André Rein

DRIVE

Dynamic Runtime Integrity Verification and Evaluation

DRIVE

Dynamic Runtime Integrity Verification and Evaluation

Dissertation
zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften
– Dr.-Ing. –

vorgelegt von

André Rein

im Fachbereich 03 (Mathematik/Informatik)
der Universität Bremen

November 2018 – Version 1.0

Datum des Kolloquiums: 20.11.2018

1. Gutachter: Prof. Dr.-Ing. Carsten Bormann (Universität Bremen, Bremen)
2. Gutachter: Associate Prof. Dr. Carsten Rudolph (Monash University, Melbourne)

I dedicate this thesis to my loving wife, my family and my friends . . .

Acknowledgements

In the years in which the results of this work have been completed, I have enjoyed the help of many amazing and extraordinary people. Without you, this work would never have been completed in this way, and I would like to express my sincere thanks to all of you.

First of all, I thank my advisor and supervisor, Professor Dr.-Ing. Carsten Bormann, for welcoming me and giving me the confidence to complete this thesis. It was a pleasure to work under your guidance, and I hope that in the future I will have the opportunity to work with you again. Next, I thank my second supervisor, Associate Professor Dr. Carsten Rudolph. We have known each other for several years now, and working under your guidance has always been a great honor and pleasure for me. Thank you very much for always believing in me; without you I would never have considered and completed a doctoral thesis. My heartfelt thanks also to Professor Dr. Nicolai Kunzte for driving me to start and finish this thesis. You have brought me to the exciting topic of security, and without your constant and untiring encouragement, help and guidance, there would be no acknowledgment because there would have been no thesis.

Furthermore, I thank everyone who was involved in the technical work of this thesis. In particular Hagen Lauer, you were a big help to me in the beginning and influenced this work decisively. In addition, many were involved in the implementation of the prototype. My special thanks to Wang Zhen, Professor Dr. Kai Oliver Detken, Marcel Jahnke, Thomas Rix, Malte Humann and Bernd Ritter for all your hard work and technical help. Without you, there would not have been such a professional prototype. Many thanks to all my current and former colleagues – Michael Eckel, Stefan Seefeldt and Mihai Serb – for all the technical discussions, advice and feedback.

My special thanks also go to all proofreaders. To Heike Stürmer, I am deeply grateful for the countless hours you have put into all the linguistic corrections. I am also especially grateful to Dr. Roland Rieke – your help and advice especially in the structuring and argumentation of this work were very helpful. Many thanks to Professor Dr. Igor Faynberg – you helped me a lot with the linguistic fine-tuning of this thesis. I would also like to express my special thanks to my former professors and especially to Professor Dr. Michael Jäger for all the enthusiasm that you have awakened in me and for working with you.

Many thanks to Huawei Technologies Germany and especially the Cyber Security and Privacy Lab team for this exciting topic and all your support during the last three years.

I thank my family and friends for all your support and for always believing in me. I thank my brother, Tommy, in particular for all the relaxing gaming sessions after many long days of writing.

Undoubtedly my deepest and most loving thanks belongs to my wife Denise. Although you would never admit it, I know you were suffering the most from the lack of time that I was unable to spend with you during the final months of completing this work. My deepest gratitude for all your love, your help and for being always there for me.

Abstract

Cyberattacks have been rapidly and continuously gaining ground over the last few years, and there is an escalating conflict between those who develop new security techniques and those who develop new attacks that circumvent these countermeasures. This means that all security measures can and will ultimately be bypassed. In this context, attackers make use of advanced attack techniques that exploit the complexity of systems by first leveraging existing countermeasures and then using simpler attack techniques to take over the attacked system.

Classic security techniques add additional layers of safeguards by introducing patterns, compiler features like canary values or tainting, or hardware features for protecting systems against runtime attacks.

A different approach as presented in this thesis is the assessment of software that is based on a comparison of the binary code loaded and the memory image found during runtime. This approach rests on information data structures that are present in systems under attack. Parts of the loaded and executed memory image and its characteristics are predictable; therefore, by taking advantage of this predictability, a reliable assessment can be performed on the basis of various available pieces of information such as loaded binary files, loading mechanisms and allocated memory addresses. By using this information, it is possible to assess the memory state during execution by defining whitelists and policies based on the software actually used. This provides a new way to detect sophisticated runtime attacks on software that are not considered and recognized in current approaches. The evaluation of runtime system states is capable of making a significant contribution to system security.

Based on the evaluation of runtime states a novel and holistic runtime protection technology is presented which performs an assessment of runtime states of systems. In particular, this thesis sets forth the background, design, implementation and evaluation of a memory protection concept at runtime. This concept is based on an assessment of memory contents and meta information that are verified using trusted binary sources and policies.

The results of this work demonstrate that the developed runtime protection technology is a suitable solution and an appropriate addition to further increase the overall security of systems used today. A careful analysis and evaluation of the presented concept on the basis of a prototypical implementation prove the effectiveness of this technology.

The work presented builds the foundation for further research in the field, since the developed concept is widely adoptable in many modern systems. Specifically, with regard to virtualized environments, mobile systems or the Internet of Things, further research is necessary because the presented details must be adapted to match these other use cases or utilize different technologies/building blocks that are specific to the particular scenarios.

Zusammenfassung

Cyberangriffe haben in den letzten Jahren rasch und kontinuierlich zugenommen. Hierbei herrscht ein Wettrennen zwischen der Entwicklung neuer Sicherheitstechniken und neuen Angriffen, die diese Gegenmaßnahmen umgehen. Diese Entwicklung bedeutet, dass letztendlich alle Sicherheitsmaßnahmen umgangen werden können und werden. Angreifer nutzen hierzu fortgeschrittene Angriffstechniken, die die Komplexität von Systemen ausnutzen, indem sie zunächst vorhandene Gegenmaßnahmen aushebeln und dann einfachere Angriffstechniken einsetzen, um das angegriffene System zu übernehmen.

Klassische Sicherheitstechniken fügen den Systemen zusätzliche Schutzebenen hinzu, indem sie Muster, Compilererweiterungen – z.B. Canaries oder Tainting – oder Hardwareerweiterungen zum Schutz von Systemen vor Laufzeitangriffen einführen.

Ein alternativer Ansatz, wie er in dieser Arbeit vorgestellt wird, ist die Bewertung von Software, die auf einem Vergleich zwischen dem geladenen Binärcode und dem während der Laufzeit gefundenen Speicherbild basiert. Dieser Ansatz beruht auf Datenstrukturen und Informationen, die in angegriffenen Systemen vorhanden sind. Dabei sind Teile des geladenen und ausgeführten Speicherabbildes und dessen Eigenschaften vorhersagbar. Dies bedeutet, dass unter Ausnutzung dieser Vorhersagbarkeit eine zuverlässige Bewertung auf der Basis verschiedener verfügbarer Informationen wie geladener Binärdateien, Lademechanismen und zugeordnete Speicheradressen durchgeführt werden kann. Durch die Verwendung dieser Informationen ist es möglich, den Speicherzustand während der Ausführung zu bewerten, indem sog. *Whitelists* und Richtlinien definiert werden, die auf der tatsächlich verwendeten Software selbst basieren. Dies bietet eine neuartige Möglichkeit komplexe Laufzeitangriffe auf Software zu erkennen, die in aktuellen Ansätzen nicht berücksichtigt wird und daher nicht erkennbar ist. Demnach leistet eine Auswertung von Laufzeitsystemzuständen einen wesentlichen Beitrag zur Systemsicherheit.

Basierend auf der Auswertung von Laufzeitzuständen wird eine neuartige und ganzheitliche Laufzeitschutztechnologie vorgestellt, die eine Bewertung der Laufzeitzustände von Systemen durchführt. Insbesondere wird in dieser Arbeit der Hintergrund, das Design, die Implementierung und die Evaluierung eines Speicherschutzkonzeptes zur Laufzeit vorgestellt. Dieses Konzept basiert auf der Bewertung von Speicherinhalten und Metainformationen, die unter zu Hilfenahme von vertrauenswürdigen binären Quellen und Richtlinien verifiziert werden.

Die Ergebnisse dieser Arbeit zeigen, dass die entwickelte Laufzeitschutztechnologie eine geeignete Lösung und sinnvolle Ergänzung ist, um die Gesamtsicherheit von heute eingesetzten Systemen weiter zu erhöhen. Eine sorgfältige Analyse und Bewertung des vorgestellten Konzepts, anhand einer prototypischen Umsetzung, belegt hierzu die Wirksamkeit dieser Technologie.

Da das entwickelte Konzept in vielen modernen Systemen weithin anwendbar ist, bildet die hier vorgestellte Arbeit eine Basis für die weitere Forschung auf diesem Gebiet. Insbesondere in

Themengebieten wie virtualisierten Umgebungen, mobilen Systemen oder dem Internet der Dinge sind weitere Recherchen notwendig. Hierzu müssen die vorgestellten Details speziell auf den jeweiligen Anwendungsfälle angepasst werden, da unterschiedliche Technologien oder Bausteine zum Einsatz kommen und somit spezifische Implementierung benötigt werden.

List of Publications

[1] Andre Rein. "DRIVE: Dynamic Runtime Integrity Verification and Evaluation". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS '17*. ACM Press, 2017, pp. 728–742. ISBN: 978-1-4503-4944-4. URL: http://dl.acm.org/citation.cfm?doid=3052973.3052975

[2] Kai-Oliver Detken, Marcel Jahnke, Thomas Rix, and Andre Rein. "Software-design for internal security checks with dynamic Integrity Measurement (DIM)". in: *2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. vol. 1. IEEE, Sept. 2017, pp. 367–373. DOI: 10.1109/idaacs.2017.8095106

[3] Andre Rein, Roland Rieke, Michael Jaeger, Luigi Coppolino, and Nicolai Kuntze. "Trust Establishment in Cooperating Cyber-Physical Systems". In: *Cybersecurity of Industrial Control Systems, Security of Cyber Physical Systems*. Springer International Publishing, 2016, pp. 31–47. DOI: 10.1007/978-3-319-40385-4_3

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Problem Description and Research Topic . . . 2
- 1.3 Use Case . . . 3
- 1.4 Research Questions, Goals and Objectives . . . 4
  - 1.4.1 Research Questions . . . 4
  - 1.4.2 Research Goal . . . 5
  - 1.4.3 Research Objectives . . . 5
- 1.5 Research Plan and Methods . . . 7
- 1.6 Contributions . . . 8
- 1.7 Outline . . . 10

2 Technical Background **13**
- 2.1 Program Execution Principles in Computer Architectures . . . 13
- 2.2 Program Loading in Modern Operating Systems . . . 14
  - 2.2.1 Organization of Program Text and Data . . . 14
  - 2.2.2 Loading Processes of Programs . . . 14
- 2.3 Program Text Variants . . . 17
  - 2.3.1 Relocatable Code . . . 17
  - 2.3.2 Position Independent Code . . . 18
  - 2.3.3 Program Text Distribution Analysis . . . 18
  - 2.3.4 Global Offset Table . . . 19
- 2.4 Memory Management, Access and Protection . . . 20
  - 2.4.1 Virtual Memory Management . . . 20
  - 2.4.2 File to Memory Mapping . . . 21
  - 2.4.3 Access Rights of Memory Mapped Segments . . . 21
  - 2.4.4 Static and Dynamic Behavior of Programs . . . 22
- 2.5 Summary . . . 23

3 Security Analysis **25**
- 3.1 Malware . . . 25
  - 3.1.1 Malware Types . . . 26

3.1.2 Malware Key Properties 27
3.1.3 Summary 33
3.2 System Memory Runtime Attacks 34
3.2.1 Attack Overview 34
3.2.2 Control Flow Attack Foundation 36
3.2.3 Types of Control Flow Manipulation and Bending Attacks 38
3.2.4 Countermeasures 50
3.2.5 Summary 55
3.3 Threat and Attack Model 56
3.3.1 Attack Goals 56
3.3.2 Attack Methods and Procedures 58
3.3.3 Multi-Step Hybrid Attacks 58
3.3.4 Attack Scenarios 65
3.3.5 Limitations 70
3.3.6 Summary 70
3.4 Trust Model and Security Assumptions 71
3.5 Security Analysis Summary and Conclusion 72

4 DRIVE High Level Attestation Concept and Architecture 75
4.1 High Level Attestation Concept 76
4.2 DRIVE High Level Architecture 77
4.2.1 Architecture Overview 78
4.2.2 Instantiated Software Architecture 80
4.2.3 Summary 83
4.3 Architecture Deployment Analysis 84
4.3.1 Isolation during Verification 84
4.3.2 Isolation during Measurement and Reporting 88
4.3.3 Design Space and Architectural Limitations 92
4.3.4 Summary 94
4.4 Concept and Architecture Summary and Conclusion 95

5 DRIVE Measurement, Verification and Reporting Concept 97
5.1 Attestation of Static Information 97
5.1.1 Measurement of Static Memory Areas 98
5.1.2 Reporting of Measured Data 102
5.1.3 Remote Attestation Protocol 104
5.1.4 Verification of Reported Static Measurement Data 109
5.1.5 Summary 111
5.2 Attestation of Predictable Dynamic Information 112
5.2.1 Measurement of Relocatable Code 112
5.2.2 Verification of Relocatable Code 113
5.2.3 Measurement of Global Offset Tables 115
5.2.4 Verification of Global Offset Tables 116
5.2.5 Summary 118
5.3 Attestation of Unpredictable Dynamic Information 119

5.3.1 Measurement of Unpredictable Dynamic Information 119
5.3.2 Verification of Unpredictable Dynamic Segments 121
5.3.3 Summary 124
5.4 Concept Security Analysis and Evaluation 124
5.4.1 Static Information Attestation Analysis 125
5.4.2 Predictable Dynamic Information Attestation Analysis 127
5.4.3 Unpredictable Dynamic Information Attestation Analysis 129
5.4.4 Summary 131
5.5 Chapter Summary 132

6 Implementation and Evaluation 133
6.1 DRIVE Proof of Concept Implementation 133
6.1.1 DRIVE Software Architecture Implementation 134
6.1.2 DRIVE Component Guideline Implementation 139
6.1.3 Summary 144
6.2 Security Evaluation 144
6.2.1 Attack-based Security Analysis 145
6.2.2 Summary and Conclusion 149
6.3 Performance and Scalability Evaluation 150
6.3.1 Evaluation Summary Original Research 151
6.3.2 Proof of Concept Measurement Evaluation 152
6.3.3 Summary 157
6.4 Implementation and Evaluation Summary 158

7 State of the Art and Related Work 161
7.1 File-based Integrity Protection during Load Time 162
7.1.1 Measured Boot 162
7.1.2 Secure Boot 163
7.2 Integrity Protection during Runtime of Programs 164
7.2.1 Integrity Protection of Static Memory Content 164
7.3 Control Flow and Data Flow Integrity 168
7.3.1 Control Flow Integrity 168
7.3.2 Data Flow Integrity 169

8 Conclusion and Outlook 171
8.1 Conclusion 171
8.1.1 Contributions 171
8.1.2 Research Questions and Objectives 173
8.2 Outlook and Future Work 179

Bibliography 183

Acronyms 195

Notations 198

List of Figures

2 Technical Background

2.1 ELF to VAS Mapping . . . 15
2.2 ELF File Representation . . . 21

3 Security Analysis

3.1 Type-0 Malware Stealthiness Classification . . . 29
3.2 Type-1 Malware Stealthiness Classification . . . 29
3.3 Type-2 Malware Stealthiness Classification . . . 30
3.4 Type-3 Malware Stealthiness Classification . . . 30
3.5 Exploitation Model and Classification of Attacks . . . 35
3.6 CFG with multiple valid Nodes and Branches. . . . 37
3.7 CFG with injected Malicious Code and Modified Branch . . . 39
3.8 CFG with maliciously modified Code . . . 40
3.9 CFG with maliciously modified Code Pointer . . . 43
3.10 CFG with maliciously modified Code pointer to Shared Library . . . 43
3.11 ROP: CFG calling Instruction Sequences (Gadgets) inside Nodes. . . . 45
3.12 CFG including malicious Data Modifications . . . 46
3.13 Hybrid CFM combining a Code Reuse, Code Injection and CPM attack . . . 48
3.14 Long Term Goals and Attack Methods . . . 57
3.15 Vulnerability Exploitation Techniques and Methods . . . 59
3.16 Different Steps in Multi-Step Vulnerability Exploitation. . . . 60
3.17 CPMAP 1: typical Step 1 Attack . . . 61
3.18 IDAP 1: typical Step 1 Attack . . . 61
3.19 CIAP 1: typical Step 2 Code Injection Attack . . . 62
3.20 CIAP 2: typical Step 2 Code Injection Attack . . . 62
3.21 RCAP 1: typical Step 2 Code Replacement Attack . . . 62
3.22 DMAP 1: typical Step 2 Data Manipulation Attack . . . 62
3.23 CPMAP 2: typical Step 3 Code Pointer Manipulation Attack . . . 63
3.24 Code Replacement Attack with optional Memory Permission Modification. . . . 64

3.25 Code Injection Attack with optional Memory Permission Modification. 64
3.26 Complex Code Injection Attack 64
3.27 A1: Create Malicious Executable Segment. 65
3.28 A2: Inject Malicious Code in arbitrary Memory Region. 65
3.29 A3: Modify Code Segment to change Semantics maliciously. 65
3.30 A5: Alter/Remove Memory Protection. 68
3.31 A4: Modify Code Pointer Data to call Malicious/Unintended Code. 68
3.32 A6: Modify Data Segment to maliciously alter the Control Flow. 68

4 DRIVE High Level Attestation Concept and Architecture

4.1 High Level Attestation Concept based on Digests 76
4.2 DRIVE Architecture for Measurement, Reporting and Verification. 78
4.3 DRIVE Measurement, Reporting and Verification Process Overview. 81
4.4 Deployment of SuE and VS Example 1 85
4.5 Deployment of SuE and VS Example 2 85
4.6 Isolated Deployment of SuE and VS Example 1 86
4.7 Isolated Deployment of SuE and VS Example 2 86
4.8 Inconclusive virtualized Deployment of SuE and VS 86

5 DRIVE Measurement, Verification and Reporting Concept

5.1 Measurement Set Composition 101
5.2 Remote Attestation Protocol for DRIVE 105
5.3 Relevant Program Header Information from */bin/bash*. 110
5.4 Verification Excerpt of the GOT Layout and Symbol Resolution 117

6 Implementation and Evaluation

6.1 Software Components Architecture 134
6.2 DRIVE Measurement Component Subcomponents 135
6.3 Excerpt of an SSR for `/bin/bash` 137

List of Tables

2 Technical Background

2.1 Program Text Variant Analysis for Shared Libraries. 19
2.2 Program Text Variant Analysis for Programs. 19
2.3 Access Permissions of VAS Segments and encapsulated ELF Sections. 22

3 Security Analysis

3.1 Stealthiness Classification of Malware. 31
3.2 Persistence Classification of Malware. 32
3.3 Stealthiness and Persistence Properties of Code Corruption Attacks 42
3.4 Properties of Code Pointer Manipulations 44
3.5 Stealthiness and Persistence Properties of Non-control Data Attacks 47
3.6 Hybrid Attack Stealthiness and Persistence 49
3.7 Memory Manipulations conducted by Attack 69

4 DRIVE High Level Attestation Concept and Architecture

4.1 Isolation Grades for different SuE and VS Deployments. 87
4.2 Reliability regarding Isolation of SuE and VA. 88
4.3 Reliability in non-virtualized Environments. 91
4.4 Reliability in virtualized Environments. 91

5 DRIVE Measurement, Verification and Reporting Concept

5.1 Hashed Measurement Sets HMS 103
5.2 DML Integrity Verification 108
5.3 GOT Calculation for `/bin/bash`. 118
5.4 Expexted Detectability of Manipulations 127

6 Implementation and Evaluation

6.1 Access Permission Manipulation Detection during Attestation 145
6.2 Manipulation Detection of predictable and unpredictable Information 147
6.3 Performance Metrics for Measurement Component. 151
6.4 Evaluation Benchmark Experiment 1 154
6.5 Evaluation Benchmark Experiment 2 155
6.6 Evaluation Benchmark Experiment 3 156

List of Listings

5 DRIVE Measurement, Verification and Reporting Concept

5.1 `Bash` ELF Program Header Excerpt 99
5.2 Using `dd` to extract the `text` Segment from Bashs' ELF file. 110
5.3 Generating a `sha256sum` of an extracted Bash `text` Segment. 110
5.4 Program Headers from Bash ELF. 121
5.5 Section to Segment Mapping Information from Bash ELF. 122

Chapter 1

Introduction

1.1 Motivation

The complexity and number of cyberattacks have been rapidly increasing over the last few years. Specifically, during the last eight years, very complex attacks were detected that targeted a broad range of industries and governmental institutions – e.g. Iran nuclear facilities StuxNet (2010) [4], Belgacom & Bengal Mobile (2013) [5], J.P. Morgan (2014) [6], German Bundestag (2015) [7], Ruag AG (2016) [8]. These sophisticated attacks are known as Advanced Persistent Threats (APTs) and pose a substantial threat to many infrastructures due to their high complexity, evasiveness and diversity, rendering effective detection exceptionally difficult. Moreover, APTs are particularly tailored to circumvent broadly adopted and widely used countermeasures, for instance, firewalls, virus scanners and intrusion detection systems. As a consequence, APTs are usually detected a long time after their initial deployment and detection is often coincidental, e.g. based on suspicious findings during network traffic analysis.

However, the behavior of a computer system or, a device acting as a part of an IT-infrastructure, is defined by the software running on the system. Thus, nearly every attack exploiting a software vulnerability interacts with the system's memory to a certain extent. Often, systems are attacked by simply replacing or adding malicious software components permanently and executing them as desired. Those illicit modifications may occur both off-line, for instance, by malicious firmware manipulation, and during system runtime, usually by exploitation of well-known vulnerabilities like buffer overflows, format string, and write-what-where vulnerabilities, c.f. [9, 10]. Yet, the detection of persistent modifications on the file system level is well researched, and anti-virus/malware tools have been available for more than 30 years. Moreover, the measurement and attestation of system states, based on integrity verification of loaded software, is also well understood.

Still, the objective is not always permanent system file modification; instead, system infiltration often utilizes runtime memory and control-flow manipulation in order to

launch a successful attack. This means that the actual manipulation is conducted only in volatile memory and leaves no evidence inside permanent storage areas on the targeted system. Particularly in these cases, integrity measurement and anti-virus tools do not provide protection, because both rely on files and do not consider memory content. For instance, anti-virus tools were found to be ineffective in terms of file-less malware, c.f. [11]. Yet even those non-permanent attacks leave trails and can be detected inside the volatile system memory. Memory forensics [12] enables the detection of maliciously tampered volatile memory. Memory forensics tools and techniques, capable of detecting even the most sophisticated attacks, can be used to analyze suspicious system behavior. However, the field of memory forensics is not thoroughly researched and well understood [13]. Nonetheless, certain memory forensics tools exist, c.f. [14–16] that are able to extract system memory content during runtime and facilitate further system analysis. But these tools are only usable with expert knowledge in both memory forensics and attack vectors and, moreover, require an initial detection of any suspicious behavior.

Because current security technologies do not provide the capability for automated runtime state assessments, this thesis enhances the classic onetime software component attestation approaches from load-time toward continuous monitoring and attestation throughout the entire software component's lifetime. This enhancement facilitates the successful detection of complex malicious threats and thus helps to significantly reduce adversaries' capabilities and the time span of successful and long-lasting attacks.

1.2 Problem Description and Research Topic

In order for operators to ensure that a system or service still fulfills its primary objective, it is necessary to determine whether the system behaves as intended. Once a system transforms into an unknown or indecisive state, its functionality becomes completely unpredictable. In some cases, this transition could simply imply that a wrong or unknown configuration was intentionally deployed to the system. But in other more severe cases, it could also indicate that the system was unintentionally tampered with by a malicious adversary, causing the system to be fully or partially compromised.

In either case, an operator requires reliable information about the system state in order to perform necessary actions. For instance, the operator could simply report the system state or do nothing if no modification was detected; deploy a known configuration or software in benign cases; observe or analyze the system more closely; or instantly isolate or disable the system in severe cases and security-sensitive environments.

The acquisition of this reliable information is currently limited to state information only covering a fraction of the system state at a single point in time, i.e. the load-time of a software component. Although this load-time information is vital for a system state classification, the vast majority, i.e. the runtime state of the software component, is not

covered by current approaches and thus not considered for any classification process.

Accordingly, the research in this work will focus on the topic of continuous acquisition and classification of reliable system state information, also known as system attestation, based on state information acquired during software components' runtime. This runtime information provides the most recent details of software components' states and can be considered as one of the last characteristics that can be effectively observed on a system without interfering with the exact runtime behavior on the instructional level. This will enable a security classification of systems based on more relevant state information and thus the application of more definite actions, such as applying remediation strategies, based on the classification carried out.

1.3 Use Case

An operator owns, manages and operates a set of devices that form an operational network. In order to be able to offer customers reliable services, it is necessary to obtain reliable information about the current status of a device. This state may include, for example, the overall state of this system, including all determinable components, or may be limited to certain relevant components required for the reliable provision of its service. It is therefore necessary to determine whether a device or considered subcomponents of its system are still in a benign state or have been compromised.

The state is mostly defined by the software that is executed on the system. Therefore, the operator needs to determine whether the software, in particular the Operating System (OS) and running applications, are still configured and executed as intended. For instance, the operator wants to know that only certain applications – in particular versions – are executed. In case an unintended application is executed, the device may be considered as malicious and should be handled accordingly, for example, being isolated from the operational network.

There are many possible solutions to the problems of detecting and preventing the execution of unintended programs. One solution would be a *Measured Boot* along with an *Attestation*, c.f. [17, 18], used to detect the execution of unintended software. In addition to that, *Secure Boot* [19] that applies code signing techniques [20] can be used to prevent the execution of software that lacks a valid signature. However, both solutions are not designed nor are they suitable to detect or prevent attacks that happen after the initial loading of the software.

Since the operator is interested in the current state of the device, the information about the initial loading of the software is not enough. This is because almost all software is susceptible to runtime attacks that can modify the intended purpose of the software during its execution and, hence, execute arbitrary malicious functions. As a result, the once-loaded software may have taken on a malicious state not detectable by the deployed

integrity protection mechanisms.

The most recent information about the actual state of the software is always maintained inside the system memory. This means the in-memory representation of the software is the most current state information available. For this reason, the operator wants to attest the software's in-memory state, which is defined as the runtime state throughout this thesis. The operator must monitor, i.e. measure and compare the current runtime state against defined directives, in order to determine whether the software is still in a well-known benign state. It is only with this information that the operator can determine the current state of a device within her operational network and act accordingly when a malicious state is identified.

This thesis focuses on individual systems with the currently dominant architecture and assumes that the devices in the operational network contain a tamper-resistant *security module*, for instance a Trusted Platform Module (TPM), able to store the current system state and facilitate its secure reporting. Once a system state is stored inside the security module, it is assumed to be securely anchored and immutable to any modification without detection. The use case considered in this thesis, further assumes that:

(1) the operator maintains a central management system in a specific trusted management network and

(2) this trusted management network is strictly isolated from the operational network.

Therefore, only the network operator has access to the management systems and any service the management systems provide. Moreover, no one but the operator has access to the trusted management network; thus, no direct access from the operational network to the trusted management network is possible. This means that no one but the network operator can observe, intercept or tamper with communication data transferred via the trusted network. As a result, the devices inside the management network are considered trustworthy and, thus, can be used to securely maintain information to conduct a secure attestation of the measured information on the devices.

1.4 Research Questions, Goals and Objectives

1.4.1 Research Questions

In order to gain a deeper understanding of how and to what extent systems are affected by attacks that target compromising software at runtime, as well as to suggest solutions for further increasing the resilience of such systems against runtime attacks, the following research questions will be addressed in the course of this work:

(Q1) What are the capabilities, limitations and characteristics of software runtime attacks?

(Q2) What is necessary to establish a protection technology that implements continuous and reliable monitoring of the runtime state of systems?

(Q3) What are the capabilities, procedures and constraints of a continuous runtime protection technology?

(Q4) How can a runtime protection technology be realized and assessed on the basis of a designed runtime protection technology?

1.4.2 Research Goal

The overarching goal of this work aims at enhancing overall security capabilities by providing a runtime protection technology that considers attacks on the runtime state of modern computer systems. The protection technology should be able to monitor continuously runtime states of systems and determine on a different system whether monitored systems are still in a benign or transformed into a malicious state. This evaluation of the system state is to be carried out in a reliable and secure fashion so that the determination of the system state is evidence-proven.

The envisioned protection technology aims to advance the state-of-the-art technology in the field through enhancing and adapting concepts from both: (1) classic system attestation concepts – to securely collect, report and verify system states of software components and (2) memory forensics – to continuously observe, capture and analyze software components' runtime states. This should enable the successful detection of aforementioned attacks and provide evidence of whether a software component was illicitly modified during its runtime or not.

Based on the presented research questions and the goal, different objectives will be addressed during the course of this thesis. These objectives are presented next.

1.4.3 Research Objectives

(RO1) Carry out a security analysis of runtime-related attacks and threats.

In order to develop a meaningful runtime protection technology, it is important to first obtain an overview of the current threat and attack landscape. For this reason, the first goal is to investigate and analyze current threats and attacks that allow software to be compromised at runtime. Next, it is also necessary to investigate well-known countermeasures that can be used to defend against attacks. It is essential to understand to what extent the countermeasures provide effective protection or how they can be circumvented. Finally, different attack scenarios have to be developed which allow a later evaluation of the security concept. This is important to examining the security technology being developed

in terms of its effectiveness in ensuring that the defense strategies developed are effective.

Accordingly, this research objective (RO1) addresses the research question (Q1) and provides input for (Q2).

(RO2) Establish and develop a flexible runtime protection technology concept and architecture.

Based on the security analysis performed, a flexible concept and architecture must be developed that considers the results of the security analysis thoroughly. Furthermore, the concept and the architecture are to implement a reliable and verifiable security concept that allows a third system to make a decision regarding the current system status and which allows it to be carried out repeatedly. Here, it is expected that concepts from well-known attestation procedures in the field of trusted computing and mechanisms from memory forensics will have to be combined in order to implement a sensible security technology that meets the given requirements. The developed concept and the architecture provide the framework by describing its building blocks. Therefore, these building blocks must be mapped to concrete components and described and analyzed in more detail.

This research objective (RO2) addresses the research question (Q2) and establishes a basis for the next research objective (RO3) that is meant to develop and provide the technical solution of the runtime protection technology.

(RO3) Describe, develop and provide an implementable runtime protection technology.

The technical details of the runtime protection technology must be refined on the basis of the developed architecture components. To do this, the components and their concrete mechanisms must be thoroughly defined, described and analyzed. It is important to ensure that the individual components of the runtime protection technology work together to attest runtime system states reliably, conclusively and repeatedly on a trustworthy third system. The level of detail of the mechanisms and procedures must be sufficiently detailed to allow the implementation of the entire designed runtime protection technology. In particular, the collection of measurement data, secure storage and transmission of these measurement data and verification of the measurement data must be ensured. It is equally important to evaluate the developed technology with regard to its security at this conceptual level. It is expected that the results of this analysis will provide important input for implementation and thus provide a coherent and secure overall solution.

This research objective (RO3) addresses the research question (Q3) and develops

a coherent, secure and implementable runtime protection technology. It will provide the basis for the next research objective (RO4), which is aimed at implementing and evaluating the runtime protection technology in a prototype in order to develop and provide the technical solution of the runtime protection technology.

(RO4) Adopt, implement and evaluate the developed runtime protection technology.

On the basis of the developed concept and architecture from (RO2) and on the basis of the technical descriptions of the mechanisms and procedures from (RO3), a prototype implementation of the runtime security technology will be adapted, implemented and evaluated. First of all, a software architecture has to be developed that realizes a concrete implementation of the architecture and implements the described mechanisms of the runtime security technology. Subsequently, the actual prototypical implementation must be carried out with the result of providing an overall operational system that implements the developed core components of the technology.

On the basis of this prototype, a security analysis will then be performed to illustrate to what extent the entire system can recognize the defined attack scenarios from (RO1). In addition, the prototype is to be evaluated for possible performance problems, and, if necessary, different optimization strategies will be tested.

1.5 Research Plan and Methods

The research in this thesis relies on a variety of related existing solutions and results. Regarding the threat and attack analysis, many results and concepts in both academia and non-academia have been published and are well understood. In order to provide a meaningful solution, potential and current exploitation techniques, attack vectors, threats and software vulnerabilities must be carefully identified and thoroughly studied. Similarly, protective measures on the OS, isolation techniques and other security solutions must be carefully examined. As a result, this initial research facilitates the design and development of a strong and appropriate attack and threat model, which in turn builds the foundation for the runtime protection technology to be developed.

Regarding the runtime protection technology, both system integrity technologies for static- and runtime-measurement and memory forensics tools and concepts are readily available. The loading and address resolution processes as well as memory management are well-known and integral parts of the OS core functionality. For these reasons, the initial research will start with a deep analysis of well-known static system integrity measurement and verification schemes and an examination of the address resolution and loading process

for kernel and user space software components. In addition, the relationship between Executable and Linkable Format (ELF) files of encapsulated program texts on Linux- and Unix-based OSs and the loaded runtime artifacts will be analyzed with regard to user and kernel space.

Based on this work and the designed attack and threat model, an architecture and concepts, as well as technical details of mechanisms and procedures of involved components, will be developed and established. The technical solution to be developed will provide secure, reliable measurement, reporting and verification of runtime system states that can be carried out repeatedly to monitor targeted systems over a long period of time. Consequently, the result will be a complete runtime protection technology that can detect attacks on software at runtime that were previously undetectable.

The designed runtime protection technology will be instantiated in a prototype implementation to demonstrate the applicability of the solution, along with several components and tools, such as: (1.) memory measurement – to measure identified user and kernel space artifacts, implementing an effective data-collection design and anchoring the measurements to a security module; (2.) procedures to generate reference values based on reliable sources for the verification of the measured artifacts, and (3.) a reference implementation for the verification of the measured artifacts based on the collected or to be calculated reference values.

The prototype implementation will further be used to verify the initial concepts, to demonstrate performance impact to the system and to prove the successful detection of runtime attacks considered as potential threats to the system's runtime security.

1.6 Contributions

This work focuses on the behavior of software and systems during their runtime and develops a runtime protection technology capable of making continuous statements about determined specific system states in a comprehensible and reliable fashion. These statements are to be attested in order to determine whether a System under Evaluation (SuE) is in a trustworthy and reliable state. In this regard, this thesis aims at making the following technical contributions:

(C1) Detailed security analysis of runtime attacks and threats.

A detailed analysis of the current threat and attack landscape with regard to runtime attacks will be presented in Chapter 3. This contribution is based on the results of research objective (RO1). This analysis will provide an overview of malware and derive different key properties for later classification of system runtime attacks. In addition, concrete attack techniques that are used by malware to compromise a system and hybrid attacks that are compositions of multiple attack techniques will be studied, described and classified based on the derived key

properties. Corresponding countermeasures will be investigated and presented in order to determine how resistant systems are if countermeasures are deployed. Lastly, attack methods will be developed that utilize multiple hybrid attacks to, first, avoid or disable countermeasures and then to compromise a system over a longer period of time. For this purpose, a simple model will be established that allows modeling of these attacks. Additionally, these complex attacks will build the basis for defining executable attack scenarios that will be used later to evaluate the other technical contributions of this work.

(C2) Novel and holistic runtime protection technology.

The main contribution of this thesis is to provide the novel and holistic runtime protection technology Dynamic Integrity Runtime Verification and Evaluation (DRIVE). This will be achieved by the combined results of the research objectives (RO1) and (RO2) and will be addressed in the Chapter 4 and 5. DRIVE will enhance established static load-time attestation concepts by enabling a granular and continuous measurement, reporting and verification of different data artifacts present in the system memory during software runtime. For this purpose, a high-level attestation concept will be developed, and a flexible architecture will be designed that supports different instantiations of DRIVE. This architecture will identify and introduce necessary systems and components that will be used to establish an instantiated software architecture describing the individual building blocks alongside corresponding procedures of operations. A deployment analysis will also be carried out to derive requirements and to identify constraints for its implementation. Subsequently, required data-structures and mechanisms for measurement, reporting and verification will be defined and described in detail. In this context, great importance will be placed on the generalization of the developed data structures and mechanisms to support a secure and reliable attestation of different types of memory data artifacts in a unified way. Finally, a security analysis based on derived attack scenarios will be carried out and discussed.

(C3) Implementation and evaluation of the runtime protection technology.

A prototypical implementation of DRIVE will be provided and evaluated with regard to effects on system performance and achieved security capabilities. This contribution is based on the results of research objective (RO3) and addressed in Chapter 6. First, the implementation will be realized as a Proof of Concept (PoC) that implements the designed software architecture, defines data-structures and describes corresponding mechanisms for runtime memory data attestation. This will include secure and repeatable data acquisition, integrity-verifiable report generation and reliable verification on the basis of well-known trusted references. Second, a security analysis on the basis of the defined attack scenarios will be

carried out. Therefore, the attack scenarios will be simulated and evaluated by the PoC. The results of this analysis will determine whether DRIVE is a practical and implementable solution capable of detecting concrete memory-based attacks during software runtime. Lastly, the component implementing the measurement mechanism will be evaluated regarding its effects on the system on which it is deployed. Different optimizations will also be evaluated that aim at limiting the effects of the measurement acquisition process.

1.7 Outline

Chapter 2 builds the foundation of this thesis and discusses basic technical concepts that will be used in the course of this thesis. For this purpose, general concepts and mechanisms are briefly discussed which introduce and present program execution, loading processes, memory management and specific object properties of the resources used.

Chapter 3 provides a detailed security analysis of threats and attacks. First, malware is introduced and classified according to various properties. This is followed by a presentation and detailed analysis of attack techniques that can be used to break into systems or software at runtime. Existing countermeasures are discussed and examined for their effectiveness. Next, the actual threats are discussed and different attack scenarios are defined using a developed model for hybrid attack techniques.

Chapter 4 introduces a high-level concept and an architecture that allows attestation of runtime system states on a trustworthy third-party system. Subsequently, on the basis of the concept and the architecture, more precise system components are developed, and an instantiation of a software architecture is presented. Furthermore, a deployment analysis is carried out which examines the isolation of different system components and identifies architectural limitations.

Chapter 5 describes the mechanisms and procedures of the instantiated architecture in detail. For this purpose, the individual mechanisms and procedures are structured on the basis of data to be attested. The attestation mechanisms are divided into measurement, reporting and verification of memory data. In the first step, simple and commonly applicable data structures are defined, and a general transfer protocol is specified. This is followed by more complex attestation concepts, which are based on the mechanisms and data structures developed for refining them further. At the end of the chapter, a security analysis is carried out to ensure that the defined attack scenarios are covered from a conceptual point of view and also to provide conclusions and suggestions for implementation.

Chapter 6 describes the implementation of the prototype. The concrete implementation of the software components is discussed and the concepts used in the development are explained. This is illustrated by an attestation example. A security analysis is then

carried out in order to prove the effectiveness of the developed concepts on the basis of their implementation. For this purpose, certain attacks on the system are simulated, and which attacks were recognized by the implementation are analyzed. The final step is an evaluation of the prototype with regard to its performance. In particular, the measurement process is evaluated here because it has direct effects on the system to be monitored, but is not necessary for strictly operational reasons.

Chapter 7 puts the results of this work in relation to related research. The related research and the state-of-the-art technologies are first presented and described. The contributions of this work are categorized and discussed for each individual research area. In particular, the actual contributions to the current state-of-the-art approaches will be discussed.

Chapter 8 summarizes the results of this work and refers to the research questions and objectives presented in the introduction. The conclusion of this chapter presents the outlook for future work that should be considered in this area.

Chapter 2

Technical Background

This chapter introduces the technical principles that this work is based on. The concept of this work is deeply integrated into the operating system's basic low-level functionality and based on various principles and technologies. First, the basics of loading and executing a program are briefly described. Particular focus is put on the transition from a file-based program to an executable in-memory program which is explained for kernel and user space components. Next, different types of programs, more precisely variants of program text, are identified and described. This is essential for the conceptual work, because the variants of program text influence the loading of programs which, in turn, affects their representation in the system memory. Last, related memory management concepts are described for which an overview of memory access and protection for memory mapped segments and sections is provided. In addition to that, this chapter discusses the static and dynamic behavior of these mappings and provides a definition for predictability.

2.1 Program Execution Principles in Computer Architectures

The execution of every program meant to run on a computer system follows a certain method. Leaving aside the specific software components, every Central Processing Unit (CPU)-based computation relies on instructions and data present in the system's main memory. Even though different architectures come with different concrete implementations for the process, all modern architectures follow certain principles and are based on a similar organization of the data involved and on the invocation of related operations that apply computations on them.

Before the CPU starts computations, architecture-specific registers are filled with a instruction and data from the system memory. Data refers to all kinds of different structures such as constants, variables or pointers. The corresponding `LOAD` instruction from the Instruction Set Architecture (ISA) moves bytes from the system memory to the registers

and the STORE instruction moves the register content from the registers back to the system memory. During a typical instruction-execution cycle, an instruction is first fetched from the system memory. This instruction may cause fetching additional data (operands) that are also present in the system memory. After the instruction's execution, the result may be stored back in the system memory, c.f. [21]. This execution cycle follows a strict sequence; the CPU executes one instruction after another, unless interrupted, terminated or completed.

The set of instructions present in memory is known as the code base and the sequential execution of these instructions is managed by one specific register, referred to as the Instruction Pointer (IP). The IP points to the currently executed instruction and once the instruction has finished its execution, the IP is incremented by one and the execution cycle starts again. However, if a control flow transition instruction has been executed – i.e. a branch instruction – the IP is altered accordingly and points to the newly computed or assigned branch value. In both cases the specified instruction is executed by the CPU as described and the execution cycle continues as described until a final instruction is executed that eventually terminates the program.

2.2 Program Loading in Modern Operating Systems

2.2.1 Organization of Program Text and Data

In Unix-based systems, the Executable and Linkable Format (ELF) is a standard file format defining the organizational structures of executables, shared libraries, and core dumps that act as containers for the program text and data portions of a program. Inside the ELF, different portions of the program text and data are represented by different sections. The most notable sections are (1) the .text, .init, and .plt sections, encapsulating executable instructions, (2) the .data, .rodata and .bss sections, holding initialized and uninitialized data, and (3) the .got section, a table that organizes data structures used for resolving function symbols. There are more sections in the ELF (e.g., program- and section headers, and procedure- and symbol resolution tables). A comprehensive overview is given in [22]. Figure 2.1 shows an ELF to Virtual Address Space (VAS) user space mapping example. As shown, multiple related sections from the ELF are organized in a single segment. This relation is specified in the Section to Segment mapping in the ELF header and can be different for each ELF.

2.2.2 Loading Processes of Programs

Before instructions can be executed by the CPU, the required program text must fully reside in the system memory. Depending on the actual program, different loaders are responsible for the loading process. There is the Bootloader, the Kernel Module Loader

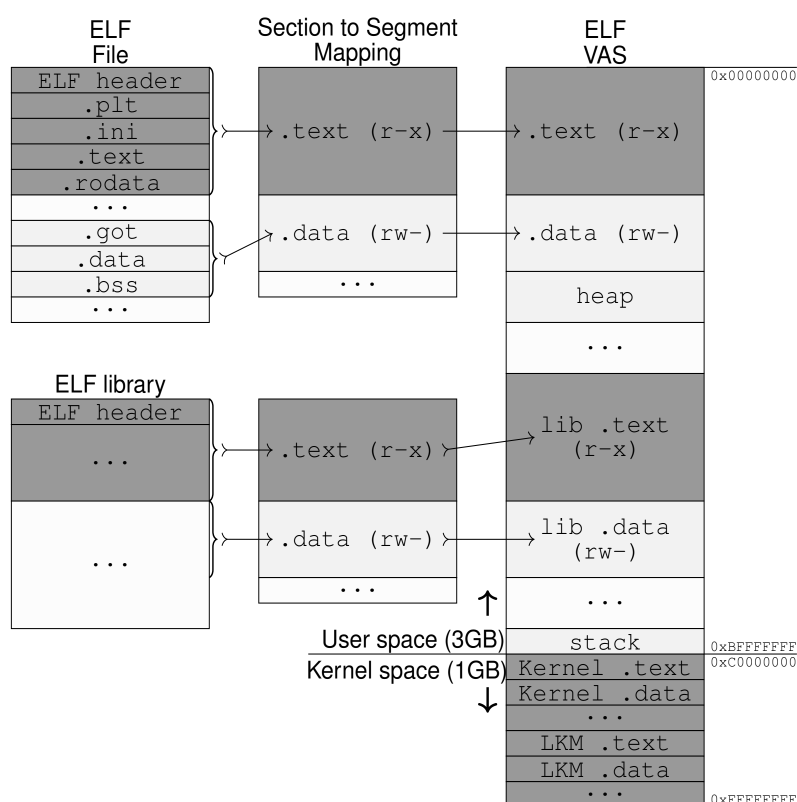

Figure 2.1 – ELF to VAS Mapping demonstrating Section to Segment Mappings for a 32-bit System.

and the User Space Process Loader for ELF. The result, after all loading mechanisms have been executed, is shown in Figure 2.1 and is described in the following for each bootloader.

System Bootloader and Kernel Setup The Bootloader instructs the CPU to load the OS Kernel into the system memory at a fixed location. From this point onward, the Kernel takes complete control over the memory management. If the Memory Management Unit (MMU) is present, the Kernel initializes it and sets up internal structures to organize the mappings between the physical and virtual memory; the managing structure is referred to as the *page table* [23]. Most importantly, in this process the virtual memory is separated into kernel space and user space. Afterwards, the Kernel continues with its execution until an Loadable Kernel Module (LKM) needs to be loaded. If there are no LKMs, control is transferred to the user space process loader and management system, conducted for instance by *init*, *systemd* or *upstart* on Linux-based OS.

