45)

[95] Long J. *No Tech Hacking: A Guide to Social Engineering, Dumpster Diving, and Shoulder Surfing.* Syngress, 2011. (Cited on page 112)

[96] MacNaught, S. Average User Picks up their Smartphone 221 Times a Day. Tecmark Survey, 2014. http://www.tecmark.co.uk/smartphone-usage-data-uk-2014/. *Last access: March 03, 2017.* (Cited on page 107)

[97] Madnick S. E. and Donovan J. J. Application and Analysis of the Virtual Machine Approach to Information System Security and Isolation. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 210–224. ACM, 1973. (Cited on page 2)

[98] MalwareTech. Phase Bot - A Fileless Rootkit (Part 1). Web Article, MalwareTech, 2014. https://www.malwaretech.com/2014/12/

phase-bot-fileless-rootki.html. *Last access: March 03, 2017.* (Cited on page 76)

[99] MalwareTech. Phase Bot - A Fileless Rootkit (Part 2). Web Article, MalwareTech, 2014. http://www.malwaretech.com/2014/12/phase-bot-fileless-rootkit-part-2.html. *Last access: March 03, 2017.* (Cited on page 76)

[100] Mantyjarvi J., Lindholm M., Vildjiounaite E., Makela S.-M., and Ailisto H. Identifying Users of Portable Devices from Gait Pattern with Accelerometers. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings.(ICASSP'05). IEEE International Conference on*, volume 2, pages ii–973. IEEE, 2005. (Cited on page 115)

[101] Maydell P. QEMU Virtual Trusted Platform Module (vTPM) Support. Project Page. http://wiki.qemu.org/Features/TPM. *Last access: March 03, 2017.* (Cited on pages 3, 25, 45)

[102] Mayer J. R. Any Person... a Pamphleteer: Internet Anonymity in the Age of Web 2.0. *Undergraduate Senior Thesis, Princeton University*, 2009. (Cited on page 85)

[103] McGuire M. and Dowling S. Cyber Crime: A Review of the Evidence. *Summary of key findings and implications. Home Office Research report*, 75, 2013. (Cited on page 1)

[104] Mehrnezhad M., Toreini E., Shahandashti S. F., and Hao F. TouchSignatures: Identification of User Touch Actions and PINs based on Mobile Sensor Data via JavaScript. *Journal of Information Security and Applications*, 2016. (Cited on pages 85, 113)

[105] Mell P. and Grance T. The NIST Definition of Cloud Computing. 2011. (Cited on pages 24, 45)

[106] Menezes A. J., Van Oorschot P. C., and Vanstone S. A. *Handbook of Applied Cryptography*. CRC press, 1996. (Cited on page 31)

[107] Meng Y., Wong D. S., Schlegel R., et al. Touch Gestures based Biometric Authentication Scheme for Touchscreen Mobile Phones. In *Information Security and Cryptology*, pages 331–350. Springer, 2012. (Cited on pages 84, 85)

[108] Merkel D. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), Mar. 2014. ISSN 1075-3583. (Cited on page 18)

[109] Miluzzo E., Varshavsky A., Balakrishnan S., and Choudhury R. R. Tapprints: Your Finger Taps Have Fingerprints. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 323–336. ACM, 2012. (Cited on pages 84, 85, 86, 115)

[110] Mironov I. et al. Hash Functions: Theory, Attacks, and Applications. *Microsoft Research, Silicon Valley Campus*, 2005. (Cited on page 31)

[111] Mohri M., Rostamizadeh A., and Talwalker A. Foundations of Machine Learning (Adaptive Computation and Machine Learning Series), 2012. (Cited on pages 20, 21)

[112] Monkey Runner Testing Tool. Android Open Source Project Documentation. http://developer.android.com/tools/help/monkeyrunner_concepts.html. *Last access: March 03, 2017.* (Cited on page 102)

[113] Monrose F., Reiter M. K., and Wetzel S. Password Hardening Based on Keystroke Dynamics. *International Journal of Information Security*, 1(2):69–83, 2002. (Cited on pages 83, 115)

[114] Monte M. *Network Attacks and Exploitation: A Framework*. John Wiley & Sons, 2015. (Cited on page 1)

[115] Mowery K., Bogenreif D., Yilek S., and Shacham H. Fingerprinting Information in JavaScript Implementations. *Proceedings of W2SP*, 2, 2011. (Cited on page 85)