Kernel Module Loader The Kernel Module Loader (KML) loads the requested LKM ELF program text into memory and transforms the LKM into a ready-to-run state. During this process, dependencies between different LKMs are resolved.

More precisely, the KML (1) inspects every unresolved symbol, either referencing procedures or data from the LKM itself, the Kernel or other LKMs; (2) resolves the symbol by determining the targeted Virtual Memory Address (VMA); and (3) patches the determined target VMA code pointer directly into the program text in LKMs' `.text` segment present in the system memory. After all transformations are applied, the LKM is in a ready-to-run state and can be invoked and executed as intended.

User Space Process Loader After the OS kernel has finished its loading process, the control is redirected to user space process management programs. Every user space process is organized in the same way: It has the same view of the available system resources, i.e. the VAS, as depicted in Figure 2.1. The layout and size of the VAS is identical for every process. The typical size for the 32-Bit VAS is 4 GB with two segmentations:

- User space 3 GB (`0x00000000 - 0xBFFFFFFF`)
- Kernel space 1 GB (`0xC0000000 - 0xFFFFFFFF`)

By contrast, 64-bit systems usually implement an address width of 48 bits, resulting in 256TB VAS:

- User space 128 TB (`0x0000000000000000` - `0x00007FFFFFFFFFFF`)
- Kernel space 128 TB (`0xFFFF800000000000` - `0xFFFFFFFFFFFFFFFF`)

The actual process-loader program invoked in Linux systems is implemented by `ld-linux.so` (*LD*) [24]. Similar to the KML, *LD* loads the program into the system memory and executes the symbol resolution and relocation process [25], where appropriate (c.f. Section 2.3). Usually, programs depend on external libraries which are, in turn, again programs consisting of program text and data encapsulated in ELF[1]. Therefore, *LD* also loads all referenced shared libraries into the process memory, before the symbol resolution and relocation phase is performed. The symbol resolution and relocation are also executed for every shared library, as dependencies between different shared libraries occur very frequently. After the dependency loading, function resolution and relocation phase of *LD* has been successfully completed, the final process image is in a ready-to-run state. In a final step, *LD* delegates the execution to the loaded program by calling its `main()` routine. From that point onward, the program is available as a process in the system. One process image of a loaded program is depicted in Figure 2.1, including the memory layout of an *LD* loaded library.

[1] Although shared libraries are programs, they cannot be executed or invoked directly; they lack a starting routine.

2.3 Program Text Variants

Program text is categorized into two variants that influence the loading process during the transformation into a ready-to-run state.

1. Position Independent Code (PIC) uses relative addressing and can thus be executed from arbitrary memory addresses.
2. Relocatable Code (RCC) depends on definite absolute memory addresses that must be resolved by the linker or loader prior to RCC's execution.

Position Independent means that the source code was compiled in a special way by eliminating the use of direct memory addresses within the program text. Necessary memory-based address access (target addresses) are resolved in PIC with a Global Offset Table (GOT) [25, 26] mapped to a specific `.got` memory region.

In contrast to PIC, RCC relies on direct memory addresses inside the program text for function calling. Inside the program text, certain placeholders exist which are replaced during the relocation process by calculating relative or absolute target addresses. These replacements take place during the link or the load time. Both variants have certain benefits and effects on the DRIVE concept which will be subsequently enlarged upon.

2.3.1 Relocatable Code

RCC is the standard mechanism for generating program text for user space programs, the Kernel and LKMs. Necessary relocations that rely on fixed memory addresses are resolved either during the linking phase or during the load-time with the help of the dynamic runtime loader [24]. More precisely, program text in user space and the OS kernel is at least deferred until the link-time because it depends on a fixed load address, which means that relocation can happen earliest during the linking phase [25].

For the link time relocated program text, the instructions already contain the concrete target addresses for all symbols or use relative addressing. This means the `.text` segment's program text within the ELF is identical to the program text in the memory segment. A relocation after the linking phase is therefore not necessary. In other words, although the link time relocated program text is based on relocatable code, it behaves more like PIC during loading and execution, explained in the next Section 2.3.2.

In contrast to this, there are LKMs that cannot be relocated during link time since they depend on dynamic addresses only available during runtime. Instead, the RCC ELF contains a specific symbol table and the program text uses placeholders for all referenced symbol addresses. During the loading process the loader analyses this symbol table, resolves the target addresses of the symbols and patches the program text placeholders accordingly with resolved concrete target addresses. In case of LKM loading, symbols

from both the Kernel and other LKMs are resolved and taken into consideration during the relocation process.

Although RCC plays a negligible role for shared libraries today, the process of symbol resolution and program text patching is similar to the one described for LKMs. Instead of the LKM loader, the user space process loader `ld` is responsible for relocation. In this case, symbols from other libraries are the main target during the symbol resolution process.

2.3.2 Position Independent Code

Generating PIC for shared libraries is the default behavior in any modern Linux system; all shared libraries in Linux are PIC.

During the loading process of PIC, the program text remains unchanged and identical to its counterpart in the ELF. As mentioned, the same principle applies for link time RCC. The main benefit of this is the possibility to share memory mapped content between multiple processes[2]. As long as a page-mapped portion remains unchanged it is shared with other processes in the system, and thus resides only one time in physical memory. Considering that many resources, such as libraries like for instance `glibc` and `libld` are shared between all processes, this deduplication of resources saves a considerable amount of physical memory.

Recently, so called Position Independent Executables (PIEs) become more relevant and are used in many current Linux distributions, replacing the link time RCC ELF. Regarding the resource sharing and behavior PIE is equal to link time RCC and PIC. Accordingly, for the rest of this work, the term PIC is also used to denote PIE if not indicated otherwise.

In contrast to link time RCC, PIC benefits hugely from Address Space Layout Randomization (ASLR), since it does not rely on fixed addresses and thus renders possible vulnerability exploitation harder, because fully resolved memory addresses are often required to conduct certain attacks. Additionally, PIC also facilitates the dynamic loading of shared libraries during runtime, which is used by many applications with the assistance of the `dlopen` system call. From the perspective of DRIVE, link time RCC and PIC are equal. This means the measurement and verification of corresponding `.text` segments are fully supported.

2.3.3 Program Text Distribution Analysis

To provide a better overview of the program text variants and which variants are used, three different distributions (Ubuntu, Debian and Fedora) for three different architectures (X86, X86_64, ARM64) were analyzed to determine the distribution of PIC, PIE and link-time RCC program text. In order to determine if PIE was used, the Linux tool

[2] This sharing, the main advantage of PIC, is not necessary for LKMs since one LKM may only be available as a single instance in kernel space.

Table 2.1 – Program Text Variant Analysis for Shared Libraries.

Operating System	Architecture	All	PIC	RCC_{load}	Canaries
Fedora Cloud (21)	X86	921	921 (100%)	0 (0%)	395 (43%)
Debian (wheezy)	X86_64	701	701 (100%)	0 (0%)	174 (25%)
Debian (wheezy)	X86	705	705 (100%)	0 (0%)	193 (27%)
Ubuntu (vivid server)	X86_64	892	892 (100%)	0 (0%)	480 (54%)
Ubuntu (vivid server)	ARM64	883	883 (100%)	0 (0%)	461 (52%)
Ubuntu (vivid server)	X86	891	891 (100%)	0 (0%)	482 (54%)

Table 2.2 – Program Text Variant Analysis for Programs.

Operating System	Architecture	All	RCC_{link}	PIE	Canaries
Fedora Cloud (21)	X86	687	482 (70%)	205 (30%)	590 (86%)
Debian (wheezy)	X86_64	650	576 (89%)	74 (11%)	406 (63%)
Debian (wheezy)	X86	676	577 (85%)	99 (15%)	416 (62%)
Ubuntu (vivid server)	X86_64	808	585 (72%)	223 (28%)	704 (87%)
Ubuntu (vivid server)	ARM64	777	558 (72%)	219 (29%)	655 (84%)
Ubuntu (vivid server)	X86	807	584 (72%)	223 (28%)	705 (87%)

`hardening-check` was run. On top of that, the distinction between PIC and RCC was made on the basis of available relocation tables or available Procedure Linkage Table (PLT) code[3].

As shown in Table 2.1, all shared libraries use exclusively PIC. Additionally, the analysis confirmed that most executable ELF files still use link-time RCC relocation. However, PIEs were found in all analyzed systems, see Table 2.2. In addition to that, the last column in each table indicates whether stack canaries were used. This will be discussed in greater detail in Section 3.2.4.

Regarding kernel space, load-time relocated RCC still plays a major role for LKM. All LKMs analyzed use load-time RCC and are thus relocated during initial loading. The Kernel images are, as expected, all statically linked.

2.3.4 Global Offset Table

Both PIC and link time RCC in user space are usually paired with a mechanism called lazy-binding [24], implementing an on-demand function symbol resolution and relocation process. This means that whenever a symbol is used for the first time during execution it is resolved by a special component of the runtime loader and afterwards invoked.

Technically, the symbol resolution involves the maintenance[4] of a table known as the Global Offset Table (GOT) (`.got`). The symbol resolution mechanism is rather complex, because it involves trampoline jumps to symbol resolution functions of the loader on first

[3] https://stackoverflow.com/questions/1340402/how-can-i-tell-with-something-like-objdump-if-an-object-file-has-been-built-wi

[4] Managed by the Procedure Linkage Table `.plt` as part of the `.text` segment.

use. After the symbol's address is successfully resolved, the initial trampoline is replaced by the resolved symbol address. This means that from this time onward, every subsequent access to the symbol does no longer trigger the symbol resolution code. Instead, the resolved symbol is directly referenced in the GOT and every subsequent access can be done by simply accessing the resolved symbol address. A concrete example of the GOT and the function resolution process is provided in Section 5.2.4.

2.4 Memory Management, Access and Protection

Memory management is a core functionality, provided by the OS kernel. To set the scene the previous sections have provided a brief explanation of the organization of program text and data in ELF sections, and an introduction of the segmentation inside the VAS. The following sections will discuss memory management in more detail. To that effect, a description of the paging mechanism regarding segmentation in VAS, the memory schemes used to protect the system memory in modern OS, and finally the static and dynamic behavior of memory mapped program text will be given.

2.4.1 Virtual Memory Management

As described, the program text and data is organized in different segments and sections within the ELF. During the loading process multiple related sections are also joined to segments, representing the organizational structure in the VAS. However, the internal structure inside the OS kernel, and at the physical hardware layer, is organized in pages of a fixed size (usually 4096 Bytes). As a result, a VAS segment is an ordered logical representation of the physical pages mapped in memory. This additional abstraction layer between VAS and physical memory enables different process' VAS to share the same physical pages, reducing the amount of required physical memory pages significantly.

In general, segments are shared between multiple processes whenever possible. However, as soon as a process writes to a shared segment, a Copy-On-Write (COW) mechanism in the page fault handler is executed allocating a new physical memory page for the process and copying the content of the old page to the newly allocated. After that, the write-operation is executed and the newly allocated physical page is no longer shared.

Whether a VAS segment is expected to change or not, is determined during the compilation and linking phase of the ELF and depends on the individual section. For this reason, access permissions are now presented in more detail. In addition the dynamic behavior of certain sections, and how this affects DRIVE, is described in more detail.

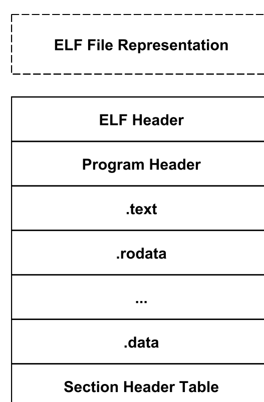

Figure 2.2 – Representation showing the inner Structure of Headers, Sections and Segments in an ELF File.

2.4.2 File to Memory Mapping

Especially in the context of loading and execution of applications the ELF files are not just copied into the system memory. As depicted in Figure 2.2, there is a structure within the ELF file, which describes the specific data needed for the execution of an application on top of the operating system.

All the information of the file is used by the systems' loader component responsible to load the individual segments of the ELF into the system memory. In addition to that, libraries used by the program are also loaded into the system memory following nearly the same procedure. Additionally, two memory areas, i.e. Heap and Stack, are created. Once everything is correctly set-up and initialized, the control flow is transferred to the application which is now running as a process within the operating system. A schematic process image is shown in Figure 2.1.

2.4.3 Access Rights of Memory Mapped Segments

Access to memory mapped segments is defined by access permission flags, controlled and enforced by the OS. Apart from security-related access control mechanisms, the access permissions also determine whether a segment can be shared or not.

Table 2.3 shows the access permissions of the `.text`, `.data`, `Heap` and `Stack` VAS segments and lists their designated ELF sections encapsulated inside segments, along with their individual access permissions, i.e. `(r)ead`, `(w)rite` and `e(x)ecute`. In specific circumstances, mappings with `rwx` permissions exist and are indeed necessary, as, e.g., virtual machine- and interpreter-based programming languages often require access permissions that are considered insecure.

DRIVE facilitates access permissions as an indicator for potential threats and therefore measures them as metadata. The metadata is analyzed during the verification phase and if unexpected changes to access permissions are detected the system is considered as

Table 2.3 – Access Permissions of VAS Segments and encapsulated ELF Sections.

VAS Segment		ELF Section		
Name	**Permission**	**Name**	**Permission**	**Type**
`.text`	`r-x`	`.text`	`r-x`	`sp`
		`.init`	`r-x`	`sp`
		`.plt`	`r-x`	`sp`
		`.rodata`	`r--`	`sp`
`.data`	`rw-`	`.data`	`rw-`	`du`
		`.bss`	`rw-`	`du`
		`.got`	`rw-`	`dp`
`Heap`	`rw-`	`--`	`--`	`du`
`Stack`	`rw-`	`--`	`--`	`du`

Permissions: `(r)ead, (w)rite, e(x)ecute`
Type: `(d)ynamic, (s)tatic, (p)redictable, (u)npredictable`

compromised and becomes untrusted.

2.4.4 Static and Dynamic Behavior of Programs

In addition to the aforementioned access rights, Table 2.3 categorizes each individual encapsulated section according to their expected dynamic behavior. The categories are `(s)tatic` and `(d)ynamic` types and predictability of the content, that is `(p)redictable` or `(u)npredictable`.

Unpredictable *means that it is impractical to compute the result of one or multiple subsequent deterministic functions in a reasonable time or with reasonable effort, even if all inputs to the function are known in numbers and values.*

For instance, it is impractical to calculate the expected content of all dynamic parts of the memory at runtime. This is because one or multiple subsequent functions with arbitrary numbers of known inputs and arbitrary known values must be considered in order to be able to compute the content, even if the functions themselves are deterministic.

Predictable *means that it is feasible to compute the result of one or multiple subsequent deterministic functions in a reasonable time or with reasonable effort, under the assumption that the number of subsequent functions and their inputs are limited in numbers, and all inputs to all functions are known.*

For instance, some static portions of memory do not depend on any input and can be derived by applying a deterministic function, i.e. the identity function. Value-dependent computations of content in dynamic portions of memory, for instance the calculation of a pointer address that is based on a known base-address and an offset to a specific function,

are also feasible. In the latter case all inputs are known, the number of inputs are known (two) and only one deterministic functions is invoked a couple of times[5].

According to the given definitions, statically classified sections in the text segment are always predictable. Modifications made inside the `.text` segment only occur during the loading process, depend on limited well-known inputs, and usually do not change at runtime. In contrast, the `.data` segment encapsulates both predictable and unpredictable sections. That means the segments' content is expected to change during runtime. If these alterations occur only once or a couple of times and the input values are well-known, then the section is considered predictable. This is the case for the `.got` section. In other cases, like for instance the `stack` segment, the content changes with every other function completely and every computation of a value inside the function changes values in the current `stack`. Here it is not enough to know all inputs to the current `stack`, but also all inputs and outputs for every function since the program started. This is because the current `stack` depends on all previously conducted computations. It is therefore impractical to compute the content of these dynamic memory portions, which is why the computation is considered as unpredictable.

To conclude, the behavior and predictability of whole segments or the individual sections affects the measurement and the verification processes. While static segments can be measured as a single instance, dynamic sections must be measured individually as sections or with an even higher granularity, i.e. a specific portion of a section. However, verification processes of dynamic sections are only applicable if the content is predictable. As explained, the verification of unpredictable memory portions is impracticable and, thus, unpredictable memory portions are not considered by DRIVE for measurement and verification. In contrast, the verification of predictable dynamic sections is considered feasible and applied accordingly by DRIVE. A more comprehensive analysis of the measurement and verification of processes can be found in Section 5.

2.5 Summary

In this chapter, basic principles related to program text loading and execution, memory management, access permissions and dynamic behavior during runtime have been presented. The execution of programs always necessitates that the program, more precisely the program text and its data, is loaded into the system memory. During the loading process, the exact memory layout is created on the basis of the information inside the ELF file and access permissions are assigned to different memory mappings that depend on the type of the mapped content. Moreover, the exact loading of the program is influenced by the type of program text encapsulated, that is either PIC or link- or load-time RCC. During the execution of the program, different in-memory parts behave differently; some

[5] A reasonable value of subsequent calls depends on the complexity of the operations applied.

parts behave statically and other parts dynamically. While static memory contents are always predictable, dynamic content is either predictable or unpredictable. This is defined by the exact type of the data in memory or the purpose of a specific segment or section. To this end a definition was provided that argues that the predictability depends on whether a computation can be made in reasonable time or with reasonable effort to compute an expected exact result based on well-known inputs.

Chapter 3

Security Analysis

This chapter identifies and analyzes threats and attacks that can compromise a system during different phases of its runtime. First, different concepts of malicious software (malware) are introduced. The main focus lies on malware key properties, their identification and further explanation.These key characteristics form the basis for later analysis and facilitate a more objective view of threats and attacks. Secondly, Control Flow Manipulations Attacks, the key building block of modern malware, are introduced and analyzed in detail. In particular all major techniques, i.e. *code injection*, *code replacement*, *code pointer manipulation*, *code reuse*, *non-control data attacks* and different combinations of them, are presented and mapped to the identified key properties of malware. Thirdly, well-known and future countermeasures to these attacks are presented and discussed. This is followed by the introduction of a threat and attack model for DRIVE. These models will be used henceforth in order to determine the effectiveness of DRIVE and explain limitations to the concept. On the basis of these models, multiple attacks are also defined that act as concrete instances of attacks in order to carry out a security analysis at concept and implementation level. Lastly, different security assumptions are described that are necessary for a secure operation of DRIVE.

3.1 Malware

Malicious software, in short *malware*, stands for a specifically crafted piece of executable code that conducts unsolicited actions on a computer system. Souppaya defines malware as follows:

"Malware, also known as malicious code, refers to a program that is covertly inserted into another program with the intent to destroy data, run destructive or intrusive programs, or otherwise compromise the confidentiality, integrity, or availability of the victim's data, applications, or operating system." ([27])

3.1.1 Malware Types

Malware is often classified imprecisely. For instance, the term (Anti) Virus, is falsely used for (Anti) malware. A virus is only one special kind of malware, but cannot be used to describe malware in general. The classic categories of malware are as follows:

Virus “[...] a virus is a program that can reproduce itself by attaching its code to another program, analogous to how biological viruses reproduce.” [28]

Worm “A worm is a process that [...] spawns copies of itself, using up system resources and perhaps locking out all other processes. On computer networks, worms [...] may reproduce themselves among systems and thus shut down an entire network.” [21]

Trojan Horse “A program that masquerades as a useful service but exploits rights of the program’s user in a way the user does not intend.” [29]

Rootkit “A rootkit is a program or set of programs and files that attempts to conceal its existence, even in the face of determined efforts by the owner of the infected machine to locate and remove it.” [28]

It is important to note that the classic categories of malware only provide a general terminology for certain properties of malware. Although the classical categories are commonly used when classifying malware, malware cannot be clearly assigned to a single category nowadays, because they almost always use multiple strategies during infection, replication and during their runtime behavior.

For instance a Bot, one of the most common types of malware today, was once an automated isolated instance able to perform predefined (malicious) actions paired with remote control functionality. Today’s Bots still perform automated predefined actions and are remotely controllable. Yet, they inherit and combine the reproducibility property from the traditional Worms to autonomously infect other systems and the stealthiness property of classical rootkits, in order to hide themselves from detection, c.f. [30]. Another well-known example – referred to as the first cyber warfare weapon – is the *StuxNet* worm, c.f. [4]. As mentioned before, the Worm-related self-propagation functionality is certainly a key property of StuxNet. In addition, StuxNet also hides itself from the detection, a function inherited from rootkits, and makes use of a remote access command and control server to interact autonomously, a function that is usually typical of bots. In addition to that, StuxNet is also able to gather and communicate information for industrial espionage, which clearly is a key property of a Trojan Horse. Last, but not least, the actual damage StuxNet caused due to the infection and manipulation of multiple programs[6]. This means that StuxNet also inherits a key property of the classical Virus category.

[6] More precisely, StuxNet infected multiple systems, including the Windows OS and Supervisory Control and Data Acquisition (SCADA) control software based on Siemens S7-300 systems [4].

For these reasons, the classic malware categories are not suitable for categorization purposes of this thesis. In order to establish a clear distinction of the designed solutions' capabilities and constraints, this thesis will provide a different classification of malware; based on certain key properties that will be identified and addressed in the following Section 3.1.2.

3.1.2 Malware Key Properties

In general, malware induces unsolicited code that performs certain malicious actions:

- disabling desired functionality, for instance, by terminating a program
- introducing new malicious functionality, for instance, by executing a malicious program
- altering benign and desired functionality to become hostile, for instance, to leak confidential information

These general malicious actions are unfit for the classification of malware in this thesis or even for a general classification. Therefore, this work does not take into consideration the severity of malware. This is because once an adversary has access to the system by exploiting a vulnerability, the actual used exploitation techniques may be constrained, but if the skill of the adversary is sufficient to circumvent deployed countermeasures, every vulnerability can lead to the most severe effects possible.

For these reasons and because malware often relies on multiple malicious actions to reach its goals, the classification in this thesis focuses mainly on the other properties of the adversary's malicious actions instead of severity or the adversary's goal itself. As a result, the malicious actions will be classified based on the following key properties: (i) stealthiness and (ii) persistence. These properties allow a classification that is independent of the adversary's final goal and is therefore not constrained to the severity or the effects of an attack. In addition, these properties support an analysis over the entire attack life cycle.

Stealthiness

One primary goal of malware is to compromise a target system or application while hiding from detection mechanisms as long as possible. This is because detection means in almost all cases that the system will be disabled, disconnected or isolated in order to prevent any further damage. For this reason, stealthiness is one core property of any sophisticated malware and considered as the first key property for the classification in this thesis.

For the analysis of the stealthiness property, the stealth malware taxonomy used is based on the previous work from Rutkowska [31]. From today's point of view and in the development of malware, however, there are certain limitations in the taxonomy, especially

since it does not facilitate a differentiation with regard to predictability of the data involved. Accordingly, the taxonomy of Rutkowska is amended in such a way that it allows a finer differentiation between the data involved and thus allows a consideration of predictability.

The reinterpreted taxonomy introduces malware types from 0 to 3, where type-0 classifies the least and type-3 the stealthiest type. Type-0 was not considered as real malware according to Rutkowska's definition. However, it will be briefly discussed for the purpose of this taxonomy.

Type-0 Just like Rutkowska, type-0 malware is seen as malicious code that runs as a process within the OS, as depicted in Figure 3.1. This means that this type does not modify or compromise other parts of the OS, i.e. neither the OS kernel nor other processes. Type-0 malware must use official Application Binary Interfaces (APIs) provided by the OS, for instance to open a Transmission Control Protocol (TCP) connection or to access files. As a result, it is not suitable to hide very well within the system and can be detected very easily and even before its execution. In order to detect type-0 malware, it is necessary to decide whether a binary ELF file is benign or malicious. However, in general this problem is considered undecidable, c.f. [32]; still, anti-virus/anti-malware tools demonstrate that detection for widespread malware is possible based on blacklists or heuristics. In addition to that, Trusted Computing integrity protection mechanisms provide white list approaches to address these threats, i.e. Integrity Measurement Architecture (IMA) [33] for detection and *Secure Boot*, aka IMA-appraisal for execution prevention.

Despite the limited possibilities regarding its stealthiness, the severity and capabilities of type-0 malware must not be underestimated as it can do serious damage to an infected system, for instance: delete or encrypt user personal files, steal personal information, or access or take part in Distributed Denial of Service (DDoS) attacks as part of a Botnet.

Still, this work focuses on the infection of programs already in memory. For this reason, type-0 malware plays only a minor role for the remainder of this thesis.

Type-1 Type-1 malware effectively modifies different system resources inside already running programs - hence, loaded and available in the system memory - to change the behavior partly or completely as depicted in Figure 3.2. This means that a program which was benign until the time of infection suddenly becomes malicious. The malware could for instance change instructions within a program to alter its semantics or inject new instructions and modify code pointers to redirect the control flow to the injected malicious code.

Rutkowska defines the targeted system resources of type-1 malware as "system resources, we can divide [...] to those which are [...] relatively constant ('read-only') and to those which are changing all the time" ([31]). As a result, the former is introduced as code, representing a relatively constant or static runtime state, and the latter as data,

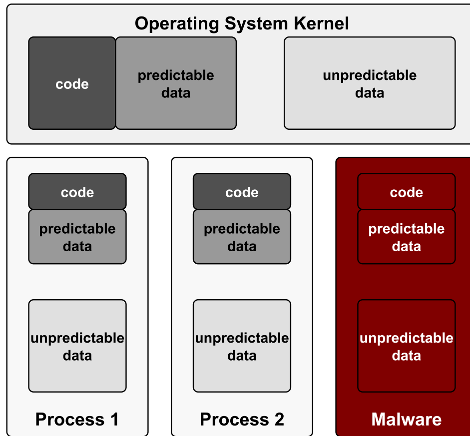

Figure 3.1 – Type-0 Malware running as a Process within the OS without affecting any other Process or OS Kernel Code or Data Sections.

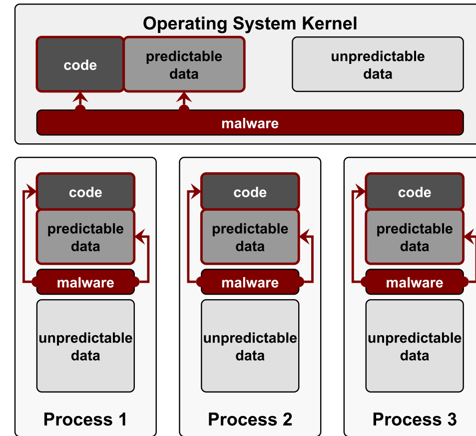

Figure 3.2 – Type-1 Malware effectively modifies predictable Code or Data Sections inside running Programs, such as Processes or the OS Kernel. This means the Behavior is modified and benign Programs become malicious.

representing dynamic runtime nature.

This broad definition of system resources is amended to achieve a more granular distinction and better fit to the provided definition of predictability in Section 2.4.4 and, thus, distinguishes between (i) code, (ii) predictable data, (iii) and unpredictable data.

System Resource 1. ***Code****: represents a memory region that consists only of instructions and pointer addresses. Code is always predictable and behaves relatively statically. Example:* `.text` *section of a program.*

System Resource 2. ***Predictable Data****: represents a memory region that consists of predictable data that behaves either statically, e.g.* `.rodata` *sections, or dynamically, e.g.* `.got` *section.*

System Resource 3. ***Unpredictable Data****: represents a memory region that consists of unpredictable data that behaves dynamically. Example:* `.data`*,* `Heap` *or* `Stack` *sections.*

Based on these three introduced system resources, the amended type-1 malware definition targets both code and predictable data as potential targets of exploitation. For this reason, the type-1 malware requires more complex detection mechanisms. If only static data is considered, it is sufficient to simply detect that a modification happened. But, in order to detect a modification in dynamic predictable data, one must be able to reliably and evidence-proof distinguish between a valid and an invalid modification. If this distinction can be made in reasonable time or with reasonable effort, for instance by mimicking program loading behavior, legitimate runtime patching or address resolution processes for lazy binding, then data is consider as predictable according to the provided definition of predictability, cf. Section 2.4.4. If this distinction cannot be made with a reasonable effort or in reasonable time, then it is considered as unpredictable.

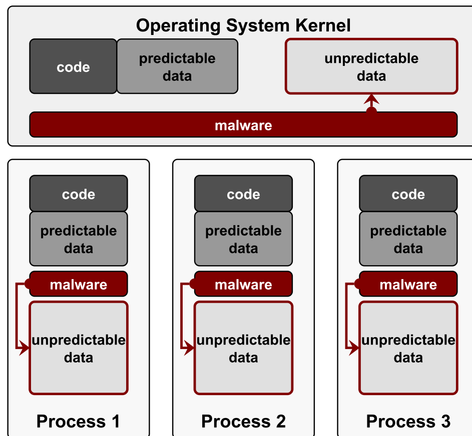

Figure 3.3 – Type-2 Malware modifies unpredictable Data Sections inside running Programs, such as Processes or the OS Kernel.

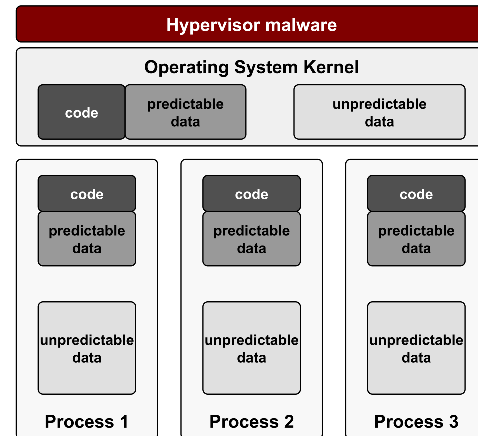

Figure 3.4 – Type-3 malware utilizes a Hypervisor to simulate an entire Runtime Environment on top of the OS Kernel. It does not make any Modification to the Code or Data Section of any Program.

One major contribution of this thesis is to provide such a reliable and evidence-proof distinction for type-1 malware according to the new amended definition. This means that this work provides a detection mechanism that is able to determine whether a modification was valid or invalid for dynamic predictable data.

Type-2 Type-2 malware also modifies system resources inside already running programs, as depicted in Figure 3.3. However, it limits itself to only modify dynamic unpredictable data in order to provide its functionality. This means a verification can no longer decide whether modification was valid or invalid without simulating the entire program execution. This is considered to be an unreasonable effort and, thus, the detection of these modifications need different paradigms for an effective detection. This does not mean that type-2 malware is generally impossible to be detectable. In fact, a lot of research has been done and there are solutions that try to solve specific parts of type-2 malware detection and prevention, for instance Control Flow Integrity (CFI) and Data Flow Integrity (DFI) addressed in Section 3.2.4. However, focusing on type-2 malware detection does not provide an effective detection of type-1 malware, because entirely different methodologies must be applied. For this reason, this thesis will focus mainly on the detection of type-1 malware in Chapter 5. Yet, an alternative solution based on metadata is proposed in Section 5.3 which allows the detection of certain type-2 malware-related attacks.

Type-3 Type-3 malware utilizes hypervisor-based approaches to hide itself from detection. This can be seen in Figure 3.4. Consequently, there is no self-contained mechanism inside the virtualized environment to detect a modification at all. The only possibility to

Table 3.1 – Stealthiness Classification of Malware.

Type	Runtime Behaviour	Stealthiness Classification
Type-0	self-contained process	very low
Type-1	predictable static	low
Type-1	predictable dynamic	medium
Type-2	unpredictable dynamic	high
Type-3	hypervisor-based	very high

detect this type of malware from inside the virtual environment would be to detect that the current system runs on top of a hypervisor. But, despite the possibility of detecting this, there is still a problem to employ any kind of countermeasure, because the hypervisor has full control of all system resources and could simply circumvent any type of countermeasure. For this reason, detection of type-3 malware is extremely difficult to accomplish when found in a virtualized environment. The only solution of detecting such malicious behavior is observing the system from an outside perspective. As a consequence, type-3 malware is not considered in the remainder of this thesis.

Stealthiness Classification According to the amended definition of malware stealthiness types, Table 3.1 provides a classification of the stealthiness property for malware. However, classification of the stealthiness ddepends solely on the exploiting and attacking techniques used. Therefore, this classification is used primarily to determine the stealthiness properties of attack techniques that will be addressed in more detail in Section 3.2.

Persistence

In addition to Stealthiness, malware can be classified based on its effects to a system over a period. For this reason, persistence is the second key property that will be used for the malware classification in this thesis. A coarse classification of persistence can be achieved by distinguishing between malware that is either non-persistent, i.e. it affects the system only for a very short period, or persistent, i.e. the system is affected by the malware over a long time, c.f.[34]. In order to achieve a finer distinction, persistent malware is further classified in memory-resident malware that affects system components only during their runtime, and resident malware that affects the entire system during its runtime but also after a restart or even a re-installation process.

Table 3.2 depicts the classification of malware based on their Persistence Class and their expected duration.

Non-persistent Malware Non-persistent malware infects a concrete system component only once during an attack. Hence, its persistence is classified as very low. This one-

Table 3.2 – Persistence Classification of Malware.

Persistence Class	Duration	Classification
Non-Persistent	One time	very low
Memory-resident	Until overwrite	low
Memory-resident	Until termination	medium
Memory-resident	Until reboot	high
Resident	Until reinstall	very high
Resident	Permanent in hardware	permanent

time effect means that the malware exploits a vulnerability exactly once to achieve the adversary's goal. However, this does not necessarily mean that non-persistent malware cannot affect the overall system security over a long period. But non-persistent malware has no persistent effects on the attacked concrete component, i.e. a program. This is a very important difference, because this property represents a major distinction between a non-persistent and persistent classification.

As an example, an adversary may aim at reading some confidential data of the attacked system components, e.g. steal an encryption key or read a particular memory address. Given a vulnerability that enables access to this information, the adversary launches the attack and acquires the information. If the adversary needs different information, such as a different encryption key or memory address, she must again exploit the vulnerability in the same way. This implies that the attacked system component was not altered and, accordingly, the attack has no persistent effect on the component itself. In another example an attacker wants to gain access to a system by spawning a shell. Again, given a vulnerability that enables this attack, the adversary exploits the vulnerability and gains access to the system, maybe even for a long time. This may indeed affect the overall security of the system, but still, the attacked component was not changed in any persistent manner. Consequently, this means, once the adversary leaves the system and needs access sometime later, she must again exploit the vulnerability in the same way as she did during the first exploitation.

In the first example, it is very unlikely that the attack will be detected at all if successful. However, in the second case, the adversary may be detected because her presence on the system may be suspicious due to the actions she performs. However, this behavior cannot be detected on the basis of an established violation of integrity, since the attacked software component has never been persistently modified.

Resident Malware The opposite of non-persistent malware is resident malware. In many cases the adversary needs full access to the system, either online by acquiring shell access or offline by manipulating the firmware of the system before deployment. In other cases, there are also vulnerabilities that enable the adversary to download malicious code

onto the system and either replace benign software with the malicious one or allow its execution. Depending on the particular attack, malware may infect files only, but there also exists malware that is able to nest itself into specific parts of the hardware, such as the Master Boot Record or the Basic Input/Output System (BIOS), and thus becomes a permanent threat to the affected system, c.f. [34]. For this reason, resident malware is classified as either very high or permanent. Very high means the malware is available on the system until the system is reinstalled or the malware overwritten by benign software. Permanent means that even a re-installation of the OS is not able to remove the malware.

Memory-resident Malware In between the non-persistent and resident malware, there is the huge spectrum of other persistent malware, referred to as memory-resident malware. In general, malware must react to system events to fulfill its intended purpose. This is usually accomplished by intercepting these events which requires that the malware is persistently available in the memory, c.f. [35, 36]. The memory-resident malware is therefore classified depending on the system component and the particular memory area it affects. For example, if the malware affects a user space process system component, the classification lies in between low and medium. Low implies that the malware modified memory data will possibly be overwritten or freed during execution, for instance inside the Heap. Medium means that the infected memory areas remain static throughout the entire runtime of the software, for instance the `.text` segment. Still, an attack can be composed of multiple related attack-steps corrupting dynamic or static parts in memory or alter certain metadata. For this reason, the classification of the persistence should be determined by the attack-step that lasts the longest. More information about attack-steps and stages is provided in Section 3.2.3 and 3.3.3. However, any attack made to a user space process eventually ends as soon as the process is terminated.

Similarly, if the attacked system component is the OS kernel and the modification was made to a static memory area, the malware will be active until rebooting and therefore its persistence is classified as high. But, once the system is rebooted the malware is no longer active and will not become automatically active again.

3.1.3 Summary

Malware comes in different forms and adopts different key properties to accomplish its malicious goals. In this thesis these the key properties are stealth and persistence. Both properties are related to one another, but it mainly depends on which specific system component is compromised by the malware itself. While type-0 malware is generally countered by well-known and adopted technologies, such as signature verification before load, i.e. secure boot, type-1 and type-2 malware is far stealthier and only detectable or preventable by very specific countermeasures. The main focus of this thesis is therefore the detection of type-1 malware, because defensive measures for this particular class are

not well researched. Type-1 malware may infect both kernel- and user space programs and reach a relative high persistence level. For instance infected kernel-based code may affect the system until reboot and user space programs until termination. Type-2 malware, is often less persistent, because most of the time it infects structures that allow a one-time execution of the malicious code. Despite the high stealthiness of type-2 malware, it is well researched and countermeasures are readily available. Type-3 malware is not considered in the remainder of the thesis. If a system is not able to determine that it is running on a compromised hypervisor, it is not possible to detect or prevent anything. The hypervisor can simply intercept any higher-level operation and prevent any mechanism from being effective inside the virtual environment.

The following Section 3.2 will present the dominant class of today's attacks, namely Control Flow Manipulation (CFM). Furthermore, the different identified CFM attack classes will be classified in accordance with the described malware properties derived in this section. This will build the basis for understanding the current threat landscape in modern systems regarding their exploitation and give reason to the significance of the DRIVE solution developed in this work.

3.2 System Memory Runtime Attacks

The last section discussed different malware types and classified them according to their behavior during their operational life-time. Yet, not many details on how malware actually accomplishes its goals were provided. For this reason, this section discusses different building blocks of malware and analyzes them regarding their capabilities and constraints. This thesis explicitly addresses runtime modifications to memory images; hence, different techniques used in today's malware are introduced, discussed and analyzed in the following sections. This will lay down the groundwork for the subsequent Attack and Threat Analysis presented in Section 3.3.

3.2.1 Attack Overview

Attacks on the system memory are manifold and occur in very different forms. For a differentiation between those attacks, the classification presented by Szekeres [37], as depicted in Figure 3.5, is adopted.

Szekeres distinguishes between four basic attacks to the system memory that were partially renamed to follow a more clear terminology throughout this work. The original names are indicated in parentheses. The attacks are: (1) Code Corruption, (2) Code Pointer Manipulation (Control Flow Hijacking), (3) Non-control data (Data-only) and (4) Information Leak.

In principle all attacks rely on an initial vulnerability that facilitates its exploitation and follow a certain pattern. It is reasonable to assume that all non-trivial programs

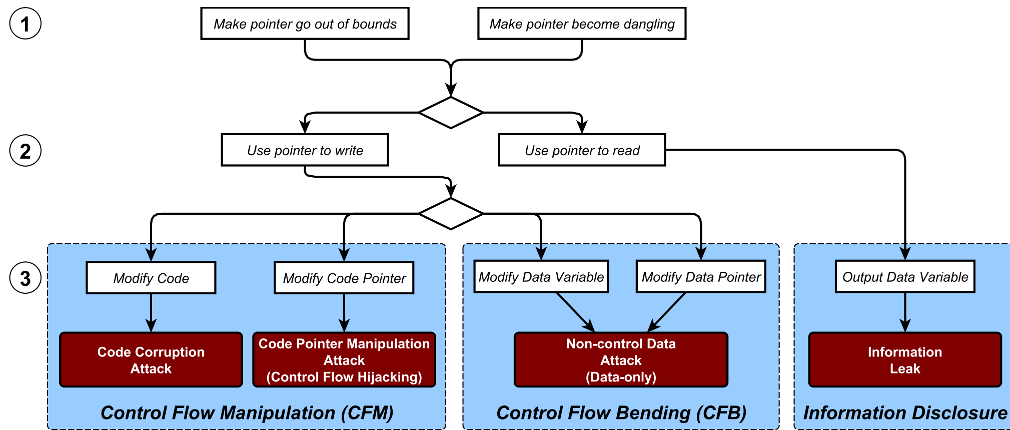

Figure 3.5 – Exploitation Model and Classification of Attacks relevant for attacking the Runtime System Memory. Adopted and simplified from Szekeres [37].

have vulnerabilities because they are written by humans who tend to make mistakes. For instance, not checking bounds when indexing an array or not freeing allocated memory properly. Consequently, if a program is implemented in a memory-unsafe programming language, which neither employs strict bounds checking nor enforces garbage collection – the C language as a primary example which does neither – every memory-related vulnerability allows either a spatial or a temporal memory safety violation. This means that the vulnerable program can ultimately be compromised by an adversary. Payer defines memory safety as well as spatial and temporal memory safety in [38] as follows:

Definition. ***Memory safety*** *"Memory safety is a property that ensures that all memory accesses adhere to the semantics defined by the source programming language. The gap between the operational semantics of the programming language and the underlying instructions provided by the hardware allow an attacker to step out of the restrictions imposed by the programming language and access memory out of context. Memory unsafe languages like C/C++ do not enforce memory safety and data accesses can occur through stale/illegal pointers."*

Definition. ***Spatial Memory Safety*** *"Spatial memory safety is a property that ensures that all memory dereferences are within bounds of their pointer's valid objects. An object's bounds are defined when the object is allocated. Any computed pointer to that object inherits the bounds of the object. Any pointer arithmetic can only result in a pointer inside the same object. Pointers that point outside of their associated object may not be dereferenced. Dereferencing such illegal pointers results in a spatial memory safety error and undefined behavior."*

Definition. ***Temporal Memory Safety*** *"Temporal memory safety is a property that ensures that all memory dereferences are valid at the time of the dereference, i.e., the pointed-to object is the same as when the pointer was created. When an object is freed, the underlying memory is no longer associated to the object and the pointer is no longer valid. Dereferencing such an invalid pointer results in a temporal memory safety error and undefined behavior."*

According to the provided definitions, the general pattern of vulnerability exploitation always follows a certain pattern. First, make a pointer *invalid*, by making the pointer go out of bounds, i.e. a spatial memory error, or make a pointer become dangling, i.e. a temporal memory error. This is depicted as step ① in Figure 3.5. Step ② in Figure 3.5 dereferences the invalid pointer to either use it to read memory, e.g. output data which is an *Information Leak*, or to write. Using the pointer to write enables different possibilities to exploit the vulnerability resulting in the following attacks used for further distinction as indicated in step ③: Modify Code w.r.t *Code Corruption Attack*, Modify Code Pointer w.r.t. *Code Pointer Manipulation Attack* and Modify Data Variable or Data Pointer w.r.t. *Non-control Data Attack*.

For the remainder of this work, the more general term CFM is used to address code corruption and code pointer manipulation attacks. This is because both attacks actively modify the control flow by manipulating either code or code pointers, discussed and analyzed in the following sections. Still, a clear distinction will be provided and the author will indicate whether a particular type is meant when necessary. Furthermore, the non-control data attacks remain, which are also partly discussed in this paper and classified as Control Flow Bending (CFB)[7], as defined by Carlini [39]. Since this work only distinguishes between predictable and unpredictable memory contents, it is irrelevant whether an attack facilitates a CFM or CFB. This means the concept does not make a distinction if code, code pointers or data is manipulated, it only matters whether the manipulated content can be predicted or not.

3.2.2 Control Flow Attack Foundation

As mentioned in the previous Section 3.2.1, CFM and CFB are general terms for different attacks or attack techniques that enable malware to (maliciously) modify the behavior of a program. This is done by adjusting the execution flow of the program itself by altering the Control Flow Graph (CFG) of the program either by direct manipulation of CFG nodes or edges in case of CFM or influencing logical decision-making code in case of non-control data attacks.

In general, there is no limit to CFM or CFB attacks regarding the potential target for exploitation. If the implementation of a system component consists of a vulnerability that can be used to subvert the control flow, it is exploitable. Different countermeasures may be active on a system that may influence the effectiveness of the exploitation or rendering a particular attack as impossible. But this does not change the fact that no ultimate protection technology exists that can eventually prevent an adversary to compromise the system; there are always countless ways.

From a security point of view an adversary will always try to compromise the system

[7] The term non-control data attack will be used in case a concrete attack is addressed and Control Flow Bending (CFB) when the general class of attacks is meant.

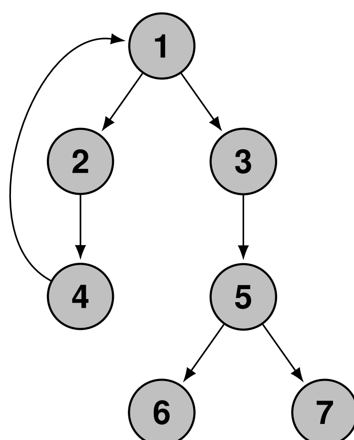

Figure 3.6 – CFG with multiple valid Nodes and Branches.

in the most effective way. At least this is the case for a professional adversary that has a particular goal in mind. For this reason, it is hard to predict which particular target is most susceptible or likely to be attacked. On a technical level, however, attacks can only be applied to software that allows a manipulation in the first place. However, as mentioned, most software today is still implemented in memory unsafe languages and therefore typically contains vulnerabilities. Specifically, most of the system programs, libraries and OS kernels are implemented in C/C++ for performance reasons; thus, any vulnerability that allows arbitrary memory manipulation can be affected and exploited by CFM or CFB attacks.

Control Flow Graphs and Execution Flow

A CFG, as depicted in Figure 3.6, represents all valid paths a computer program may traverse throughout its execution. A node in the CFG represents a basic block of one or multiple related instructions without any branches, i.e. without any direct or indirect jumps. Branches, on the other hand, are represented in the CFG by directed edges connecting other individual nodes. This means that traversing from a node ① to node ② is a valid operation because both nodes are directly connected via a directed edge. In contrast, directly traversing from node ① to node ④ is an invalid path, because there is no direct connection between both nodes. In order to get from node ① to ④ the only valid path is ① → ② and afterwards from ② → ④: ① → ② → ④.