[116] Murdock, I. Debian Package Management System (DPKG). Debian Project. https://wiki.debian.org/Teams/Dpkg. *Last access: March 03, 2017.* (Cited on page 70)

[117] Nance K., Bishop M., and Hay B. Virtual Machine Introspection: Observation or Interference? *Security & Privacy, IEEE*, 6(5):32–37, 2008. (Cited on pages 3, 50)

[118] Neverova N., Wolf C., Lacey G., Fridman L., Chandra D., Barbello B., and Taylor G. Learning Human Identity from Motion Patterns. *arXiv preprint arXiv:1511.03908*, 2015. (Cited on pages 115, 116)

[119] Nobel, A., SyNetDev. Stay Alive. Google Play Store App. https://play.google.com/store/apps/details?id=com.synetics.stay.alive. *Last access: March 03, 2017.* (Cited on page 102)

[120] NTT DATA Corporation. TOMOYO Linux – A Security Module for System Analysis and Protection. Official TOMOYO Project Website. http://tomoyo.osdn.jp. *Last access: March 03, 2017.* (Cited on page 50)

[121] Owusu E., Han J., Das S., Perrig A., and Zhang J. Accessory: Password Inference using Accelerometers on Smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, page 9. ACM, 2012. (Cited on pages 83, 115)

[122] Patil S., Kashyap A., Sivathanu G., and Zadok E. I3FS: An In-Kernel Integrity Checker and Intrusion Detection File System. In *Proceedings of the 18th Annual Large Installation System Administration Conference (LISA'04)*, 2004. (Cited on page 77)

[123] Payne B., de Carbone M., and Lee W. Secure and Flexible Monitoring of Virtual Machines. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 385–397, 2007. (Cited on page 78)

[124] Payne B. D., Carbone M., Sharif M., and Lee W. Lares: An Architecture for Secure Active Monitoring using Virtualization. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 233–247. IEEE, 2008. (Cited on pages 51, 53, 78)

[125] Pedregosa F., Varoquaux G., Gramfort A., Michel V., Thirion B., Grisel O., Blondel M., Prettenhofer P., Weiss R., Dubourg V., Vanderplas J., Passos A., Cournapeau D., Brucher M., Perrot M., and Duchesnay E. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. (Cited on page 96)

[126] Peotta L., Holtz M. D., David B. M., Deus F. G., and de Sousa R. A Formal Classification of Internet Banking Attacks and Vulnerabilities. *International Journal of Computer Science & Information Technology*, 3(1):186–197, 2011. (Cited on page 1)

[127] Petroni Jr N. L. and Hicks M. Automated Detection of Persistent Kernel Control-Flow Attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 103–115. ACM, 2007. (Cited on page 51)

[128] Petroni Jr N. L., Fraser T., Molina J., and Arbaugh W. A. Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor. In *USENIX*

Security Symposium, pages 179–194. San Diego, USA, 2004. (Cited on page 51)

[129] Popek G. J. and Goldberg R. P. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7): 412–421, 1974. (Cited on page 15)

[130] Primo A., Phoha V., Kumar R., and Serwadda A. Context-Aware Active Authentication using Smartphone Accelerometer Measurements. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 98–105, 2014. (Cited on pages 84, 85)

[131] Proskurin S., Weiss M., and Sigl G. seTPM: Towards Flexible Trusted Computing on Mobile Devices Based on GlobalPlatform Secure Elements. In *Revised Selected Papers of the 14th International Conference on Smart Card Research and Advanced Applications - Volume 9514*, CARDIS 2015, pages 57–74, New York, NY, USA, 2016. (Cited on page 46)

[132] Proudler G., Chen L., and Dalton C. *Trusted Computing Platforms - TPM2.0 in Context*. Springer, 2014. ISBN 978-3-319-08743-6. doi: 10.1007/978-3-319-08744-3. (Cited on page 11)

[133] Purcher, J. Apple 3D Touch. Web Article, Patently Apple, 2015. http://www.patentlyapple.com/patently-apple/2015/09/apples-reported-3d-force-touch-is-supported-by-a-patent.html. *Last access: March 03, 2017*. (Cited on page 104)

[134] Quynh N. A. and Suzaki K. Xenprobes, a Lightweight User-Space Probing Framework for Xen Virtual Machine. In *USENIX Annual Technical Conference Proceedings*, 2007. (Cited on pages 51, 53, 78)

[135] Raj H., Saroiu S., Wolman A., Aigner R., Cox J., England P., Fenner C., Kinshumann K., Loeser J., Mattoon D., Nystrom M., Robinson D., Spiger R., Thom S., and Wooten D. fTPM: A Software-Only Implementation of a TPM Chip. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 841–856, Austin, TX, 2016. USENIX Association. (Cited on page 46)

[136] Ramzan Z. Phishing Attacks and Countermeasures. In *Handbook of Information and Communication Security*, pages 433–448. Springer, 2010. (Cited on page 113)

[137] Rascagnères, P. Poweliks: The Persistent Malware without a File. Blog Article, G-Data, 2014. https://blog.gdatasoftware.com/2014/07/23947-poweliks-the-persistent-malware-without-a-file. *Last access: March 03, 2017.* (Cited on page 76)

[138] Rattani A. and Poh N. Biometric System Design under Zero and Non-Zero Effort Attacks. In *Biometrics (ICB), 2013 International Conference on*, pages 1–8. IEEE, 2013. (Cited on page 98)

[139] Rescorla, E. Hypertext Transfer Protocol (HTTP) Over TLS. RFC 2818, 2000. (Cited on page 113)

[140] Richardson R. CSI Computer Crime and Security Survey. *Computer Security Institute*, 1:1–30, 2008. (Cited on page 1)

[141] Roberts J., Yaya L., and Manolis C. The Invisible Addiction: Cell-phone Activities and Addiction Among Male and Female College Students. *Journal of behavioral addictions*, 3(4):254–265, 2014. (Cited on page 107)

[142] Rosenblum M. and Garfinkel T. Virtual machine monitors: Current technology and future trends. *Computer*, 38(5):39–47, 2005. (Cited on page 15)

[143] Roth J., Liu X., and Metaxas D. On Continuous User Authentication via Typing Behavior. *Image Processing, IEEE Transactions on*, 23 (10):4611–4624, 2014. (Cited on page 115)

[144] RSA Security Inc. Minimizing the Identity Attack Vector with Continuous Authentication. RSA white paper, EMC Corporation, 2016. (Cited on page 4)

[145] Russell R. virtio: Towards a de-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008. (Cited on pages 54, 69)

[146] Sadeghi A.-R., Stüble C., and Winandy M. Property-based TPM Virtualization. In *Information Security*, pages 1–16. Springer, 2008. (Cited on page 14)

[147] Saevanee H. and Bhatarakosol P. User Authentication Using Combination of Behavioral Biometrics over the Touchpad Acting Like Touch Screen of Mobile Device. In *Computer and Electrical Engineering, 2008. ICCEE 2008. International Conference on*, pages 82–86. IEEE, 2008. (Cited on pages 84, 85, 87, 104)

[148] Sailer R., Zhang X., Jaeger T., and van Doorn L. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, Berkeley, CA, USA, 2004. USENIX Association. (Cited on pages 2, 25, 28, 42, 59)

[149] Saltzer J. H. and Schroeder M. D. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. (Cited on page 114)

[150] Sandberg R., Goldberg D., Kleiman S., Walsh D., and Lyon B. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX conference*, pages 119–130, 1985. (Cited on page 53)

[151] Santos N., Rodrigues R., Gummadi K. P., and Saroiu S. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 175–188, 2012. (Cited on page 45)

[152] Schaufler, C. The Smack Project. Official Smack Project Website. http://schaufler-ca.com. *Last access: March 03, 2017.* (Cited on page 50)

[153] Schepers D., Brubeck M., Barstow A., and Moon S. Touch Events. W3C Recommendation, W3C, 2013. (Cited on pages 95, 109)

[154] Schepers D., Moon S., Brubeck M., and Byers R. Touch Events Extensions. W3C Working Group Note, W3C, 2013. (Cited on page 95)

[155] Schneier B. *Secrets and Lies: Digital Security in a Networked World.* John Wiley & Sons, 2011. (Cited on page 2)

[156] Schneier, B. Whitelisting vs. Blacklisting. *Schneier on Security*, Official Blog, 2011. https://www.schneier.com/blog/archives/2011/01/whitelisting_vs.html. *Last access: March 03, 2017.* (Cited on page 61)