As depicted in Figure 3.6 other valid execution paths are, for instance:

① → ② → ④ → ①
① → ③ → ⑤ → ⑥
① → ② → ④ → ① → ③

The aforementioned nodes are represented by one or more non-branch instructions the CPU executes and the branching mechanism is realized by branch instructions that modify the IP in the CPU. Once the IP is adjusted (branch), the CPU simply executes the next instruction the IP points to (node).

It should be noted that the CPU has no knowledge of the validity of the paths being executed; the CPU simply tries to execute what the IP references. If the IP points to a destination that does not contain a semantically valid instruction, for instance by pointing to a null-pointer due to a bug, an exception is thrown and the program is terminated instantly. Yet, the CPU only recognizes this during the execution of the targeted instruction. This means there is no mechanism that verifies whether the IP points to a valid or invalid target position before the actual execution. Even if such a mechanism would exist, it would not make any difference from the CPU's point of view, because once a valid execution path is left, the CPU cannot predict a correct alternative path on its own.

Another error that may be detected by the CPU occurs whenever the IP points to a semantic valid instruction, but the memory position of this addressed instruction is not marked as executable by the CPU; hence, the instruction is in a region that may not contain executable and, thus, IP addressable instructions, cf. Section 2.4 and Section 3.2.4. In the latter case, again an exception is thrown and the program is terminated, once the CPU tries to execute the falsely addressed instruction.

Definition and Differentiation

For the presented reasons, successful manipulation of the control flow must always target a valid instruction. Valid means in this case that the instruction is semantically valid and located at a memory address, more precisely in a memory page, marked as containing executable instructions.

Consequently, the two fundamental prerequisites for any successful CFM are:

CFG Prerequisite 1. *The CPU's IP must always point to a memory address referencing a semantically valid instruction.*

CFG Prerequisite 2. *The CPU's IP must always point to a memory address marked as executable.*

3.2.3 Types of Control Flow Manipulation and Bending Attacks

CFM and CFB attacks are classes of different techniques that allow the manipulation or bending of the program's control flow. The different types and their variants are presented hereinafter and described in detail. In addition to that, all attacks are categorized according to the identified and defined malware properties stealth and persistence.

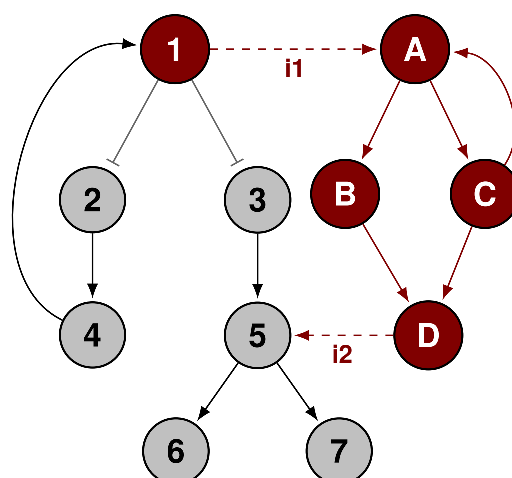

Figure 3.7 – CFG with injected Malicious Code and Modified Branch that alters the Control Flow so that the Malicious Code (①→Ⓐ) is always called instead of the Valid Paths (①→② or ①→③).

Code Corruption Attacks

Code Corruption Attacks are well-known and come in different variations. In any case, they always consist of code instructions that either:

(1) Inject code, i.e. introduce malicious code instructions in memory areas, as depicted in Figure 3.7; or

(2) Replace code, i.e. modify existing benign code instructions with malicious instructions, as depicted in Figure 3.8.

Code Injection The classic buffer overflow exploit as introduced by Aleph One [9] is a code injection attack that injects new instructions (also often called shellcode) into an executable marked memory segment. This is achieved by writing instructions directly into a buffer variable. These malicious instructions are represented by the nodes Ⓐ, Ⓑ, Ⓒ and Ⓓ in Figure 3.7. But simply injecting malicious code is not sufficient because the injected malicious instructions are not yet part of the programs' control flow graph. This means that in addition to the injected code a code pointer must also be adjusted in order to point to the start address of the malicious instructions, i.e. the used buffer.

As an example this code pointer modification is represented by *i1* in Figure 3.7. Once the code pointer is successfully adjusted, the control flow is redirected from benign node ① to malicious node Ⓐ. Whether the introduced malicious code redirects a pointer back to a benign control flow (*i2* in Figure 3.7) depends on the goals of the attacker and, thus, is optional.

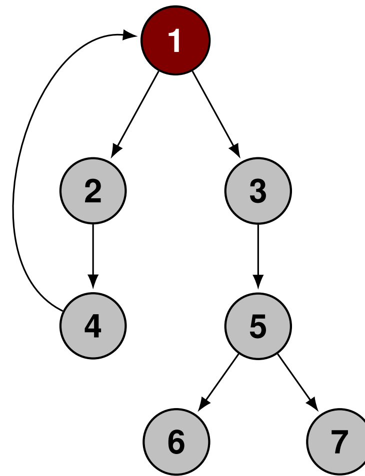

Figure 3.8 – CFG with maliciously modified Code that alters the Semantics of (1).

Code Replacement Code replacement is very similar to code injection. One could even argue that both are semantically equal and the differentiation is purely based on its presentation. But, there is one detail that can be used to distinguish between the two types. As explained, code injection always involves the modification of a pointer. This means that in order to trigger the execution of the injected code, a pointer must always be adjusted (actively or passively) to point to its start address. In contrast to this, code replacement does not rely on any pointer modification to be triggered. This is because the replaced instructions become a valid part of the control flow and, thus, are implicitly executed during the normal execution.

A code replacement example is depicted in Figure 3.8. In particular, instructions in node (1) are replaced by different instructions and, thus, the semantics of node (1) are persistently changed and executed implicitly. As explained, this modification does not rely on an initial adjustment of the IP.

Code Corruption Properties Both types of attacks are equally dangerous but differ in the key properties identified in the previous Section 3.1.2, i.e. (1) persistence and (2) stealthiness.

The main difference between both types is whether a pointer must be actively redirected or not and whether the code remains persistently inside the memory. The execution of injected code always relies on a pointer modification acting as a triggering mechanism to actively redirect the control flow to the malicious code start address. For instance, in many cases, such as the classic buffer overflow attack, the adjusted pointer is the return address located on the stack. If this is the case, the code injection can almost always be classified as a one-time attack. Once the return address has been adjusted, which ultimately redirects the IP, the malicious code is executed and the attack vector is finished[8]. If,

[8] This does not mean that the entire attack is finished.

for whatever reason, the injected code should be executed another time, the return address must be manipulated again which redirects the IP implicitly. However, if the manipulated pointer is not the return address, but a different pointer that eventually adjusts the IP, a classification of the attack must also consider in which memory region the pointer has been manipulated. Hence, for the remainder of this thesis, IP is used as a representative of any adjusted pointer that is eventually used by a branch instruction. This is because the ultimate result of any branch instruction can be reduced to the adjustment of the IP.

To conclude, a leveraged pointer influences the stealthiness and persistence of a code injection attack. For this reason, the stealthiness and persistence of the attack is influenced by the memory region used to inject the malicious code and the pointer that was manipulated to trigger the attack:

Code Injection 1: code injection and pointer manipulation in unpredictable short term data, such as the `stack`

Code Injection 2: code injection in unpredictable long term data, such as the `heap` and pointer manipulation in unpredictable short term data, such as `stack`

Code Injection 3: code injection and pointer manipulation in unpredictable long term data, such as the `heap`

Code Injection 4: code injection in predictable dynamic data and pointer manipulation in unpredictable short term data, such as `stack`

Code Injection 5: code injection in predictable static data, such as program text, and pointer manipulation in unpredictable long term data, such as the `heap`

Code Injection 6: code injection in predictable static and code pointer manipulation in predictable dynamic data, such as the `GOT` in user space or `syscall` table in kernel space

Please note, for the sake of completeness all possibilities have been listed to point out different properties to distinguish mainly between the persistence of the code injection. For example a code injection in a predictable segment combined with a code pointer manipulation on the stack is very unpractical, yet possible.

In contrast to code injection, code replacement becomes a constant part of the control flow. This means whenever the control flow is expected to execute the benign code, the malicious code is executed implicitly instead and without the need to actively modify any pointer. Code replacement usually tries to modify the control flow persistently during the entire lifetime of a program; therefore, it is classified as high. As expected, the in-memory persistence also has a major influence on the stealth property. Consequently, the

Table 3.3 – Stealthiness and Persistence Properties of Code Corruption Attacks. Stealthiness Classification: low (predictable static data), medium (predictable dynamic data), high (unpredictable dynamic data). Persistence Classification: very low (one time), low (until overwrite), medium (until termination), high (until reboot), very high (until reinstall).

Type	Stealthiness	Persistence	
		User space	Kernel space
Code Injection 1	high	very low	very low
Code Injection 2	high	very low	very low
Code Injection 3	high	low	low
Code Injection 4	medium	very low	very low
Code Injection 5	low	low	low
Code Injection 6	low	medium	high
Code Replacement 1	low	medium	high
Code Replacement 2	medium	medium	high
Code Replacement 3	high	low	low

stealthiness of code replacement depends on whether the manipulation was applied in predictable or unpredictable data; thus, the stealthiness classification is based on these two variants:

Code Replacement 1: manipulation in predictable static data, such as program text

Code Replacement 2: manipulation in predictable dynamic data, other than program text

Code Replacement 3: manipulation in unpredictable dynamic data, such as the `heap`

The results of the stealth and persistence classification of Code Corruption Attacks are illustrated in Table 3.3.

Code Pointer Manipulation

Another type of manipulation attack is a Code Pointer Manipulation (CPM) shown in Figure 3.9. Here, a pointer address in node ① is replaced that disables the two paths to the intended nodes ② and ③ and redirects the control flow directly to the node ⑤, depicted by *m1*. This means that whenever the control flow reaches node ① it is always implicitly redirected to node ⑤ by following the adjusted branch target pointer address.

But CPM is not limited to nodes that are part of the CFG. They can also redirect the control flow to any node that is available in the program's entire address space. For instance, in Linux every compiled program depends on at least two shared libraries, i.e. `libc` and `libld`. Nevertheless, the program uses only a subset of all defined shared library functions. As an example, Figure 3.10 depicts two unconnected nodes, i.e. (L1)

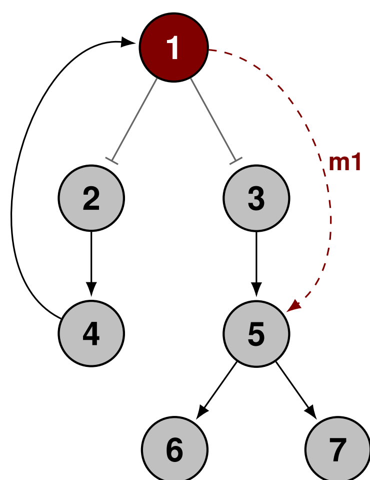

Figure 3.9 – CFG with maliciously modified Code Pointer that alters the Control Flow (from (1) → (2) and disables (1) → (3) to (1) → (5)).

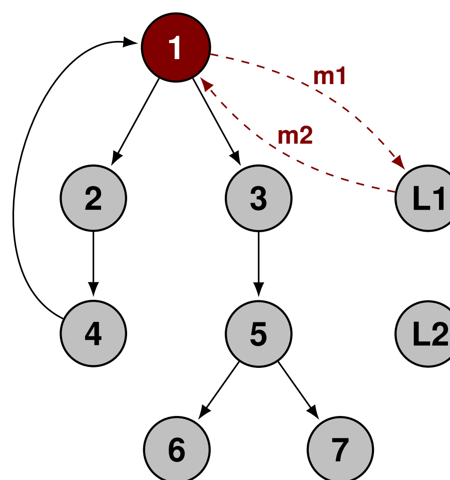

Figure 3.10 – CFG with maliciously modified Code Pointer that alters the Control Flow (from (1) → (L1) and (L1) → (1)).

and (L2), that represent such unreferenced functions. In particular, the control flow is first redirected from (1) to (L1), and then, back from (L1) to (1)[9].

Code Pointer Manipulation Properties In contrast to the code corruption attacks, CPM is quite limited in its actions because it only allows to manipulate pointer addresses of branch instructions; thus, it can only redirect the IP so that a different control flow is selected. However, considering that redirections to arbitrary functions in the address space can be used shows that CPM can be very powerful under certain conditions.

CPM can be applied in code and predictable data, such as the `.text` segment or the GOT, as well as in unpredictable short-time data, such as the `.stack` segment and unpredictable long-time data, such as the `.heap` segment.

For this reason, a distinction between three variants of manipulations are necessary:

CPM 1: manipulation in code, such as program text, or predictable data, such as the `GOT` in user space or `syscall` table in kernel space

CPM 2: manipulation in unpredictable short term data, such as the `stack`

CPM 3: manipulation in unpredictable long term data, such as the `heap`

Furthermore, the properties are classified in terms of stealthiness and persistence as depicted in Table 3.4.

Please note, manipulation of code pointers in predictable data is implicitly called and persistent during the entire lifetime of the program. Moreover, it does not rely on an

[9] For instance, one could assume that (L1) is a function that alters certain access privileges

Table 3.4 – Properties of Code Pointer Manipulation Attacks. Stealthiness Classification: low (predictable static data), medium (predictable dynamic data), high (unpredictable dynamic data). Persistence Classification: very low (one time), low (until overwrite), medium (until termination), high (until reboot), very high (until reinstall).

Type	Stealthiness	Persistence	
		User space	Kernel space
CPM 1	medium	medium	high
CPM 2	very high	very low	very low
CPM 3	very high	low	low

initial IP adjustment to trigger the attack. In contrast to this, CPM in unpredictable data is almost always used in a separate attack type. This is further discussed in the following Section 3.2.3.

Code Reuse

As mentioned, code reuse attacks are a constrained variant of the aforementioned CPM attacks and are based on the idea that the control flow of a program is modified by just altering the IP. Hence, one major difference between code reuse attacks and the described CPM is that code reuse attacks never modify any instructions or pointer addresses in code or predictable data. Instead, they only inject and manipulate so called control-data, i.e. the control structures that explicitly manage the edges of a CFG[10].

Return-to-Libc The basic idea behind code reuse attacks is to utilize existing code of the attacked program. Thus, equally to the example given in Figure 3.10, a code reuse attack can also utilize redirections to arbitrary functions in the address space of the program. In fact, the first variant of code reuse was indeed of this kind and is known as return-to-libc (`ret2libc`) initially demonstrated by Solar Designer [40].

Return Oriented Programming Today, CPM plays a major role in exploit and malware development in the form of Return Oriented Programming (ROP) [41, 42]. ROP is a generalization of the `ret2libc` attack, but it does not utilize arbitrary functions in the address space. Instead, ROP uses sequences of instructions, called gadgets; these gadgets can be chained together by intelligently arranging control-data in unpredictable regions, such as the `stack` or `heap`. Under the assumption that enough gadgets are available, this provides a Turing complete programming language [41, 43].

This behavior is depicted exemplary in Figure 3.11. As shown, instructions in the nodes ④, ⑤ and ⑥ are chained together to one gadget. The order of their invocation is

[10] Please note: Although the GOT can be seen as a control-data only structure, manipulation of GOT pointers are not interpreted as a classical code reuse Attack.

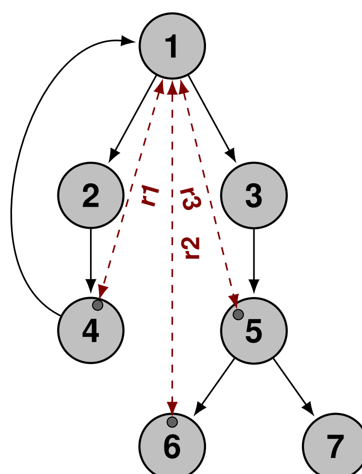

Figure 3.11 – CFG calling multiple chained Instruction Sequences (Gadgets) inside Nodes.

① → ④, ① → ⑥ and ① → ⑤. Because the injected pointers are only replaced inside control-data, the node itself does not become malicious.

Classic ROP attacks rely on control-data managed on the `stack`. This control data is the address to which the return instruction refers. Yet, the generic principle of ROP about borrowing code chunks can also utilize other control-data. For instance, Jump Oriented Programming (JOP) facilitates jump instructions and, thus, can be located anywhere in writable memory [44]. Sigretrun Oriented Programming (SROP) utilized signal handling control structures [45] but is limited to control-data in the `stack`. The newest class of CPM attacks is Counterfeit Object Oriented Programming (COOP), described by Schuster et al. [46]. This attack is the most advanced attack in the family of CPM and utilizes indirect branches and object-oriented C++ semantics to implement a code reuse attack by modifying code pointers to virtual C++ functions managed in the Heap.

Control Flow Bending

Non-control data attacks, as the sole representative attack in CFB, are another variant used to subvert the control flow of a program. In contrast to the aforementioned code reuse, Non-control Data attacks do only manipulate control structures that are used in decision-making logic. For this reason, non-control data attacks can only manipulate edges in the CFG that are directly dependent on the manipulated data. Next, a more precise example of a non-control data attacks, limited to Boolean variable modification, is presented.

Particular attacks that implement non-control data attacks-alike behavior, either partially or fully, are further discussed in 3.3.2.

Figure 3.12 depicts an example of a non-control data attack. In this particular case, node ① must make a decision whether path ① → ② or ① → ③ is selected to continue. This decision is based on a conditional branch inside the node and depends on `data`.

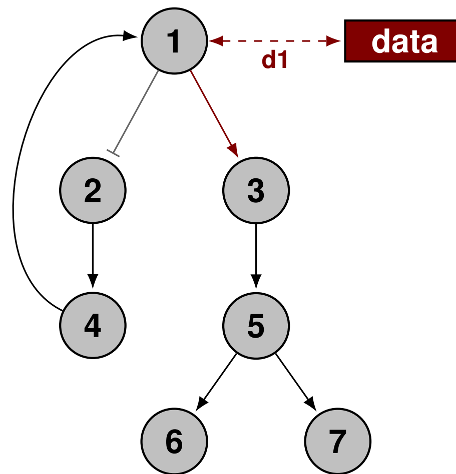

Figure 3.12 – CFG including malicious Data Modifications to disable a valid Path (① – ②) and always force a fixed Transition ① → ② .

Assume that `data` is a Boolean value that is either `True` or `False`. In the case it is `True`, path ① → ② is selected, and if it is `False` path ① → ② is chosen. A classic CFM attack would target the pointer addresses and manipulate the addresses in a way that a particular path is selected, cf. Figure 3.9. This enables the possibility to redirect the control flow to arbitrary nodes. In the given example, however, the Boolean value was manipulated by the operation `d1` and set to `False`. As a result, the path ① → ③ is always selected, as long as the related value does not change. Precisely this behavior of indirect manipulation through data variables or pointers is meant by the term bending.

Non-control Data Attack Properties Non-control data attacks are very constrained and complex. Yet, under specific circumstances, they can become a major threat to the victim. A classic example would be to manipulate data that controls whether a user has certain privileges in a program, for instance whether she is an administrator or not. As expected, there are countless possibilities how non-control data attacks may cause serious damage to a system. And, in addition to that, in many cases these crucial data can be manipulated not only during runtime of a program, but also offline, for instance based on configuration files. For this reason, a distinction between different persistence levels of Non-control Data attacks are as follows:

Non-control Data 1: manipulation in short term runtime data, such as the `stack`

Non-control Data 2: manipulation in long term runtime data, such as the `heap`

Non-control Data 3: manipulation in persistent runtime data, such as global variables or data-structures

Non-control Data 4: manipulation in persistent data, such as volatile configuration files

Non-control Data 5: manipulation in resident data, such as configuration files that remain after a reboot

Moreover, non-control data attacks are exceptionally stealthy, because they are often applied in unpredictable data. Furthermore, without the semantic meaning in the program's current context, it is impossible to make a decision whether a value is in a good or bad state. For instance, if a program controls access to a resource based on a single value, such as a Boolean variable `is_administrator`, an external observer cannot decide whether a particular user is indeed an administrator or not by solely looking at the Boolean value. In order to make this decision, the current context, i.e. the user, her general access permissions and the Boolean runtime value of the variable `is_administrator`, must be known. In other words, there is no effective mechanism for an external observer to detect runtime-only attacks without a contextual understanding.

Still, if a runtime modification is carried out in data that is well-known or can be derived from other information, it can be visible to an external observer and eventually be detected, cf. Section 5.2.4.

Table 3.5 contains a classification of stealthiness and persistence, according to the aforementioned attacks.

Table 3.5 – Stealthiness and Persistence Properties of Non-control Data Attacks. Stealthiness Classification: low (predictable static data), medium (predictable dynamic data), high (unpredictable dynamic data). Persistence Classification: very low (one time), low (until overwrite), medium (until termination), high (until reboot), very high (until reinstall).

Type	Stealthiness	Persistence	
		User space	Kernel space
Non-control Data 1	very high	very low	very low
Non-control Data 2	very high	low	low
Non-control Data 3	very high	medium	high
Non-control Data 4	high	medium	high
Non-control Data 5	high	very high	very high

Hybrid Attacks

Over the last decade many countermeasures have been designed that effectively prevent or restrict certain described attack variants. These countermeasures will be discussed in more detail in Section 3.2.4. This Section describes the general concept of hybrid attacks that facilitate the use of different attack variants for control flow manipulation. A more practical example of a concrete hybrid attack will be provided in Section 3.3.4.

Hybrid attacks utilize multiple attacks or variants in order to manipulate the control flow of a program. In Figure 3.2.3,a code reuse is combined with a code-injection attack which is a very common combination. According to Rains, almost all exploits discovered

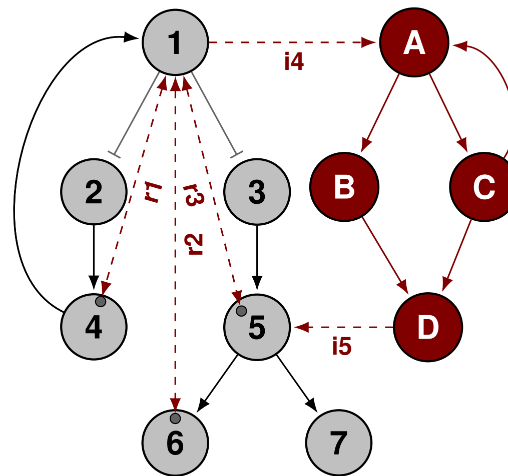

Figure 3.13 – Hybrid CFM combining a Code Reuse, Code Injection and a CPM Attack. First, a code reuse attack is applied `r1`, `r2`, `r3`. Second, malicious code is injected Ⓐ, Ⓑ, Ⓒ and Ⓓ. And third, code pointers are adjusted so that the malicious code gets executed `i4`, `i5`.

in 2014 and 2015 used ROP or a different code reuse variant [47]. The main idea behind the attack is to: (1) Apply a code reuse attack to disable countermeasures, in particular disable Data Execution Prevention (DEP), (2) Inject code into the altered program memory, and (3) Redirect the control flow accordingly so that the injected code is executed. This hybrid attack is illustrated in Figure 3.13. First, the code reuse attack is applied by using CPM on unpredictable control-data. In particular, it is assumed that the attack enables the executable permission for a memory region, for instance the `heap`, by calling relevant gadgets in `r1`, `r2` and `r3`. Afterwards, malicious code is injected into the altered memory region, represented by the node Ⓐ, Ⓑ, Ⓒ and Ⓓ. In a final step, the pointer addresses are adjusted accordingly; this redirects the control flow of ① to always execute the injected malicious code persistently during the runtime of the program ① → Ⓐ represented by `i4`. Optionally, the assumption can be made that the malicious code redirects the control flow back to Ⓓ → ⑤ at the end of its full execution, as indicated by `i5`.

Hybrid Attack Properties The properties of stealthiness and persistence depend on the individual attacks variants that are combined. In many cases, the multi-Step attacks consists of an initialization step which is requirement in order to enable a successful exploitation and arbitrary subsequent steps, i.e. the real attack vectors used to reach defined (long term) goals of the attack. Obviously, the initialization step is preferably conducted as a highly stealthy attack and the persistence only plays a minor role. This means that the detection of the initial step is usually considered very hard since only unpredictable control or non-control data is involved for a very short period. This renders the successful detection of such initialization attacks as extremely complicated.

For these reasons, the classification of hybrid attacks is based on the stealthiness and

Table 3.6 – Stealthiness and Persistence Properties of Hybrid Attack 1 and 2. Stealthiness Classification: low (predictable static data), medium (predictable dynamic data), high (unpredictable dynamic data). Persistence Classification: very low (one time), low (until overwrite), medium (until termination), high (until reboot), very high (until reinstall).

Step	Hybrid Attack 1		Hybrid Attack 2	
	Stealthiness	**Persistence**	**Stealthiness**	**Persistence**
Initialization	very high	very low	very high	very low
Deployment	high	**low**	**medium**	medium-high
Activation	**medium**	high	very high	**low**

persistence of subsequent deployment and activation steps, and thus on the attack vectors that attempt to reach the adversary's ultimate goal. Accordingly, the earlier presented classification can be used to determine the stealthiness and persistence of a hybrid attack. Therefore, it is not possible to provide a general classification of hybrid attack properties.

For example, Hybrid Attack 1 uses three individual attack steps, as illustrated in Table 3.6:

Step 1 Initialization: Code reuse disabling DEP in unpredictable short term data

Step 2 Deployment: Code injection in unpredictable long term data, such as `heap`

Step 3 Activation: CPM in predictable code, such as program text

Accordingly, the stealthiness property is classified as medium, since detection of a single step is already sufficient. Although the persistence for the activation step is considered high, the overall persistence classification of the attack is low for both the user space and the kernel space. As soon as the injected code is replaced or freed, the attack is no longer active. More specifically, in this case, it is very likely that the attacked program will be terminated immediately after branching into the memory that has now been replaced or released.

In contrast to this, for an additional Hybrid Attack 2, the individual attack type variants have been modified, see Table 3.6:

Step 1 Initialization: Code reuse disabling DEP in unpredictable short term data

Step 2 Deployment: Code injection introducing malicious code in predictable data, such as program text segment

Step 3 Activation: CPM in long term unpredictable data, such as the `heap`

In this case, the stealthiness property is also classified as a medium, but now the classification is based on a potential detection of the code injected during the deployment step. The classification of the persistence property lies in a range between low (step 3) and high

(step 2). However, in this case again, the overall persistence classification is determined by the property that defines the shortest duration; it is hence classified as low due to the activation step 3.

Note, the persistence classification for the Deployment step 2 in this case, depends on the component that was infected. A code injection into a user space process' program text segment is classified as medium (until termination), but for the kernel it is classified as high (until reboot). In addition, a classification of persistence based on dynamic memory contents must be considered with caution, since in this case the best case is assumed. It is impossible to determine exactly whether and when this memory will be overwritten or freed. For this reason, it is also not unlikely that the persistence classification tends towards more persistent classifications, up to the point where the maximum persistence level of the respective attack has been reached.

In conclusion, hybrid attacks must always be classified based on their individual attacks, the attacked component and by taking all involved steps into consideration. This means that a hybrid attack which implies a very high stealthiness or persistence property in a one step, must be reevaluated when the stealthiness or persistence properties are adjusted in a different step. For stealthiness this means that the least stealthy property is always significant. Similarly, the persistence classification also depends on the least persistent property of the Deployment or Activation step, but is influenced by the attacked system component and subject to other interactions in the case of dynamic memory parts.

3.2.4 Countermeasures

Today, there are many countermeasures addressing all different kinds of attacks that can compromise the system during runtime. The very first kind of these attacks is the well-known classic buffer overflow exploit as introduced by Aleph One [9]. Attacks and countermeasures are in a constant arms race, and the classic technique of buffer exploitation is the root from where both offensive and defensive measures have evolved.

At the time the buffer overflow exploit and its different variations appeared, the targeted memory region, i.e. the `stack`, was entirely unprotected. In other words, the `stack` was considered as an ordinary memory region where reading, writing and code execution was not limited. In addition, compiled programs behaved statically and relied on fixed addresses in almost all cases. Once the program and its dependent libraries had been compiled and linked, the memory layout was fixated. This means that every time a program was started, the same addresses for branches were used. For this reason, the buffer overflow attack was very simple to execute. Once a buffer overflow vulnerability was identified, the buffer inside the `stack` was simply overwritten with shellcode and random data until the return address was reached. The return address was then overwritten with the start address of the shell-code. Once the function finished its execution, the manipulated return address redirected the `IP` to point and jump to the beginning of the

shell-code, which was then executed immediately after the IP was set.

The buffer overflow exploit represents the most trivial code injection attack, as described in Section 3.2.3. Different variants that inject malicious code or adjust pointers in different memory regions are available. However, code injection, in its pure form, always relies on executable permissions to succeed. Today, these simple buffer overflow attacks are no longer applicable because many countermeasures exists that prevent successful exploitation.

In the following, the most common and widely adopted protection mechanisms are presented.

Countermeasure: Data Execution Prevention

The first defensive countermeasure that was designed to prevent the successful exploitation of buffer overflow based exploits, or more broadly code injection attacks, was to limit the permissions of memory segments. This concept is known as Data Execution Prevention (DEP). In particular, memory regions that do not contain executable instructions can be marked as non-executable. As a result, malicious code inserted in a buffer marked as non-executable is not executed and a runtime exception is generated. Although this countermeasure does not prevent the initial injection, it is very effective and prevents the successful exploitation of code injection. For this reason, the concept was adopted by hardware manufacturers who have implemented this feature under different names. For the X86 architecture, AMD added the so called *NX* bit, which stands for *No-eXecute*, in their AMD64 architecture's Page Table Entrys (PTEs). This makes it possible to control the execution rights with the granularity of a single memory page. Intel adopted the concept later and called it *XD* bit, which stands for *eXecute Disable* and ARM included the feature in ARMv6 and called it *execute-never*-bit[11]. Apart from the different names, the technical concept remains exactly the same. Accordingly, if DEP is supported by the hardware, the OS can set the corresponding bit in the PTE and, thus, once code is executed in a non-executable region, the program is terminated and an exception is thrown.

It has to be noted, that code injection attacks are still a major threat and must be considered as very serious. If malicious code can be injected into a memory region that is or can be marked executable, the attack is still feasible and extremely dangerous. As an example, many Just in Time (JIT) compilers used by interpreted languages, the Java Virtual Machine (JVM) and many legacy systems or software rely on executable memory segments that can still be altered during runtime. Thus, they can also be used by code injection attacks.

Regarding the exploitation of DEP, it is possible to utilize system calls, such as `mprotect` on Linux and `VirtualProtect` on Windows, in order to disable the initially

[11] In addition to that, all major CPU manufacturers at that time, i.e. SPARC, PowerPC, Alpha, PA-RISC, added support for the NX bit. So the function was and still is available on almost all CPUs.

assigned access permissions. For instance, Code Reuse Attacks can be used to invoke the mentioned system calls and effectively disable the protection provided by DEP. For this reason, GrSecurity *PaX* ($W \oplus X$), a Linux kernel security patch-set, implements certain policies to limit the exploitation of the `mprotect` system call. This means that it is not possible to map or change page permissions that are writeable and executable at the same time when PaX is activated. Consequently, this measure further complicates the code injection exploitation techniques, since it requires multiple calls to the system call instead. In particular, a page would need to be first set to `rw-`. Second, the code injection needs to take place. And lastly, the page permissions must be reset to `r-x`, in order to allow the execution of the injected code. Furthermore, and in order to obtain the full effectiveness of the PaX security functionality, all the policies have to be enforced on system wide scale. As it turns out, legacy software often relies on page permissions that violate PaX policies by default. Examples for those default violations are e.g. self-modifying code, any software utilizing Virtual Machines (e.g. Java) and programming language based on interpreters. Although PaX increases the overall system security, aforementioned constraints limit its applicability hugely. Consequently, and mainly due to breaking the compatibility with many legacy systems by providing only a minor improvement, PaX is not widely adopted.

Countermeasure: Buffer Overflow Protection

The general principle used by buffer overflows is to write data outside of an intended fixed-size buffer and, thus, overwrite values that are located outside the boundaries of the buffer, for instance, the return address on the `stack`. Buffer overflow protection now introduces a well-known value that is located directly before the return address. In cases where a buffer overflow occurs, which never happens during a valid execution of the program, the overwriting of the buffer can be detected by introduced verification code that constantly checks the well-known value. If a value is no longer equal to the well-known value, a protection mechanism can react to this violation. For instance, throw an Exception and terminate the program.

Usually, the code for Buffer Overflow Protection is automatically generated during compilation. If enabled, the compiler automatically adds the well-known value before the return address and adds verification code that checks the value and employs the protection mechanism. This feature is also often referred to as *stack smashing protection* or *stack canaries* and is available in all major compilers.

Countermeasure: Address Space Layout Randomization

Code injection attacks often rely on resolved memory addresses of targeted data-structure in memory. Injecting malicious code into the process memory is only the initial part of an injection attack. Thus, in order to trigger the execution of the injected code, the control

flow must also be redirected accordingly. This means, for example, that the return address must be modified so that it redirects the IP to the beginning of the injected code. Without protection, these memory addresses can easily be calculated based on known sizes of the involved data-structures, or they can be guessed by simply trying different offsets and base-address combination. This means identification is no serious hurdle for a successful exploitation.

Address Space Layout Randomization (ASLR) is a first line defense mechanism that randomizes these initial start addresses for all loaded code and data segments, if supported by the corresponding program; the executable program or shared library must be compiled as PIC as described in Section 2.3.2. Thus, if available, ASLR randomizes all loading addresses for every execution. This means loading addresses are different for every single execution. As a result, all code and data segments are distributed all over the Virtual Address Space and the loading addresses can no longer be easily guessed.

If ASLR is enabled, an attacker can no longer simply calculate correct memory addresses. The only opportunities left to identify correct memory addresses are: brute-force all possible pointer locations by trying all possible variations; exploiting secondary software vulnerabilities that leak the required addresses or launch side-channel attacks [48] to find them. ASLR is enabled in almost all modern OS and a very important building block for modern defense strategies against exploitation.

Please note: As explained earlier, the IP must always point to executable code or else the program is instantly terminated. Thus, an attacker cannot sequentially try all possibilities during the brute-force process. This is because once the program has been terminated and restarted the addresses will be reassigned differently. In other words this means the brute-force approach is very inefficient.

Countermeasure: Control Flow Integrity

CFI is a mechanism to protect running software against malicious redirection of its execution flow, which is used by many modern malware during an attack. The malicious redirection of the execution flow means that an intended target of a branch instruction, for instance a `ret` or `jmp` instruction, is controlled and eventually modified by an adversary. Since the adversary fully controls the corresponding target of the branch instruction, she can redirect the current execution flow to any available memory address that points to executable instructions. This enables the adversary to adjust the defined path in the CFG and, thus, enables the execution of arbitrary adversary-controlled instructions instead. As a result, the modifications of the branch targets are the building-blocks of code reuse Attacks that utilize these underlying mechanisms to alter the execution flow in a structured manner.

The goal of CFI is to prevent the execution of maliciously altered execution flows. This can be achieved either by preventing the modified instructions to be executed or by

preventing the modification of the branch target in the first place, cf. [49].

CFI countermeasures have been extensively researched in the last years and many solutions have been proposed. Regarding code execution prevention, the solutions can be roughly classified in solutions that protect forward-edges, i.e. indirect branches using architecture-specific `jmp`-instructions that rely on dynamic memory contents, and backward-edges, i.e. the architecture-specific `ret` instructions. Direct branches are not vulnerable to modifications since they do not rely on dynamic target addresses, and, hence, remain in read-only portions of the code. They cannot be controlled directly. CFI relies on an either coarse or fine-grained analysis of the CFG that specifies valid branch targets either derived directly from source code or program text. As expected, source code-based CFG analysis is always superior in terms of preciseness, but may not always be available. Based on the results from the CFG analysis, the forward-edge protection is usually based on the introduction of additional validation code. Before a branch instruction is executed, the branch target is compared to valid CFG targets by this code and only executed if the branch target is considered valid. Depending on the accuracy of the CFG analysis, which ideally also considers language-specific semantics, such as virtual function calls in C++, the solutions themselves vary hugely regarding their preciseness.

Forward-edge protection is typically implemented by a compiler. Implementations are available in recent compilers such as *gcc* and *Clang* [50, 51], or Microsoft C/C++ *Stack Guard* [52]. However, all these solutions implement a coarse CFI policy only. This means that they do not provide a strict and precise target branch validation. Consequently, they do not fully protect against code reuse attacks, they only make them harder to execute during exploitation. A detailed overview is presented in a recent work of Burow [53].

The backward-edge protection does usually not rely on an initial analysis, since the backward-edge is defined by the functionality in the Application Binary Interface (ABI)-specific function epilogues. Since the defined standard behavior of a returning function is always to return to its caller, validation code can always check whether the return address points back to the caller or not. As it turns out, this context-sensitive functionality can be implemented very efficiently in hardware as a so called *Shadow Stack* [54]. Each caller issuing a function call is pushed onto the Shadow Stack and verified by hardware just before the `ret` instruction is executed. If the `ret`-target is valid, the function returns and the execution flow continues. If it does not match, an exception is generated and the program terminated. In conclusion, enforcing stack-based integrity with a context-sensitive mechanism, such as a Shadow Stack, guarantees backward-edge protection for returning function calls in the program's scope. However, it must be noted that Shadow Stacks, whether implemented in hardware or not, can be circumvented in different cases, as illustrated by Conti in [55]. Alternative hardware-based solutions are also available, a very recent overview is provided by de Clerk [56].

In conclusion, CFI is a very strong countermeasure that limits the adversary's capa-

bilities to compromise software during runtime. The quality of CFI policy depends on its correct implementation and, most significantly, on the preciseness of the CFG analysis of identified branch targets. However, CFI does not prevent the adversary from causing memory corruptions and, thus, may allow a constant redirection to arbitrary branch targets that were falsely identified as valid. This is because the CFG construction is only an imperfect approximation of the runtime CFG. An hardware-implemented Shadow Stack provides strong and precise protection for returning function calls, but can also be bypassed in specific cases.

In this work, CFI is considered to be a future protection technology. Even though CFI is partly available in recent compilers, the level of protection it provides relies heavily on the concrete implementation. As of today, there is no complete implementation of CFI available; most of the mentioned compiler-addons do only implement a subset of particular CFI schemes. Please note that the DRIVE approach presented in this thesis is not designed to implement CFI-like analysis and protection, this issue will be further discussed in Section 4.3.3.

3.2.5 Summary

CFM and CFB attacks represent the most significant threats to system security, since they enable the adversary to modify or bend the control flow and, thus, allow the execution of adversary defined arbitrary functionality in the context of an attacked software component. Once a software component is compromised and controlled by an attacker, the integrity of the attacked portions of the system can no longer be assured. The major difference between the described variations, i.e. CFM and CFB, is not defined based on their capabilities. Ultimately, and through skillful combination of hybrid-attacks, all described techniques are similarly effective and dangerous. The main difference is thus determined by the complexity of the concrete attack, and further by its stealthiness and persistence. While the stealthiness and persistence of an attack mainly depend on the attacked specific data-structures inside a program's memory, its complexity is mainly defined by the available countermeasures that must be eluded in order to reach the attacker's goal. In comparison to the described Classic Buffer Overflow attack, today's systems are far more resilient, even in their default configuration, than they were a decade ago. However, the complexity of the individual attacks has also increased during this time, since for every countermeasure a new offensive technique has evolved to circumvent it. Still, achieving a certain attack goal may be too complex, or, in other words too expensive, to be realized by only one technique, such as code reuse or non-control data. Thus, hybrid attacks are designed to disable countermeasures by using highly sophisticated techniques initially. Directly afterwards, hybrid attacks then eventually fall back to simpler attack techniques, such as code injection, to realize the adversary's goal regarding functionality, persistence and stealthiness requirements.

To conclude, every countermeasure increases the complexity of an attack and, thus, forces the adversary to circumvent countermeasures in certain ways. No known countermeasure exists that protects a system from all possible techniques, they can all be circumvented. Forcing the adversary to utilize different techniques to achieve her goals increases the visibility of the attack and eventually a successful detection or prevention of it.

The next Section 3.3 contains more details on the application of hybrid attacks and defines a more precise Threat and Attack Model based on the results from the previous two Sections 3.1 and 3.2.

3.3 Threat and Attack Model

This section presents the Threat and Attack Model related to the conceptual work of this thesis. First, the potential threats based on defined attack goals are presented. These identify the adversary's long term goals and describe potential reasons for the exploitation of a system. Second, available attack patterns and procedures are briefly identified and example attacks are presented. Third, the so called concept of Multi-Step Hybrid Attacks is presented that relies on the application of multiple composable steps in order to successfully exploit a vulnerability and compromise a system. This is a reasonable assumption, because systems today employ different standard countermeasures that render a single-step exploitation to reach the defined attack goals as very unlikely. In particular, three steps are introduced and a model is created that allows the combination of these steps to conduct the Multi-Step Hybrid Attacks. Traditional threat modeling, like STRIDE [57, 58], is unfit for the modeling in this work because available threat models consider security characteristics, like for instance Integrity, Confidentiality, Authenticity, Information disclosure, Privilege Escalation, etc., on the application layer or communication channels. However, these characteristics are not applicable or meaningful in this case, although they could classify the long-term objectives, which is not the goal of this analysis. For this reason, this thesis defines its own model to describe potential threats and attacks, based on the previously identified methods and properties. Lastly, the section finishes with the definition of concrete attacks that are used in the following chapters as a reference to apply a security analysis for the concept and implementation of DRIVE.

3.3.1 Attack Goals

The adversary's initial goal is the implantation and execution of arbitrary code in volatile memory for further exploitation. Specifically, as depicted in Figure 3.14, the adversary is assumed to launch an attack with particular long term goals such as:

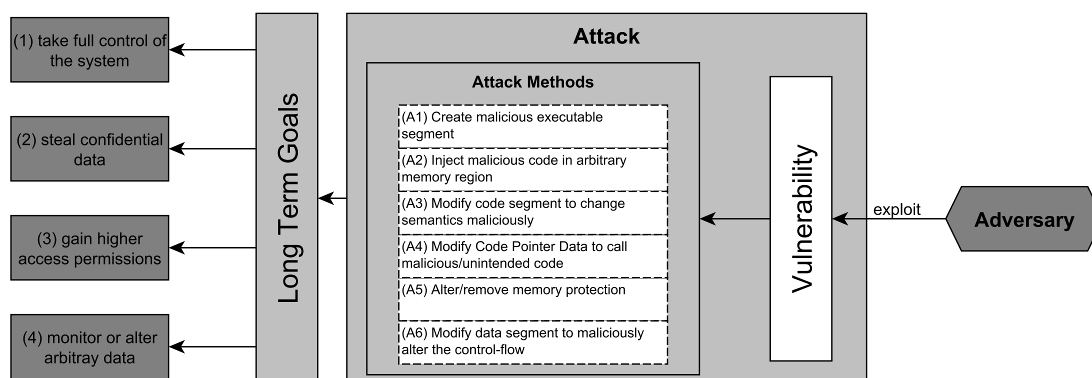

Figure 3.14 – Threat Model: Long Term Goals and Attack Methods in a typical Attack Scenario.

(1) Take full control of a system or network (e.g. implant rootkit)

(2) Steal confidential data (e.g. cryptographic keys or passwords)

(3) Gain higher access permissions (to circumvent access control mechanisms)

(4) Monitor or alter arbitrary data (e.g. data of a program, network traffic, routing tables, etc.)

This means the target of interest is not the initial attack; instead, long term system modification and monitoring are the primary goals of the adversary. Moreover, it is assumed that the adversary wants to remain hidden in order to carry out malicious actions for as long as possible. Consequently, this means the adversary is assumed to rely on sophisticated techniques that allow a successful disguise of the attack from early detection through well-known defensive security mechanisms. In general, the adversary wants to reach high stealthiness and persistence; however, as explained in Section 3.1.2 both properties influence one another.

In addition to that, it is assumed that the adversary's long term goals can be realized by an attack from the malware type-1 or type-2 category, c.f 3.1. In particular, it is assumed that attacks are implanted in code, predictable or unpredictable data. This is indeed a reasonable assumption, since these classes of attacks provide a high stealthiness level and no effective protection technology for detection exists at this time. Moreover, the persistence property that can be accomplished in these categories is also relatively high as the attacks enable to take over processes until they are restarted, LKMs until they are reloaded or the kernel until the system is rebooted.

Still, if the attack is carried out in dynamic data portions, the persistence levels depend on the target of the corrupted memory part. This means that any attack that is conducted in short-term dynamic data does not exceed short persistence levels, because the data is only valid for a short time. Similarly, any attack conducted only in long-term dynamic data does not exceed a medium persistence level since it may be overwritten or freed during normal operations such as intended memory allocation, memory writing or garbage collection processes.

3.3.2 Attack Methods and Procedures

In general, the described attack methods, or in short referred to as attacks, are utilized to maliciously modify arbitrary predictable or unpredictable regions inside the system memory in order to implant a desired malicious behavior consistently and over a long period. It is assumed that the adversary may utilize the following different attacks, shown in Figure 3.14:

A1 Create new executable segment, e.g. load new (`mmap`) or map existing code (`dlopen`)

A2 Inject malicious code in arbitrary memory region, e.g. inject code into .text segment's padding space

A3 Modify code segments to change semantics maliciously, e.g. replace instructions or code pointers in .text segment

A4 Modify Code Pointer Data to call malicious/unintended code, e.g. modify memory jump addresses in the GOT (`.got`)

A5 Alter/remove memory protection, e.g. disable or circumvent DEP mechanisms [59, 60] by utilizing the `mprotect()` system call

A6 Modify data segments to maliciously alter the control-flow, e.g. change configuration option to fixate a particular control-structure path

In addition to that, it is assumed that the adversary is able to carry out multi-step hybrid attacks that utilize a composition of related attack techniques as shown in 3.15. Available attack techniques, according to the definition from Section 3.2, are as follows:

- Code corruption attack
  - Code injection
  - Code replacement
- Code pointer manipulation attack
- Code reuse attack
- Non-control data attack

The following Section 3.3.3 presents different examples of attacks which are described in detail by the different steps during exploitation.

3.3.3 Multi-Step Hybrid Attacks

According to the presumed attack techniques, a successful exploitation requires almost always multiple steps to actually perform a persistent attack. For this reason, attacks

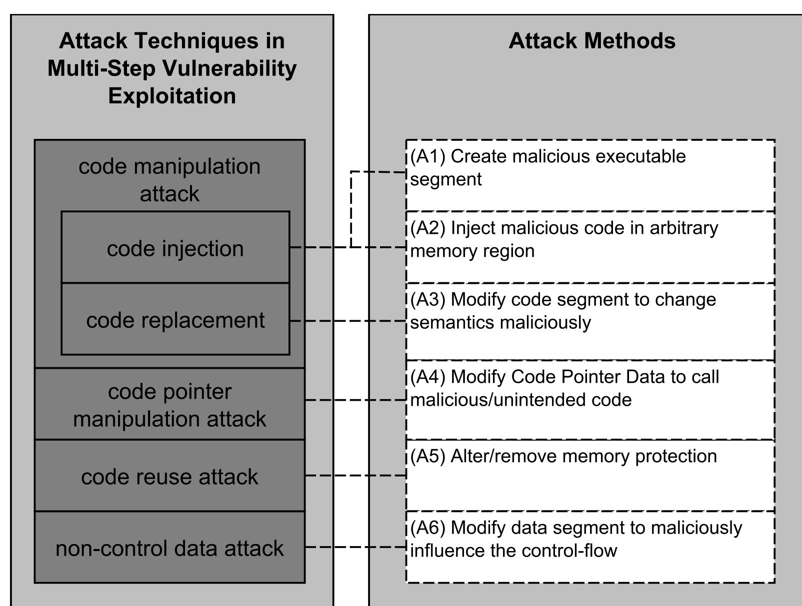

Figure 3.15 – Attack Techniques and related Attack Methods for Vulnerability Exploitation.

are divided into three kinds of steps, which are also illustrated in Figure 3.16: (Step 1) Initialization, (Step 2) Deployment, and (Step 3) Activation.

For each step there are typical attack techniques that are usually used and depending on the concrete attack different variations of these steps are applied. More precisely this means that a particular long term goal may be accomplished by an attack technique during the deployment step. *(A1) Create malicious executable segment* is for instance a traditional code injection attack. But, in order to be able to inject the code in a targeted memory area, an Initialization Step may be necessary. This means that in order to enable the code injection into a certain memory area, an initialization step must be used for resolving the writable preconditions or the executable post-condition of the attack. Similarly, an Activation Step may be necessary to trigger the execution of the injected code during the program execution. As expected, the Activation Step may also require additional Initialization Steps to work. For these reasons, the application of multiple attack techniques in different steps, relies on: (1.) the selected attack technique to deploy the malicious behavior, (2.) suitable attack techniques for its activation and (3.) pre- or postconditions that require the application of one or multiple initialization steps.

For the individual steps, described hereinafter, it is assumed that the adversary has access to a known vulnerability and exploit enabling a successful attack on the system. This may be enabled by any vulnerability that allows memory access or code execution (e.g. buffer overflow, array over-indexing, or format string vulnerability), once exploited [9, 61, 62]. For instance, the adversary may utilize a successful code reuse attack (e.g. ROP, JOP or SROP), or facilitate any other kind of attack in order to inject or load arbitrary data into volatile memory and exploit it at arbitrary times [43, 44, 63, 64]. Next, the individual

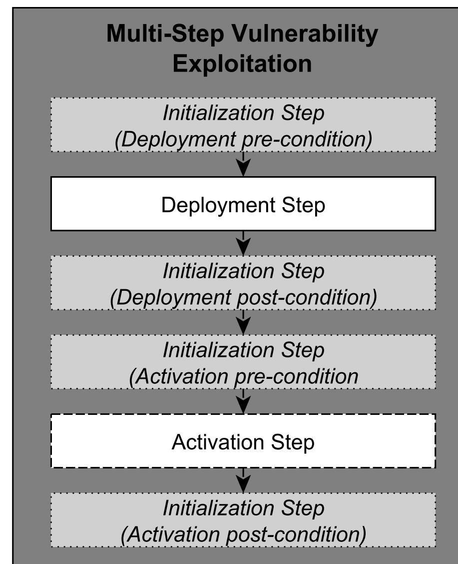

Figure 3.16 – Different Steps in Multi-Step Vulnerability Exploitation.

steps are presented and put into relation to typical attacks.

Initialization Step

In general, Initialization Steps are considered as optional, since they almost always fulfill pre- or postconditions of subsequent Deployment or Activation steps. However, the Deployment Step requires a fulfilled precondition in most of the attacks described in this thesis. For this reason, an attack is provided to alter memory protection mechanisms for arbitrary memory regions. This is shown in Figure 3.17 as an Initialization Step Code Pointer Manipulation Attack Pattern (CPMAP) 1 that effectively disables DEP protection for arbitrary memory pages or segments. Furthermore, the adversary must disable or circumvent defensive mechanisms such as ASLR or Stack Canaries [65], before she can successfully apply the CPMAP 1 attack. Therefore, it is assumed that the adversary already has knowledge of targeted memory addresses randomized by ASLR, for instance through reverse engineering or provided by an Information Disclosure Attack Pattern (IDAP) 1 as illustrated in Figure 3.18. Moreover, it is assumed that the adversary already circumvented canaries based protection by disclosure of the relevant canary value and considering it during exploitation. In other words, these preconditions for CPMAP 1 are already satisfied; for the purpose of simplicity, they will not be modeled in every related attack.

Deployment Step

As previously mentioned, the Deployment Step represents the core of the actual attack that tries to accomplish the long term goals of the adversary. In particular this means that the

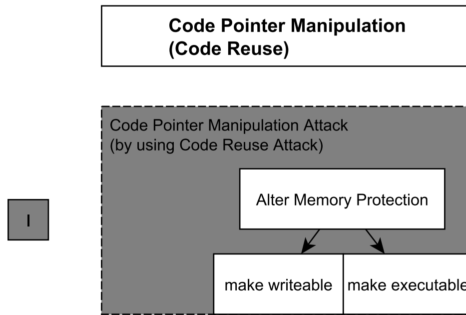

Figure 3.17 – CPMAP: typical Step 1 Code Pointer Manipulation Attack to alter Memory Permissions.

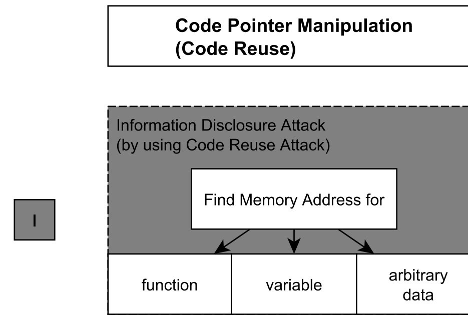

Figure 3.18 – IDAP 1: typical Step 1 Information Disclosure Attack to identify Memory Addresses of Functions, Variables or arbitrary Data.

adversary typically executes a Code Corruption Attack. As defined in Section 3.2.3, a distinction between general types is made: (T1) code injection attacks, i.e. adding executable code to the program's runtime memory; (T2) code replacement attack, i.e. alter/replace instructions inside the program's runtime memory; (T3) code pointer manipulation attacks, i.e. manipulate/modify code pointers; and (T4) non-control data attacks manipulating data variables or data pointers. Code injection attacks almost always require an activation step, because the injected code is not referenced inside the CFG. Code replacements modify existing instructions inside the CFG which means that the malicious instructions are automatically invoked during the program's natural execution, therefore usually no activation step is needed. Code pointer and non-control data manipulation attacks are very similar in this regard, because they are used to directly influence the CFG to always take a certain execution path. Therefore, they usually also do not need an Activation Step.

For a code injection attack, two different Code Injection Attack Patterns (CIAPs) are presented: (CIAP 1) a classical code injection that can inject code into mapped arbitrary predictable or unpredictable data segments, cf. Figure 3.19; and (CIAP 2) a Code Injection Attack that utilizes code reuse functionality to load previously unreferenced code into the program's runtime memory via different system calls, cf. 3.20. As an example, CIAP 1 can be used to inject code into unpredictable data, such as a buffer on the stack or heap, or to inject code into predictable data, such as the page padding area inside the `.text` segment. Similarly, CIAP 2 can be used to load code into the program's runtime memory by using the `mmap()` system call that maps an arbitrary file with executable code or by using `dlopen` to load an unreferenced shared library.

In addition to that, a Code Replacement Attack Pattern (RCAP) was defined as an example for a code replacement attack, cf. Figure 3.21. RCAP 1 replaces arbitrary code[12] in either unpredictable data, such as instructions of interpreted programming languages in the Heap, or in predictable data, such as the `.text` segment for normal programs not

[12] It is assumed that code pointers can be replaced by this attack. For the sake of the argument, ii is considered as a code pointer modification made by a code replacement attack as part of the deployment step.

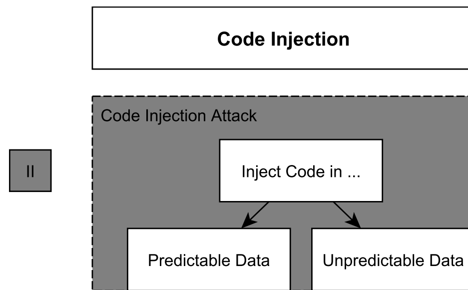

Figure 3.19 – CIAP 1: typical Step 2 Code Injection Attack to inject arbitrary Code in Predictable or Unpredictable Data Regions.

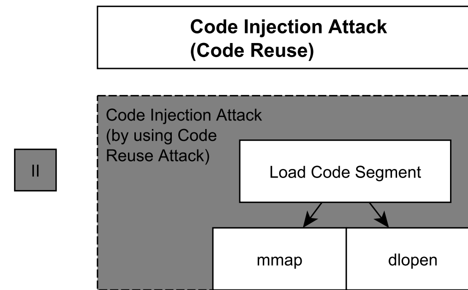

Figure 3.20 – CIAP 2: typical Step 2 Code Injection Attack to create new Memory Segments or to load Libraries.

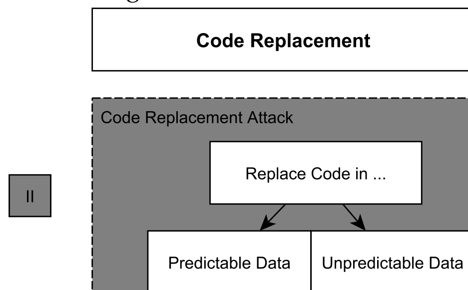

Figure 3.21 – RCAP 1: typical Step 2 Code Replacement Attack to replace arbitrary Parts in Predictable or Unpredictable Data Regions.

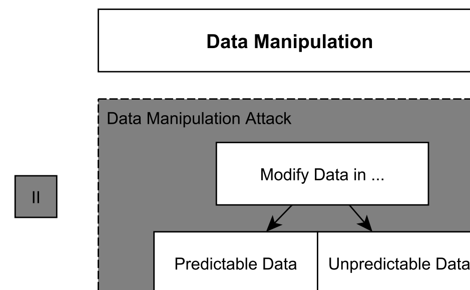

Figure 3.22 – DMAP 1: typical Step 2 Data Manipulation Attack to modify arbitrary Data in Predictable or Unpredictable Data Regions.

relying on runtime code generation.

Moreover, the last Attack is defined as Data Manipulation Attack Pattern (DMAP), cf. Figure 3.22. DMAP 1 can manipulate arbitrary data in either unpredictable data, such as configuration data inside the Heap, or in predictable data, such as constants inside the data-segment of a program's memory.

Activation Step

The Activation Step fulfills an important role for code injections. As mentioned earlier, code injection in this thesis introduces new, but most importantly, unreferenced code into the program's memory. This means that even though the code is present in the program's memory, it is not part of the actual CFG and thus not reachable during normal execution. As a result, the Activation Step is responsible for manipulating a code pointer in such a way that the injected code becomes a part of the CFG. This means that the IP eventually references an instruction of the injected malicious code. As depicted in Figure 3.23, a manipulation of a code pointer can be applied in predictable data, such as modification of a branch instruction's target address inside the program's `.text` segment or inside the GOT. Alternatively it can be applied in unpredictable data, such as the return address of

the Stack or a function pointer that is maintained in the Heap. This is defined as CPMAP 1.

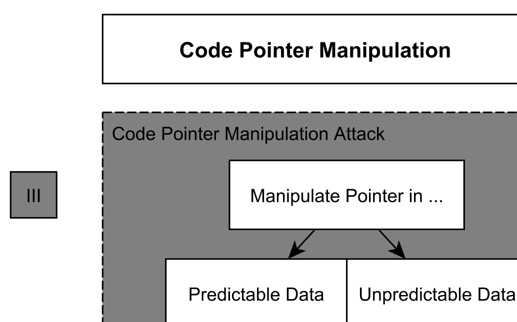

Figure 3.23 – CPMAP 2: typical Step 3 Attack to Manipulate Code Pointer to trigger the Execution of previously Injected Code.

Multi-Step Techniques and Methods

As explained earlier, a single attack is in many cases not sufficient to exploit a vulnerability to accomplish the adversary's long term goal. This is because often pre- or postconditions must be fulfilled or the attack would become too complex to be reliable or cost effective.

For these reasons, most attacks nowadays rely on the application of multiple attacks and techniques that are chained together in a specific order.

For instance, the code replacement attack based on RCAP 1, depicted in Figure 3.24, can only be successful if the replacement target is writable before the replacement (pre-condition) and executable after the replacement happened (post-condition). For example, replacing code in the `.text` segment does not fulfill the Deployment precondition unless write-protection is disabled. Therefore, the *make writable* variant of CPMAP 1 must be applied before the Replace Code Attack (O1). Usually a Code replacement attack target memory area is always executable since it would not work as intended otherwise. However, for the sake of completeness, if this Deployment post-condition is not met, the application of the *make executable* variant of CPMAP 1 must be applied before or, at latest, after (O2) the Code Replacement Attack.

As a second example of a Multi-Step attack, illustrated in Figure 3.25, a code injection attack-based CIAP 2 is defined. CIAP 2 itself has no Deployment pre- or postconditions, but, as mentioned, it is not yet a part of the CFG. For this reason, CPMAP 2 must be applied to reference the injected code. However, CPMAP 2 has indeed an Activation precondition, that is, the targeted area of the code pointer modification must be writable. This means that if the code pointer target is not writable, for instance if the GOT is mapped as read-only [66], CPMAP 1 must be applied and make the memory area writable in order to fulfill the precondition of CPMAP 2 (O1).

The third example Multi-Step Attack, depicted in Figure 3.26, is very similar to the first attack, cf. Figure 3.24. The difference is that CIAP 1 is used as the attack technique

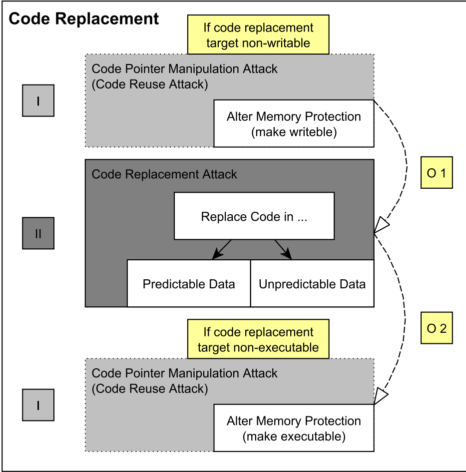

Figure 3.24 – Code Replacement Attack with optional Memory Permission Modification.

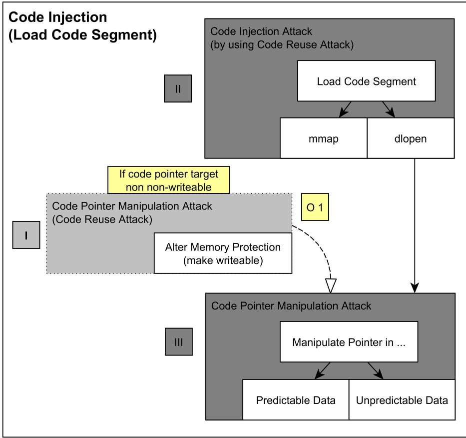

Figure 3.25 – Code Injection Attack with optional Memory Permission Modification.

instead. This means that arbitrary code can be injected somewhere in the address space. However, CIAP 1 has a Deployment precondition that requires the targeted memory area to be writable and a Deployment post-condition that requires that the injected code resides in an executable memory area. For this reason, the *make writable* variant of CPMAP 1 must be applied if the precondition is not met (O1). The *make executable* variant of CPMAP 1 applied, in cases the post-condition is not met (O2). The Activation Step applying CPMAP 2 is equal to the previous example. If the Activation precondition is not met, the make writable variant of CPMAP 1 must be applied accordingly (O3).

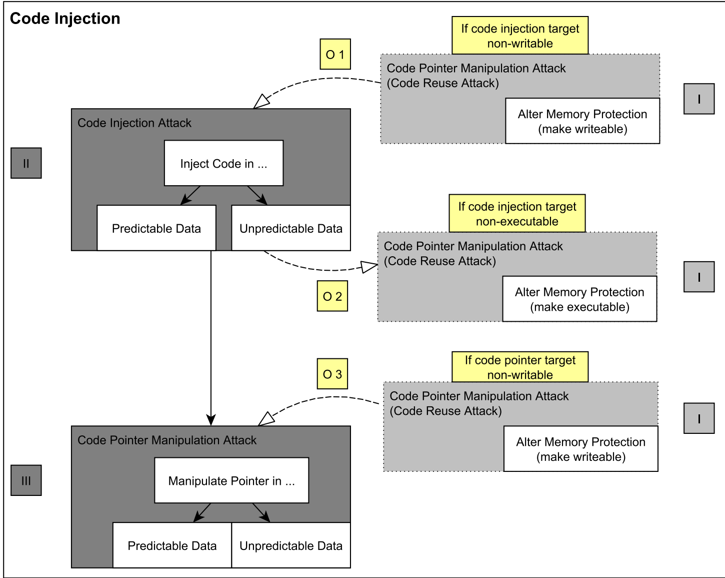

Figure 3.26 – Complex Code Injection Attack with multiple optional Memory Permission Modifications.

3.3.4 Attack Scenarios

This section presents concrete attacks of the attack methods **A1** to **A6** defined in Section 3.3.2 and based on the introduced single or multi-step attacks from the previous section. The goal of the different attack scenarios is to model attacks that alter data in very different predictable and unpredictable data segments. This makes it possible to analyze the attacks objectively whether the proposed solution in this thesis is able to detect them or not.

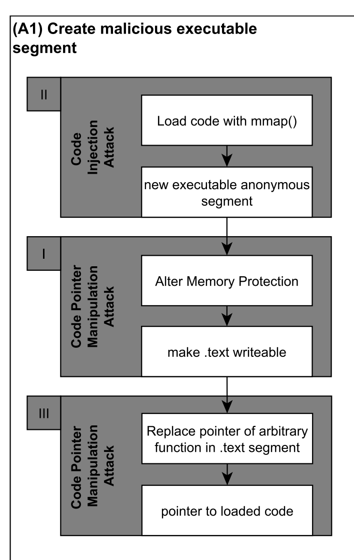

Figure 3.27 – A1: Create Malicious Executable Segment.

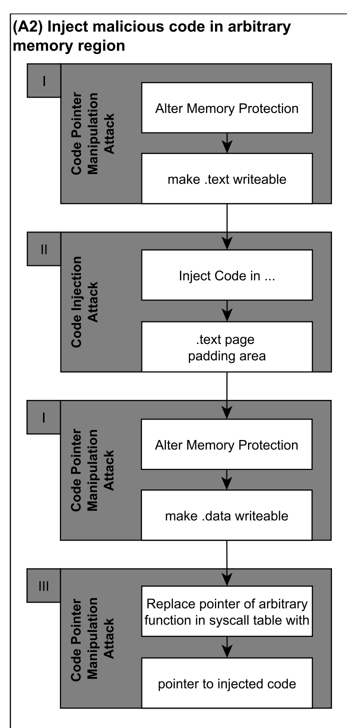

Figure 3.28 – A2: Inject Malicious Code in arbitrary Memory Region.

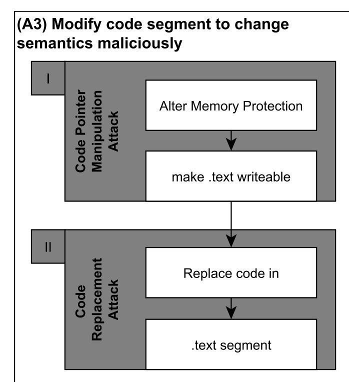

Figure 3.29 – A3: Modify Code Segment to change Semantics maliciously.

A1: Create Malicious Executable Segment This attack is based on the injection of malicious code into the process' VAS in a separate memory segment and depicted in Figure 3.27. The attack starts with an initialization step that creates a new anonymous mapping in the VAS. The allocation of a new mapping is usually done with the `mmap` system call and, depending on its parameters, the segments' access permissions can be chosen based on the parameters given to mmap. If malicious code is already available on the system, e.g. if it was downloaded earlier, the access `r-x` permissions are sufficient to conduct the attack. Alternatively, an anonymous segment with `rwx` permissions can be created and malicious shellcode can be inserted by writing to the segment.

Once the malicious code is available on the system in a process VAS, it must become

a part of the CFG. In order to do this, active code, for instance the `.text` segment of the process, must be modified accordingly. Thus, in the second step of the attack, the `.text` is made writable by calling the `mprotect` system call. Since the `.text` segment is now writable, the third step of the Attack modifies a code pointer inside `.text` so that the injected code from step 1 is called instead, replacing a previously defined benign function. As a result, the malicious code invocation becomes persistent until the process is terminated.

A variation of the attack could also modify the return address of a function call on the stack instead of conducting step 2 and 3. However, in this case, the malicious code would only be triggered once, since after its execution it would no longer be part of the CFG.

A2: Inject Malicious Code in Arbitrary Memory Region The attack, depicted in Figure 3.28, injects code into `.text` segment's padding space and is similar to attack A1; but, in this case the malicious code is not injected into a separate memory area. Instead, the core idea of the attack is to place the malicious code into a memory location that is already present but not actively used for any other purpose. This will eventually decrease the detectability significantly since the appearance of an additional mapping in the address space may raise suspicion even if no integrity-based countermeasure such as DRIVE is available.

In order to initialize the attack in the first step, the `.text` segment of the kernel must be made writable to hide the malicious code. This is done by emitting the behavior of `mprotect` by setting the corresponding writable flag in the *page table*[13].

As mentioned, the goal is to place the malicious code not in an additional segment mapping and, in particular, it shall not replace any existing code. Instead, the code is placed into the padding area of the `.text` segment in the second step. This hides the malicious code in a place that is usually not considered to contain any code and, thus, helps to avoid its detection. In addition to that, even the permission-meta data will not raise any suspicion in the long term if the permissions are restored after the manipulation. In the third step, the data-segment, holding the system call table (`.rodata`), is made writable by calling `mprotect` again.

Similar to the first step, the third step alters the memory permissions of the corresponding `.data` segment.

In the fourth step, the pointer to the designated system call, i.e. in case of process hiding `getdents` or `getdents64`, is replaced by a pointer to the injected malicious code that hides certain defined processes. The modification is active until the system is rebooted.

In order to give a more concrete example for this last step, attacks commonly hide malicious activities in the system by modifying the system call table of the kernel. This

[13] Inside the kernel space the `mprotect()` is not available.

technique is called a hooking and in the particular case system-call table hooking, cf. [67, 68]. Many rootkits apply these techniques to avoid detection or to hide certain processes which may indicate that the system has been compromised.

In addition, the attack may remove the writable access permissions from `.text` and `.rodata` after the modification was made. This will further increase the stealthiness of the attack and render a detection based on segment access permissions impractical.

It has to be mentioned that the idea to facilitate a hiding mechanism in unsuspicious memory areas can also be applied for user space programs.

A3: Modify Code Segment to Change Semantics Maliciously This attack is a simpler variation of Attack 2 and depicted in Figure 3.29. Similar to A2, in this case, the `text` segment is made writable in its initialization step by calling `mprotect`. Afterwards, arbitrary instructions inside `.text` are replaced by malicious instructions. If desired by the attacker, the original access permissions of `.text`, i.e. `r-x` can be restored after the modification, to avoid detection by inspecting the access permission flags.

A3 can modify `.text` segments in the kernel and user space. If a modification happens in kernel space, the malicious code is active until the reboot of the system. If it is applied to a process, it is active until the process is restarted.

A4: Modify Code Pointer Data to Call Malicious/Unintended Code Attack A4, depicted in Figure 3.31, represents a complex attack that utilizes an Information Disclosure Attack in its initialization step. Once the memory address of `system()` is known to the attacker, the second step modifies a code pointer so that the original intended function call is always redirected to `system()`.

This attack was described by Roglia [69]. In particular, Roglia's attack itself utilizes ROP gadgets to identify and calculate memory addresses to arbitrary `libc` functions and patches the `.got` accordingly. As a result, all calls to the attacked function are redirected to the patched one that may cause severe effects enabling arbitrary commands to be executed on the attacked system with the privileges of the owner of the exploited process.

The attack itself is very stealthy since it modifies a dynamic data segment. The attack is active in the system, until the infected process is terminated.

A5: Alter/Remove Memory Protection Attack A5 represents a basic initialization step attack that is used in many other complex attacks. The attack itself is a CPM with the single purpose to disable the DEP countermeasure ([59, 60]). As explained earlier, nearly all modern OS and platforms provide this protection mechanism. As a result, any complex attack that needs specific access rights to a targeted section utilizes a particular variant of A5. The considered attack facilitates a ROP chain to explicitly call libc's `mprotect()` function. Figure 3.30 depicts three variants that change different targeted segments. The

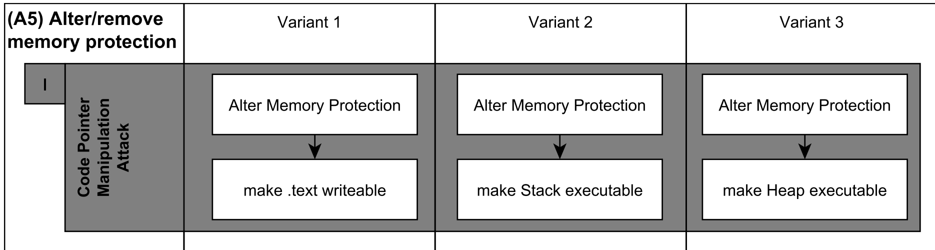

Figure 3.30 – A5: Alter/Remove Memory Protection.

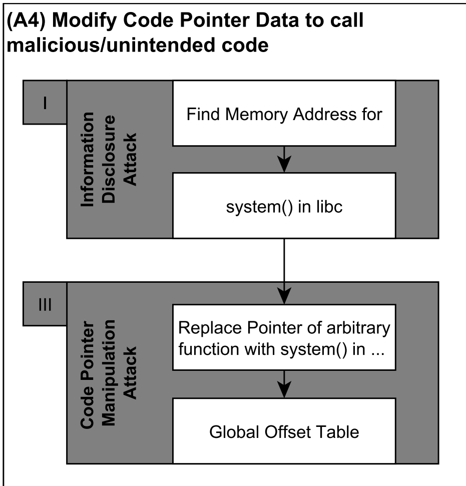

Figure 3.31 – A4: Modify Code Pointer Data to call Malicious/Unintended Code.

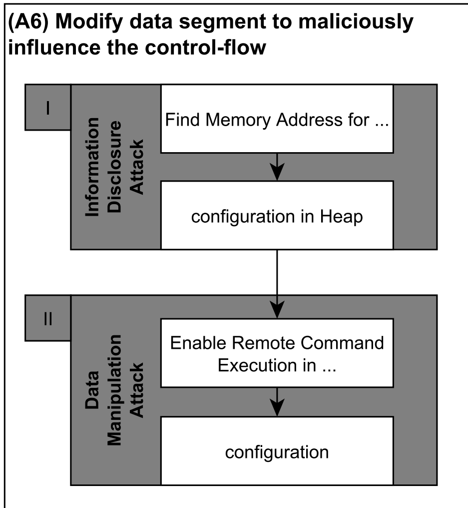

Figure 3.32 – A6: Modify Data Segment to maliciously alter the Control Flow.

first variant targets the `.text` segment and enables the injection of arbitrary code by disabling the write protection flag. The second and third variant alters the executable flag, enabling the execution of code into the targeted segments, i.e. the `.stack` segment in variant two and the `.heap` segment in variant three.

A6: Modify Data Segment to Maliciously Alter the Control-flow Attack A6, depicted in Figure 3.32, represents a non-control data attack and uses the infamous ghost vulnerability [70], a buffer overflow inside the `.heap`. This means that the attacker gets arbitrary read and write access to the `.heap` by exploiting the vulnerability. The possible effects of this vulnerability were shown by spawning a remote shell in an application that did use the `gethostbyname` function. In particular the PoC demonstrated the `Exim` mail server exploitation by executing a shell command. In the initialization step of the attack, the required configuration option, i.e. `helo_try_verify_hosts` address, is determined. Afterwards, a shell command is built and injected into the `smtp_cmd_buffer`, exploiting

the heap overflow and finally triggering the execution of the shell command[14].

The technical details of the exploit are not important for this thesis, since they are mainly related to Exim and its configuration. Still, the attack is interesting, because it demonstrates a Non-control data attack that only relies on the modification of data inside .heap. Naturally these attacks are the stealthiest since there is no known mechanism to distinguish between a good or bad command in general. The only possible solution in this case would be to implement a white-list inside the program logic that only allows the execution of pre-defined commands. The exploit itself is available whenever an Exim process with a vulnerable glibc version is executed. Still, the attack itself is only active until the Exim process or the executed command is terminated.

Table 3.7 – Memory Manipulations conducted by Attack (p:=persistent, t:=temporary. For Variants V1-V3 for A5: Permission Manipulation for V1 & V3 must be persistent and for V2 temporary.

Type	Attack Scenario					
	A1	**A2**	**A3**	**A4**	**A5**	**A6**
Permission Metadata	t	t	t	-	p/t/p	-
Predictable Static Data	p	p	p	-	-	-
Predictable Dynamic Data	(p)	(p)	(p)	p	-	-
New Mapping	p	-	-	-	-	-
Unpredictable Data	-	-	-	-	-	p

Attack Persistence Table 3.7 summarizes the manipulations conducted by each attack to become active on a system. Although the attacks A1-A3 appear to be very similar regarding their applied manipulations, they significantly differ in their details. Moreover, it must be mentioned that A1-A3 were described in this Section to compromise static data only. For this reason, Table 3.7 indicates with the (p) symbol, that all described attacks are also applicable in dynamic predictable data; specifically in load-time RCC which may appear in both kernel, i.e. LKMs and user space, i.e. shared libraries. As a result, the differences between the attacks A1-A3 and their application in dynamic predictable areas will be further discussed in Section 5.4 after the DRIVE attestation concept has been introduced.

Additionally, the access permission manipulations are almost always temporary, since they are only necessary to prepare the actual persistent attack. An exception are the attacks A5 in variant V1 and V3. In these two cases, access permissions must remain executable in order to allow continuous exploitation of the system, for instance inject shell-code directly

[14] More technical information on the exploit is available at https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0235 and https://blog.qualys.com/laws-of-vulnerabilities/2015/01/27/the-ghost-vulnerability)

into the `stack` or `heap` and trigger or wait for its execution. The results of this section will be used later during the conceptual security analysis and evaluation in Section 5.4 in order to determine which particular DRIVE verification step is able to detect the related manipulation.

3.3.5 Limitations

The detection of attacks in unpredictable data is very complex and, not a particular design focus of DRIVE. The problem of detecting malicious modification in unpredictable data is that the measured information cannot be compared to anything which is well-known to the verifier. In certain cases, a detection can be done in a meaningful way. For instance, as explained, code execution always relies on executable memory access permissions. In some cases, these malicious access permissions can be detected and verified based on well-known properties and metadata. As an example, an executable Stack can always be considered an attack. But in other cases, a decision based solely on metadata information is not always meaningful.

Another problem that influences successful detection is to capture the event that triggers a detectable change in the metadata. One particular approach could be to monitor all system calls that may create particular access permissions (e.g. `mmap`) or alter memory access permissions (e.g. `mprotect`). Another approach could be to modify the scheduler so that every scheduled task is first measured and anchored and then executed.

Although both approaches should be generally implementable and perfectly possible in theory, the vast amount of generated data for verification and the caused delay for the eventual execution of the task is impractical under the consideration of an attestation process that is to be carried out by a remote system. As a result, these approaches are not considered in this thesis and left open for future research.

3.3.6 Summary

This section presented the Attack and Threat model the remainder of this thesis is based on. At first, the adversary's attack goals were discussed. The general assumption was that the adversary tries to accomplish different long term goals while avoiding detection and compromising the system for as long as possible. Next, different attacks were defined that the adversary can use in oder to accomplish desired long term goals. The attacks themselves rely on multiple attack techniques for a successful exploitation. This is because the complexity to exploit a system with a single attack technique is either: impossible, due to deployed counter measures; or impractical, because the complexity would become unreasonably high.

For this reason, a multi-step attack model was presented that considers the utilization of different related attack techniques. The defined steps are: Initialization, Deployment

and Activation. Typically, an attack always consists of one Deployment Step. This step introduces the core malicious functionality and is responsible for accomplishing the actual goal of the attack. If necessary, the malicious functionality must be made active; this is usually accomplished during the Activation Step by altering a pointer to modify the CFG. Still, Deployment and Activation rely often on pre- or post-conditions. To resolve these conditions, one or multiple Initialization Steps must be applied before or afterwards. Next, on the basis of this behavior, a multi-step attack model was introduced. It enables the flexible chaining of related attack techniques and facilitates the definition of concrete attacks models (attack scenarios) to compromise the system.

Next different attack scenarios were defined on the basis of the presented multi-step attack model. The defined attack scenarios considered and covered all mentioned attack techniques from Section 3.2 and act as concrete and implementable examples for further analyses in the remainder of this thesis; they will be used during conceptual analysis in Section 5.4 and for the evaluation of the implementation in Section 6.2. The attack scenarios were introduced and discussed and a brief analysis of their persistence properties provided. As a last step, limitations for detecting attacks that target unpredictable data were briefly discussed. These limitations will be revisited in the following sections and discussed in more detail in Section 5.4.3.

3.4 Trust Model and Security Assumptions

The envisaged solution DRIVE can and should be combined with all available defensive mechanisms to be most effective. As a non-intrusive solution, targeted to further improve the overall system security, all previously mentioned preventive security mechanisms are compatible and provide significant obstacles for any adversary.

Nonetheless, DRIVE also depends on some security requirements. Most importantly, it relies on enabled security mechanisms to successfully detect or prevent execution of illicitly modified ELF files such as executables, libraries, kernel modules, and the kernel. In particular, it is assumed that the system boots into a well-known and reliable state and behaves as expected until an initial attack is executed. This requirement can be enforced by carrying out a secure boot or attested by utilizing a measured boot of the OS as described in Trusted Computing [17]. As a result, the adversary must not be able to modify and load system binaries on disk, modify or replace OS components, or disable those mechanisms without being detected.

Similar to well-known integrity protection schemes, such as the IMA, cf. [33], DRIVE may utilize a tamper-resistant component, for instance a TPM security module, to continuously record, track, and report system states securely and arbitrarily. First and foremost, tamper-resistance is the essential property required. This means that the measurements can no longer be modified – or can only be modified at considerable expense – as soon as

they are in a tamper-resistant component.

In addition to that, one security-sensitive part is the measurement accumulation of the system memory contents, specifically before measurements become tamper-resistant. For this reason, it is assumed that the attacker cannot interfere with the measurement process or disable it altogether. This means that the component that takes the measurements is considered isolated from and immutable by the attacker. A straightforward implementation may employ OS-based process isolation or user/kernel-level isolation; but, as soon as threats are considered that effectively bypass kernel level security protection mechanisms [71–73], more sophisticated isolation techniques and mechanisms must be considered. A lot of research and solutions exist that provide different isolation mechanisms, for instance: (1) Virtualization- or hypervisor-based approaches, [18, 74–79]; (2) Sandboxing approaches, [80–82]; (3) Hardware-backed approaches, for ARM TrustZone [83, 84] and Intel Software Guard Extensions (SGX), [85–87]; (4) other isolation mechanisms, for instance [88] for ARM; or (5) discrete hardware security coprocessor-based approaches, similar to [89, 90].

DRIVE should therefore not be limited to a particular isolation mechanism; however, the exact implementation and available isolation mechanisms mainly depends on use-case-specific requirements and conditional architecture and platform capabilities. A more detailed analysis of virtualization and isolation mechanisms is provided in Section 4.3.

3.5 Security Analysis Summary and Conclusion

In this chapter, a detailed security analysis of threats and attacks on system security at runtime was presented. In a first step, malware was described and examined on the basis of its specific characteristics, persistence and stealth. The malware was classified according to its respective properties. Classification for stealth was carried out on the basis of an extended taxonomy of Rutkowska [31]. The target malware in this work is thus type-1 and type-2 malware, which generally describes runtime attacks. For persistence, the applied changes in the actual data during the attacks were considered. Depending on which data was modified in the memory, the persistence changes accordingly. As determined, runtime attacks can reach any kind of persistence, from volatile to resident. Furthermore, it was determined that both properties are related to each other. However, this depends on which system component and which memory area is attacked.

In the second step, various attacks were introduced and investigated which can be executed at runtime. After an overview of relevant memory attacks and their definitions were introduced, various types of attacks on the control flow were discussed in detail. For this purpose, certain types of CFG attacks were presented. The attack types are divided into attacks that either manipulate the control flow, so-called CFM attacks or bend the control flow, so-called CFB attacks. In addition, there are also information disclosure

attacks that can be used to obtain specific information, for instance canary values or memory addresses. All attacks were examined on the basis of their specific characteristics and classified into the identified characteristics of stealth and persistence. Furthermore, it has been found that nowadays almost always multiple attacks are necessary to compromise a system. For this purpose, the concept of hybrid attacks was introduced which can be divided into different steps. Related to hybrid attacks, it has been found that there are interactions that affect the original attack classification based on stealth and persistence. Furthermore, common countermeasures have been discussed which are available today or will be available in the coming years. It was determined that hybrid attack methods eventually bypass all countermeasures. Furthermore, it was found that the specific attacks considered in this work are insufficiently protected. There are countermeasures that prevent other attacks, but there are no specific countermeasures that are tailored to the target attacks of this work.

In the third step, an attack model was developed that allows the modeling of hybrid attacks on system security. After the abstract objectives of the attacks were introduced, various procedures were explained and example attacks were defined. For this purpose, the hybrid attacks discussed before were used again and a model was developed that considers the necessary steps during exploitation. Based on the defined model, attack scenarios were then developed which will be used in the further course of this work to evaluate the concept and implementation. The attack scenarios specify hybrid attacks on system security by the intelligent chaining of individual attacks to avoid countermeasures. Attention was paid to the fact that the attack scenarios consider all previously defined attacks and thus all possibilities for exploitation.

In the final step, the security assumptions of the concept to be developed were discussed. Since the concept is meant to be an orthogonal security technology, it is not possible to view the concept in isolation. For this reason, system properties have been identified which must be fulfilled in order for the concept to be used effectively. In particular, it was determined that an attacker must not be able to disable the necessary technical components.

To conclude, the most important finding of the security analysis is that malware today is hugely complex and uses multiple highly sophisticated techniques in order to compromise a system. Recent developments show that the attack techniques used are particularly designed to circumvent existing countermeasures and thus eventually compromise the system. However, adversaries try to use simpler attack techniques as soon as countermeasures are bypassed. Particularly in this context, the work in this thesis is of significance because it closes a gap between the traditional exploitation based on file modifications and the sophisticated code reuse or non-control data attacks. For this reason, DRIVE makes a valuable contribution to the state-of-the-art in order to further increase today's system security.

Chapter 4

DRIVE High Level Attestation Concept and Architecture

This chapter introduces the high level attestation concept and architecture of DRIVE. At first, the general concept and idea of an attestation scheme is presented. After the relevant systems involved and different phases of an attestation are identified and introduced, the corresponding system memory areas and operations to conduct an attestation are discussed. Next, the high level architecture of DRIVE is introduced and described. The high level architecture is developed by considering the abstract attestation scheme described earlier. Initially, an overview of the architecture is presented that identifies and describes the necessary building blocks.

Afterwards, the building blocks, their purpose and mechanisms are discussed in more detail. This is done in particular by identifying the relevant information DRIVE utilizes and by describing the individual mechanisms to conduct a successful and secure attestation on an abstract level.

After the high-level architecture and concept, a deployment analysis of the developed architecture is carried out in the next section. More specifically, this analysis seeks to answer important questions relating to secure deployment with regards to available isolation mechanisms. The analysis discusses variations of architectural deployments and attempts to determine how certain isolation techniques affect the reliability of the attestation scheme.

In a final step, the design space and architecture will be analyzed for specific constraints covered by other security technologies. In particular, the analysis provides a distinction to CFI technologies and explains the limitations of DRIVE on an architectural level.

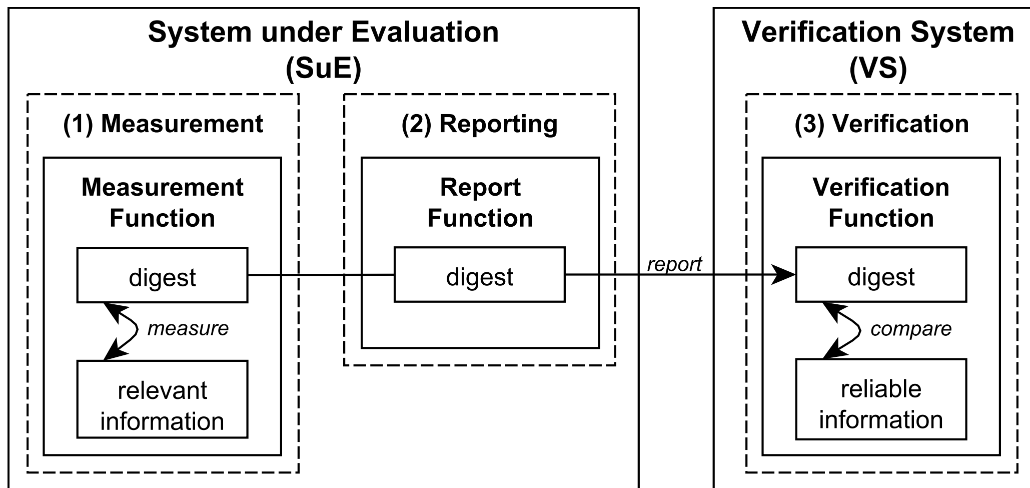

Figure 4.1 – High Level Attestation Concept of relevant Information based on Digests.

4.1 High Level Attestation Concept

An attestation describes certain mechanisms to make a decision about the trust state of a particular system. The basic idea behind an attestation is to: acquire information from that particular System under Evaluation (SuE) and verify this information by comparing it against available reliable information on an initially trusted system. An attestation usually consists of three individual mechanisms as depicted in Figure 4.1:

(1) Measurement: conduct a measurement on the SuE, by calculating a digest of relevant information.

(2) Reporting: report the calculated digest to a trusted Verification System (VS).

(3) Verification: verify the calculated digest by comparing it to a second reliable digest previously derived from originally trusted ELF.

Regarding DRIVE, different runtime artifacts residing in the system memory represent the relevant information for the attestation. More specifically, the runtime artifacts are specific memory areas that can be identified by their memory addresses and which are further described by additional metadata, for instance size, access permissions or name. The measurement mechanism must securely acquire this information and generate a digest of these memory artifacts and their related metadata. As previously described in Section 2, the targeted memory artifacts are represented by ELF segments and sections and the OS is responsible for managing their memory addresses, metadata and runtime environment. Also, as explained, there exists a strict separation in the memories' VAS that separates the kernel and the user space. Consequently, this means DRIVE must consider relevant artifacts and metadata from both the kernel space and the user space during an attestation.

The details of DRIVE's developed measurement process and digest generation will be described in Section 5.1.1.

Once a digest of a targeted memory artifact and its metadata is generated, the digest must be securely reported to the trusted system. Securely means that the reporting mechanism must protect the measured information during transit. In particular, the freshness, authenticity and integrity of the data must be assured and protected. For these reasons, an attestation protocol should be used that provides the necessary security properties during the reporting step. Consequently, before the digest is transmitted from the SuE to the trusted system additional information is exchanged and cryptographic operations are applied that guarantee the necessary security properties. DRIVE's reporting mechanism will further be discussed in detail in Section 5.1.2.

The last step in an attestation is the verification of the reported digest and its related metadata. The verification involves an initial step that determines whether the received information fulfills the previously described security properties. Thus, the freshness, authenticity and integrity of the digest and the metadata are verified by applying the related cryptographic operations. If all security properties are conclusive, the digest and metadata are verified by comparing them to well-known reliable information that is present on the trusted system. If the calculated digest and metadata can be successfully compared against the corresponding reliable information, it is proven that the measured data has not been illicitly tampered with and thus represents a well-known state on the SuE.