[157] Serwadda A., Phoha V. V., Wang Z., Kumar R., and Shukla D. Toward Robotic Robbery on the Touch Screen. *ACM Transactions on Information and System Security (TISSEC)*, 18(4):14, 2016. (Cited on pages 98, 123)

[158] Shatilin I. Google Project Abacus. Official Kaspersky Lab Blog, Kaspersky Lab, 2015. https://blog.kaspersky.com/google-projects-soli-jacquard-vault-abacus/9135/. *Last access: March 03, 2017.* (Cited on page 116)

[159] Shi W., Yang F., Jiang Y., Yang F., and Xiong Y. Senguard: Passive User Identification on Smartphones using Multiple Sensors. In *Wireless and Mobile Computing, Networking and Communications (WiMob), 2011 IEEE 7th International Conference on*, pages 141–148. IEEE, 2011. (Cited on pages 84, 85)

[160] Shotton J., Sharp T., Kipman A., Fitzgibbon A., Finocchio M., Blake A., Cook M., and Moore R. Real-Time Human Pose Recognition in Parts from Single Depth Images. *Communications of the ACM*, 56(1): 116–124, 2013. (Cited on page 93)

[161] Shrestha P., Mohamed M., and Saxena N. Slogger: Smashing Motion-based Touchstroke Logging with Transparent System Noise. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 67–77. ACM, 2016. (Cited on page 115)

[162] Silberschatz A., Galvin P. B., Gagne G., and Silberschatz A. *Operating System Concepts*, volume 4. Addison-wesley Reading, 1998. (Cited on page 18)

[163] Smalley S., Vance C., and Salamon W. Implementing SELinux as a Linux Security Module. *NAI Labs Report*, 1:43, 2001. (Cited on page 50)

[164] Smith J. and Nair R. *Virtual Machines: Versatile Platforms for Systems and Processes*. Elsevier, 2005. (Cited on page 15)

[165] Stevens M., Bursztein E., Karpman P., Albertini A., Markov Y., Bianco A. P., and Baisse C. Announcing the first SHA1 collision. Google Security Blog, CWI Amsterdam and Google, 2017. https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html. *Last access: March 03, 2017.* (Cited on pages 12, 32)

[166] Stumpf F. and Eckert C. Enhancing Trusted Platform Modules with Hardware-Based Virtualization Techniques. *Emerging Security Information, Systems, and Technologies, The International Conference on*, 0:1–9, 2008. (Cited on pages 25, 44, 46)

[167] Symantec. Security Response. Kovter Malware Learns from poweliks with Persistent Fileless Registry Update. Symantec Official Blog Article, Symantec, 2015. http://www.symantec.com/connect/blogs/kovter-malware-learns-poweliks-persistent-fileless-registry-update. *Last access: March 03, 2017.* (Cited on page 76)

[168] Tangelder J. Hammer.js Open Source Gesture Recognition Library. Project Page. https://hammerjs.github.io/. *Last access: March 03, 2017.* (Cited on page 86)

[169] The WebKit Open Source Web Browser Engine. Project Page. http://www.webkit.org/. *Last access: March 03, 2017.* (Cited on page 109)

[170] Tool Interface Standards Committee. Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification. Version 1.2, 1995. (Cited on pages 65, 66)

[171] TrouSerS. The Open Source TCG Software Stack. TSS Project Page. http://trousers.sourceforge.net. *Last access: March 03, 2017.* (Cited on page 42)

[172] Trusted Computing Group (TCG). TCG Project Page. https://www.trustedcomputinggroup.org/. *Last access: March 03, 2017.* (Cited on pages 2, 11, 24)

[173] Trusted Platform Module (TPM) 1.2. Main Specification, Level 2, Version 1.2, Revision 116, 2011. (Cited on pages 2, 11, 12, 13, 24, 26, 33, 57)

[174] Trusted Platform Module (TPM) 2.0. Library Specification, Family 2.0, Level 00, Revision 01.16, 2014. (Cited on pages 2, 11, 13, 24, 34, 36)

[175] Trusted Platform Module (TPM) 2.0. TCG TPM 2.0 Automotive Thin Profile, Family 2.0, Level 00, Version 1.0, 2015. (Cited on pages 13, 26)

[176] Trusted Platform Module (TPM) 2.0. PC Client Platform TPM Profile (PTP) Specification, Family 2.0, Level 00, Revision 00.43, 2015. (Cited on pages 13, 26)