As expected, the measurement, reporting and verification of a single runtime artifact does not provide enough information to evidently prove that the whole SuE is in a trustworthy state. For this reason, multiple measurements from both the kernel and user space must be acquired, reported and verified by the trusted system. If enough information from the SuE is successfully verified, there is a high propability that the entire SuE resides in a trustworthy state. It has to be mentioned that proving the trustworthiness of the SuE cannot be verified beyond any doubt. However, if enough relevant information has been provided, the trusted system can make a decision on whether reported crucial parts of the SuE behave as expected and thus conclude that the entire system is in a trustworthy state with a very high level of assurance. DRIVE's verification mechanism and the decisions made based on the verification results will further be discussed in detail in Section 5.1.4

4.2 DRIVE High Level Architecture

The main objective of DRIVE is to repeatedly measure, report, and verify runtime information present in the system memory. Since the measurements only reflect the system state at the exact time, reporting and verification only represent the status of the trustworthiness of the systems at this point in time. For this reason, continuous attestation of the

system should be carried out, starting directly after start-up and ending with shutdown. It is generally not possible to determine at which interval or based on which events an attestation should be carried out, since this depends on the requirements of the operator.

In general, it can be assumed that depending on the use-case, adaptations to DRIVE's architecture will be necessary. Bearing this in mind, this section introduces the architecture in an abstract way. This helps with the implementation of the architecture in different manifestations, without having to revise the originally specified architecture. However, the flexibility of the architecture also influences the attestation. Depending on the actual implementation of the architecture, certain components may or may not be available. For this reason, architecture and attestation are closely intertwined and therefore cannot be discussed individually. As a result, this section will examine one particular instantiation of the architecture and discuss the attestation concept for this case. The following Section 4.3, however, will analyze different manifestations of the architecture and discuss how the attestation is affected in these cases. This analysis will be based on the reliability of the attestation processed information.

4.2.1 Architecture Overview

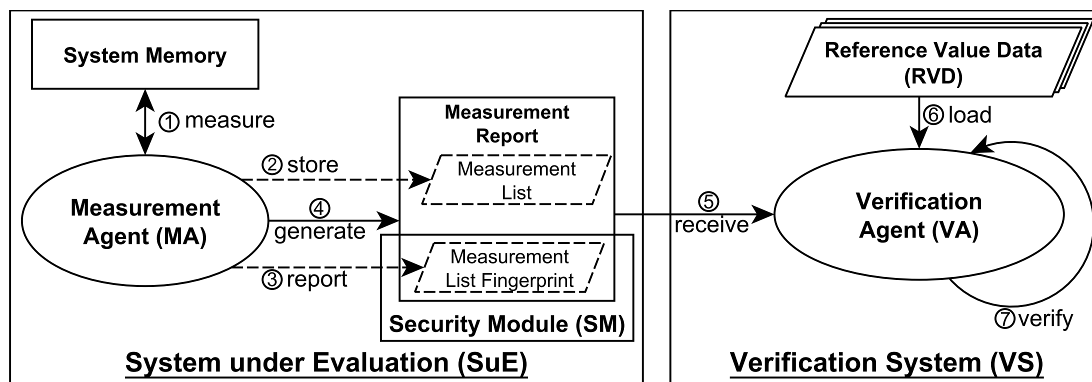

Figure 4.2 – DRIVE Architecture for Measurement, Reporting and Verification.

Figure 4.2 presents the high level architecture that is defined and analyzed in this section. It consists of two systems: 1. SuE and 2. VS. The SuE implements the Measurement Agent (MA) that is responsible for acquiring the relevant measurements from the memory. The MA produces a Measurement Report that includes a Measurement List, comprising a list of corresponding measurements and a Measurement List Fingerprint that is anchored in a security module. The VS implements the Verification Agent (VA) that receives the produced Measurement Report for verification. Most importantly, the VA implements a verification component that conducts the verification process under the assistance of Reference Value Data (RVD).

It has to be noted that the architecture shares some similarities with the architecture described by Sailer in [33]. It utilizes similar conceptual measurement and verification

agents, and also a security module. However, measurement acquisition and verification differs significantly, because DRIVE repeatedly measures and verifies the runtime system's states including dynamic information, whereas Sailers' work only considers static one-time measurements taken before the loading process.

Measurement Agent: Measurement and Reporting of Measurements

The MA is the core component of the SuE and responsible for secure measurement and reporting. DRIVE's Measurement and Reporting architecture is depicted in Figure 4.2 and is presented in the following.

In order to conduct a secure measurement process, DRIVE's Measurement Component implements the following operations: ① Measure and receive the targeted system memory; ② store individual measurements in a Measurement List; ③ generate and report a Measurement List Fingerprint; and ④ generate a Measurement Report encapsulating the Measurement List and Measurement List Fingerprint.

As expected, the MA needs specific rights to access particular regions in the system memory. Typically, if the MA is implemented as a software component, the access rights are granted by the OS and depend on the specific measurement target. Alongside the access permissions that would be sufficient for implementations at the kernel level, the MA should preferably run at a higher privilege level and thus isolated from the to-be-measured target. This means that an implementation at the kernel privilege level may be sufficient regarding the access rights, but insufficient in scenarios that consider attacks to the OS kernel itself. In those cases an implementation should utilize more complex isolation mechanisms or employ a hardware-based coprocessor for measurement acquisition, as discussed in Section 3.4 and Section 4.3.

Moreover, the security module is expected to be resistant against manipulation. Once a Measurement List Fingerprint is anchored within the security module, it should be nearly impossible to manipulate this value. Ideally, a discrete TPM is used as a security module since tamper resistance is a particular property defined in the TPM's specification. However, if a TPM is not available in the target system, alternative solutions using soft- or hardware-based isolation techniques must be considered. This will further be discussed in Section 4.3.

Verification Agent: System Report Verification

The VA implements all necessary functionality to verify a Measurement Report. Involved operations include: ⑤ receive a Measurement Report from a SuE, ⑤ load trusted RVD and calculate or extract concrete reference values and ⑦ verify the integrity, based on the Measurement List Fingerprint, and individual measurements from the Measurement Report.

The Measurement Report is received by the VA either by direct exchange of the SuE or, preferably, by using an attestation protocol. Direct exchange in this case means that the SuE provides an interface to access the Measurement Report remotely or locally, for instance by offering a network service or internal API to access it. However, especially if SuE and VS are two different independent systems, for instance connected only over a network, the Measurement Report should be securely transmitted by using an attestation protocol, if supported by the components involved. Securely transmitted means integrity protection, authenticity, and freshness of the Measurement Report is assured during transit. If the measurements themselves reflect security critical data, confidentiality of Measurement Report during transit must also be guaranteed. The provided solution in this work utilizes an attestation protocol to transmit the Measurement Report between SuE and VS. Still, as a attestation protocol is not always available, Section 4.3.1 will revisit this topic and discuss alternative solutions and their security.

4.2.2 Instantiated Software Architecture

The previous section introduced and discussed the components involved and the architecture from a high level point. As described, different manifestations of the high level architecture can be instantiated and architecture design decisions are often based on the particular use-case further influencing every single component and their implementation. In this section one instantiated attestation architecture is described and the components of DRIVE and their individual tasks are introduced.

DRIVE's Measurement and Reporting Process

DRIVE's instantiated measurement architecture is depicted in the upper half of Figure 4.3. DRIVE uses its own terminology of the components and data-structures involved. The terminology is defined as follows:

DRIVE Measurement Component (DMC): Instantiation of a Measurement Agent (MA) that implements the functions: *measure*, *anchor* and *report*.

Security Module: Tamper resistant component used to securely store fingerprints of measurements.

System State Report (SSR): Instantiation of the Measurement Report container data-structure. Comprising the Fingerprint and the Dynamic Measurement List (DML).

Dynamic Measurement List (DML): Instantiation of the Measurement List. It comprises all accumulated measurements in an ordered list.

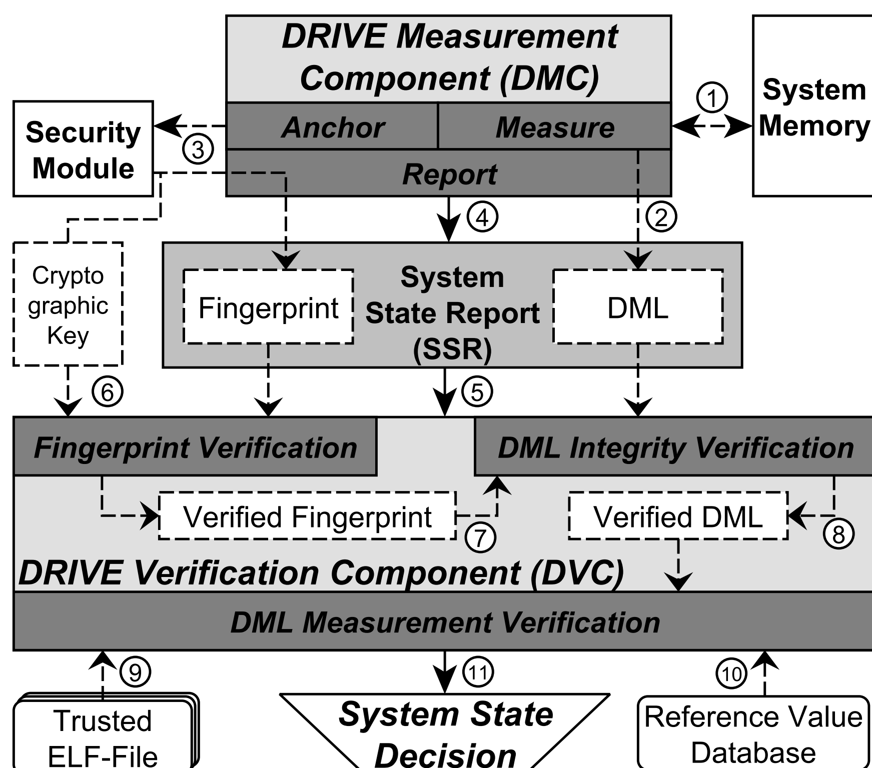

Figure 4.3 – DRIVE Measurement, Reporting and Verification Process Overview.

Fingerprint: Instantiation of the Measurement List Fingerprint data/structure. An internally calculated value of a security module to prove the integrity of a DML to an external system. The calculation is based on the acquired measurements.

The DRIVE Measurement Component (DMC) is the core component of the architecture during measurement, implemented on the SuE and responsible for *measuring*, *anchoring* and *reporting* the acquired measurements. First, the system memory is measured ①. Measured in this case means that the individual targeted memory artifacts, alongside their metadata, are measured and stored in internal data-structures. Afterwards, these internal measurements are appended to an ordered DML ②.

Second, directly after the measurement is appended, a fingerprint of the measurement list is created and anchored inside the security module ③. The fingerprint is constructed in such a way that it represents all accumulated measurements in a single value; thus, the fingerprint can later be used to prove the integrity of the DML to the VS.

Third, the report function is responsible for generating a System State Report (SSR) ④. This data-structure is used during the integrity verification process. It consists of the DML and the corresponding fingerprint that is anchored inside the security module. In addition to that, the Fingerprint is digitally signed by the security module, with a private cryptographic key that is only known to the security module. This is later used during verification to verify the authenticity of the Fingerprint.

DRIVE Verification Process

DRIVE's instantiated verification architecture is depicted in the lower half of Figure 4.3. The terms used are defined as follows:

DRIVE Verification Component (DVC): Instantiation of a Verification Agent (VA) that implements the functions: *Fingerprint Verification*, *DML Integrity Verification* and *DML Measurement Verification*.

Verified Fingerprint: Fingerprint data-structure after successful verification based on a well-known cryptographic key.

Verified DML: DML data-structure after successful integrity verification based on the verified Fingerprint.

The DRIVE Verification Component (DVC) is the core component of the architecture during verification, realized on the VS and implements the functions: *Fingerprint Verification*, *DML Integrity Verification* and *DML Measurement Verification*.

After the SSR was successfully received by a VS ⑤, the DVC verifies different properties of the SSR. At first, the authenticity of the Fingerprint is verified. This is done by: ⑥ loading a well-known Cryptographic Key that identifies the security module of the SuE and ⑦ verifying the authenticity of the Fingerprint based on the loaded Cryptographic Key. If the authenticity can be successfully verified, i.e., the digital signature is valid, the Verified Fingerprint is used in the subsequent operation.

Second, as indicated by ⑧, the integrity of the transmitted DML is verified. During this process a cryptographic hash sum is generated by the DML Integrity Verification function. This is done by extracting all significant individual values that were used to calculate the fingerprint inside the security module. Since the DML is an ordered list, the sequence is already correct. Furthermore, this self-calculated DML-based Fingerprint is compared against the Verified Fingerprint from the previous step. If, and only if, both values are equal, the integrity of the received DML is verified successfully and, thus, the DML is considered untampered and reliable.

Third, the individual measured values are further analyzed to determine whether the system state is in a trusted state or not. In order to verify whether the measurements were modified during runtime, the VA compares every measured value present in the DML against references. Every successfully verified measurement draws the conclusion that the measured memory part did not change unexpectedly and thus can be considered as trustworthy. The process of DML verification is depicted in Figure 4.3 and briefly described next.

The subsequent verification of the individual measurements is as follows: For every individual measurement in the DML the DRIVE Measurement Verification function tries to generate or find valid reference values. This can either be done by ⑨ generating a

reference value by loading process simulation of a Trusted ELF file or, alternatively, by (10) searching for a valid reference value in the precomputed RVD. In case a valid reference value has either been successfully generated or found, the individual measurement represents a well-known state. Accordingly, the process is repeated for every accumulated individual measurement. If all measurements can be verified successfully, the measured system parts are considered to behave benign, meaning no illicit modification has been detected. This is indicated by the System State Decision (11). In addition to the aforementioned steps, further or more complex verification operations may be used. This depends on the type of measurements and cannot be defined in general. For this reason, the verification process is described in more detail in Section 5.1.4.

4.2.3 Summary

The previous sections introduced the high level attestation concept, architecture and presented DRIVE's instantiation. The objective of the attestation is to determine the status of a particular system by determining whether the system is trustworthy or not. The attestation concept was the first to be introduced. Two systems have been defined for this purpose: 1. SuE and 2. VS. The following operations are then performed on these two systems: Measurement, Reporting and Verification. Measurement and reporting is performed on the SuE. It collects relevant information about the system runtime as digests and implements a mechanism to provide this information. The collected information is then used on the VS to determine the trust state of the system based on well-known reference values. If all presented information could be correlated with references, then it can be assumed that SuE behaves as intended and is therefore considered trustworthy.

Next, an architecture was presented that defines general building blocks used to implement the defined operations and that allows to distribute the building blocks on two distinct systems. The defined MA is responsible for implementing the measurement and reporting function and for creating a Measurement Report that represents this information. In addition to that it makes use of a security module, in order to allow a tamper-resistant storage of this Measurement Report. The introduced counterpart of verification side is the VA. This uses the Measurement Report and performs verification based on the reference values. The integrity and authenticity of the Measurement Report is first checked and then the system status is assessed.

The instantiation used in this work is based on this attestation concept and architecture. For this purpose, the instantiation has defined different components and data-structures that are used during the attestation process by DRIVE. The components for SuE are: DMC and Security Module. The DMC collects the information, anchors them in a security module and provides a verifiable SSR stored in a tamper-proof fashion. The SSR can subsequently be verified by the DVC. At first, integrity and authenticity of the SSR is verified by DVC and afterwards, all collected measurements are verified on the basis of

different reference data sources, i.e. Trusted ELF files to compute ad-hoc references and a RVD that consists of pre-calculated references. Only if all individual verifications are successful, SuE system state is considered as trustworthy.

4.3 Architecture Deployment Analysis

The provided high-level architecture and concept is flexible and supports different instantiations. The previous section contains the instantiated architecture used in this work. However, other use-cases may not provide necessary components, for instance they may not provide a security module or do not allow a strict isolation between SuE and VS. Still, they might provide other components or functions that can be used to derive a similar functionality instead.

Regardless, the most important resource during attestation is the processed information. This information is acquired during measurement, reported after the measurement and verified during the verification process. Hence, and in order to provide a meaningful implementation of a solution, the acquisition, reporting and verification of the processed information affects whether the attestation can be considered as a reliable source to determine the trustworthiness of the deployed system.

In other words, the reliability of the information is influenced by: (1.) the design decisions regarding the concrete architecture based on the use-case and (2.) the implementation choices based on the chosen architectural decisions. Therefore, this section analyzes the components presented, the systems involved, and seeks to develop a set of recommendations that should be considered when implementing the solution.

At first, the verification process is analyzed. The particular goal in this analysis is to get an understanding how different isolation techniques affect the attestation. Afterwards, the measurement and reporting are discussed. More specifically, it is discussed which isolation techniques are available to acquire reliable measurements and what possibilities exist in order to protect the acquired measurements from manipulation.

4.3.1 Isolation during Verification

An established isolation between SuE and VS is essential to determine the reliability of the system used. Ideally, both systems are deployed on two physically isolated systems. In this case, it is impossible for an opponent who has compromised the SuE to actively manipulate the verification process that makes the final decision about the trustworthiness of the SuE. Still, in many scenarios, it is not possible or desired to set up a strict physical isolation. This section introduces different possible deployment scenarios, that consider physical and logical isolation mechanisms. After the scenarios are presented, they are compared based on a determined level of reliability.

Physical Isolation of System under Evaluation and Verification System

As an example for a typical deployment scenario, it is assumed that a SuE and the VS are implemented on two distinct physical devices. As depicted in Figure 4.4 this means both systems are physically separated. They share no information other than the information that was exchanged during a communication protocol and, thus, they are considered fully isolated.

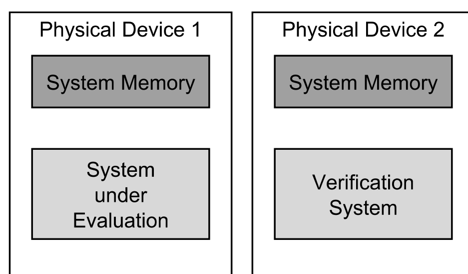

Figure 4.4 – System under Evaluation and Verification System Deployment on physically isolated Devices.

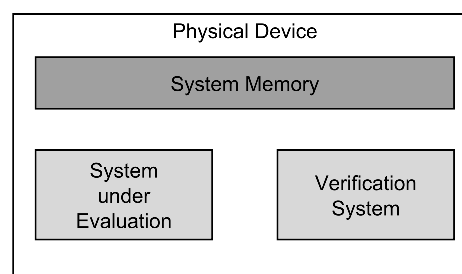

Figure 4.5 – System under Evaluation and Verification System Deployment on one Physical Device.

In this described scenario, the attestation process is commonly known under the term *remote attestation* in accordance with the terminology of the Trusted Computing Group Module (TCG). Typically, the VS is assumed to be initially trusted and it can be operated by a Trusted Third Party (TTP). This means that the verification process relies on certain assumptions, i.e., the VS and all its data are considered reliable; it is not possible to compromise this system. One could argue that this assumption is inadmissible. However, considering multiple deployed VSs in a highly secure and monitored network that conduct verification processes in parallel could be one strategy to basically eliminate the possibility of all VSs being compromised at the same time.

Logical Isolation of System under Evaluation and Verification System

Alternatively, SuE and VS can be implemented on the same physical device, see Figure 4.5. In these cases volatile and non-volatile memory are typically shared between SuE and VS.

The OS provides certain levels of isolation such as process isolation or access control mechanisms. However, when utilizing these less strict isolation techniques, the attestation process is more of an introspection and uses an Local Verification (LV) process instead. As a consequence, no initially trusted system exists during LV and thus the verification processes and all involved verification data must be secured and protected explicitly. More precisely, the VS must assure that the involved verification process and its corresponding data is protected against any sort of tampering.

Logical Isolation of System under Evaluation and Verification System with Virtualization

Both described scenarios represent two extremes. In between there are other scenarios that provide a stronger isolation between SuE and VS utilizing stronger isolation techniques. For instance, a SuE and a VS could be deployed as two individual Virtual Machines (VMs), as depicted in Figure 4.6. Although both systems share the same physical memory, the introduced virtualization layer logically isolates the VMs and prevents them from accessing each other's memory. This means, under the assumption that there is no vulnerability in virtualization implementations, the SuE cannot access the memory of VS, and vice versa.

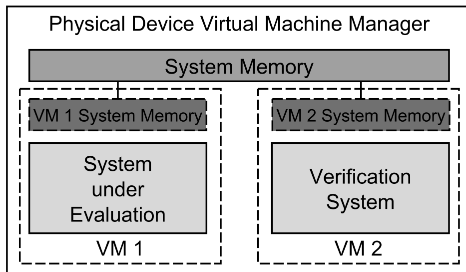

Figure 4.6 – System under Evaluation and Verification System Deployment in two logically isolated VM's.

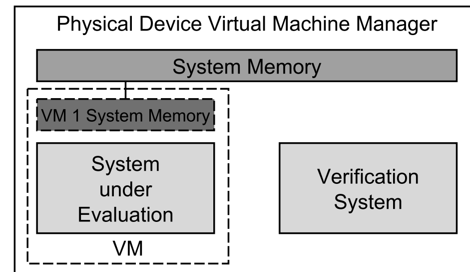

Figure 4.7 – System under Evaluation Deployment in VM and Verification System Deployment in Virtual Machine Manager (VMM).

The same principle can be applied if the VS is implemented on a VMM and the SuE on a VM, as depicted in Figure 4.7. Again, there exists a logical isolation between SuE and VS and thus the SuE cannot access the memory of VS. It is important to note that in this deployment the VS must always reside on the VMM. This is because the SuE is not originally a trusted system and thus performs the verification in a less privileged context. If done anyway, as illustrated in Figure 4.8, the systems should not be considered as properly logically separated and the results of the VS should be regarded as inconclusive.

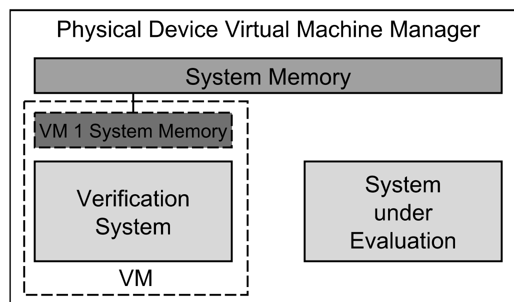

Figure 4.8 – Inconclusive: System under Evaluation Deployment in VMM and Verification System Deployment in VM.

The reliability of the accumulated and processed information varies significantly with

regard to their specific implementation. Regarding the SuE and VS deployment in the designated use-case, utilizing the strongest isolation possible is recommended. This means that physical isolation of both systems provides the most reliable solution and no isolation the least. The different analyzed deployment scenarios are summarized in Table 4.1.

Table 4.1 – Isolation Grades for different SuE and VS Deployments.

SuE location	VS location	Isolation Technique	Isolation Grade
Physical Device 1	Physical Device 2	physical	very high
Physical Device 1	Physical Device 1	process	low
VM 1	VM 2	virtualization	high
VM	VMM	virtualization	high
VMM	VM	virtualization	very low

System under Evaluation and Verification System Isolation Discussion

Under the assumption that an adversary successfully launched an attack against a SuE, every component of the SuE is considered compromised. However, in order to circumvent the measurement process, the adversary needs very specific knowledge about the measurement process and a successful attack in order to circumvent the measurement process completely; this issue will be addressed in the following Section 4.3.2. Consequently, the following discussion addresses the implications under the assumption that the measurement itself provides reliable data and only the verification process of this data is vulnerable to the adversary.

If SuE and VS are deployed on the same system without any isolation, the attack surface for the adversary increases significantly. This ranges from simple tampering with the Reference Data up to the manipulation of the VA implementation, for instance completely disabling the verification mechanism involved. Hence, avoiding detection by circumventing the initial measurement or modification of measured data anchored in the security module is far more difficult than interfering with the verification process or the reference data involved. Although the OS provides process isolation and access control mechanisms, the default configuration is not considered as sufficient to protect the verification process implicitly. As a result, the VA must be implemented in a tamper-resistant way, in order to provide reliable results.

However, as soon as specific isolation techniques are utilized, for instance Trusted Execution Environments (TEEs) [91], Intel SGX [92, 93], AMD SME [94] or ARM Bowmore[15] that are specifically tailored to make these kinds of attacks exceptionally more difficult or prevent them altogether, the adversary that compromised the SuE cannot easily interfere

[15] At the time of writing this thesis, no public available documentation or information exists on this technology. However, the functionality is comparable to AMD SME/SVE technologies.

Table 4.2 – Reliability regarding Isolation of SuE and VA.

Type	Component	Isolation	Reliability
OS	Basic OS Isolation and MAC	SW	weak
VE	SuE & VS isolated by VMs	SW	medium
VE	SuE & VS isolated by VMs	SW + HW	strong
TEE	VS isolated in TEE	SW + HW	strong
PH	Physical Isolation of Devices	HW	very strong

or access the VA. Therefore, the verification results become more reliable, as potential threats of interference are considered less likely. The technologies mentioned are, however, rather new and not deployed widely. For this reason, their security guarantees cannot be precisely defined at this time. Nevertheless, the assumption is that they offer a relatively high level of security, comparable but not equivalent to physical isolation.

As a result, physical isolation provides the strongest isolation possible, because in this case the VA can be considered as an initially trusted system. Table 4.2 offers an overview in order to assess the reliability regarding utilized isolation techniques.

Please note that a kernel space implementation for the VA has intentionally not been considered in the basic OS protection scenario. In general, a kernel space implementation for the VA is not a solid concept, since it would introduce complex functionality in critical parts of the OS, while reliability would only be comparable to the software isolation in VMs. This is because the VM's software isolation is also enforced by the OS kernel eventually.

To conclude as a general rule: The higher the level of isolation between SuE and VS, the more reliable are the results. This means that if a SuE and a VS are implemented on the same device, additional isolation techniques must be used to obtain reliable results during attestation. Ideally, in this case, virtualization or hardware-based isolation techniques should be used. If SuE and VS are implemented on distinct and independent devices, the VS represents an initially trusted device. This scenario provides the most reliable attestation results, since interferences during the verification process are eliminated.

4.3.2 Isolation during Measurement and Reporting

The measurement and reporting inside the SuE are also security-critical functions and must be protected accordingly. Similarly, as described in the previous section, different isolation techniques or components can be considered that are used to limit the capabilities of an attacker to interfere with the measurement and reporting.

Secure Reporting Mechanisms

If a tamper-resistant security module is available, a secure reporting of measurements depends on the security guarantees provided by the security module. For instance, a TPM provides highly secure anchoring mechanisms that do not allow the modification of any previously anchored measurement. This means that once a measurement has been anchored inside the TPM, it is assumed that an adversary is no longer able to modify an SSR or any previously anchored measurement without becoming detectable. This limits the adversary's capability to manipulate the measurement process itself by disabling or circumventing it altogether. As a consequence, the adversary can only manipulate measurements or reports that are not already anchored.

A similar principle applies if measurements are reported directly after being measured. For instance in a deployed scenario that implements a VA inside a TEE, it is possible to transmit the measurements directly after they were acquired and either store and maintain an SSR in a memory location protected by TEE or conduct a verification of the measurements directly. In these cases, the security guarantees provided by the isolation technique used applies. To conclude, once a measurement or SSR is no longer under the direct control of the SuE, an attacker must first compromise the VS in order to manipulate the results during the attestation.

Security during Measurement

In its simplest form the MA could be a measurement and anchoring implementation inside the observed program itself. But, this solution is considered unreliable, since an adversary that takes control of the execution flow, could simply disable the measurements and anchoring functionality altogether or always only anchor well-known forged measurements; consequently, isolation techniques must be used to isolate the measurement and anchoring process from the actual program execution.

However, there is one major difference to the isolation techniques used to separate SuE and VS for verification. An MA cannot be deployed in a strictly isolated distinct location.

This is because only the OS kernel has the semantic knowledge to support or successfully conduct a measurement of any program under its control due to its responsibility to manage the virtual memory. Thus, the OS kernel represents the single instance that has knowledge about internal structures of a program under its control and, hence, an observer outside of this context has only very limited capabilities in order to find or interpret any relevant information. For this reason, a deployment relying on a strict physically isolated MA is considered as impractical. To conclude, the MA always relies on the information provided by the responsible OS kernel running and managing a program. This is indeed a significant limitation because strict physical isolation would provide the highest reliability of measured information. Still, different isolation techniques can be used to

increase the reliability of the measurement.

Isolation Techniques The weakest form of isolation is provided by the enforced process isolation inside the SuE. That is, only a privileged process may access memory of an unprivileged process. This isolation is enforced by the OS kernel by providing access control functionality to the relevant system calls for memory interactions.

The next level of isolation is provided by the clear distinction between user space and kernel space. A process within user space may not access kernel space memory unless explicitly granted by defined APIs. Virtualization provides the next level of isolation and is also managed by the OS kernel and relevant APIs that guarantees that a VMM may not access a different VM's memory or the memory of the VMM unless explicitly provided by an API in a controlled way. In addition to these three isolation techniques, hardware-supported extensions can be used to further increase the isolation level. However, the main problem with these hardware-supported technologies is that they operate very often in a separate enforced physical or logical memory area and do not allow direct access to arbitrary memory controlled by the OS kernel. This means, in many cases, that these technologies cannot conduct a measurement on their own, because they lack the contextual information managed by the OS kernel. In fact, this strong isolation in both directions is the main benefit they provide. Although many technologies, like for instance a TEE, have unrestricted access to all memory in a system, it still relies on external support from the OS kernel to get the internal contextual knowledge of memory to access relevant information.

Interestingly, some information is available without that contextual knowledge, for instance, if the OS kernel itself resides in a fixed memory area. If the exact location and the size of the kernel are well-known, a measurement of the Kernel's code segment can be conducted from a TEE or any other system able to access the memory. For every other OS component, i.e. LKM and managed processes, it might be still possible to conduct a measurement, but an implementation of this solution would by extremely difficult. This is because the TEE operates on its own representation of virtual to physical memory and, hence, must implement its own contextual view of the OS memory before it can even locate the necessary high level structures maintained by the OS kernel. In the scope of this thesis, TEE and other self-reliant isolated MA implementations are not considered due to their complexity; this topic is left open for further research.

To conclude, although the basic measurement operations, i.e. reading arbitrary memory and computing a digest, can indeed be conducted inside distinct isolated components, they must be instrumented by the OS kernel or, at least, utilize high level structures managed by the kernel. As a result, isolated measurement operations do not provide a significant benefit regarding the reliability of the measured information themselves.

This means the presumed highest possible isolation level for an MA, besides the measurement of the kernel itself, is ultimately represented by the OS kernel for non-

Table 4.3 – Reliability in non-virtualized Environments.

Environment	Measurement		Isolation Technique	Reliability
	Agent	Target		
NV 1	User space	User space	No Isolation	very weak
NV 2	User space	User space	Process Isolation	weak
NV 3	Kernel space	User space	Kernel/User space Isolation	strong
NV 4	Kernel space	Kernel space	Kernel/User space Isolation	medium

Table 4.4 – Reliability in virtualized Environments.

Environment	Measurement		Isolation Technique	Reliability
	Agent	Target		
V1	VMM User space	VM Kernel / User space	Virtualization	strong
V2	VMM Kernel space	VM Kernel / User space	Virtualization	very strong

virtualized environments. Therefore, reliable information can only be provided by the OS kernel, and in particular, by an isolated implementation inside the OS kernel space, as depicted in Table 4.3. However, it should be noted that these are assumptions which are neither confirmed nor refuted and are therefore mentioned again as an open research topic in Section 8.2.

Reliability Classification in non-virtualized and virtualized Environments The non-virtualized environment NV 1 is classified as very weak since the exploitation of the Measurement Target, i.e. the program to be measured, also compromises the MA. NV 2 is classified as weak because it would allow to compromise a Measurement Target with a high privilege level which could be used, for instance, to terminate or restart the MA arbitrarily. Moreover, MA itself could become the victim of an attack, which would inevitably lead to unreliable measurements. NV 3 represents the strongest reliability, since it would require a kernel level exploitation to circumvent or disable the MA. NV 4 is classified as medium because although it requires a kernel exploitation, the measurement target itself is also inside the kernel space and thus a potential victim of the attack. Still, it is considered more reliable than NV 2 since a successful exploitation for NV 4 is far less likely than gaining higher access permissions for a process.

Accordingly, a classification of virtualized environments is depicted in Table 4.4 representing an MA deployed on a VMM and a Measurement Target inside a VM. In cases where MA and Measurement Target are deployed inside a single VM, the classification from Table 4.3 applies.

V1's reliability is classified as strong, because it would require an exploit inside the vir-

tualization code of the VMM. Moreover, V2's reliability is considered as very strong, since it would require an additional exploit inside the virtualization code's kernel component in order to circumvent or disable detection, which is very unlikely.

Still, it has to be noted that deploying an MA inside a VMM to measure user space programs inside a VM is nearly as complex as described for the approaches that are based on hardware extensions. The major difference is that the MA inside the VMM can facilitate the Virtual Memory Management information directly by introspecting the VM's memory and *page tables* in the relevant memory of the VMM. The semantic disconnection between a VMM and a VM is known under the term *semantic gap*, cf. [95], and further addressed by Pfoh in [96]. Similar to the hardware extensions, the specific mechanisms for a self-reliant MA in a VMM are considered as out of scope in this thesis and left open for further research, cf. Section 8.2.

4.3.3 Design Space and Architectural Limitations

The DRIVE Architecture is based around the concept of continuously monitoring a system and conducting an attestation based on the accumulated measurements from the SuE. A core concept of DRIVE is that the Measurement and Verification can be separated; thus, DRIVE enables an external observer of a system to determine its state. On the one hand, this enables DRIVE to detect code injection attacks and code pointer manipulation attacks which happen in predictable memory areas. This is the unique characteristic provided by DRIVE, since the architecture was designed to specifically solve this open problem. On the other hand, DRIVE is not designed to solve the detection or prevention of code reuse or non-control data attacks. While detecting or preventing non-control data attacks is still under research with no proposed solution, code reuse attack can be mitigated by applying described CFI counter-measures as described in Section 3.2.4.

CFI is an orthogonal technology which does not incur any impact to DRIVE. Both approaches are complementary and significantly enhance system security, especially when used alongside each other. DRIVE's responsibility in this case is to monitor the state of the system over a longer period and CFI is used to further reduce the risk of the system being compromised by an adversary.

Similar to DRIVE's architectural design, CFI counter-measures must also be specifically designed to enable a successful detection and prevention. In particular, CFI must overcome certain limitations and constraints, in order to be usable in practice. In particular these involve:

(1) Contextual Runtime Information: distinguish between valid and invalid branch targets that depend on specific information only available during runtime in the current context

(2) Effective time and duration: time the actual attack is carried out and how long it lasts

(3) Scalability: time or computational effort necessary to prevent the attack

Regarding (1), the branch target modifications usually happen in highly dynamic data, for instance the `Stack` or `Heap`. This means that an external observer cannot easily determine whether a branch target is valid or not, since the contextual runtime information, necessary to make this decision, is usually not available. Consequently, it is infeasible to predetermine valid branch target addresses without this information. Even though necessary information itself could be accumulated outside of the current software's runtime scope, for instance the kernel measuring a process, the information would still lack required cohesiveness. Thus, an external component would need to establish necessary relations manually based on the measured information by itself, which would basically require a simulation of the software's exact state at this time.

Regarding (2), the time between the modification of the branch target and the execution of the corresponding code is almost always very short. Especially inside the `Stack`, the altered execution flow becomes effective almost immediately, for instance as soon as the current function returns in its epilogue.

Most significant in this regard is that the software is executed on the CPU without any further supervision. Although the OS determines through its scheduling when a software is executed it does not supervise its execution. This means that once the software is executed on a CPU, the OS is not further informed which exact operations the software is executing at the moment. Apart from explicitly executing operations that are implemented by the OS, for instance system calls, I/O operations or waiting, the OS has no exact knowledge or control about the instructions currently executed.

Consequently, the operating system does not know when a branch will occur, when a function will be called or when a function will return; this would require a debugging session with single stepping through the individual instructions. As a result, it may happen that a code reuse attack is executed or has already finished, before the OS even gets the chance to detect it. For this reason, the only practical approach, without supervising individual instruction execution, is to explicitly invoke a validation function as part of the software's program logic and execution flow. Whether these validation functions are an implicit part of the current software to be validated, provided by the OS or realized otherwise does not make a difference in this regard.

Scalability (3) is another important constraint that must be considered when applying CFI counter-measures. As mentioned earlier, it is indeed possible to accumulate all necessary runtime information and supervise software instructions by single stepping them in a debugging session. But, from a scalability point of view, the solution would become arguably infeasible due to the significant overhead required to distinguish between a valid and invalid branch, right before it is executed. Since the prevention of code reuse attacks is

the main goal of CFI counter-measures, they must be applied before the attacker controlled code is invoked. As a result, the validation of the branch target should be conducted as fast as possible, to allow the software to continue its desired functionality.

Research demonstrates that the overhead introduced by various CFI counter measures differs a lot, although they are already implemented efficiently as part of the program's logic. Burow et al. reported that the geometric average for the SPEC CPU2006 benchmark varies between 1.1% and 5.5%, cf. [53]. Supervised instruction analysis and validation are not considered in any of the CFI counter-measures. It is unclear whether they were not considered due to the expected massive overhead they would incur or simply overseen.

To conclude, it seems validation processes for CFI are always implemented as part of the program logic; obviously supervision of software on instruction level is too slow for a practical solution. In addition to that, the required contextual runtime information is already present in this execution scope. Thus, no simulation or additional computations need to be done in order to simulate the current runtime information. Moreover, the necessity of attack prevention is significant for code reuse attacks. If Code Reuse is implemented correctly, it leaves no trail inside memory and, thus, a detection of the code reuse attack after an attack cannot be proven in most cases. As a result, for a feasible CFI counter-measure, validation logic must be part of the software's inner logic. Thus, external analysis is unfit for this task and can only be used to detect repercussions of the code reuse attack. For these reasons, DRIVE is unfit to provide a practical solution regarding the protection against code reuse attacks.

Therefore, it is generally advisable to use existing CFI solutions for protecting a system against code reuse attacks and DRIVE for long-term monitoring of the system state.

4.3.4 Summary

This section provided an analysis of the developed attestation concept in regards to the instantiated architecture. Based on available components and isolation mechanisms, the reliability of the attestation result varies. At first, an analysis for different possible deployment scenarios dependent on the isolation between SuE and VS was presented. As discussed, the level of isolation between both systems involved affects how an attestation can be conducted and how reliable the attestation result is. The highest level of reliability can only be achieved if both systems are physically isolated and a remote attestation is applied. However, if strict physical isolation is not possible, other isolation mechanisms, for instance virtualization or hardware-based isolation, can be used to provide more reliable results instead.

Following, isolation mechanisms during measurement and reporting were analyzed and discussed. In cases where the SuE does not provide a discrete security module, for example a TPM, alternative technologies should be considered. For instance an implementation that facilitates a TEE or similar hardware-based technologies. To conclude,

measurement must be stored in a tamper-resistant way to provide meaningful information and the reliability of the measurements corresponds to the security properties provided by the technology used. The measurement process itself must also consider isolation mechanisms, but is mainly limited to security guarantees provided by the OS kernel of the SuE. At least for runtime information regarding processes, a measurement cannot be easily conducted by an external system or component; this data is managed by kernel internal dynamic data structures. This means that although it is technically possible to conduct a measurement from an OS kernels' outer scope, the measurements would not become more reliable. For the given reasons DRIVE implements the MA inside the kernel space and provides a reasonably high level of assurance for its measurements.

To complete this section, an analysis regarding the architectural limitation were discussed. Since this thesis addresses runtime information a comparison was made between DRIVE and CFI. Both concepts are complementary but have no overlapping characteristics, they address different data and states. On the one hand, CFI is tailored to make the system more resilient regarding code reuse attacks; DRIVE has not the technical capabilities to detect or prevent code reuse attacks at all. On the other hand, DRIVE is tailored to monitor continuously the state of a system over a long period. In this case CFI lacks technical concepts to facilitate such a monitoring and state determination. This means that both concepts should be used in parallel to maximize the level of security.

4.4 Concept and Architecture Summary and Conclusion

This chapter presented a high level attestation concept and architecture for the work to be developed in this thesis. At first, a high level attestation concept was introduced and described. An attestation involves two systems: 1. SuE and 2. VS which is used to determine a trust state of a SuE. This trust state is determined by analyzing acquired relevant information and based on three basic mechanisms: 1. Measurement, 2. Reporting and 3. Verification. The measurement and reporting is conducted on the SuE and used to obtain and present the accumulated measurements to a VS. The VS, in turn, is responsible for verifying these measurements, on the basis of well-known reference information present.

Second, the architecture of DRIVE was introduced. The architecture describes different building blocks that implement the mechanisms necessary to successfully conduct an attestation for the systems involved. The architecture defines an MA that implements all mechanisms necessary to acquire, securely store and report the relevant information from a SuE with the help of a security module, and a VA that is deployed on a VS and that implements the corresponding verification routines.

Third, DRIVE's concrete instantiation of this architecture was presented. For this purpose, specific components and data structures were defined and necessary mechanisms

discussed. The components on the SuE are: 1. DMC and 2. Security Module that eventually creates an SSR that can be verified by an external system. The verification is conducted on the VS by DVC. It uses the SSR to determine the trust state of SuE and relies on described well-known reference data. In addition to that, the attestation process implemented by the defined components was briefly described.

Fourth, an analysis of the architectures' implementation and deployment was carried out. This was done by considering different alternative isolation techniques during the attestation. In particular a distinction between the measurement and verification process was made. In both cases, the level of isolation determines whether an attestation result is reliable or not. The general rule that applies during verification is: The higher the isolation between the systems is, the more reliable are its results. Consequently, the highest level of assurance is provided by strict physical isolation of SuE and VS. Measurement and reporting mechanisms rely on the secure accumulation and storage of the measurements. Secure storage capabilities are provided ideally by a discrete security module, for instance a TPM, but other implementations can also be considered if a discrete security module is not available. The measurement process, in turn, is limited to isolation provided by OS kernel. This is because external systems lack contextual information necessary and thus always rely on information provided and maintained by the OS kernel. For this reason, a kernel space implementation of DMC is recommended and used by DRIVE.

Following, a design space analysis was described that presents architectural limitations for DRIVE. In particular, during the analysis a comparison with CFI solutions was made and discussed where DRIVE and CFI are typically applied. The conclusion was that both concepts are complementary and address different runtime information; hence, they should be applied in parallel. CFI to make SuE more resilient against code reuse attacks and DRIVE for continuous monitoring of the trust state.

Chapter 5

DRIVE Measurement, Verification and Reporting Concept

In this chapter the technical details of the measurement, verification and reporting concepts are presented, developed and analyzed which are necessary to carry out reliable attestation of runtime memory artifacts for determining the trustworthiness of a SuE.

The overall objective of this chapter is to provide a technical concept that adopts and refines the described architecture from the previous section. For this purpose, it is important to describe the required mechanisms in detail in order to derive a complete solution, which can be implemented in specific software components. These specific implementations may use different methods and components of measurement acquisition and algorithms for verification; nevertheless, the goal is not only to precisely describe the technical details, but also to facilitate different implementations of the defined concepts.

In addition to that, all concepts will be analyzed with regard to their capabilities and limitation relating to various attacks. For this purpose the hybrid attack scenarios defined in Section 3.3.4 are used to refine the concepts and determine their effectiveness regarding the detection of given attack techniques.

The solution developed in this chapter is a core contribution of this thesis. Consequently, the solution, along with its dependencies, assumptions and limitations, represents DRIVE's fully designed and developed protection technology. This builds the basis for the implementation and evaluation phase of this work, which is described in Chapter 6.

5.1 Attestation of Static Information

This section introduces the technical attestation concept to measure, report and verify static memory portions, i.e., segments and sections. As explained in section 2.4.4, the different segments and sections of a program behave differently, i.e. they are either static or dynamic. Furthermore, it has already been discussed that static memory portions

always behave predictably. This is because the compiled and linked program is not meant to change during runtime and, thus, the memory content of these areas is an exact copy of the content within in the ELF.

In particular, the `.text` segment of link-time-relocated RCC and PIC are always static and therefore predictable. This means that the corresponding loading mechanism extracts the `.text` section from the ELF and loads its content into memory without any modification. For this reason, the information used during verification can be directly derived from the ELF; no further transformation is necessary.

Accordingly, static memory areas are the simplest to measure, report and verify in this work. For this reason they are introduced first. In addition to that, this section introduces details used to construct an SSR. The SSR consists of two parts, 1. a security module anchored Fingerprint and 2. a DML comprising all measurements. In particular, this section explains how the information in the DML must be arranged to provide a more effective anchoring mechanism.

This section also establishes a remote authentication protocol that is used for secure transmission of the information collected between SuE and VS. The protocol itself is responsible for requesting and receiving a fresh attestation from SuE to apply a verification process. This verification process is explained in detail afterwards and concludes this section. For this purpose, the integrity verification of an SSR is explicitly discussed and finally the verification process of DML comprised measurements are explained.

5.1.1 Measurement of Static Memory Areas

The measurement of static areas – or more precisely segments or sections – in memory is straightforward provided that sufficient access permissions, i.e. reading rights for the memory portions, are set. As explained, every segment or section resides in such a memory area; moreover, the memory area is defined by a well-known memory address and a particular size.

As depicted in Figure 2.1, a memory segment is a composition of different corresponding ELF sections. For instance, the well-known Unix shell Bash defines the `.text` segment as a composition of the following sections: `.interp`, `.note.ABI-tag`, `.note.gnu.build-id`, `.gnu.hash`, `.dynsym`, `.dynstr`, `.gnu.version`, `.gnu.version_r`, `.rela.dyn`, `.rela.plt`, `.init`, `.plt`, `.text`, `.fini`, `.rodata`, `.eh_frame_hdr` and `.eh_frame`. As a result, these sections form the `.text` segment that is loaded into memory in the ELF's Program Headers, as indicated in Listing 5.1.

```
Program Headers:
  Type   Offset             VirtAddr           PhysAddr
         FileSiz            MemSiz              Flags  Align
  LOAD   0x0000000000000000 0x0000000000400000 0x0000000000400000
         0x00000000000eefdc 0x00000000000eefdc  R E    200000
```

Listing 5.1 – Program Header Excerpt from the `readelf` output of the `.text` Segment of the Unix Shell `Bash`. This information is used by the loader to load the `.text` segment into the system memory.

The first step during the measurement process is to read the targeted memory area. In order to do that, the in-memory `.text` segment's initial loading address and size must be determined. In the given example, the initial loading address and size are already known. The memory start address is `0x400000` and the size is `0xeefdc` (978908 Bytes). In cases where PIC code is available and ASLR is activated, the memory addresses cannot be acquired by the ELF file. In these cases, the memory layout of the process must be analyzed to determine the relevant memory start addresses. This information is available by using tools like `pmap` or by consulting the mapping available in the `/proc/` file system.

Accordingly, once the targeted memory address and the size of the to-be-measured memory area is determined, the measurement process executes designated functions provided by the OS. As discussed during the Architecture Deployment Analysis in Section 4.3, one particular way of reading the memory area is to use the `ptrace` [97] system call on Linux, which corresponds to the NV2 case providing weak reliability of the data, c.f. Table 4.3 and . A different approach is to conduct the measurement process inside the kernel, providing strong reliability of the measurements; this corresponds to the NV3 case in Table 4.3.

The second step during the measurement process is the generation of the measurement itself. One possibility, which is used by many memory forensic tools and approaches (e.g. [89], [16] and [90]), is to create a snapshot from the targeted memory area without any modifications or reduction functions. As expected, the created snapshot has the same size as the measured memory area. As a result, the measurements become very comprehensive and, thus, do not scale well for use-cases that attest a system repeatedly. In order to reduce the size of the measurements, while preserving information measured, DRIVE's measurement process uses Cryptographic Hash Functions (CHF) that provide a secure and reliable representations of the measured content. The concept of applying CHF to measurements is well-known; yet, no known related work applies CHF to measure and verify individual memory segments or sections. In fact, CHF-based measurement and verification of runtime memory contents with dynamic behavior are not considered at all in previous or recent work. In the given example, the generated hash digest for the measured static code part of the Bash is: `be2a741a5df4d05b80cdb62b6ee086cf41af1d8c83702840aa7b0ce6666f493e`[16].

[16] The value can be compared to the hash extracted during reference value generation in Section 5.1.4.

Accumulation and Management of Measurements

Until now only the program's `.text` segment located in memory has been considered. But, the memory layout of a program usually consists of many other `.text` segments that belong to shared libraries used by the program. For instance, Bash depends on the shared libraries `libtinfo.so.5`, `libdl.so.2`, `libc.so.6` and `vdso`. In addition to that, Bash also depends on the loader `ld-2.19.so` and additional libraries that are dynamically loaded during runtime via `dlopen()`. These libraries are: `libnss_files-2.19.so`, `libnss_nis-2.19.so`, `libnsl-2.19.so` and `libnss_compat-2.19.so`[17].

As it turns out, all `.text` segments of the shared libraries and the loader `ld-2.19.so` happen to be also static. Consequently, all related `.text` segments from the referenced libraries are also measured as previously described. Even though the measurement process is similar on all different hardware architectures, details like memory addresses or architecture specific behavior slightly influence the measurement process. A concrete instantiation of the measurement concept is presented in Chapter 6.

Encapsulation of Measured Information The measured information (mi) can be designed in a flexible way, tailored to the particular type of the memory area. Yet, only the memory start address (msa), memory size (ms) and the measured hash digest (mhd) have been considered. But, in addition to that, further metadata, such as memory end address (mea), access permissions (map) and, if available, the related mapped filename (mf) can also be gathered and added to all measurements. However, in the particular case, i.e. measurement of static memory areas, the mhd is the only value necessary during a verification. In other words, mhd is the only mandatory information necessary to verify measured static content. Consequently, all other metadata information is optional for the time being and will be considered in the other verification methods discussed later if they contribute to a concrete verification procedure.

As mentioned in Chapter 4, all measurements are stored in an ordered DML whereas every single list entry forms a set S with variable information. In this particular case, measuring static memory segments, the set S is represented by:

$$S = \{mhd\}$$

A more general form of the set S can be defined as $S = \{mi_0, mi_1, \ldots, mi_n\}$, whereas mi_n represents the variable content, for instance: mhd, mf, msa, ms, map. As an example, a set S comprising all previously introduced measurement information can be represented as:

$$S = \{mf, msa, mea, ms, map, mhd\}$$

In addition to set S, a Measurement Set (MS) is defined that comprises individual sets

[17] Please note, this is only an excerpt. There are additional shared libraries that are not mentioned.

of S. In other words, the Measurement Set acts as a container for the individual sets that encapsulates related measurements in a tree data-structure.

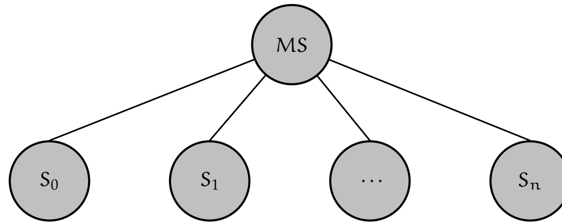

Figure 5.1 – Measurement Set Composition in a Tree Structure. The Measurement Set MS represents the Root Node and is a Composition of the individual Measurement Sets $S_0 \dots S_n$.

As depicted in Figure 5.1, the Measurement Set MS forms the root of a tree structure and the individual sets $S_0 \dots S_n$ represent its children, so that:

$$MS = \{S_0, S_1, \dots, S_n\}$$

This encapsulation makes it possible to put related individual measurements in a particular logical context and thus a Measurement Set can be used to represent instantiated programs like for instance a process of an OS. But, most importantly, the Measurement Set enables a significant improvement of the anchoring process, which will be discussed in the following Section 5.1.2.

For instance, a measured Bash process is represented by a single MS, that consists of $6 + n$ individual Sets S representing the individual `.text` segments. S_0 for the Bash program, S_1 for the loader, $S_2 \dots S_5$ for shared libraries always loaded and $S_6 \dots S_n$ for dynamically loaded shared libraries.

Dynamic Measurement Lists As a consequence the DML structure is defined that composes all Measurement Sets $MS_0, MS_1 \dots MS_n$, so that:

$$DML = \{MS_0, MS_1, \dots, MS_n\}$$

The DML itself is a chronologically ordered list. This means that the order of the DML is defined by the sequence in which the Measurement Sets have been inserted in this list: $MS_0 < MS_1 < \dots < MS_n$. The strict adherence to the defined order is very important during the verification process because the integrity of the DML will be audited during the verification phase. Without the strict adherence to the order, the integrity verification would become impractical for numerous encapsulated Measurement Sets. This will further be discussed in Sections 5.1.2 and 5.1.3.

Continuous Measurements As expected, and in contrast to established static integrity measurement concepts, like for instance IMA, DRIVE is not limited to onetime measurement and can thus conduct its measurements anytime and repeatedly after the software component is fully loaded. As a result, DRIVE can be used to perform continuous monitoring of relevant to-be-measured parts of the system. DRIVE's measurement process can be triggered at any time. For instance the measurement process can be hooked to or triggered after relevant system calls, like e.g. `mprotect` or `dlopen`, or run on a timer, executing the measurements on a defined time interval. The concrete trigger mechanism or the interval between subsequent measurements cannot be defined in general terms, since the definition depends on the actual use-case and its security requirements. For some systems it may be reasonable to conduct a measurement every time a particular process is scheduled. In turn, for other systems, a time interval ranging from a couple of second to hours may be sufficient. Similarly, other systems do not require an automated measurement at all and trigger a measurement manually or whenever the system is meant to be attested.

In conclusion, defining when and how often a measurement should happen must be defined for the particular use-case. From a conceptual perspective, the only important factor for DRIVE is that the architecture and implementation do not restrict when or how often the measurement is done.

5.1.2 Reporting of Measured Data

In the following, the concept of the reporting mechanism is explained and details of the related anchoring process presented. As explained, the encapsulated measured information does not affect this process and thus it is equal for static and dynamic measurements. In particular, it is described how the SSR is constructed to enable an integrity verification based on an anchored value.

As discussed in Section 4.3.1, DRIVE's architecture encourages a design to separate measurement and verification processes preferably on two distinct physically isolated systems. In order to report the MA measured data to the VA, the SSR has previously been defined and introduced. The SSR is a data-structure that contains a DML and a security module's protected fingerprint that enables an integrity verification process of the DML.

In other words, the measurement process implements an additional step during the reporting mechanism to anchor the Measurement Sets MS in a security module. Consequently, this enables a VA to verify whether the reported DML was illicitly modified during the reporting or not.

One particular problem with anchoring Measurement Sets is that the anchoring mechanism is a sequential operation by nature and, more importantly, may take long to complete. For instance a TPM, which is designed to support this anchoring mechanism as one of its core functionality, takes about $10ms - 15ms$ to complete the anchoring operation. Without

Table 5.1 – Hashed Measurement Sets HMS represent a Hash Value of the Data of all Measurements S entailed in a particular Measurement Set MS. | denotes Concatenation.

Hashed Measurement Set	Measurement Set	Entries
HMS_0	$digest(MS_0)$	$digest(MS_{0S_0} \| MS_{0S_1} \| MS_{0S_{...}} \| MS_{0S_n})$
HMS_1	$digest(MS_1)$	$digest(MS_{1S_0} \| MS_{1S_1} \| MS_{1S_{...}} \| MS_{1S_n})$
...	...	...
HMS_n	$digest(MS_n)$	$digest(MS_{nS_0} \| MS_{nS_1} \| MS_{nS_{...}} \| MS_{nS_n})$

the encapsulation, each Set $S_0 \dots S_n$ had to be anchored individually, resulting in an significant overhead caused by the anchoring process. For this reason in particular, the design of the Measurement Sets MS was adopted as containers for the individual measurements $S_0 \dots S_n$.

The anchoring process is as follows: For every individual Measurement Set MS a hash digest is calculated that creates a hashed measuremet set (HMS), so that:

$$
\begin{aligned}
HMS_0 &= digest(MS_0) \\
HMS_1 &= digest(MS_1) \\
&\dots \\
HMS_n &= digest(MS_n)
\end{aligned}
$$

HMS_n now represents a value that enables the integrity verification of each individual MS_n and, therefore, all included measurements as depicted in Table 5.1. Note, since a Measurement Set only represents a container, the actual operation calculates the hash digest over the encapsulated sets[18] $S_0, \dots, S_n$ so that:

$$digest(MS_0) = digest(S_0|\dots|S_n)$$

All calculated Hashed Measurement Sets HMS are then extended into the security module preserving their initial order as defined by the DML. The extend-function for the security module's anchored fingerprint (FP) is defined as follows:

$$
\begin{aligned}
FP_0 &= digest(00000000000000000000|HMS_0) \\
FP_1 &= digest(FP_0|HMS_1) \\
&\dots \\
FP_n &= digest(FP_{n-1}|HMS_n)
\end{aligned}
$$

Accordingly, FP_n securely reflects a fingerprint for all Measurement Sets $MS_0 \dots MS_n$, including the most recent MS_n and all previous measurement sets $MS_0 \dots MS_{n-1}$ in a

[18] | denotes concatenation

single value. Therefore, FP_n provides integrity protection over the entire DML including all comprised individual measurements.

For this reason, the verification process can rely upon the fact that if the integrity protected by FP can be successfully proven, all data in the DML can also be treated as reliable and untampered.

As a final data-structure, the SSR is defined. It comprises the latest anchored fingerprint FP_n and the entire DML:

$$SSR = \{FP_{DML_n} \cup \{DML_0, \ldots, DML_n\}\}$$

The SSR is then transmitted to the VA and used during the measurement verification to determine the SuE's runtime state. The transmission of the SSR to the VA will be discussed Section 5.1.3 and the verification in Section 5.1.4.

It has to be noted, that the anchoring process and the construction of the SSR is independent from the actual content, because it only operates on defined abstract data-structures. This is because the introduction of the Measurement Sets decouples the information necessary for the management of the DML and its actual content. This means that although this section focuses on the measurement, reporting and verification of static memory contents the described construction of the SSR, DML and its anchoring process remains the same for all other measured memory contents types, whether static, predictable or unpredictable. As a result, only the content of the encapsulated sets, i.e. $\{\{S_0, \ldots, S_n\} \forall MS_n\}$ will change based on the type.

5.1.3 Remote Attestation Protocol

The *remote attestation protocol* links together the attestation process between SuE and VS. Thus, the main goal of the protocol is to exchange an SSR as it represents the main data-structure for the later verification of the system state. The SSR itself should be transmitted over a secure channel from SuE to VS. Therefore it is assumed that a secure channel has been established, for instance, by means of a Transport Layer Security (TLS) connection between SuE and VS.

Figure 5.2 depicts the sequences of the remote attestation protocol for DRIVE. In particular, the sequences involve the following individual steps:

Step 1: Initiate the protocol by sending a Nonce from VA to MA. The Nonce is later used to verify whether the generated SSR belongs to the current session for replay protection and to verify that a fresh SSR was generated.

Step 1.1: Generate a fingerprint $FP_{Nonce AK}$ that protects the integrity of the DML by means of the received Nonce, sign with a cryptographic private attestation key AK and add it to SSR.

Step 1.2: Extract the DML and add it to SSR.

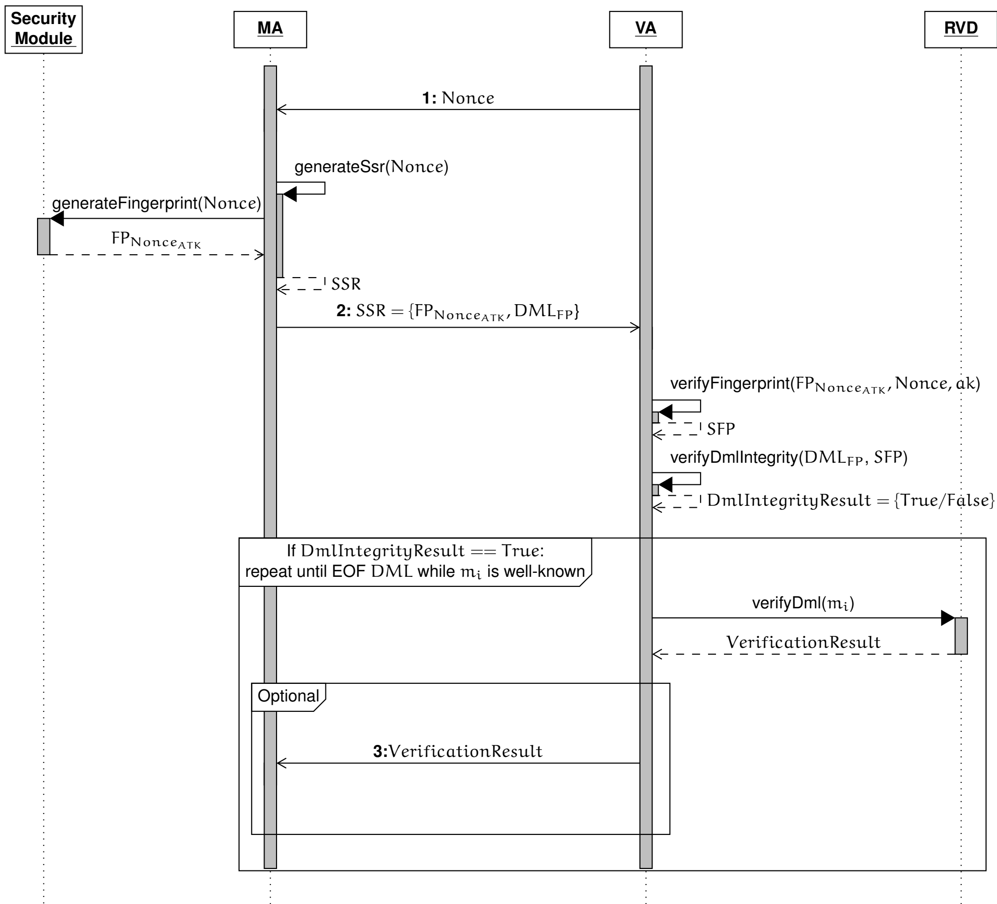

Figure 5.2 – Remote Attestation Protocol for DRIVE to transmit an SSR from MA to VA with optional Result.

Step 2: Send SSR to VA.

Step 2.1: Verify the signature of the fingerprint based on public attestation key ak and the generated Nonce with the operation verifyFingerprint($FP_{NonceAK}$, ak). If successful, the fingerprint is trusted and addressed as SFP.

Step 2.2: Verify the integrity of DML on the basis of SFP.

Step 2.3: Verify the DML by iterating over every individual Measurement Set $MS_{0...n}$ under the assistance of Reference Database.

Step 3: (optional) Report the verification result to MA to trigger additional mechanisms.

Following, all steps but 2.3 are described in more detail, because these are independent from the actual MSs encapsulated inside the DML and thus remain the same for different encapsulated contents of a DML. The actual verification of the MSs, i.e. step 2.3, however, varies depending on the MSs' types. These type-based content verifications are described in Sections 5.1.4, 5.2.2, 5.2.4 and 5.3.2.

Step 1, 1.1 and 1.2 SSR Generation

Step 1 initiates the remote attestation protocol by sending a Nonce from VA to MA. Using this Nonce is very important for remote attestation since it adds freshness as a property to the protocol. Therefore it is possible to detect replay attacks and to make sure that the SSR has been generated during the initiated session. In particular, the Nonce is used as an input parameter for the `generateSsr()`-operation in Step 1.1 that invokes the actual `generateFingerprint()`-operation in 1.2 provided by the security module. For instance, in case a TPM is used, the `generateFingerprint()`-operation corresponds to the `getQuote()` TPM operation. The fingerprint itself is maintained automatically in the security module and used implicitly by the `generateFingerprint()` operation. It represents the most recent security module Anchored Fingerprint FP_n. Additionally, the `generateFingerprint()`-operation takes the provided Nonce and the internal fingerprint FP_n and generates the data-structure $\mathsf{FP}_{\mathsf{Nonce}_{AK}}$ by calculating a digital signature over both values on the basis of the private key AK that is only available inside the security module.

Accordingly, $\mathsf{FP}_{\mathsf{Nonce}_{AK}}$ is returned to the `generateSsr()`-operation that creates an SSR that consists of:

$$\mathsf{SSR} = \{\mathsf{FP}_{\mathsf{Nonce}_{AK}}, \mathsf{DML}_{FP}\}$$

The value DML_{FP} represents the most recent DML and all of its encapsulated values. As a final step, SSR is sent to VA as a response message.

Step 2.1: DML Fingerprint Verification

The verification of the Fingerprint uses the transmitted signed fingerprint $\mathsf{FP}_{\mathsf{Nonce}_{AK}}$, the Nonce generated by itself in the initial Step 1 and the public key ak that is well-known. At first, the `verifyFingerprint`-operation loads the public AK key ak and verifies the digital signature of $\mathsf{FP}_{\mathsf{Nonce}_{AK}}$. Only if the digital signature has successfully been verified, the process continues.

Next, the `verifyFingerprint`-operation verifies that the message received, belongs indeed to the current session. As mentioned, VA generates for each session and has knowledge of the current Nonce in Step 1. During verification the transmitted Nonce_{AK} is extracted from $\mathsf{FP}_{\mathsf{Nonce}_{AK}}$ and compared to the VA generated Nonce. If both values are equal $\mathsf{Nonce}_{AK} == \mathsf{Nonce}$ the received fingerprint is generated by using Nonce and, hence, belongs to the current session.

After the successful verification of the digital signature of the fingerprint $\mathsf{FP}_{\mathsf{Nonce}_{AK}}$ and its encapsulated Nonce, both values are considered reliable and are hereinafter referred to as Signed Fingerprint (SFP) and Signed Nonce (SNonce).

Step 2.2: DML Integrity Verification

If a VS wants to attest the state of a remote SuE, the fingerprint SFP signed by the security module and the entire DML are mandatory. The sole purpose of the SFP is to verify the integrity of the DML. In order to verify the correctness of the DML, an expected fingerprint value, i.e. CFP, is computed by simulating the anchoring functionality as described in Section 5.1.3. If CFP is equal to SFP, the DML has not been altered and its entries reliably reflect the current system state.

Accordingly, the first stage of each DRIVE verification process is to verify the integrity of the transferred DML on the condition that only the information from SSR is used. As each measured hash digest is anchored in the tamper-proof security module, each measurement is also part of the signed fingerprint SFP. Consequently, SFP is sufficient to verify the integrity of the DML.

Consequently, CFP is calculated based on the reported DML. This calculation is carried out by extracting every encapsulated reported Measurement Set MS from the SSR's DML and by computation of a hash digest on the basis of these values, applied in the same sequence as extracted. In other words, the same functions must be applied on these values and in the same order that was used during the security module's anchoring process. The hash generation is defined as $CFP_n = \mathtt{digest}(\mathtt{digest}(MS_{n-1})|MS_n)$ for both the anchoring and the verification process.

$$CFP_0 = \mathtt{digest}(00000000000000000000|HMS_0) \quad (5.1)$$

$$CFP_1 = \mathtt{digest}(CFP_0|HMS_1) \quad (5.2)$$

$$\dots \quad (5.3)$$

$$CFP_n = \mathtt{digest}(CFP_{n-1}|HMS_n) \quad (5.4)$$

The process is as follows: 5.1 generates the first intermediate CFP_0 by generating a hash digest on the start value concatenated with HMS_0; 5.2 generates the subsequent CFP_1 by concatenating CFP_0 and HMS_1 and applies a hash function; 5.3 repeats step 5.2 for any subsequent CFP; 5.4 generates the final CFP_n by concatenating CFP_{n-1} and HMS_n. A concrete example is presented in Table 5.2.

After CFP is calculated, it is compared against the reported SFP. If, and only if, the expected value is equal to the reported, that is $CFP_n == SFP_n$, the DML is considered as untampered and hence benign. This implies that the DML represents the measured runtime system state at the time the report was generated. After the integrity of the DML is proven, the verification process continues to verify every individual measurement, described in the following section.

Table 5.2 – DML Integrity Verification: Calculation of an expected value CFP compared to a reported Value SFP on the basis of Measurement Sets.

Computed Fingerprint CFP	Hashed Measurement Set HSM_i	Measurement Sets $MS_{i_{0...n}}$
Starting Value: `000000000000000000000000000...`		
d92e7c344f84b5d192d57baba89f...	1856d531548655b76d35105c318b...	$MS_{0_{0...n}}$
7725dd11391eb230fdd346895c9c...	ef411bae164fd624ea94fc9ef82f...	$MS_{1_{0...n}}$
8835a4fa603e0f50bbe96df0dc52...	bd32e452e14f84eb22d6ac9e9e1c...	$MS_{2_{0...n}}$
3ca003b89aaf977a8654f205e948...	eefd4a6bebd6b001ff587c2335a3...	$MS_{3_{0...n}}$
bf24d17301b89eaa485ed6c7a5be...	cd7c653f0a6691c0d723393bc732...	$MS_{4_{0...n}}$
CFP (expected **SFP**): **bf24d17301b89eaa485ed6c7a5be...**		

Step 3: Report Verification Result to MA

Optional Step 3 can be used to trigger additional mechanisms. Bearing in mind that the DML can become indefinitely large, a mechanism can be implemented that addresses scalability with respect to transferred data and the computational effort involved in verification. The basic idea of this mechanism is to transmit only the difference between the previous and the current DML, which is referred to as ΔDML. ΔDML is created by building the relative complement between the previously verified $DMLi - 1$ and the current DML_i so that:

$$\Delta DML = DML_i \setminus DML_{i-1}$$

Therefore, any subsequent attestation process following the first transmits ΔDML to VA instead. During the DML Integrity Verification in step 2.2, the VA computes CFP by using a stored value SFP_{i-1} as its start value in 5.1, so that:

$$CFP_0 = \mathtt{digest}(SFP_{i-1}|HMS_0)$$

The rest of the operations remain equivalent as described. In case the overall attestation process is successful, SFP_i is generated, is stored on VA and used during the subsequent attestation process to calculate CFP_0. For the described process, VA must always store the current SFP, but other mechanisms can be used to make the protocol stateless. For instance, VA could encrypt or digitally sign SFP, and send it alongside the verification result to MA. In turn, MA adds the encrypted or digitally signed SFP to the SSR and sends it in step 2 back to VA, so that:

$$SSR = (FP_{Nonce_{AIK}, \Delta DML_{FP}, SFP_{i-1}})$$

In this case, VA extracts SFP_{i-1} from SSR after the fingerprint verification step 2.1 and uses SFP_{i-1} as described. By adopting such an additional mechanism, the DML contains unverified or, in other words, unseen MSs only. This means that the DML's size is always

minimal, or the DML is empty, in case no new measurement processes were triggered between two distinct attestation processes. As a result, this decreases the amount of transmitted data and the verification performance significantly. In addition, the MA can clean up the current DML and save a considerable amount of memory by removing MSs from the DML that have already been verified and thus are no longer required.

5.1.4 Verification of Reported Static Measurement Data

DRIVE's verification concept is based on the idea of unaltered reference measurements. As previously described, the measurement process calculates a hash over predictable measured memory areas. This means that in case the verification process can (1) calculate an expected hash digest (ehd) based on a reliable source and (2) successfully compare ehd against the measured hash digest mhd, DRIVE can determine whether mhd has illicitly been altered or not. In the following section, a description of the individual verification steps is provided and, moreover, details on the mechanism of obtaining the necessary reference values are presented.

DML Measurement Verification

The verification of the DML encompasses all individual measurements $MS_{0...n}$ depending on the content of the measured memory portion. In this section, verification schemes for static memory portions, i.e. segments or sections, are explained. This is done in correspondence with the DML verification Step 2.2 `verifyDML()` function as depicted in Figure 5.2.

The verification is based on the principle of reference value calculation derived from ELF files by: (1) Data extraction of segments, sections and metadata from ELF files, (2) composition and modification of extracted data to represent measured memory portions, and (3) calculation and comparison of reference values against reported information to make a decision about the reliability of the SuE

As mentioned earlier, in the case of static measurements the reference values do not depend on any dynamic data from SuE. For this reason, the reference values can be generated independently any time before being used during a verification. The process of the reference value generation in this case is based on the extraction of the static data, more specifically the static program text encapsulated inside the `.text` segment, from the ELF file and the generation of a hash digest on the basis of this extracted data.

Static Program Text Extraction All information necessary to extract the targeted data is present in the `ELF-Header`. Basically, DRIVE utilizes the same information as the ELF file loader. In Figure 5.3, an excerpt from the Program Header of the ELF executable file */bin/bash* on a X86_64 system is shown. The type LOAD instructs the loader to load

Type	Offset	Size	Flags
LOAD	0x00000000	978908 Bytes	R-E
LOAD	0x000efdf0	36536 Bytes	RW-

Figure 5.3 – Relevant Program Header Information from */bin/bash*.

the program text into memory. The relevant information DRIVE uses are (1) the Offset, determining the position of the first to-be-loaded program text byte and (2) FileSize determining the number of to-be-loaded bytes relative to the given position. As indicated in Figure 5.3, the `.text` segment starts directly at the beginning of the ELF file, i.e. Byte `0`, with the size of 978908 Bytes.

In order to calculate the reference value for the relevant segment, the program text is extracted by loading the file and extracting the data according to the determined offset and size. An extraction of the program text is typically performed by the reference value generation application, but can also be done by the Unix tool `dd`, as exemplified in Listing 5.2. The command extracts the targeted program text, starting at offset `0` (defined by `skip=0`) and reading `978908` Bytes from the file. The extracted data is saved to a temporary program text file (tptf) `extracted_text_segment` that is used for generating the corresponding reference value.

```
$ dd if=/bin/bash of=extracted_text_segment bs=1 skip=0 count=978908
978908+0 records in
978908+0 records out
978908 bytes (979 kB) copied, 6.31637 s, 155 kB/s
```

Listing 5.2 – Using `dd` to extract the `text` Segment from Bashs' ELF file.

Reference Value Generation and Verification of static DML Entries The verification of individual DML entries is the core verification process that uses pre-calculated reference values or triggers an ad-hoc reference value calculation process to derive them on demand. For the reference values of static information, pre-calculated reference values are already generated on the basis of `tptf`. This is done by applying a hash function on `tptf` so that $RV_{\mathtt{tptf}} = \mathtt{digest}(\mathtt{tptf}_{\mathtt{file}})$.

Typically, reference values are calculated on a reference system in a trusted environment or derived otherwise from a trusted source, for instance a digitally signed software package that was received from a trusted third-system. For instance from an official or internally maintained software repository. In the current example, the command presented in Listing 5.3 generates a reference value for extracted static part `tptf` from Bash. As expected, the generated hash value is equal to the measured hash value from Section 5.1.1.

```
$ sha256sum extracted_text_segment | cut -f 1 -d " "
be2a741a5df4d05b80cdb62b6ee086cf41af1d8c83702840aa7b0ce6666f493e
```

Listing 5.3 – Generating a `sha256sum` of an extracted Bash `text` Segment.

The verification process iterates over the entire DML and compares the measured static program text digests to the already calculated reference values for every individual measurement $S_0 \dots S_n$ of every entailed Measurement Result $MS_0 \dots MS_N$. If, and only if, the complete DML can be verified successfully, that is for all measured digest a valid reference value based digest was found, the SuE is considered as trustworthy.

5.1.5 Summary

In this section the technical concept attesting static information has been developed and discussed with the goal of assessing the system state of a SuE. For this purpose, all involved processes, i.e. measurement, reporting and verification were defined, introduced and presented. In addition, the internal structures and procedures for the implementation of the mentioned process of attestation were explained and discussed in detail. At first, the measurement process was developed and discussed. Necessary for the successful application is the memory address and size of the targeted memory area. Given sufficient access rights, OS-related functions can be used to access and read the targeted content to eventually create a hash digest of the read content. This measured cryptographic hash digest now represents the main information used during verification in order to determine if the content is considered trustworthy or not. As stated above, no other known related work uses CHF to measure and verify individual memory segments or sections.

Next, the accumulation and management of measurements was presented. To facilitate a structured accumulation of the measurements different container structures were introduced. One top level structure was defined as a DML, a chronologically order list, which encapsulated all measurements in a structured and flexible manner. During its construction, which is a dynamic process due to the possibility of continuous measurements, so called measurement sets that represent all measured memory portions are anchored into a security module, as part of the described reporting process. This anchored value, defined as the Fingerprint, can be used to prove the integrity of the DML during verification and is encapsulated in the defined SSR together with the DML.

Subsequently, a remote attestation protocol was specified and discussed in detail. The protocol involves different steps to transmit the SSR from the MA to VA for verification. Most importantly, the goal is to derive a fresh, authenticated and integrity proven DML. For this purpose, all relevant protocol steps were explained. In particular, the process verifies a Nonce, a digital signature of the Fingerprint provided by the security module and, finally, compares the signed Fingerprint to a self-calculated Fingerprint based on the transmitted DML. If all involved steps are successful, all necessary properties are proven and the encapsulated measurements can be verified. In addition, a mechanism was introduced to minimize the amount of data transferred and data to be verified. This was achieved by an additional optional protocol step, which incorporates the results of previous attestations and can therefore make use of a relative complement of a DML.

Furthermore, the verification process of the measurements was presented. However, before the verification, an independent upstream process is carried out to generate reference values, which are later compared to the measurements. The generation of reference values for static memory portions extracts corresponding portions of the related ELF file and generates a hash digest of this extracted content. Thus, the generated hash digests represent the reference values used during verification. The final step during the verification is the actual comparison of the reference values for each encapsulated measurement in the DML. In case all measurements were successfully compared with the generated reference values, the system state of the SuE is considered trustworthy.

5.2 Attestation of Predictable Dynamic Information

The attestation of predictable dynamic information considers different specific areas inside system memory. This work will analyze and discuss two identified particular predictable areas. First, `.text` segments of RCC-based programs are analyzed and discussed. These programs are relocated during their actual loading process[19]. In contrast to the previously discussed link-time-relocated RCC or PIC, additional information must be gathered and considered during the measurement process and added to the reporting, in order to enable a successful verification. Second, Global Offset Tables (GOT) that are used to perform on-demand function resolution during the runtime of programs are analyzed and presented. For this purpose, measurement and verification processes involve additional steps and data-structures that must be analyzed, collected and verified.

For these reasons, this section is structured to first analyze and discuss the entire attestation process of RCCs and afterwards the entire attestation process of GOTs. Although both of these different predictable areas share some data-structures that are used in both attestation processes, an independent analysis and discussion is more appropriate. However, both attestation processes do not require any change to the described reporting mechanism from the previous Section 5.1.2, only the measurement values in the sets S require additional data to be added. These additions will be discussed in the related measurement sections of each attestation.

5.2.1 Measurement of Relocatable Code

The measurement process of RCC is very similar to the one described in Section 5.1.1. The main difference is that the mapping start address $\mathtt{msa}$ inside the defined set S becomes mandatory for each individual $S_0 \ldots S_n$ composing a Measurement Set MS that represents an RCC based program with dependencies to external functions.

In other words, this means that the corresponding MS is composed of set $S_0 \ldots S_n$ that contains at least $\mathtt{mhd}$ and $\mathtt{msa}$ as mandatory information besides other optional fields,

[19] If not indicated otherwise RCC always refers load-time relocation in this thesis

such as mf, ms and map. Accordingly, the measurement process must first determine the dynamically assigned msa and the ELF-based size of the program, then read the memory content and generate a hash digest of the identified memory portion. Therefore, this measurement process is very similar to the described procedure for static measurements for PIC, besides the necessity to add msa to MS.

Alternatively, it is also possible to reverse the relocation of RCC during the measurement process. In this case, the measurement process must first identify all relevant patched symbol addresses and replace their contents with `0x0`. The identification information of the relevant symbol addresses is available in the ELF file's relocation section, explained in more detail in Section 5.2.2. After reversing the relocation, the RCC becomes static and, thus, equal to its ELF file representation. Consequently, msa is not needed during the verification and can be omitted in the corresponding MS. However, there are some disadvantages that stem from applying a reverse relocation during the measurement. First and foremost, security relevant information is lost during the process. For instance, if an attacker replaces only relocated symbol addresses, the attack itself cannot be detected during a verification. This is because the malicious addresses would be overwritten during reverse operation process. Second, the reversing process increases the computational effort during the measurement and must be repeated during continuous monitoring for every individual measurement of the RCC. Although the individual operations necessary to conduct the reversing are simple and not expected to influence the computational effort hugely, the necessary logic during measurement is expected to become more complex in comparison to a measurement without. As initially described in Section 5.1.1 the measurement process without reversing relocations is very simple since all information is readily available in individual high-level data-structures.

For these reasons, DRIVE does not apply a reverse relocation during the measurement process. Instead, the verification process applies an ad hoc relocation on the basis of msa and the designated ELF file. The verification process is described in the following Section 5.2.2.

5.2.2 Verification of Relocatable Code

Reference Value Generation for Relocatable Code

To calculate reference values for RCC, a correlation between the loaded ELF file and the reported information MS must be established. The first step during the ad hoc reference value calculation is the extraction of the related memory start address msa from MS. After msa is extracted and tptf loaded by the verification program, the corresponding loading process is simulated using the following steps: (1) Load relocation rel from the ELF relocation sections, i.e. `.rela.text` or `.rela.dyn`, (2) extract symbol file offsets sfo from rel, (3) locate referenced symbol offset rso in ELF symbol tables, i.e. `.symtab`,

(4) calculate absolute symbol address $\mathtt{asa} = \mathtt{msa} + \mathtt{rso}$, and (5) patch tptf at position sfo with asa accordingly.

Once every relocation has been applied in tptf, the hash digest is calculated by using: $RV_{RCC} = \mathtt{digest}(\mathtt{tptf})$. However, the calculated RV_{RCC} is only valid for the given RCC with exactly this msa. With regards to LKM, where this process is typically applied, this means that unloading and reloading the LKM would render the RV_{RCC} outdated, because the memory start address would most likely change. In this case, RV_{RCC} must again be recalculated on the basis of the new msa.

Verification of Relocatable Code

As the analysis shows, load-time-relocation is not used for user space program text today. Therefore, the RCC load-time relocation concept has been reimplemented for LKM. LKM and shared program text relocation in user space are very similar; however, LKM utilizes additional indirection through trampoline-jump tables in certain cases[20]. Those mechanisms are very architecture-specific and hence the technical details are not discussed in detail. Nevertheless, DRIVE's concept is implemented as described in Section 5.1.4, corresponding to the architecture specific behavior. During verification, the LKM ELF file is first loaded from the persistent storage maintained by the reference value database. Once loaded, the described mechanisms are applied: extract the `.text` segment from the ELF file and save it in a temporary file; obtain the relocation, analyze it, and calculate the symbol target address based on DML information; finally, patch the calculated symbol address into the temporary file. Afterwards, the following steps are performed: calculate the expected hash digest ehd, and compare it to mhd. The measured program text is considered trustworthy if and only if both values are equal.

Verification Kernel Image As previously mentioned, the Linux kernel image is statically linked to predefined addresses. This means its `.text` segment does not depend on addresses determined during runtime; thus, it is possible to calculate a reference value for the kernel in advance, add it to the reference value database and just compare mhd from the DML to the reference value. Accordingly, a mechanism calculating a kernel reference value hash digest was implemented so that the verification component is able to detect illicit runtime tampering of the kernel's `.text` segment.

But, in contrast to statically linked user space objects, like ordinary executable ELF files, the kernel implements sophisticated fix-up mechanisms, self-applied very early during its initial loading phase, as described by Kittel et al. [98]. These fix-ups are not only architecture-dependent, but also specific to the particular CPU and MMU on the system and, thus, this behavior renders the reference value generation quite complex. On the

[20] E.g. in PPC 32 certain relative jumps exceed a maximal jump-length of 24 Bit for a target symbol address. This can only occur in kernel space, due to its size.

basis of the kernel's source tree, the fix-up mechanisms was re-implemented to facilitate the calculation of reference values for the evaluated systems in advance. As it turned out, the kernel's reference value remains constant for a particular hardware configuration. Thus, an alternative approach for reference value generation is to conduct a one-time measurement on a trusted system and use this trusted measured value as the reference value for the particular kernel and hardware configuration.

In addition to that, the kernel implements under certain configurations a mechanism called *runtime-patching*. If enabled, the kernel maintains during its runtime certain internal tables, deciding whether a particular function is used. For instance, the kernel has the ability to activate certain debugging capabilities dynamically during its runtime. Activating a function means triggering a functionality that replaces the corresponding function address code pointers within the .text segment. One example for this behavior is utilized by the kernels function tracer `ftrace`. This tracer can be enabled for any function call inside the Kernel to enable dynamic debugging.

Moreover, the kernel also supports a mechanism called *Code Label Patching*, which works quite similar to the `ftrace` function patching. In this case, however, only particular function pointers are replaced. Furthermore, in cases where the kernel is executed on top of a Hypervisor as a VM, additional modifications to the kernel are applied. Depending on the actual features and capabilities of the Hypervisor used, different values inside the kernel are adjusted. It has further been found that this behavior may also change with the version of the hypervisor software used. Still, the described runtime patching and on load patching mechanisms do not result in unpredictability of the kernel's text segment. In this thesis, however, the kernel's runtime patching mechanism is not further explored and would go beyond the scope of this work. The reason for that is that the modifications rely too much on implementation details of the kernel and other software. This means that there is no general approach on how to solve this issue and, thus, every kernel on a particular system must be analyzed properly as soon as runtime patching capabilities are enabled.

5.2.3 Measurement of Global Offset Tables

The GOT is represented as a separate section (`.got`) inside the composed `.data` segment. This means that the measurement process itself cannot simply measure the entire `.data` segment, defined by a start and end address, but has to determine the exact location and size of the `.got` section and conduct the measurement on the basis of this information. In addition to that, this process becomes even more complex because the exact location of the `.got` depends on runtime information due to ASLR. This means that in order to actually measure the `.got`, its start address and size must be determined at runtime, and identified on the basis of the allocated dynamic addressing. The necessary offsets and addresses, however, are maintained in internal data-structures and are used to calculate the exact

memory locations. In order to measure the `.got`, the relevant in memory ELF section, i.e. the `.dynamic` section, must first be located, analyzed and interpreted correctly. Based on this information the allocated memory area of the `.got` can be calculated. In a final step the measurement of the GOT's content and relevant metadata is conducted.

In principle, the GOT's content measurement is very similar to the already described measurement process for static information, c.f. Section 5.1.1. Once the exact location and size are determined, the measurement process generates a hash digest of the GOT's content and adds this information to the defined set S as mhd. Besides the GOT's content hash, the access permissions may also be measured and added as metadata map to S.

However, additional metadata is necessary for a successful verification, because the verification process needs to know whether a function was already resolved or not.

As a result, the set S_{got} consists of only the necessary mhd and, optionally map:

$$S_{got} = \{mhd, map\}$$

It has to be mentioned that the measured GOT information is not single-handedly sufficient for its verification. Since the `.got` contains resolved addresses to library functions that reside at random memory locations – the location depends on addresses allocated during loading – the DML's measurement set MS of a process must include at least all msa of all loaded shared libraries to support a successful GOT verification. This means, for the described bash process measurement from Section 5.1.1, that the DML's MS consists of S_0 and $S_{0_{got}}$ for the Bash program, S_1 for the loader[21], $S_2 \dots S_5$ and $S_{2_{got}} \dots S_{5_{got}}$ for shared libraries always loaded and $S_{6_{got}} \dots S_{n_{got}}$ for dynamically loaded shared libraries.

5.2.4 Verification of Global Offset Tables

Reference Value Generation for Global Offset Tables

The verification of `.got` relies on different information about the measured process. Specifically, all processes' shared library code sections' memory addresses must be known during verification. The other necessary information is contained in the ELF files involved in the process execution and gets extracted during verification.

The verification itself depends on the re-calculation of the GOT based on the library memory start addresses (msa_{lib}), the resolved symbol's offset ($offset_{symbol}$) and the GOT's location address $GOT_{address}$[22].

The required memory loading addresses msa_{lib_n} are part of the SSR's DML, as depicted in Figure 6.3, and available during verification. The GOT's location address $GOT_{address}$ and the symbol name can be derived directly from the ELF file's relocation

[21] `ld` is statically linked and thus has no GOT.
[22] Used to determine the order of the `.got` table.

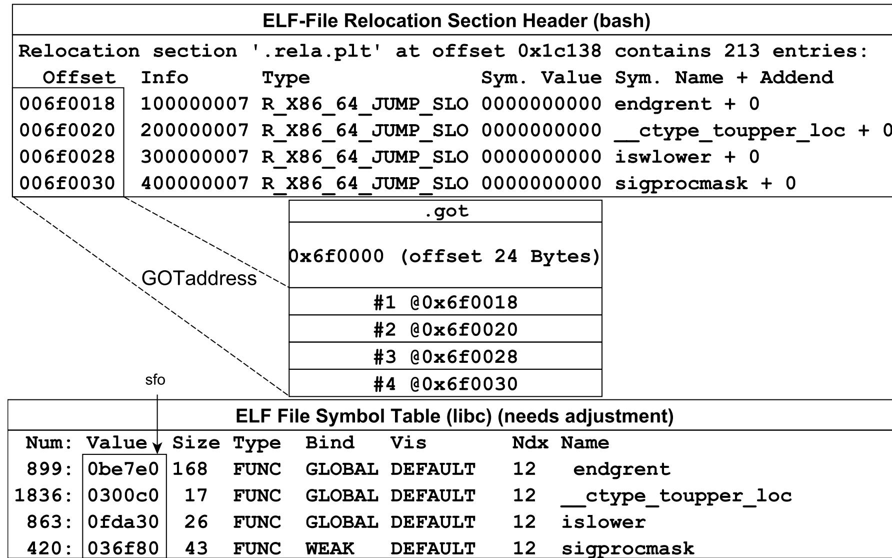

Figure 5.4 – Verification Excerpt of the GOT Layout and Symbol Resolution for `/bin/bash` 4.3.

section's header (`.rela.plt`). Most importantly, the symbol file offset (sfo) can be derived from the symbol tables, i.e. `.symtab`, of referenced library ELF files by their symbol name. Once the required information is extracted, the expected `.got` entry address can be calculated by:

$$GOT_{asa} = msa_{lib} + sfo$$

The process is repeated for every single relocation entry in the order given by $GOT_{address}$. Once all got addresses are calculated and sorted, CHF can be applied in order to calculate the `.got` reference value RV_{GOT} as follows:

$$RV_{GOT} = digest(GOT_{asa_0} || GOT_{asa_{...}} || GOT_{asa_n})$$

Finally, it can be compared against the measured hash digest of the `.got` (mhd_{GOT}).

Verification of Global Offset Table contents

The verification of the `.got` is an important operation and always applied for PIC and link-time-relocated RCC by DRIVE. As depicted in Figure 5.4, the symbol name and the GOT's location address $GOT_{address}$ can be extracted from the ELF file of the GOT target.

For instance, the `endgrent` function used by `/bin/bash` is implemented in `libc-2.19.so` and its designated GOT address $GOT_{address}$ is `0x006f0018`. By analyzing the `libc-2.19.so` ELF file, the `endgrent` symbol resolves to the offset `0x0be7e0`, relative to the loading address `0x7fbd32f4b000` [c.f. Figure 6.3, 5.4]. Thus, the resolved address for the `endgrent` function is `0x7fbd330097e0`.

Table 5.3 – GOT Calculation for `/bin/bash`.

Symbol Name	sfo	Library Name	msa_{lib}	$GOT_{address}$	GOT_{asa}
endgrent	0x0be7e0	libc-2.19.so	0x7fbd32f4b000	0x006f0018	0x7fbd330097e0
__ctype_toupper_loc	0x0300c0	libc-2.19.so	0x7fbd32f4b000	0x006f0020	0x7fbd32f7b0c0
iswlower	0x0fda30	libc-2.19.so	0x7fbd32f4b000	0x006f0028	0x7fbd33048a30
sigprocmask	0x036f80	libc-2.19.so	0x7fbd32f4b000	0x006f0030	0x7fbd32f81f80

Based on this, Table 5.3 shows the calculated GOT which is in line with the measured GOT. It has to be noted that before the CHF can be applied on the calculated GOT, the target architecture's endianness must be considered; this means, if the target architecture uses little-endian, the resolved addresses must be converted into that format.