[177] United States Federal Government. Secure Hash Standard (SHA-1). Federal Information Processing Standards (FIPS) Publication 180-4, National Institute of Standards and Technology, 2015. (Cited on page 32)

[178] Van Hensbergen E. and Minnich R. Grave Robbers from Outer Space using 9P2000 under Linux. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, page 45, Berkeley, CA, USA, 2005. USENIX Association. (Cited on pages 19, 51, 53, 59)

[179] Velten M. and Stumpf F. Secure and Privacy-Aware Multiplexing of Hardware-Protected TPM Integrity Measurements among Virtual Machines. In *15th International Conference on Information Security and Cryptology (ICISC 2012)*, Lecture Notes in Computer Science. Springer, 2012. (Cited on page 24)

[180] Velten M., Wessel S., Stumpf F., and Eckert C. Active File Integrity Monitoring Using Paravirtualized Filesystems. In *5th International Conference on Trusted Systems (INTRUST 2013)*, Lecture Notes in Computer Science. Springer, 2013. (Cited on page 50)

[181] Velten M., Schneider P., Wessel S., and Eckert C. User Identity Verification Based on Touchscreen Interaction Analysis in Web Contexts. In *11th International Conference on Information Security Practice and Experience (ISPEC 2015)*, Lecture Notes in Computer Science. Springer, 2015. (Cited on page 82)

[182] Vogl S., Kilic F., Schneider C., and Eckert C. X-TIER: Kernel Module Injection. In *International Conference on Network and System Security*, pages 192–205. Springer, 2013. (Cited on page 51)

[183] Wallom D., Turilli M., Taylor G., Hargreaves N., Martin A., Raun A., and McMoran A. myTrustedCloud: Trusted Cloud Infrastructure for Security-Critical Computation and Data Managment. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 247–254. IEEE, 2011. (Cited on page 45)

[184] Weaver R., Weaver D., and Farwood D. *Guide to network defense and countermeasures*. Cengage Learning, 2013. (Cited on page 2)

[185] Wessel S. and Stumpf F. Page-based Runtime Integrity Protection of User and Kernel Code. In *5th European Workshop on System Security*, 2012. (Cited on pages 77, 79)

[186] White, B. Package Management Runtime Library libapt-pkg. Debian Project. (Cited on page 70)

[187] Wood P., Nahorney B., Chandrasekar K., Wallace S., and Haley K. Internet Security Threat Report. Technical Report, Volume 21, Symantec Corporation, 2016. (Cited on page 1)

[188] Xu H., Zhou Y., and Lyu M. R. Towards Continuous and Passive Authentication via Touch Biometrics: An Experimental Study on Smartphones. In *Symposium On Usable Privacy and Security (SOUPS 2014)*. USENIX Association, 2014. (Cited on page 99)

[189] Zhao F., Jiang Y., Xiang G., Jin H., and Jiang W. VRFPS: A Novel Virtual Machine-Based Real-time File Protection System. In *Proceedings of the 2009 Seventh ACIS International Conference on Software Engineering Research, Management and Applications*, SERA '09, pages 217–224, Washington, DC, USA, 2009. ISBN 978-0-7695-3903-4. doi: 10.1109/SERA.2009.23. (Cited on pages 51, 78)

[190] Zheng N., Paloski A., and Wang H. An Efficient User Verification System via Mouse Movements. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 139–150. ACM, 2011. (Cited on pages 83, 115)

[191] Zheng N., Bai K., Huang H., and Wang H. You Are How You Touch: User Verification on Smartphones via Tapping Behaviors. Technical report, Tech. Rep. WM-CS-2012-06, 2012. (Cited on pages 83, 84, 85, 86, 111, 115)

[192] Zhu J., Wu P., Wang X., and Zhang J. Sensec: Mobile Security through Passive Sensing. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 1128–1133. IEEE, 2013. (Cited on pages 83, 84, 85, 111, 115)