5.2.5 Summary

In this section the technical details of attesting predictable dynamic information was introduced, explained and developed on the basis of two examples. At first, the attestation concept for RCC was presented. For this purpose, the measurement and verification process was analyzed and developed. The measurement of RCC was found to be very similar to static memory information. The main difference is that for RCC it is necessary to add the memory start address in the defined measurement set for each measurement, because the verification process relies on this information. Next the verification process was presented. It was found that up to the point of extracting the loaded memory content, the reference value generation process is equal to the previously developed process for static memory reference generation. However, after the extraction of `tptf`, RCC requires a simulation of its loading process based on the determined msa encapsulated in the measurement set. After the loading process was simulated and `tptf` patched accordingly, a hash digest was generated and compared to the measurement. In case of a successful comparison, the measurement was considered as trustworthy. Applying a reverse-relocation during measurement was also discussed, but, in conclusion found to be less secure than the presented solution.

Second, the attestation concept for GOTs sections was analyzed and developed. In this case, the measurement process needs to first locate the `.got` section inside the `.data` segment. Because `.got`'s size and location in memory is dynamically appointed, msa and ms were calculated on the basis of information available in the `.dynamic` section during runtime. After msa and ms was determined, the `.got` was accessed, read and a hash digest calculated based on the read content. Afterwards, the verification process was presented. The reference value generation relies on a calculation of an expected `.got` section on the basis of the programs' ELF file and the transmitted information present in the DML encapsulated measurement sets. Once this information is available, the expected `.got` can be generated and a hash digest calculated that is compared against the measurement during verification. Additionally, it was found that the verification of

GOTs requires the information of all msa's of measured shared libraries. As a result, if a GOT verification should be applied, all measurement sets must also include these corresponding memory start addresses.

5.3 Attestation of Unpredictable Dynamic Information

The previous sections considered predictable information that was fully encapsulated in memory areas, i.e. segments or sections. Since the complete memory portions considered fulfilled the property of predictability, it was possible to generate reference measurements. This generation was done either by direct extraction out of an ELF file or by applying different functions that mimic a particular predictable behavior. For instance, applying a relocation process for RCC or mimicking the dynamic library function resolution process. In contrast, it is not possible to rely on reference values for the entire target area for memory portions that consist of or contain unpredictable information. As a result, these areas must be attested based on other information, i.e. metadata, gathered during the measurement process. Still, all unpredictable areas are composed of unpredictable and predictable information. This means that although the complete portion cannot be correlated to a reference measurement, there are some parts inside the area that are predicted and, thus, can be verified based on their content.

In this section the technical attestation concept of unpredictable dynamic information is analyzed and presented. This involves the measurement and verification of unpredictable memory areas based on metadata. Similar to the attestation of RCC, the attestation process of unpredictable information does not require a revision of the described reporting mechanism in Section 5.1.2. Only the measured information, encapsulated in set S, must be adapted accordingly. Moreover, this section will provides some examples for encapsulated predictable information and discuss to what extent an attestation of the predictable information would be meaningful. Since the predictable encapsulated information is tightly coupled with the observed component, there is no general solution for an attestation of encapsulated predictable information. As a result, the section will particularly focus on the metadata-based attestation approach and only briefly touch on the attestation of predictable encapsulated information.

5.3.1 Measurement of Unpredictable Dynamic Information

The content of unpredictable dynamic memory portions does not provide any conclusion towards the current integrity state of the measured program or the system. For this reason the contents of those memory areas are not measured and considered in DRIVE. Still, in some cases, there may be unpredictable areas composed of both, predictable and unpredictable sections or data. In this case, depending on whether these predictable portions can be identified, their content can be measured similarly to the described approaches in

the Sections 5.1.1 and 5.2.1. Still, deriving reference values for these areas may be very complex and, in many cases, it may not provide evidence towards the actual state of the system, incurred from the limitations of DRIVE's Architecture, described in Section 4.3.3. For this reason, the measurement of unpredictable segments, composed of predictable and unpredictable information, is considered to go beyond the scope of this thesis and is left open for further research.

Instead of measuring the content or parts of the content, DRIVE limits itself to the measurement of metadata for unpredictable memory areas. In addition to that, the metadata is measured for all described content-based measurements as well, since the metadata provides relevant information about the measured program and its current state. In particular, the metadata is measured by collecting the information from the control structures defined and managed by the OS. For instance, the memory access permissions (map) are maintained in the `task_struct` control structure inside the kernel and also exposed to processes fulfilling the necessary access permissions to read them. In user space, for instance, these mapping information are available in the `/proc/` file system and are accessible by root or the process owner. Regarding DRIVE's DMC implementation this means: If the DMC has access to the content of a memory segment, it also has access to all relevant management structures. Consequently, the information is accessed, read and added as a measurement information alongside other information about the particular memory area.

As mentioned earlier in Section 5.1.1, all information is accumulated in the DML in individual Measurement Sets MS, which, in turn, consist of the individual Sets $S_{0 \ldots n}$. Thus, a set S representing an unpredictable segment has the exact same structure as a set representing predictable segments, except the missing content-dependent memory hash digest mhd. As a result, the general form of Set S is used, so that:

$$S_u = \{msa, mea, map, mf, ms\}$$

Moreover, all other information except mhd is considered to represent metadata in terms of DRIVE, whereas the fields msa, mea and map are considered as mandatory for all segments and all other fields, e.g. mf and ms, are optional. The reason why msa and mea are considered mandatory for all MSs – when metadata verification is active – is a result of demonstrated *stash clash* attacks that are further mentioned in the corresponding verification in Section 5.3.2. Stack and heap grow towards each other and can either overlap at some point or overlap with other segments that are usually located between stack and heap in memory.

All other parts during measurement are equal as described in Sections 5.1.1 and 5.2.1. The unpredictable measured set S_u becomes part of a Measurement Set MS as described and, consequently, part of the corresponding DML.

5.3.2 Verification of Unpredictable Dynamic Segments

Reference Value Generation for Metadata

Metadata consists of data that is already known. This means that metadata can be extracted by analyzing relevant information based on ELF files, configurations, policies or properties at runtime. For example, the program headers depicted in Listing 5.4 contain information like: sizes of all encapsulated segments and sections and access permissions for segments. Similarly, the composition of the individual sections into segments is available in the Section to Segment mapping of the ELF file, as presented in Listing 5.5.

```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x000000000040 0x000000400040 0x000000400040
                 0x0000000001f8 0x0000000001f8  R E    8
  INTERP         0x000000000238 0x000000400238 0x000000400238
                 0x00000000001c 0x00000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x000000000000 0x000000400000 0x000000400000
                 0x0000000f3cac 0x0000000f3cac  R E    200000
  LOAD           0x0000000f3df0 0x0000006f3df0 0x0000006f3df0
                 0x000000008e98 0x00000000ea68  RW     200000
  DYNAMIC        0x0000000f3e08 0x0000006f3e08 0x0000006f3e08
                 0x0000000001f0 0x0000000001f0  RW     8
  NOTE           0x000000000254 0x000000400254 0x000000400254
                 0x000000000044 0x000000000044  R      4
  GNU_EH_FRAME   0x0000000d8c90 0x0000004d8c90 0x0000004d8c90
                 0x000000004094 0x000000004094  R      4
  GNU_STACK      0x000000000000 0x000000000000 0x000000000000
                 0x000000000000 0x000000000000  RW     10
  GNU_RELRO      0x0000000f3df0 0x0000006f3df0 0x0000006f3df0
                 0x0000000000000210 0x0000000000000210  R      1
```

Listing 5.4 – Program Headers from Bash ELF.

Moreover, the ELF also contains information about which additional dependencies are used by the analyzed program or shared library. Based on the extracted metadata, verification could specify the layout of the process in memory and also ensure that the program does not load any unexpected additional libraries, which could pose a security risk. However, many programs use dynamic loading of libraries, for instance by using the `dlopen()` function. In these cases, the information cannot be determined solely by analyzing the ELF headers, since the information is only available within the program code itself. This means that a source code or runtime analysis may be necessary to determine all dependencies.

```
Section to Segment mapping:
 Segment Sections...
  00
  01 .interp
  02 .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
     .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt
     .plt.got .text .fini .rodata .eh_frame_hdr .eh_frame
  03 .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
  04 .dynamic
  05 .note.ABI-tag .note.gnu.build-id
  06 .eh_frame_hdr
  07
  08 .init_array .fini_array .jcr .dynamic .got
```

Listing 5.5 – Section to Segment Mapping Information from Bash ELF.

Apart from the ELF file encapsulated information, other information can be considered to define additional metadata information or policies. For instance, if a system is known to use DEP, a general policy valid for all processes is that no writable segment is also mapped as executable. This policy applies in general for all processes, but exceptions could be defined for programs known to violate this policy due to their particular requirements, for instance writable and executable Heap for Java or interpreted languages.

In this thesis, however, only the following verification mechanisms are considered: (1.) Ensure that only valid executable segments of the program and related libraries are loaded, and (2.) only valid access-permissions are present for any mapped memory area. Both represent the most important metainformation to be verified. The information is extracted by analyzing the relevant information inside the described ELF file structures, by considering the system configuration, i.e. enabled DEP and based on a runtime analysis conducted at a reference system, to determine the dependent libraries not referenced in the ELF file.

Metadata Verification

As previously mentioned, the verification based on metadata is not limited to unpredictable information only. All VAS segments and sections, whether predictable or unpredictable, can and should apply a metadata verification. As discussed during the Security Analysis in Section 3.3.2 it is a common practice during an attack to alter memory permissions, for instance changing the access permissions of the `.text` segment to `rwx`. As such, detecting the violation of an access permission policy for any segment is a strong indicator that something malicious happened on the system. The same is also valid for unexpected loaded libraries, more precisely mapped executable segments, from unrelated files in the processes' VAS.

However, the main reason for metadata verification in DRIVE is that for unpredictable

memory areas it is the only reliable option that can be applied. In other words, since a content-based verification is not possible due to the unpredictability of the measured information, only a metadata verification provides significant results during the attestation process.

Hence, metadata verification is mandatory for non-reproducible unpredictable memory areas, i.e. in particular `data.data`, `data.bss`, `stack`, and `heap`. Moreover, it is optional and conclusive for all other VAS segments as well. As soon as an adversary is able to modify memory access permissions, every memory mapping can potentially be used for successful exploitation. For instance, executing shell-code would require access permissions (`rwx`) for `stack`, `heap` or anonymous mappings which is generally forbidden. For these reasons, DRIVE defines access permission policies for well-known segments and sections. Every deviation regarding the defined policy is considered a violation of the policy and detected during DRIVE's metadata verification.

For instance, one particular policy states that no memory segment or section – other than the `heap` in different circumstance, c.f 3.2.4 – may adopt the access rights `rwx` at the same time. Although there are security patches available for Linux that prevent the setting of both `w` and `x` permissions at the same time, c.f. [99] and Team PAX `mprotect` documentation [100], this enforcement is not widely adopted. As a result, the system does not prevent these special access permission settings, and therefore recognizing a `rwx` permission for a memory area – if not defined as an exception for described cases – violates DRIVE's access permission policy and is regarded as an attack on the system.

Similarly, a policy can be defined that prohibits the executable permission flag for all anonymous or file mappings. Since the content of these areas is also considered unpredictable – unless otherwise required, e.g. by hot-patching mechanisms – no reference value is available and therefore it cannot be determined whether the content is benign or not. Therefore, the appearance of a file or anonymous mapping with executable permission is also treated as a security violation by DRIVE.

While access permissions represent a strong indicator whether a system has been attacked, other metadata can also be analyzed during verification. For this reason, the verification process also verifies that only expected executable segments of related and well-known libraries are loaded. While it is perfectly possible that in some cases the exact dependencies cannot be determined during the analysis, an unexpected executable segment does not necessarily mean that a process has been compromised. However, the information can be used to either refine the list of expected dependencies or trigger an alarm to further inspect the system by an expert in the domain. In addition to that the verification process should also consider the memory start address (msa) and memory end addresses (mea), since different attacks were demonstrated that exploited a behavior where the Heap or Stack overlapped with other segments, known as *stash clash* attacks, c.f. [101]. For this reason, the verification process may check whether an msa overlaps with

an mea or a different measured segment in order to detect such stack-clash attacks.

Still, to conclude, the most meaningful metadata verification and, thus, most significant during metadata verification, is the access permission verification on the basis of defined policies.

5.3.3 Summary

In this section, the technical details for the verification of unpredictable dynamic information were presented, explained and elaborated. In contrast to the other concepts for static and predictable information, it was explained why a content-based approach is not practicable for unpredictable memory areas. For this reason, it was found that an attestation based on metadata is the only reasonable approach to provide evidence that contributes the to attestation of system state. Although it was argued that under specific circumstances a content-based approach could be applied, for instance if predictable and unpredictable information are mixed, there is no general concept on how to approach this specific issue. At first, the measurement process was introduced and explained. It was pointed out that unpredictable memory areas could be represented in set S with the exception that in this set the content-based hash mhd is omitted. Furthermore it was explained that for all other static and predictable memory areas it is meaningful to also include metadata in their corresponding sets. In particular, map, msa and mea were identified to be mandatory metadata which is meaningful for all measurements. Subsequently, the verification of metadata-based properties was analyzed and discussed. As pointed out, reference values, i. e. in this case reference data, can be obtained by analyzing the ELF file. In addition, reference data can also be obtained based on other information, for instance by defining policies that restrict invalid anonymous memory allocations or invalid combinations of access permissions. To conclude, metadata-based attestation cannot prove the system state beyond any doubt, but represents a strong indication whether a system was attacked or not. In addition, it is the only meaningful attestation mechanism for unpredictable memory portions and thus provides a huge contribution for increasing the overall system security.

5.4 Concept Security Analysis and Evaluation

This section provides a security analysis and evaluation of the attestation concepts presented in the previous sections. In Section 3.3 different attacks were introduced. These attacks will be revisited and used as the basis for the security analysis in order to determine which particular attestation concept is able to detect them.

As explained, the defined attacks represent a composition of different attack techniques and targeted different types of data. In particular, a distinction between (1) predictable static data, (2) predictable dynamic data and (3) unpredictable dynamic data was made.

In order to further distinguish between these three categories, two additional categories were used: (4) permissions metadata, that are related to changes to permission flags of a memory area, and (5) new mapping, which relates to an unintentional mapping within the process memory.

For the following analysis, it is assumed that the related information is successfully measured and reported to the attestation's verification service. However, since identified unpredictable dynamic memory contents are not measured, it is assumed that no content measurements are available during the related verification steps in this case.

5.4.1 Static Information Attestation Analysis

The concept for the static information attestation is able to detect deviations between a measurement and its related reference, an overview is presented in Table 5.4. Each attack that manipulates predictable static information will eventually be detected during verification. Hence, attacks A1 (Create Malicious Executable Segment), A2 (Inject Malicious Code in Arbitrary Memory Region) and A3 (Modify Code Segment to Change Semantics Maliciously) are covered partially depending on the concrete target of manipulation. If a manipulation was applied in static area that is considered as predictable, it is detected during static verification. In the other case it not detected.

It has to be noted that A1 also introduces a new executable mapping in the initial step of the attack. In this case, it is generally assumed that every executable mapping infers predictable static information. Hence, the static information attestation does detect two manipulations for A1. (1) A manipulation within the `.text`, i.e. a modification of an arbitrary function pointer in order to redirect the CFG to the newly added executable code inside the new mapping and (2) the new mapping itself.

However, it might be possible that during an attack a mapping is created that corresponds to a well-known reference value. For instance, a shared library could be loaded into a process' VAS that actually refers to a well-known reference, but not in the context of the measured process. Consequently, it depends on the actual implementation of a policy for these mappings. In particular there could be a strict policy implementation that would detect a violation in case of a mapping with a valid reference within the wrong process context or a relaxed policy which could not draw this conclusion. The major challenge in this case it to derive all valid executable code mappings within the process context. Sometimes, this is not easily possible, especially in cases where certain libraries are loaded dynamically with `dlopen` within the programs' logic. This is because in these cases the actual ELF file does not contain any information about the corresponding library and, thus, this information can only be derived during runtime or by explicit source code analysis. Nevertheless, this is not a conceptual flaw but a detail that must be decided and solved during the actual implementation. In any case, referencing different executable code in the program's intended execution flow always requires a modification of additional in-

formation. Consequently, the attack is covered as long as the manipulation appears in a predictable memory area.

Regarding attack A2 the manipulations described modify static predictable data, in particular inside the kernel's `.text` segment and system call table (`.rodata`). Both areas represent data that is considered static within this thesis. The attack A2 itself seems very similar to the previous attack A1. The major differences are that no additional memory mapping is required and the control-flow manipulation is applied in specific jump-table, that is the system call table located inside the `.rodata` section of the kernel. However, the most interesting part of this attack is that it introduces code inside a padding area. This renders the attack stealthier, but, above all, this affects the actual implementation of the actual measurement and verification concept. This is because all segments located inside the VAS are page aligned. For all segments marked as executable this means that malicious code can be hidden despite the necessity to apply persistent changes to its permissions. Regarding the measurement process, this means the implementation must always measure the last segment including its padding area. If padding areas are not considered during the measurement, malicious code can be injected without the possibility of detection. In addition to that, the program text within the ELF file is not always page aligned. This is indicated in the corresponding ELF `pheader` in the `p_align` flag. For this reason, the generation of reference values must consider the ELF-specific alignment and implement a manual padding algorithm if the ELF's `.text` segment is not page aligned. To conclude, the padding area must always be measured to prevent code hiding and the reference values generation process must explicitly page align all non-aligned ELF sections accordingly.

Regarding attack A3, no exceptions apply for the successful detection of the manipulation. In this case the semantics of the programs is altered inside the related `.text` segment directly by the attacker. As a result the manipulation infers a deviation between the measurement and its reference and, hence, is detectable in any case.

To conclude, attacks to predictable static areas, such as `.text` segments in user space, are generally covered by static information attestation during the verification process. Regarding introduced new mappings in the VAS unknown mappings are always detected, since no valid reference is available. However, for well-known but unintentional mappings the implementation must apply a strict detection policy. In cases where a relaxed policy is used, mappings that refer to a well-known reference cannot be correctly identified during verification and therefore detection is not possible. As expected, manipulations in predictable dynamic `.text` segments, for example in relocatable code in user space or for LKMs, cannot be detected during the static information attestation. Still, all previously described attack are also applicable in dynamic predictable memory areas. For this reason they are revisited and discussed in the next section.

Table 5.4 – Expected Detectability of Manipulations during Static Information Attestation (SIA), Predictable and Unpredictable Dynamic Information Attestation (DIA) and Metadata Attestation (MDA). Symbols: ✔ (detectable), ✗ (undetectable).

Attack	Manipulation in .../ Details	Policy	SIA	DIA	MDA
A1	Predictable static area `.text`		✔	✗	✗
	Predictable dynamic area `.text`		✗	✔	✗
	Added well-known unintended mapping	strict	✔	✗	✗
		relaxed	✗	✗	✗
	Added unknown mapping		✔	✗	(✔)
A2	Predictable static area `.text`,`.rodata`		✔	✗	✗
	Predictable dynamic area `.text`,`.got`		✗	✔	✗
A3	Predictable static area `.text`		✔	✗	✗
	Predictable dynamic area `.text`		✗	✔	✗
A4	Predictable dynamic area `.got`		✗	✔	✗
A5	Altered access permissions		✗	✗	✔
A6	Non-control data		✗	✗	✗

5.4.2 Predictable Dynamic Information Attestation Analysis

The presented concept of Dynamic Information Attestation supports two different modes of attestation operations. In particular, these are the measurement, reporting and verification of Relocatable Code and Global Offset Tables (GOTs). Since both methods are very different, they are discussed separately. First the Relocatable Code Attestation is discussed. Specifically, the attacks A1, A2 and A3 are revisited to solve the remaining issue regarding manipulations within predictable dynamic areas (`.text`).

Second, the GOT attestation process is discussed; specifically attack A4 (Modify Code Pointer Data to Call Malicious/Unintended Code) is analyzed because it involves only dynamic data manipulations that are not detectable otherwise.

Relocatable Code Attestation Analysis

The Relocatable Code Attestation is a dynamic variant of the Static Information Attestation described in Section 5.4. During the analysis, there remained some unsolved issues regarding the detection of manipulation of predictable dynamic data. Since the Static Information Attestation relies on a comparison between its measurement and a fixed reference value, it is not applicable for any dynamic information that alters the program during load or runtime. For relocatable code, function pointer addresses are resolved during its link or load-time. For link-time relocatable code, these addresses are determined during the linking phase. Once resolved these function pointer addresses become static and do not change further. Consequently, the link-time relocated code is already considered and behaves as described in Section 5.4. Load-time relocated code applies

its function resolution during its initial loading phase, and, as such, does not resolve to a unique and static reference. DRIVE introduces a particular ad hoc reference value generation process in Section 5.2.2 and a corresponding verification process utilizing the ad hoc reference measurement generation, c.f. Section 5.2.2, to resolve this issue. This enables DRIVE to apply an attestation process of load-time relocated code of user space programs and LKM. This procedure fully resolves the issues for the detectability of attacks A1, A2 and A3 for predictable dynamic `.text` areas. Accordingly, the attacks A1 and A3 are now fully covered by the analyzed concept. For A3, however, there remains another issue on closer inspection still, i.e. if a dynamic jump-table is used, this attack cannot be discovered during the Relocatable Code Attestation. For A3 this is less critical because the padding-injected code is detected in this case. But, considering also attack A4, which applies a modification to a jump-table only, A3 must be revisited again to resolve this last identified problem.

Global Offset Table Attestation Analysis

As mentioned, the assumption from A3 was that the padding-injected code was included in the execution flow by patching a particular jump table, i.e. the system call table of the kernel. But, as soon as we release ourselves from these specifics, other jump table could also be a potential target to include the injected code. For instance, the GOT of user space programs can be used for this purpose. This means the modification of jump-table are a generic threat that must be resolved by DRIVE. As long as the jump-table information is static, as initially assumed in A3, static attestation procedures are sufficient for detection. However, the most valuable and utilized jump-table for this purpose is the GOT which is either initialized with dynamic pointer addresses during load-time once or, in case lazy-loading is activated, even during the program's runtime.

For this reason, the attack A4 (Modify Code Pointer Data to Call Malicious/Unintended Code) was presented. This attack does not rely on any previously injected code. Instead, it utilizes an information disclosure bug in order to determine a dynamic valid function pointer and subsequently patches the GOT to further exploit the system. DRIVE considers and is able to detect any unintended modification to the GOT as described in Section 5.2.4. Consequently, the GOT Attestation concept is able to fully detect the applied manipulation of A4 and, in addition to that, resolves the remaining issue of A3, as long as it applies its modification in the GOT as well. There may exist other jump-tables that also rely on dynamic function pointer addresses, specifically if the program implements a particular jump-table on its own. However, in this thesis other dynamic jump-tables are not considered or explicitly addressed in the concept.

To conclude, Predictable Dynamic Information Attestation is a very important part within the DRIVE concept. Without it, many attacks to the system cannot be detected for different possible attacked targets. Table 5.4 depicts the capabilities of the DRIVE

concept considering both static and dynamic predictable information attestation concepts. With both concepts in place, all attacks from A1-A4 are now detected fully. Consequently, all predictable memory areas are fully covered by DRIVE and, thus, they can no longer be active in a system without the possibility of detection. Regarding attestation of content, DRIVE considers the most important parts utilized by programs today. Apart from runtime-patching, not considered in this thesis, the context-based attestation approach is fully exhausted. Next, the attestation concept for unpredictable dynamic data, specifically the attestation of metadata, is discussed and analyzed.

5.4.3 Unpredictable Dynamic Information Attestation Analysis

As mentioned earlier DRIVE is not designed to protect against manipulations inside unpredictable data for different reasons. Most importantly, unpredictable data cannot be correlated to any information without additional contextual knowledge. As soon as a program relies on or operates with arbitrary inputs, there is no reasonable way to generate reference values. Consequently, no comparison between a well-known reference value and a measurement can be done. In addition to that, the frequency of data-alteration during program runtime is too high and arbitrary. Considering that function call-related data may be present inside the `heap` or `stack` only for a very limited time and a function may be called at any time, it would be unreasonable to attest data expired already.

Metadata-based Attestation

For these reasons, content-based attestation is considered as unpractical for unpredictable data. If required, other concepts, such as CFI, must be applied in addition to DRIVE. Still, many attacks rely on modification of metadata of predictable and unpredictable memory artifacts. Most importantly the manipulation of access permissions is used for exploitation techniques today. In contrast to the actual content, the metadata information is well-known. For instance, it is well-known that a `stack` must not be writable and executable at the same time. As a result, DRIVE utilizes these well-known permission constraints as a reference for the comparison against the current measured access permission settings. If the values are the same, certain assumptions can be made. For instance, if the `stack` is currently not marked executable, injected code cannot be executed.

Attack A5 (Alter/remove memory protection, e.g., disable or circumvent DEP) represents an attack that is widely used today. In particular, the core idea of the attack is used by code reuse attacks to disable DEP in order to enable the further exploitation of a targeted system. Without a manipulation of access permissions, malicious code can be injected, but not executed on a system. Consequently, code injection attacks rely on a manipulation of access permissions in almost all cases[23]. The only attacks, which do not

[23] This is true for all programs that do not rely on `rwx` permissions on one or multiple memory segments.

rely on access permission manipulations are self-reliant code reuse attacks, c.f. attack A4 and non-control data attacks, c.f. attack A6.

Since the attack A5 is so widely adopted, DRIVE measures the access permission settings of all memory segments within the program's address space. During metadata attestation, the measurements are then compared against the well-known references. If a deviation from the reference is detected then the system is expected to be attacked and is no longer considered as reliable. With the addition of metadata attestation DRIVE provides a very strong procedure to detect access-permission manipulations. Consequently, the attack A5 is fully detectable without any exceptions. Moreover, the attacks A1-A3 also utilized attack A5 in their initialization step. Thus, these initialization attacks are also countered implicitly by the metadata attestation. Still, it has to be noted that metadata attestation does not replace the content-based attestation models. In fact, metadata attestation is most meaningful for attacks that do not provide any additional indication such as for unpredictable memory areas. More precisely, metadata attestation is most effective in cases a persistent manipulation is required. For this reason, a distinction between temporary and persistent access permission manipulation were introduced in Table 3.7. For A1-A3 and A5 Variant V2, the access permissions are classified as temporary, because the permissions can be restored to their original settings while the injected code is still active. For A5 Variant V1 and V3, the access permissions cannot be restored, because the attack would eventually end.

This means that the persistent manipulated access permissions can be measured and detected as long as the attack is active. In contrast to this, the temporary variants can only be measured and detected in a very limited time. This affects the actual implementation of DRIVE significantly, since every single access permission change would require a distinct measurement process. Under the consideration that changes to access permissions are not uncommon, the measurement process could become a bottleneck for the system performance. That said, it is perfectly possible to track all access permission changes within a system by hooking the corresponding `mprotect` system call. However, since other detection mechanisms are available for the temporary variants, it is not necessary to conduct a measurement for all access permission changes. Instead, an intelligent way would be to only monitor the changes to unpredictable memory areas, which should almost never happen during normal program operation. Still, this matter is left open for the particular implementation and the actual use-case.

Limitation for Unpredictable Dynamic Attestation

DRIVE's ability to detect attacks ends when there is a modification in memory that does not involve metadata manipulation for an unpredictable memory segment. In particular, this is the case for code reuse and non-control data attacks that are self-contained, i.e. they implement their malicious behavior without interfering with any other predictable area

or metadata. Although complex self-contained code reuse and non-control data attacks are not widely used in the field – they are mostly used as proof of concepts in academia – some attacks were demonstrated.

In general this means DRIVE is not able to detect those types of self-contained attacks. Attack A6 (Modify Data Segment to Maliciously Alter the Control-flow), more precisely the described example in Section 3.3.4, represents a concrete real-world attack that was demonstrated and, perhaps, even used for active exploitation of systems in the field. Consequently, attack A6 is indicated as not detectable in Table 3.7.

In other words, DRIVE cannot detect self-contained code reuse attacks. Instead, other defensive concepts, such as CFI must be used, c.f. 3.2.4. With regard to non-control data attacks, DFI is conceptually available as a countermeasure to certain variants, but there are no known methods that can generally detect or prevent non-control data attacks. For this reason, DFI is not addressed in this thesis any further and left open until effective mechanisms are available. Similarly, DRIVE is incapable of detecting attacks that target unobserved system resources. For example, the recently discovered famous side channel attacks Spectre [102] and Meltdown [103] are not detectable as they do not manipulate the system memory at all and are therefore invisible to DRIVE[24].

5.4.4 Summary

In this section a security analysis and evaluation of the developed attestation concepts was carried out. The objective of this analysis was to clarify whether the concepts developed were sufficient to detect the attacks presented and defined in Section 3.3.2. In this context, the verification concepts developed were examined with regard to these attacks and an assessment was made as to whether a particular concept considers and recognizes the attack techniques used or not. In order to provide a more detailed analysis, specific criteria were used to determine in which cases certain concepts are effective or not. These criteria were based on predictability, access permissions and arbitrary anonymous mappings of memory areas.

The most important results during the analysis were that all but one attack were detectable by at least one verification approach. The exception was the attack A6, which realizes an attack based on non-control data. This attack is undetectable due to DRIVE's restrictions on unpredictable data verification, but this behavior was expected. In addition, changes to the access permissions are not detectable unless they are made within long-term or access control operations such as `mprotect()` that implicitly triggers a measurement on each invocation. Furthermore, it has been found that arbitrary anonymous mappings are only recognized if a strict policy is applied. For this reason, the implementation must also implement a strict policy that explicitly links programs and dependencies.

[24] A detailed summary and overview of the Spectre and Meltdown attacks can be found at https://meltdownattack.com/

5.5 Chapter Summary

The objective of this chapter was to define, develop and analyze technical details of measurement, verification and reporting concepts carrying out reliable attestations to determine the trustworthiness of a SuE. To achieve this objective, at first the different concepts were classified based on the nature of memory portions they address.

The concepts were separated into static, predictable and unpredictable memory regions and the technical details were introduced and developed. The procedure was consistent, which means first of all the measurement, then the reporting and finally the verification concept were described. First the necessary data structures for all parts were defined and then the technical details were discussed.

The developed measurement concept implemented the technical details of the MA and was used to collect and provide the measurement data from the SuE memory. In addition, the measurement data was anchored in a security module in order to be able to perform different verification steps. To this end, certain procedures have been used to enable verification of authenticity and integrity. Finally, the data was provided in an SSR. The SSR was then transferred to the VA on the VS using a remote attestation protocol, which then performed verification of the transferred data. After successful verification of the transferred data, the actual verification of the measurement data was then carried out on the basis of calculated reference values. First, the process of reference value generation was described, which was defined in different variations depending on the measured data. The verification of the measurement data was then carried out on the basis of the calculated reference values or other policies. In the case that all verification steps were successful, the system was considered trustworthy.

The second part of the chapter was a security analysis of the developed concepts. For this purpose, certain attack scenarios defined in Chapter 3.3.4 were used. The objective of this analysis was to determine whether the concepts developed were sufficient to detect certain attack techniques. In addition, it was to be determined whether certain restrictions with regard to attestation concepts existed or whether additional requirements had to be met. The results of the security analyses were therefore a classification of the developed concepts with regard to the detection of the defined attacks. These results were compiled and presented in tables. All attacks that were supposed to be detected were covered by the attestation concepts. However, there have been expected limitations regarding non-control data attacks and access permissions can only be detected under certain conditions.

The results of the security analyses are taken into account in the design phase of the software implementation. As a result, they are addressed during implementation and revisited in the security analysis section of the next Chapter 6, which describes the actual software implementation and provides an evaluation of the implementation.

Chapter 6

Implementation and Evaluation

This chapter presents and evaluates the software implementation of DRIVE on the basis of the developed architecture and concept from Chapter 4 by implementing the technical solution as described in the previous Chapter 5.

The main objectives of this chapter are:

1. Verify the feasibility of the DRIVE's architecture and concepts. This will be achieved by designing and implementing a PoC that is based on the previously developed high level architecture and the technical concepts.

2. Verify DRIVE's established security assumptions by performing a security evaluation of the PoC through attack simulation of defined attack scenarios.

3. Evaluate the scalability of DRIVE's measurement through benchmarks of compute-intensive operations based on the PoC implementation.

In addition to that, the developed PoC should support an attestation of user space and kernel space memory artifacts. Consequently, this will also be considered during each objective.

6.1 DRIVE Proof of Concept Implementation

This section describes and presents DRIVE's PoC that implements a concrete software architecture based on the previously presented high level architectures, concept and the technical detail provided.

At first, the software architecture itself is described and developed. This software architecture is the foundation of the implementation and defines implementable subsystems that provide a framework for the integration of components. Next, the implementation details of the developed components are presented. In particular, the adopted guideline concept that allows a decoupled and flexible implementation of corresponding functions

across the components is introduced and described. Last, one concrete implementation of a guideline triplet will be established. This triplet realizes the actual technical software implementation for the developed components, i.e. DMC, Reference Value Generator (RVG) and DVC.

6.1.1 DRIVE Software Architecture Implementation

According to the specified and instantiated architecture described in Section 4.2.2, the developed software architecture in this section implements the MA and VA and supporting functions in three different components. These components are: DMC, DVC and RVG, see Figure 6.1.

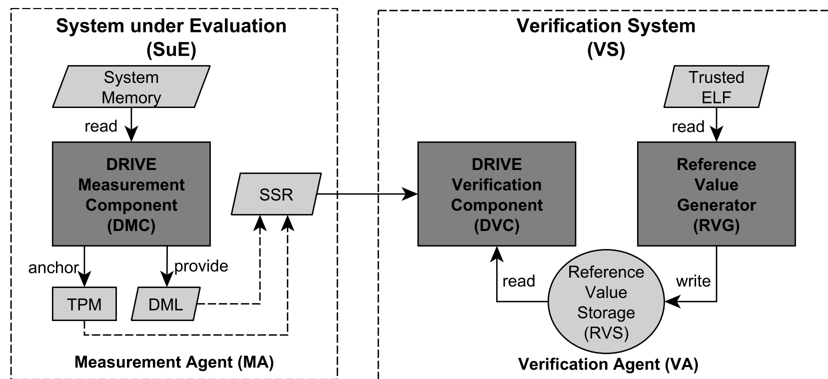

Figure 6.1 – Architecture Overview for implemented Software Components.

As mentioned, these components represent frameworks that allow a high decoupling and flexibility. This is necessary because the user space and kernel space attestation supported by the PoC varies considerably depending on the implementation. These differences are implemented as guidelines and discussed in more detail in the following Section 6.1.2.

On the SuE the MA is realized by the DMC. The DMC is responsible for performing the measurement, anchoring and reporting of user space processes, loadable kernel modules and the running kernel image. As described, this involves accessing and reading the system memory to collect required measurement information.

The second component, RVG is responsible to generate RVD on the basis of well-known and trusted ELF files. Typically, this process is executed on the verification system or a different initially trusted system. As expected, access to the ELF files used on SuE is necessary. The RVG maintains a Reference Value Storage (RVS) that is used by the DVC during verification processes.

The last component DVC is responsible to receive the measurements from the DMC, to carry out the required verification mechanisms and finally take a decision about the system state of SuE. For this purpose a report is generated that states the verification

process result for further use. Following, the details of each component are established in more detail.

Drive Measurement Component

The DMC is implemented as an LKM for the Linux kernel. It provides the framework for measurement and anchoring of user and kernel space memory artifacts and maintains an accessible measurement list to generate an SSRs. DMC's architecture is illustrated in Figure 6.2 and can be split into four main subcomponents.

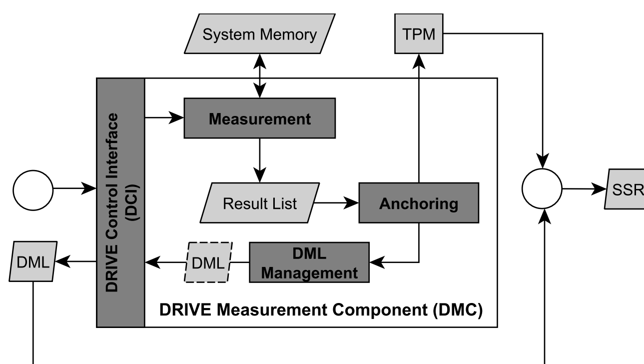

Figure 6.2 – Subcomponents of DRIVE Measurement Component and SSR Generation.

The first subcomponent realizes a communication interface to communicate with user space programs via a character device or a SecurityFS node, both are further referred to DRIVE Control Interface (DCI). The DCI allows invoking measurements of individual targets or the whole system via a command string. For instance, a measurement target can be a single user space process, an LKM or the image of the running kernel. Additionally, a read operation on DCI is used to retrieve the list of stored measurement results. The results are encoded into a binary format and sent to the process reading from the DCI. The PoC implementation uses the Concise Binary Object Representation (CBOR), cf. [104], to represent the measurement results, but the implementation can be modified to produce other binary or plain text formats as well.

The core subcomponent of DMC performs the actual measurement of the targeted memory artifacts. This includes enrichment of internal data structures with additional measurement information about the measurement target, selection and execution of matching guidelines 6.1.2, and collection of the generated results. The measurement operations inside this subcomponent are performed asynchronously by using a Linux kernel work queue. Measurements may either be invoked by commands received via the DCI or, if configured, by a reoccurring timer. The latter is realized by injecting a delayed work package into the work queue that invokes a full system measurement once its timer has run out. After invocation of the full system measurement, the work package re-queues

itself into the work queue, using the same delay as before. The timer can be configured or disabled at module load time via a module parameter.

The measurement results from the DMC are injected into a linked list, i.e. the result list, storing the values for further processing. The contents of this list are consumed by the anchoring subcomponent. It processes the results generated by removing unnecessary information that is no longer required for the following steps. Consequently, this process generates so-called reduced results, which only contain the list of relevant measurement information. Internal data, such as timestamps or counters created during the measurement process, are dropped to save memory. The reduced results are then further processed. In particular, they are added to the DML and anchored inside a security module.

The PoC implementation uses a TPM as its security module. Both TPM 1.2 and TPM 2.0 are supported, however, the current TPM 2.0 API implementation inside the Linux kernel does only support TPM 1.2 operations. Consequently, TPM 2.0-specific features are not supported. For example, only SHA-1 hashes are supported during the anchoring process. On the technical level, reduced results generated in the previous step are serialized to the same binary format used for the DML output. The resulting byte string acts as input for the `extend` operation during the anchoring process, as described in Section 5.1.1.

The last subcomponent implements the DML management. It comprises all results that have been produced since the module was loaded. Hence, it manages the insertion of new results into the list and allows read access to its content. Since the DML is required to be immutable, results can only be added to this list. In other words, the DMC appends the measurements to the DML and anchors the fingerprint to a TPM by calling the `TPM_extend()` functionality.

Figure 6.3 depicts an excerpt for the accumulated user space SSR for the `/bin/bash` process. Once the measurements have been successfully appended to the designated DML and reported to the TPM, an SSR can be generated and verified by the DVC. It has to be noted that a direct generation of an SSR is not possible, due to the unavailable operation `tpm_quote` inside the kernel space. Hence, an SSR is generated by receiving a DML from the DMC, acquiring a quote from the TPM by calling `tpm_quote` operation and combining both results, as shown in Figure 6.3 and 6.2.

The kernel space DML is generated similar to the algorithm for a single process. Still, the data-structures for LKMs are organized in a list utilizing the `module` struct and the Kernel solely relies on `mm_struct`[25].

Reference Value Generator

The RVG is a complement to the DMC. It also implements the concept of guidelines, providing a framework for generating reference values used during the verification by the

[25] All struct definitions can be found in the soruce code of the Linux kernel, c.f. https://github.com/torvalds/linux/blob/master/include/linux/mm_types.h

```
{
 "quote":"7100ff5443..",
 "dml":
 [[[[
     ["p_mem", "/usr/bin/bash|code|digest|sha1", "9e04d9ca50.."],
     ["p_mem", "/usr/bin/bash|code|flags", "0x8000875"],
     ["p_mem", "/usr/bin/bash|code|pte_xor", 0],
     ["p_mem", "/usr/lib64/libnss_files-2.17.so|code|digest|sha1", "bc18c214c5.."],
     ["p_mem", "/usr/lib64/libnss_files-2.17.so|code|flags", "0x8000075"],
     ["p_mem", "/usr/lib64/libnss_files-2.17.so|code|pte_xor", 0],
     ["p_mem", "/usr/lib64/libc-2.17.so|code|digest|sha1", "d7519f2545.."],
     ["p_mem", "/usr/lib64/libc-2.17.so|code|flags", "0x8000075"],
     ["p_mem", "/usr/lib64/libc-2.17.so|code|pte_xor", 0],
     ["p_mem", "/usr/lib64/libdl-2.17.so|code|digest|sha1", "0633b562d5.."],
     ["p_mem", "/usr/lib64/libdl-2.17.so|code|flags", "0x8000075"],
     ["p_mem", "/usr/lib64/libdl-2.17.so|code|pte_xor", 0],
     ["p_mem", "/usr/lib64/libtinfo.so.5.9|code|digest|sha1", "54b0476b4e.."],
     ["p_mem", "/usr/lib64/libtinfo.so.5.9|code|flags", "0x8000075"],
     ["p_mem", "/usr/lib64/libtinfo.so.5.9|code|pte_xor", 0],
     ["p_mem", "/usr/lib64/ld-2.17.so|code|digest|sha1", "93c6424858.."],
     ["p_mem", "/usr/lib64/ld-2.17.so|code|flags", "0x8000875"],
     ["p_mem", "/usr/lib64/ld-2.17.so|code|pte_xor", 0],
     ["p_mem", "vdso|digest|sha1", "e4375a0e93.."],
     ["p_mem", "vdso|flags", "0x8040075"],
     ["p_mem", "vdso|pte_xor", 0]
 ]]]]
}
```


Figure 6.3 – Excerpt of an SSR for a `/bin/bash` Process on the X86_64 Platform in JavaScript Object Notation (JSON).

DVC. The RVG is a user space program and requires a trusted ELF file of all targets to be attested. As expected, these ELFs files must match the exact same ELF counterpart on the target system, but should originate from a secure and initially trusted source. This guarantees that none of the ELF files were compromised at time of generating the reference values.

During the generation process, the RVG reads all relevant ELF files and generates corresponding reference values. The exact generation process is defined by the guideline implementation and varies depending on the measurement target, c.f. Section 5.1.4. In case that the verification for a particular measurement target relies on ad-hoc verification, for instance for RCC or GOT, the ELF is stored in the RVS and is accessible during verification instead.

In addition to the generation of reference values, the RVG manages and maintains other properties and configurations. For this purpose, it implements additional operations to manage these configurations. The configurations and properties managed include, but are not limited to, Internet Protocol addresses of devices to be measured, public portions of cryptographic keys or the version of the TPM internal data structure. These values are also part of the RVS and used during verification, explained in the following.

DRIVE Verification Component

The DVC completes the set of components required to implement the software architecture of DRIVE. Hence, the DVC is responsible to verify received SSRs. First the integrity of the DML and second, the individual measurements are verified. The results of this verification determine whether a system is considered trustworthy or not. Any other action that happens after the system state was determined, specifically in cases the SuE is considered as compromised, is not in the scope of this component. This means that processing of verification results to take remediation actions is not addressed.

As mentioned, the input data of the DVC is encapsulated in an SSR. The SSR is also a CBOR encoded data structure and consists of two elements: (1) a TPM-signed Fingerprint, created SSR generation time and (2) the DML, comprising the current measurement results.

The DVC is divided into three subcomponents which corresponds to the functions *Fingerprint Verification*, *DML Integrity Verification* and *DML Measurement Verification*, defined in Section 4.2.2. Consequently, these subcomponents verify different aspects of the received SSR. The first submodule verifies the authenticity of the SSR by verifying the signed Fingerprint. Afterwards, the second submodule verifies the integrity of the DML by simulating the `extend` operation and comparing the computed result with the comprised TPM Fingerprint.

Only if the integrity of DML was verified successfully, the DML measurement verification is carried out. This is implemented by the third verification subcomponent on the basis of guidelines that implement the measurement-specific verification process, described in corresponding Sections 5.1.4, 5.2.2 and 5.3.2.

In order to process measurements found in the DML, corresponding reference values are retrieved from the RVS and compared to the measurements or ad hoc reference value generation is triggered based on a trusted ELF file obtained from the RVS. If any deviation is found, the SuE is considered compromised and a corresponding attestation report is generated. In addition, it is possible for a guideline to define a result that must always match a specific value. For instance, it verifies a memory permission access policy which assures that no measurement was mapped as `rwx`, c.f. Section 5.3.2.

After processing all results inside the DML a report for the entire verification process is generated and stored in the verification storage. The verification can only be considered successful and valid, if the integrity verification and all individual processes were considered valid. If only a single measurement or any other data-structure involved in the process was considered invalid, the verification state results in a failure state, which means the SuE is not in a trustworthy state and considered as being compromised.

6.1.2 DRIVE Component Guideline Implementation

As mentioned, the components described in the previous sections are frameworks that require additional business logic to perform their operations. This business-logic is represented as so-called guidelines. Guidelines provide a flexible and dynamic way to extend and modify the whole system and facilitate decoupling of functionality inside the components. On a technical level, this is done by implementing a single interface function with code, tailored to fulfill the implementation-specific behavior.

Typically, guidelines come in triplets that realize an implementation for each component and the implementation depends on the actual measurement targets they will process. Hence, the guidelines are specific in terms what they accomplish. First and foremost, all guideline-triplets must agree on the measurement information they collect, the reference data they create and how the measurement information is verified based on the generated reference data. For this purpose, they also need to agree on a unique guideline name, which enables identification in all corresponding components. These build the basic rules the guideline-triplets should consider.

DMC guideline implementation must collect measurement information for at least one type of measurement target. Possible measurement targets are user space processes, LKMs or a kernel image. Second, it must implement a public function that matches the interface definition provided by the DMC. Typically, this function receives a data structure that describes the measurement target to be processed by this guideline. If the guideline implementation is used for multiple target types, it must provide its own way to distinguish between those to dispatch and perform the correct operations.