List of Acronyms

AA . . . . Attestation Agent
ACE . . . . Access Control Entry
ACL . . . . Access Control List
ADB . . . . Android Debug Bridge
AES . . . . Advanced Encryption Standard
AIK . . . . Attestation Identity Key
AK . . . . Attestation Key
CAM . . . . Continuous Authentication Monitor
CIFS . . . . Common Internet File System
CMML . . . . Concealed Multiplexed Measurement List
CPVM . . . . Complementary Privileged Virtual Machine
CRTM . . . . Core Root of Trust for Measurement
DAA . . . . Direct Anonymous Attestation
DOM . . . . Document Object Model
EA . . . . Enhanced Authorization
ECC . . . . Elliptic Curve Cryptography
EDE . . . . Execution Detection Engine
EK . . . . Endorsement Key
ELF . . . . Executable and Linking Format
EPS . . . . Endorsement Primary Seed
FAR . . . . False Acceptance Rate
FOM . . . . File Operation Monitor
FPE . . . . File Protection Enforcer
FPGA . . . . Field-Programmable Gate Array
FRR . . . . False Rejection Rate

HDD . . . . Hard Disk Drive
HIDS . . . . Host-based Intrusion Detection System
HMAC . . . . Hashed Message Authentication Code
HTTP . . . . Hypertext Transfer Protocol
HTTPS . . . . Hypertext Transfer Protocol over TLS
IMA . . . . Integrity Measurement Architecture
KVM . . . . Kernel-based Virtual Machine
LSM . . . . Linux Security Modules
MA . . . . Measurement Agent
MITM . . . . Man-in-the-Middle
MML . . . . Multiplexed Measurement List
MMU . . . . Memory Management Unit
MPA . . . . Multiplexing Agent
NFS . . . . Network File System
NLKVM . . . . Native Linux KVM Tool
OS . . . . Operating System
PCR . . . . Platform Configuration Register
PIN . . . . Personal Identification Number
PME . . . . Package Maintenance Engine
RAM . . . . Random Access Memory
RBAC . . . . Rule Based Access Control
SHA-1 . . . . Secure Hash Algorithm 1
SMB . . . . Server Message Block
SML . . . . Stored Measurement Log
SoC . . . . System on Chip
SSD . . . . Solid State Drive
SSL . . . . Secure Sockets Layer
SVM . . . . Support Vector Machine
TBV . . . . Touch Behavior Verifier
TCB . . . . Trusted Computing Base
TCG . . . . Trusted Computing Group
TLS . . . . Transport Layer Security
TPM . . . . Trusted Platform Module

VLAN . . . . Virtual Local Area Network
VM . . . . Virtual Machine
VM-ID . . . . Virtual Machine Identifier
VMI . . . . Virtual Machine Introspection
VMM . . . . Virtual Machine Monitor
vTPM . . . . Virtual Trusted Platform Module

List of Figures

1.1 Main components explored in this thesis 7

2.1 Authenticated boot 13
2.2 Comparison of type1 and type2 hypervisors 16
2.3 Comparison of paravirtualization and OS-level virtualization 17
2.4 Visual representation of binary classification 21

3.1 System architecture showing main components and workflow 29
3.2 Multiplexing remote attestation protocol 35
3.3 Average processing time of virtual machines 44

4.1 Filesystem relocation scenarios 54
4.2 Paravirtualized monitoring architecture 56
4.3 Installation and upgrading of packages via CPVM 67
4.4 Write performance for different environments 71
4.5 Read performance for different environments 72
4.6 Required time to read and write files over the network 74

5.1 System architecture with web server, TBV, and CAM 87
5.2 State diagram maintained by CAM 90
5.3 Vertical scroll gestures with pronounced curvatures 94
5.4 Vertical scroll gestures with predominantly straight lines . . . 95
5.5 Minimum, mean, and maximum of bounding box heights . . . 97
5.6 Minimum, mean, and maximum of bounding box widths . . . 98
5.7 Feature value comparison of scroll gestures by different users 99
5.8 FAR and FRR based on single scroll gesture 100
5.9 FAR and FRR based on sequences of scroll gestures 101
5.10 CPU load of enabled user identity verification 104
5.11 Battery usage of verification with WiFi and mobile data . . . 106
5.12 Relative battery usage comparison 107
5.13 Number of fired events of event listeners 109

List of Tables

2.1 Comparison of TPM 1.2 and TPM 2.0 12
2.2 Set of 9P2000 operations 19

3.1 Average processing time for measured files 43

4.1 Critical requests of the Plan 9 9P2000.L protocol 58
4.2 Critical requests mapped to policy checks using only predicates. 62
4.3 Policy example utilizing predicates and reference hash values 64

5.1 Tested devices and browsers supporting force attribute 105
5.2 Comparison of generated network traffic per browser 110