RVG guideline implementation must generate reference data that is used during verification to verify measurement information provided by the DMC guideline. Current implementation of the RVG requires the guideline to implement also a public function that matches the interface definition specified by the RVG. Typically, this function receives an ELF file as one of its inputs. The ELF file is then processed by the guideline implementation. Depending on the measurement target, additional information relevant for reference data generation may also be provided.

Reference data can be of different types. For instance, they may represent simple reference values, for instance hash digests that can be directly compared with values found in measurement information. Or, in different cases, represent prerequisites or data that enables the DVC to compute actual reference data on its own. For example, RCC verification relies on mimicking the loading process, so in the provided reference data is the ELF file itself. Accordingly, the implementation of the particular guideline triplet must be aware and consider all details regarding which values are required and how the reference data must be generated.

DVC guideline implementation must be able to receive measurements from a DML collected by the DMC and, extract relevant measurement results and information and compare this information to reference data generated by the RVG. Hence, the guideline implements a corresponding interface with at least two input parameters. As mentioned and depending on the measurement target, it may be required that the verification guideline performs additional calculations on the basis of provided inputs. For instance, this is required for results that rely on memory addresses only known at runtime, for instance RCC verification. In this case the RVG generates base values, i.e. storing of the ELF file, required for additional calculations and DMC measurement containing corresponding metadata information with the relevant memory addresses, present at time of measurement. Consequently, the verification guideline must make use of these base values and the received metadata to derive reference data and compare the calculated reference values to the corresponding measurement information.

In the following, one particular example of a guideline-triplet implementation will be discussed in more detail.

Example Guideline-Triplet Implementation

This section will discuss an example of a guideline-triplet implementation for static information attestation for user space processes. In addition, a metadata verification for memory-access permissions will also be considered. For this purpose, all three parts of the guideline-triplet and their individual operations are presented and discussed in more detail.

In general, the components and their corresponding guideline-triplets support (1) the measurement of the OS kernel and all loaded LKMs, in kernel space and (2) the measurement of all active running processes, including executable and shared program text for the X86_64 platform. In addition to that, the implementation was also tested on other hardware architectures, i.e. PPC32, ARM32 and ARM64 on different Linux kernel versions including (3.10, 3.13, 4.2, 4.9).

However, during the evaluation it was found that the adaptation of the Linux Kernel image, LKM- and PTE-granular access permission measurements are highly architecture-dependent. Therefore, these features were disabled for all other platforms other than X86_64. This means that the measurement of user space processes is available for all platforms, whereas Kernel image, LKMs- and PTE-granular access permission measurements are only available on X86_64. Apart from that, related hardware architecture-based effects also affected the concrete implementation of the RVG and the DVC hugely and thus go beyond the scope of this work. For this reason, this section will describe the user space guideline, applicable on all evaluated platforms.

DMC implementation Once a measurement target has been identified, the DMC passes two data structures to the guideline. The first data structure, i.e. the *task structure*, references the measurement targets and contains all the information required to identify and process the measurement target. The second data structure represents the measurement results that are filled by the guideline. The measurement target corresponds to the Measurement Set MS which contains the measurement results that correspond to several sets S, as defined in Section 5.1.1.

The objective of this sample guideline is to access the VAS of a user space process, identify, access and read all static memory portions containing executable code, calculate a hash digest of the contents of each portion and fill the data structures of the measurement results accordingly. The measurement hash digest is only one result produced by this guideline. In addition, the access rights of the corresponding memory portions are read and a test is performed to check whether all corresponding PTEs have the same memory permission rights. These two results are also added to the measurement result. This means that this guideline measures three pieces of measurement information for each memory segment containing executable code.

To identify and access measurement targets that are typically the `.text` segments in this case, the guideline accesses the provided *task structure* and iterates over a list of so-called *vm area structures*, each representing exactly one memory segment used by the user space process. These identified segments are then analyzed to determine whether they need to be considered by the guideline. For this purpose, their memory permission access flags are checked. If the flags are set to readable and executable, the segment becomes a potential measurement target. This guideline also checks whether the segment is an anonymous mapping, since only private segments associated with a file are to be considered[26]. If the flags match the requirements, they are stored as metadata measurement information in the measurement result.

For each identified valid segment, the guideline then uses the start and end addresses of the *vm area structure* to calculate its size and the number of pages this segment is composed of. The guideline must make sure that each of these pages are available in the system memory; therefore, the guideline implicitly causes the kernel to load them into the memory[27]. As soon as a page is loaded and available, its contents are read and supplied to a hash algorithm. Additionally, the guideline resolves the PTE of the current page to test whether the PTE memory permission flags match its counterpart of the segment flags. If a mismatch of the flags has been found[28], a counter will be incremented which allows to recognize this mismatch later on.

Once all pages for a segment are processed, the guideline finalizes the hash operation,

[26] There must be no anonymous mappings that are readable and executable at the same time.
[27] Provisioning of unloaded pages, i.e. loading them from disk to system memory, causes a slight time delay and consumes additional system memory.
[28] As mentioned, the PTE granular access permission are only supported on X86_64 platform.

receives the digest and adds it as measurement information to the measurement result. Finally, the counter mentioned above is read and its value is also added. The guideline then continues with the next segment until all identified segments for the user space process are measured as described. This concludes the measurement guideline operations on the DMC.

RVG implementation The RVG operates on the basis of the ELF files of the measurement targets and with all related libraries available in the ELF header information of the process. To generate reference values for the *vm area flags* metadata, the guideline builds a fixed bit mask that matches the expected flags. In case of the static memory segments, processed by this guideline triplet, the reference value is agreed to be `0x5`. The process for generating the expected measurement hash digests of memory segment contents is as follows:

First, the segment is located inside the ELF file and copied into a buffer. For this purpose, the ELF is loaded with certain library functions and a data structure is generated that allows access to individual parts and information stored inside the ELF.

Once the memory segment has been copied, it must be page-aligned first. For this purpose, `0x0` bytes are appended until a multiple of the page size of the target system is met. Accordingly, RVG allocates additional memory for the buffer and fills the allocated space with `0x0`. After the memory segment has been page aligned, the guideline calculates several hash digests, i.e. SHA-1, SHA-256 and SHA-512, of its contents and stores them as reference values inside RVS, identified by a string that represents the hash algorithm used. Based on this string, the verification can later determine which reference value must be used for comparison. No reference value is generated for the PTE flags test, because the guideline implementations agreed that this value always must be zero to be considered valid. Any other value indicates a compromised system.

After all information are successfully collected and calculated, the RVG stores all reference value data persistently in an *SQLite*[29] database. In addition to that, the RVG also maintains copies of RCC ELFs files on the file-system for ad hoc calculation; however, this particular guideline-triplet does not make use of these files. Once the RVG has finished the data collection, the reference value data is ready to be used during by the verification guideline, discussed next.

DVC implementation As previously described in Section 5.1.4, the verification consists mainly of two parts. At first, the integrity of the received DML and second, the measurement results inside the DML, are verified.

The DML integrity verification mechanisms is implemented as described in Section 5.1.3 and 5.1.3. As explained, this process is not guideline specific and thus always

[29] https://www.sqlite.org/

applied for any encapsulated measurement targets. For this reason, only the guideline-specific implementation details will be discussed.

The DVC guideline takes the verified DML, provided by the DMC, extracts all measurement targets and corresponding measurement results, and compares the encapsulated measurement information to reference data provided by the RVG. The process of verification is simple in this case, because it requires no complex calculations.

For each measurement target that was processed by the DMC, the verification extracts all of its measurement results. Afterwards, the verification guideline iterates over all measurement results, extracts the measurement information and performs the necessary verification steps. For each entry, a report stating the verification state is generated and stored in the verification storage. These reports can later be used by reporting tools for generating a detailed verification report for the attested system.

The PTE flag test is a comparison to the value zero, since the guideline triplet agreed on this particular result value if no deviation was detected. If it is different to zero, this individual measurement information is considered as invalid.

For other measurement information, the guideline retrieves the relevant reference values from the RVS and performs a comparison. This means that the measured hash digest and reference hash digests are compared bit wise. Only if both values are equal, the result is considered valid.

To test the access permissions, the set of flags received from the DMC are compared to the corresponding reference value. If both values are equal, they are considered valid. Alternatively, the guidelines could have also agreed on the comparison of the fixed value `0x5`, instead of relying on the RVG generated data. However, the current solution provides more flexibility on a very minor cost and thus it was implemented as described.

In case no reference value has been found for a specific result entry, the result entry is considered invalid. This is because unexpected measurement results must have been found, which are not meant to be present for the particular user space process. In addition, the guideline implements also a strict policy to determine whether unintended but known segments are mapped in the process' VAS, as discussed in the security analysis in Section 5.4.1. Consequently, if a known reference value was found, but the segment is not expected in the context of the measured process, the mapping is also considered as invalid and the process as being compromised.

DMC implementation will only provide one configured digest per segment. This means that once one hash digest was found and verified, the other reference values, for different hash digests, are ignored. The guideline execution stops once all provided measurement results are processed.

6.1.3 Summary

This section described the details of the developed and implemented PoC used to attest different memory artifacts described in this thesis. For this purpose, the PoC implemented the MA and VA based on the high level attestation concept and architecture from Chapter 4 and mechanisms described in Chapter 5.

The PoC realized and implemented three different components. These were: 1. DMC collecting, anchoring and reporting measurements on SuE, 2. RVG generating reference data used by DVC during verification and 3. DVC receiving and verifying reported measurement on VS.

The components themselves are composed of different subcomponents. For this reason, each of the corresponding subcomponents were introduced and described in more detail. In addition to that, the PoC implemented a concept called guidelines. These guidelines provided a flexible mechanism to realize different business-logic for measurement, reference data generation and verification inside their corresponding components. To exemplify how these guidelines were implemented, one particular guideline-triplet implementation was presented. This guideline-triplet provided an implementation for each component and realized a static information attestation for user space processes and some additional verification steps based on additional metadata.

6.2 Security Evaluation

In this section, the evaluation of DRIVE's implementation is presented. In particular, several attack types were evaluated in order to determine whether the designated verification modules were able to detect them.

According to the threat analysis presented in Section 3.3 and 5.4, the security evaluation consists of an analysis that is carried out for different applications, represented mainly by the `/bin/bash` application or by other applications when an attack was not applicable for `/bin/bash`. Specifically, the behavior and detection capabilities for the attacks introduced in Section 3.3.2 have been analyzed with additional variants:

A1 Create new executable segment, e.g. load new (`mmap`) or map existing code (`dlopen`)

A2 Inject malicious code in arbitrary memory region, e.g. inject code into .text segment's padding space

A3 Modify code segment to change semantics maliciously, e.g. replace instructions or code pointers in .text segment

A4 Modify Code pointer data to call malicious/unintended code, e.g. modify memory jump addresses in the GOT (`.got`)

Table 6.1 – Detection of Access Permission Manipulation (APM) during Static Information Attestation (SIA), Predictable and Unpredictable Dynamic Information Attestation (DIA) and Metadata Attestation (MDA) for Attack A5. Symbols: ✔ (detected), ✗ (undetected), ✓/✗ (detectable).

Attack	APM of	SIA	DIA	MDA
A5: .text	Bash process	✗	✗	✔
	Kernel	✗	✗	✔
	LKM	✗	✗	✔
A5: stack	Bash process	✗	✗	✔
	Kernel	✗	✗	✔
	LKM	✗	✗	✔
A5: heap	Bash process	✗	✗	✔
	Kernel	✗	✗	✔
	LKM	✗	✗	✔

A5 Alter/remove memory protection, e.g. disable or circumvent DEP mechanisms by utilizing the `mprotect()` system call

A6 Modify data segment to maliciously alter the control-flow, e.g. change configuration option to fixate a particular control-structure path

In order to keep the experiments simple and reproducible, most of the attacks mentioned were simulated. It is generally assumed that a sophisticated attacker can utilize these attacks if an initially required vulnerability is available for the particular software that is to be exploited. For the analysis itself, a python tool was developed that utilizes the `ptrace()` system-call for attaching to a process and apply the modifications directly in memory (A2, A3, A4). In other cases, the tool utilizes the GNU Debugger (GDB), to simulate system calls (`mmap()` A1 and `mprotect()` A4, A5) or calls to external library functions (`dlopen()` A1)[30]. Attacks that involved manipulations inside the kernel space, e.g. manipulation of `.text` segments inside the kernel or LKM, were simulated by implementing the necessary attack in a different malicious kernel module that applied the changes when loaded [31].

For attack A6, which is based on a real-world vulnerability attacking the `Exim` mail-sever, Metasploit[32] was used for the actual exploitation. After the attacks were simulated, a measurement and verification process was conducted in order to determine whether the attack was detected or not, and which part of DRIVE detected the modification.

6.2.1 Attack-based Security Analysis

[30] The `/proc/sys/kernel/yama/ptrace_scope` was set to `0` enabling process hooking by `ptrace`
[31] c.f. https://github.com/maK-/Syscall-table-hijack-LKM/blob/master/template.c
[32] https://www.metasploit.com/

Attack A5 The simplest attack simulation is A5, i.e. changing memory access permissions. Experiments covered manipulations of `.text`, stack and heap segments for processes and manipulations of `.text` segments for LKMs and the kernel. The goal was either to enable write access to the `.text` segments or enable executable permissions to the stack or heap. For the bash process access permission manipulation, the python tool generates a command that calls `mprotect()` with the determined memory addresses and instructs GDB to execute the command. After the command was executed by GDB, the memory permissions for the attacked process are modified persistently until another call to `mprotect()` is made to reset the access permissions to the original values. For the kernel and LKMs, a persistent modification was done by manipulating the corresponding memory access permissions inside the page table, applied by a specifically crafted attack simulation LKM.

In addition to these experiments, the manipulation of individual pages is also considered when carrying out the measurement. For example, it is possible for an attacker to only change the exact page within a mapping that is needed to inject malicious code. This means that the attacker only changes the permissions of a single page and leaves the other page permissions in their original form. The access authorizations of a mapping are therefore not reflected in metadata that is maintained directly in higher-level structures, but are a composition of the access permissions that are maintained for each page within the page table. The related measurement guideline in DRIVE considers this behavior: the guideline iterates over all pages of a mapping and calculates whether there are deviations from the permissions within a mapping. This information is added to the DML and evaluated during the verification phase. Consequently, all deviations are detected and considered during verification. If the deviation results in a malicious state, for example granting executable rights to the stack, the mapping is considered as compromised.

In all 9 experiments, during the metadata verification of DRIVE, the modifications of access permissions during the verification phase were detected as long as the modification remained active. These results are shown in Table 6.1. As already mentioned, it is irrelevant for DRIVE whether an individual page permission has been changed within a mapping or whether the entire mapping permission has been modified. DRIVE's guideline implementation was able to detect the modifications in each of these two cases. It is also important to note that the stack and heap require executable rights for the entire duration of an attack. As a result, these stack or heap-based injection attacks are detectable at least as long as they are active.

Attack A3 The next simulated attack was attack A3, i.e. modify code segment to change semantics maliciously. The results are depicted in Table 5.4. The simulated attacks rely on the application of the attack A5 during an initialization step, followed by the actual manipulation attack. However, the simulation applies attack A5 after the successful

manipulation attempt for a second time in order to reset the access permissions back to their original value. Consequently, this avoids the detection during Metadata Attestation, because no deviation to the original access permissions remain persistent. Regarding the detection of the actual code manipulation, the corresponding Static and Predictable Dynamic Information Attestation modules detected both attacks, depending on which component they were applied. In case of manipulating the text segment of the bash process, the Static Information Attestation module could not find a valid reference hash value during verification; consequently, the manipulation attack was detected based on an unknown reference hash. When manipulating the LKM and kernel text segments, no valid hashes could be generated during ad hoc reference value generation. The Predictable Dynamic Information Attestation was therefore able to recognize that an unintentional manipulation must have taken place and has thus successfully detected the attack.

Table 6.2 – Detection of Manipulations in Static Information Attestation (SIA), Predictable and Unpredictable Dynamic Information Attestation (DIA) and Metadata Attestation (MDA). Symbols: ✔ (detected), ✗ (undetected), ✓/✗ (detectable).

Attack	Manipulation in .../ Details	Policy	SIA	DIA	MDA
A1	Adjust pointer in Bash `.text`		✔	✗	✗
	Adjust pointer in LKM `.text`		✗	✔	✗
	Adjust pointer in Kernel `.text`		✗	✔	✗
	Load Bash unused library	strict	✔	✗	✗
		relaxed	✗	✗	✗
	Load unknown LKM		✗	✔	✓/✗
	Create random executable mapping		✔	✗	✓/✗
A2	Insert code into Bash `.text` padding		✔	✗	✗
	Insert code into kernel `.text` padding		✗	✔	✗
	Insert code into LKM `.text` padding		✗	✔	✗
A3	Alter instruction in Bash `.text`		✔	✗	✗
	Alter instruction in LKM `.text`		✗	✔	✗
	Alter instruction in kernel `.text`		✗	✔	✗
A4	Alter Bash `.got`		✗	✔	✗
A5 V1	For details, see Table 6.1		✗	✗	✔
A5 V2			✗	✗	✔
A5 V3			✗	✗	✔
A6	Complex non-control data attack		✗	✗	✗

Attack A4 Attack A4, i.e. modify code pointer data to call malicious/unintended code, is a specific attack that can only be applied to processes. This is because the attack manipulates the GOT which is only available in user space processes. In order to confirm

a successful detection of the attack, it is sufficient to alter any address inside the GOT to an arbitrary value. As a result, the attack simulation spawned a bash process and manipulated a pointer value inside the '.got' section. As depicted in Table 5.4, the Predictable Dynamic Information Attestation module detected the manipulation successfully. During verification, it was not possible to calculate a reference hash on the basis of the available information that was equal to the measured hash value, c.f. Section 5.2.4. As a consequence, the .got was considered to be altered with an unintended value and hence was no longer considered to be reliable.

Attack A1 Attack A1, i.e. create new executable segment, consists of multiple related attacks with many possible variants. Important in these cases is that the activation involves an additional step, so that the newly loaded code becomes active within the CFG of the attacked program. Since the manipulation inside .text segments and other predictable dynamic segments has already been simulated for attack A3 and A4, the simulation of attack A1 focused only on the loading of foreign code portions by utilizing corresponding functions. Hence, as depicted in Table 6.2, the manipulation of .text segment pointers is similar to the instruction manipulation in A3. In case the manipulation is applied in the GOT, the verification for attack A4 successfully recognizes this manipulation.

Accordingly, the particular simulation involved for attack A1 consisted of:

(1) Loading of a random file into bash VAS with executable rights

(2) Loading of an unknown LKM

(3) Loading of a well-known, but unused library into bash VAS or unused LKM

As expected, (1) and (2) have been detected by the corresponding attestation modules. For (1) no valid reference hash value was found in the RVD. For (2) there was no valid ELF file within the RVD and, as such, it was not possible to trigger an ad hoc reference value generation process. This means that in both cases, the attacks have been successfully detected on basis of missing valid reference values. In the case (3), however, the situation was different and depended on the actual implementation of the Attestation module. If a relaxed policy was used, i.e. the verification system only considers whether a library or an LKM is known, then the attacks were not detected by any attestation module.

For this reason, the implementation of the Static Information Attestation module applies a strict policy that also establishes a correlation between programs and libraries the program relies on. Similarly, the Predictable Dynamic Attestation module does only contain LKM's that are expected to be loaded on a particular system. With this additional information, the corresponding attestation module can make a decision whether a library is valid to be loaded for a program or if an LKM is expected to be loaded on a particular system and, thus, determine if the well-known but unexpected component is valid to be loaded or not. To conclude, the strict policy detected the attack variant (3) successfully.

Attack A2 Attack A2, is a special case of the attacks A3 and A1. The major difference regarding the A2 attack is that the manipulation does not introduce a new mapping within the address space, but, instead, uses padding areas to inject malicious portions of code. Conceptually, this means, the concrete attacks, i.e. code injection and code pointer manipulation, are covered by the corresponding attestation modules. However, in order to be able to detect them, the actual implementation details are important, as described in the Section 5.4.1. For this reason, the attack simulation injects the malicious code in the padding areas of the `Bash` process and the kernel `.text` areas. The experiments have confirmed that the implementation of the corresponding attestation modules were able to detect these injections as expected, the results are shown in Table 5.4.

Attack A6 Attack A6 was the only attack that has not been detected by any DRIVE attestation module. Nevertheless, this result was expected and shows that there are strict limits regarding DRIVE's detection capability, especially for non-control data attacks in unpredictable memory. The experiment, carried out for A6, was different to all other experiments, because it relied on a specific `Exim` configuration and a vulnerable `glibc` implementation. For this reason, a VM was prepared that consisted of the vulnerable `glibc` version and `Exim` was configured to allow the actual exploitation. It must be noted that for the experiment no default security features, such as ASLR or canaries, were disabled. The actual attack was carried out on an external system by using the Metasploit penetration testing framework[33].

The experiment has been carried out as follows:

(1) Launch the VM and start `Exim`

(2) Load DRIVE LKM on VM

(3) Start Metasploit on remote system and launch attack

(4) Conduct an attestation of the `Exim` process' VAS while the attack is still active

As expected, there was no indication of the attack visible during DRIVE attestation. Hence, as shown in Table 5.4, attack A6 has not been detected by any of DRIVE's attestation modules.

6.2.2 Summary and Conclusion

To conclude, this section has evaluated the implementation of the attestation modules by simulating different attacks as defined in Section 3.3.4. It has been shown that all attestation modules, consisting of measurement, reporting and verification procedures,

[33] https://www.rapid7.com/db/modules/exploit/linux/smtp/exim_gethostbyname_bof

behave as expected; all attacks that were intended to be detectable by DRIVE have been detected successfully during, at least, a single verification step.

This means that DRIVE's current implementation is able to detect attacks on different kinds of levels and with different granularity. Code injection attacks and Code pointer manipulation attacks in predictable memory areas are well suited for DRIVE as anticipated. Similarly, Code pointer modifications in specific memory areas, such as the GOT, can also be detected reliably. The metadata attestation used to detect Attack A5 has the potential to detect complex attacks on the system that involve changes that are not based on manipulated, predictable content. Regarding the unpredictable memory areas, metadata analysis is the only applicable strategy DRIVE can rely on to make a decision about the system state.

However, as described and shown, self-contained code reuse attacks or non-control data attacks are currently not detectable by DRIVE. This is because they usually do not modify predictable memory areas. The mechanisms of CFI and DFI are specifically designed to detect these attacks and it would therefore be very interesting to investigate and analyze these protection mechanisms in order to possibly integrate some concepts into DRIVE or vice versa. Very recent research on attacks that solely utilize non-control data to implant malicious actions seem to be resistant even against CFI. However, if data-structures are altered that rely on static information or modify metadata that can be measured and verified successfully, DRIVE can be used to detect at least specific variants of those attacks. This advanced topic, however, goes beyond the scope of this work and is left open for further research.

6.3 Performance and Scalability Evaluation

This section presents a performance and scalability evaluation based on the PoC implementation described in Section 6.1. The primary focus of the evaluation is to determine the impact of the DMC implementation with regards to the SuE. This is because the DMC is typically deployed as a security measure on a SuE; but, the main purpose of the SuE is to fulfill operational functions. Any security feature deployed should therefore only protect the system from malicious actions and therefore affect only the operation or basic functions of the SuE in a reasonable manner. For this reason, this section will evaluate to what extent DRIVE affects SuE's resources.

The PoC implementation presented in this work is a far more sophisticated implementation than used in the initial research published by Rein in [1]. For this reason, the evaluation is carried out in this work on the presented PoC implementation. However, the evaluation from the original research is first summarized and then the evaluation based on the new PoC is presented in more detail.

6.3.1 Evaluation Summary Original Research

The evaluation has been carried out on two different systems representing SuEs on different hardware-architectures:

System 1: X86-64 Bit Intel Core i5-4570 CPU @ 3.20GHz on a standard Ubuntu 14.04 (3.13 Kernel) server installation (X86_64)

System 2: 32 Bit PPC e500mc @ 1.2 GHz (4 cores) on Windriver embedded Linux (2.6.34 Kernel) (PPC32)

Both systems were equipped with a discrete TPM 1.2 chip. Furthermore, two time-critical operations were identified by utilizing the kernel's *ftrace*[34] debugging mechanism. The identified time-critical functions were:

1. Hash calculation of the individually measured memory portions
2. TPM `extend()` operation

One measurement cycle included the measurement and anchoring process of all active processes on a SuE. The evaluation results are illustrated in Table 6.3. For X86_64, ~ 28 different code and ~ 393 shared library code segments (~ $96-97$ MB) were measured. These were aggregated from ~ 70 individual shared libraries. For PPC32 ~ 30 different code and ~ 698 library segments (~ $277-282$ MB) were measured. In this case, the library segments were aggregated from ~ 230 individual shared libraries.

Since libraries' unmodified `.text` segments are usually mapped one-time in physical memory, it was concluded that multiple measurements of the same library were applied unnecessarily. An enormous optimization potential was therefore anticipated if a more sophisticated measurement strategy was implemented, taking into account the deduplication of measurements already carried out.

Table 6.3 – Performance Metrics for Measurement Component.

Hash	Arch	(1) Code/Library Segments			(2) TPM Extend Function			(1 + 2) Cumulative		
		time	percent	size	time	percent	single	time	percent	overall
none	X86_64	0.0291s	8.21%	97.06MB	0.3177s	89.47%	10.96ms	0.3468s	97.68%	0.3551s
SHA-1	X86_64	0.3947s	54.44%	96.41MB	0.3188s	43.98%	11.07ms	0.7135s	98.42%	0.7249s
SHA-256	X86_64	0.6315s	66.25%	97.06MB	0.3169s	33.25%	10.93ms	0.9484s	99.50%	0.9532s
none	PPC32	1.0010s	64.77%	279.33MB	0.4591s	29.71%	15.30ms	1.4601s	94.48%	1.5454s
SHA-1	PPC32	7.6695s	92.65%	277.88MB	0.4536s	5.48%	15.26ms	8.1232s	98.13%	8.2779s
SHA-256	PPC32	7.7194s	92.51%	281.42MB	0.4664s	5.59%	15.19ms	8.1854s	98.10%	8.3442s

The TPM `extend()` operation was applied one-time for every individual process and consumed ~ $10.9-11.1$ ms on X86_64 and ~ $15.19-15.3$ ms on PPC32 on average. This value was considered independent from utilized hash algorithms.

[34] https://www.kernel.org/doc/Documentation/trace/ftrace.txt

The time difference between hash algorithms were found to behave as expected on X86_64, i.e., the SHA-1 algorithm was substantially faster than SHA-256. On PPC32 no substantial difference could be measured between the two algorithms.

The overall computational time during hash calculation was: For X86_64, SHA-1 $\sim 0.3947s$ and SHA-256 $\sim 0.6315s$ and for PPC32, SHA-1 $\sim 7.6695s$ and SHA-256 $\sim 7.7194s$ respectively.

Regarding the verification process, it was concluded that research by Rein et al. already analyzed integrity verification of an SML [3]. It was therefore concluded that the verification of freshness, authenticity of a TPM `Quote` and the verification of DML's integrity based on a TPM `Quote` is identical regarding time and computational effort. Effects from the larger individual data-sets were considered to be negligible. PIC measurement verification relied on a comparison of the measured hash digest mhd and a calculated reference value. Hence, verification times were expected to be equal to the referenced SML verification and found to depend mainly on the implementation details. The verification of GOT and RCC verification involved ad hoc reference data generation for measured `.got` and LKMs memory portions. In both cases, it was found that the symbol resolution process was the most time-consuming operation; it took $\sim 1.53s$ to generate the symbol table for the `.got` of `/bin/bash` and $\sim 1.66s$ to generate the symbol table for analyzed LKMs on average. Once the symbol tables had been generated, the remaining operations, i.e. calculation of correct jump addresses, `.got` generation (`/bin/bash` application), the patching process (LKM), and hash calculation, took for the `/bin/bash` application $\sim 14ms$ and for a single LKM $\sim 52ms$ on average.

6.3.2 Proof of Concept Measurement Evaluation

This evaluation was primarily conducted on an Intel® NUC Kit NUC5i3MYHE[35]. The device consisted of 4 CPUs Intel® Core i3-5010U CPU @ 2.10GHz and was equipped with 8GB of RAM. The device was intentionally chosen, because it was one of few systems that was equipped with a discrete Infineon SLB9665TT2.0 TPM 2.0. The OS used was CentOS Linux release 7.3.1611 (Core) and the kernel version was `3.10.0-514.16.1.el7.x86_64`. It has to be mentioned that CentOS kernels receive many functional updates back-ported from the mainline kernel. This means that the version number is not very meaningful in this case.

The PoC code of the DMC implemented internal benchmarking functions that can be enabled during compile time. Once the benchmarking has been enabled, the DMC collects different benchmarks values internally and reports these accumulated values via the kernels logging mechanisms. The benchmark values are then accessible via the `dmesg` program. The time based benchmarks start a timer when a functional block enters

[35] https://www.intel.de/content/www/de/de/products/boards-kits/nuc/kits/nuc5i3myhe.html

execution and stops when a functional block finishes its operation. Based on previous research and summarized evaluation results in Section 6.3.1, the functions that affect the SuE the most, were already known. For this reason, the DMC functions that implemented hash digest generation and TPM-related operations have been benchmarked. Still, the described PoC implementation of the DMC includes some additional parameters that enable optimizations that influence the behavior of certain operations.

In particular, the following three adjustments can be defined during compilation and module load time:

Hash algorithm Selects the hash algorithm used, typically these are `SHA-1` and `SHA-256`.

Batch size Control how many measurement sets are anchored at once. A batch size of 1 anchors every single measurement set, for example one user process, separately. Every batch size $x > 1$, collects x measurement sets first, groups them together and anchors them in a single operation.

PTE caching Controls which strategy is applied when identical measured memory segments are encountered during user space process measurements. Possible options are:

- ***NOPTE*** Caching is disabled. Every segment is measured every time.
- ***PTECLEAR*** Caching based on PTEs is enabled. The cache is reset after a measurement task was finished. This means that when a single process is measured, it adopts the same behavior as the *NOPTE* option. In contrast, if multiple measurements are measured, for instance measuring all processes, the caching is active until the last process has been measured.
- ***PTEFULL*** Caching based on PTEs is enabled. The cache is initialized at module load time and reset only on module unload.

Experiment 1: Baseline

The first experiments carried out during evaluation applied the hash algorithms SHA-1 and SHA-256 and was configured with no optimizations enabled. This means Batch size= 1 and PTE caching = NOPTE. Therefore, this experiment was comparable to the summarized results from Section 6.3.1.

The acquired benchmark results, presented in Table 6.4, build the baseline for the other benchmark experiments that will be discussed. The results are evaluated as follows. First, both separate measurement benchmarks operated on comparable amounts of data, i.e. ~ 2456MB. As expected, and similar to the previous evaluation, the SHA-256 hash algorithm takes considerably longer than SHA-1. In summary, SHA-1 took ~ 9.29s with throughput

Table 6.4 – Evaluation Benchmark Experiment 1: SHA-1 and SHA-256 Hash Digests with no Optimizations enabled. The Experiment consisted of 10 Full System Measurements measuring: all active user space Processes, all loaded LKM's and the Kernel Image.

Algorithm	Hash				Anchor	
	size	**time**	**average**	**performance**	**time**	**count**
SHA-1	2455,61 MB	9.29s	929 ms	254,87 MB/s	150.06 s	1571
SHA-256	2455,61 MB	15.70s	1570 ms	156,43 MB/s	150.08 s	1571

of ~ 254, 9MB/s and SHA-256 took ~ 15.7s with a throughput of ~ 156.4MB/s. These results are indeed comparable to the previous evaluation which acquired a throughput of ~ 244.7MB/s for SHA-1 and ~ 153.7MB/s for SHA-256. In addition, it has been found that the hash algorithm used, did not influence the anchoring process.

The measured benchmark results for the anchoring process, however, showed a very surprising and unexpected result. In both cases, the overall time took ~ 150s and was carried out ~ 1571 times. However, this high time value was not expected under any circumstance, because it was significantly higher than the values acquired in previous evaluations or any other research conducted in this field. For this reason, a more detailed analysis of this benchmark result was carried out. Due to the asynchronous implementation of the measurement process used by DMC and an enhancement that enabled a secure truncation of a DML, the anchor mechanism applied a `tpm_extend()` followed by a `tpm_pcr_read()` operation. Thus, at first, both operations need to be considered individually. On average, a single `anchor` operation took ~ 95.52ms, `tpm_extend` took ~ 59.40ms and `tpm_pcr_read` took ~ 36.05ms. Nevertheless, the value for `tpm_extend` operation was still far higher than expected.

Finally, it could not be clarified why the `tpm_extend` operation took so long. No values could be taken into account because no comprehensive analysis of this operation was available for this discrete TPM. Microsoft research specifies a maximum value of 20ms for this operation[36], but no published values were found to make a comparison or further analysis. In addition, `tpm_extend` was measured on a *Raspberry PI* on ARM32 with kernel version 4.9.6 mounted with an equal TPM chip, i.e. Infineon SLB9665TT2.0, and connected via an Serial Peripheral Interface (SPI) bus. Again, the average time of the operation was ~ 59ms. Similarly, reading a Platform Configuration Register (PCR) value by using `tpm_pcr_read` operation also took a considerable amount of time. The origin of this behavior can therefore only be speculated on. Potential reasons that could be responsible are: the TPM chips themselves have performance problems, the bus used is too slow or the implementation in the Linux kernel driver is faulty.

Despite the identified problem with the `extend()` operation, the effects this operation

[36] https://msdn.microsoft.com/en-us/library/windows/hardware/dn293575(v=vs.85).aspx

has on the implementation can be significantly reduced by changing the Batch size value. If this value is increased, the effects of the anchoring process becomes less significant.

For this reason, the second experiment will adjust the batch size and briefly discuss its impacts and results.

Experiment 2: Batch Size Adjustment

As explained, adjusting the batch size of the DMC can be used to group multiple measurement sets and anchor them during a single anchoring operation. For this purpose different batch sizes were evaluated. In particular the batch sizes were: $1000, 500, 200, 100, 50, 20, 10, 5, 2, 1$. Equal to Experiment 1, the benchmark carried out 10 full system measurements during the benchmark.

Table 6.5 – Evaluation Benchmark Experiment 2: Adjust Batch Sizes to influence Anchoring Behavior.

Batch Size	Anchor			Extend		PCR read	
	count	time	average	time	average	time	average
1	1571	150.06 s	95.52 ms	93.32 s	59.40 ms	56.64 s	36.05 ms
2	790	74.83 s	94.72 ms	46.27 s	58.56 ms	28.48 s	36.05 ms
5	320	29.58 s	92.43 ms	17.96 s	56.13 ms	11.54 s	36.06 ms
10	160	14.47 s	90.46 ms	8.64 s	53.99 ms	5.76 s	36.02 ms
20	80	7.22 s	90.30 ms	4.27 s	53.43 ms	2.88 s	35.98 ms
50	40	3.66 s	91.43 ms	2.14 s	53.62 ms	1.44 s	36.06 ms
100	20	1.86 s	93.05 ms	1.07 s	53.56 ms	0.72 s	36.06 ms
200	10	0.96 s	96.13 ms	0.54 s	53.73 ms	0.35 s	35.46 ms
500	10	0.96 s	96.49 ms	0.54 s	53.58 ms	0.36 s	36.06 ms
1000	10	0.97 s	96.63 ms	0.54 s	53.76 ms	0.36 s	36.06 ms

As the results in Table 6.5 show, doubling the batch size led to a ~ 50% reduction of the total time for all three related benchmark results. This means that the reduction is almost linear. This result was expected and confirmed the anticipated reduction regarding overall time for the anchoring operation. As indicated by these values, no significant difference between the batch size with regard to the actual average time of the operations has been found after increasing the batch size to 200. Considering that a total of 1571 operations were performed for a batch size of one, it is concluded that ~ 157 readings were anchored on average. This explains why the experiment showed only a negligible difference between batch sizes of 200, 500 and 1000, since in this case only one anchoring operation was performed for a single full system measurement.

To determine a reasonable value for the batch size, the corresponding hash digest generation process must be taken into account. Typically, there should be a balance between the two values in order to reduce the attack surface between measurement and anchoring. For example, it has been determined that the hash time for SHA-1 is 929 ms

and for SHA-256 1570 ms on average for a full system measurement. Consequently, this would allow to first collect ~ 157 (SHA-1) or ~ 89 (SHA-256) measurements and anchor them as one group.

Accordingly, a batch size of 157 would be ideal for SHA-1 and 89 for SHA-256. It is important to note that these determined values only apply to this particular system. To determine the ideal batch size for another target system, the analysis of the SuE to be measured must be performed again.

Experiment 3: Measurement Caching

The original research evaluation expected a huge optimization potential by avoiding to measure onetime physically mapped resources multiple times. For this reason, a PTE-based caching mechanism was implemented in the DMC. The caching function builds a correlation between virtual and physical memory addresses and establishes a cache that recognizes identical mappings. This means that if virtual mappings from different VASs resolve to the same physical resource-mapping, then they are considered as identical. For this purpose, the caching function calculates a hash digest of the physical PTE addresses. If two PTE hashes from different VAS are equal, they map to the same resource and hence their content must be identical. If the resource had been changed, the operating system's COW mechanism would have assigned a different physical address to the changed page.

The behavior of the cache can be configured by selecting the previously mentioned strategies: *NOPTE*, cache is disabled; *PTECLEAR*, cache is enabled but reset after each individual measurement; and *PTEFULL*, cache is enabled and not reset.

Table 6.6 – Evaluation Benchmark Experiment 3: Hash Time Optimization Measurements on the basis of PTE Caching Mechanisms.

Hash	**Type**	$\mathrm{hash}_{\mathrm{total}}$	$\mathrm{hash}_{\mathrm{uncached}}$			$\mathrm{hash}_{\mathrm{cached}}$		
		time	**count**	**time**	**size**	**count**	**time**	**size** (saved)
SHA-1	*NOPTE*	9,582s	8098	9,582s	2455,6 MB	0	0,000s	0,0 MB
SHA-1	*PTECLEAR*	1,340s	1828	1,223s	523,9 MB	6275	0,118s	1934,0 MB
SHA-1	*PTEFULL*	0,314s	220	0,159s	64,9 MB	7882	0,155s	2392,3 MB
SHA-256	*NOPTE*	15,691s	8071	15,69s	2444,2 MB	0	0,000s	0,0 MB
SHA-256	*PTECLEAR*	2,674s	1828	2,545s	523,9 MB	6275	0,129s	1934,0 MB
SHA-256	*PTEFULL*	0,482s	219	0,319s	64,5 MB	7879	0,163s	2391,1 MB

The results of the experiment are presented in Table 6.6. They have been collected from all measured batches in order to evaluate a larger amount of data. To this end, all relevant values were aggregated and a mean value was calculated. These calculated mean values show that by using caching mechanisms a significant reduction of the total time $\mathrm{hash}_{\mathrm{total}}$ could be achieved. However, it was important to consider that the total time $\mathrm{hash}_{\mathrm{total}}$ includes both the content hash calculation and the PTE hash calculation. Here applies:

$hash_{total} = hash_{uncached} + hash_{cached}$. In detail, the total value of the required hash time for *PTECLEAR* SHA-1 decreased by approx. 715% to 1,340 seconds and *PTECLEAR* SHA-1 by approx. 3055% to 0,314 seconds, in relation to the initial value 9,582 seconds for *NOPTE* SHA-1. Similarly, the values for *PTECLEAR* SHA-256 decreased by approx. 587% to 2,674 seconds and *PTECLEAR* SHA-256 by approx. 3252% to 0,482 seconds, in relation to the initial value 15,691 seconds for *NOPTE* SHA-256.

These results clearly show the potential of cache-based optimizations. Nevertheless, it should be noted that the use of a cache has an impact on the security of the solution. The basic premise of the solution is based on the COW mechanism of the OS kernel. It can be assumed that the COW mechanism works reliably and that any modifications made can be detected by the cache algorithm. However, if attacks are carried out directly on the physical memory, such as done by the Rowhammer attack [105], they can only be captured by an explicit measurement. Since the physical addresses do not change in these attacks, the cache mechanism would falsely retrieve the hash value from the cache. For this reason, the *PTEFULL* mode should only be used if attacks on the physical memory can be excluded. The *PTECLEAR* mode is less susceptible to this, because the cache is reset after each measurement. This means that attacks on the physical memory are detected at the latest during the next measurement.

To conclude, this experiment showed that a caching strategy has the potential to significantly reduce the amount of time spent and thus the impact on the system. Whether caching is necessary and which caching strategy should be used depends on the individual case; a general recommendation as to whether a caching strategy should be used cannot be given. However, if caching is considered a viable option for a particular use case, it is recommended to use the caching strategy *PTECLEAR*. The *PTEFULL* strategy is not recommended from a security point of view, as attacks on physical memory are very difficult to rule out.

6.3.3 Summary

This section established a scalability analysis on the basis of the PoC implementation of DMC. For this purpose different experiments have been carried out to analyze and optimize different aspects of the implementation.

Initially, the original research was summarized that was based on a previous PoC implementation. In that case it was already found that the hash calculation and TPM-related operations consume the most computational effort and time. For this reason these parts were revisited in the experiments carried out during this evaluation. The first experiment was to confirm that the new PoC is comparable to previously determined results and thus to establish a baseline. One major finding here was that the hash generation process is indeed comparable. But, the TPM-related operations for anchoring showed a significant increase in duration. The reason of this behavior could not be determined.

The next experiment 2, investigated how and to what extent the anchoring processes can be reduced by grouping measurement sets. In this case, batches of measurement sets were first collected and then anchored in a single anchor operation as a group. Different batch sizes were evaluated and analyzed. The most important finding was that the increase in batch size reduced anchor time almost linearly, until only one single anchor operation was necessary. In conclusion, this means that the anchoring processes can be controlled granularly if a batch mechanism is used.

The last experiment 3 evaluated an optimization of the hashing time duration by implementing a cache mechanism. Since the hashing operation is usually carried out by the CPU, this operation was suspected to affect the system performance the most. Reducing the amount of required data that must be hashed thus also contributes to effectively reducing the impact of DMC on the SuE in terms of computational effort. The PTE-based caching mechanism relied on the physical memory addresses of storage pages and took advantage of the strict deduplication of OS resources. Three optimization strategies were evaluated. It could be shown that using this caching mechanisms significantly reduced the overall amount and consequently the time required for hashing. In comparison to the evaluated results that applied no caching strategy, the reduction was for SHA-1 ~ 715% for *PTECLEAR* and ~ 3055% for *PTEFULL* and for SHA-256 ~ 587% for *PTECLEAR* and ~ 3252%. However, selecting a caching strategy must be made carefully under the consideration of security implications.

6.4 Implementation and Evaluation Summary

This chapter presented and evaluated the PoC implementation that was developed during the course of this thesis. The PoC was able to measure all relevant types of system components, i.e. user space processes, LKMs and the kernel. Furthermore, the PoC could be deployed and was evaluated on a multitude of different hardware-architectures, different Linux distributions and for different kernel versions.

At first, the PoC implementation was presented. The architecture of the PoC was based on the high level attestation concept and architecture from Chapter 4 and it implemented the mechanisms described in Chapter 5.

The PoC consisted of three different components: DMC, RVG and DVC. The DMC was implemented as an LKM on the SuE and responsible for secure measurement and reporting. For this purpose DMC utilized a TPM to securely store its measurements. The second component, the RVG was responsible for generating reference value data on basis of trusted ELF files and storing them in a RVS. It was deployed on as part of the VS and thus carried out its tasks on an initially trusted system. The third components was the DVC. It was also deployed on the VS, received the measurements and carried out different verification steps. At first, it verified the freshness and authenticity of the received SSR,

then the integrity of the DML and last extracted the measurements and carried out the verification on basis of the reference value data available in the RVS.

All three components were composed of different subcomponents and implemented related guideline-triplets that implemented the actual business-logic. These guidelines provided a flexible way to attest system components and thus provided attestation mechanisms for user space processes, LKMs and the kernel.

Second, a security evaluation that simulated attacks on a SuE for different processes was carried out and its results were presented. The objective of this security assessment was to confirm that the attacks to be detected by DRIVE are actually detected. For this purpose a DMC was deployed on the system and took measurements before and after the attacks. The attacks were the same as introduced in Section 3.3.4 which were also used during the security analysis in Section 5.4. Afterwards, the measurement were reported to the DVC and an attestation was carried out. The results of the attestation, and in particular which part of the verification actually recognized the attack, were presented in different Tables. The study has demonstrated that all attacks that were expected to be detectable were detected during at least one verification step. As anticipated, the non-control data-based attack A6, could not be detected by DRIVE. This means the PoC implementation worked as intended and all anticipated results could be confirmed successfully.

Third, a scalability evaluation based on the PoC was carried out and analyzed. The objective was to evaluate how and to what extent DRIVE affects the systems involved when it is deployed. In particular, the DMC deployed on SuE was evaluated. Since the SuE system's actual task is not to perform an attestation, the effects of DRIVE on this system are particularly significant.

Nevertheless, the effects on the system during the actual measuring process are perceptible, especially if all components were measured each time. Depending on how extensive the measurements are, i.e. how often they are carried out or how many individual measurements are made, the system is affected for longer or shorter periods of time. However, the system itself remained constantly reactive and could be used without interruption. The effects were not felt to be too great to carry out complete system measurements at more or less frequent intervals, as long as there was no overlap of several measurement processes at the same time. During the evaluation, different optimization strategies were evaluated and compared to acquired baseline values. All optimization strategies were able to decrease the overall time of corresponding operations significantly and thus reduced the impacts the DMC has on the SuE considerably. However, it was argued that all optimization strategies also come with security implications. For this reason, it must be carefully analyzed and decided if optimization should be applied at all or which type is most adequate on a case to case basis. Finally, it can be concluded that DRIVE is very well suited to realize a continuous measurement on SuE for the defined use case. Various strategies can be used to minimize the influence on the SuE. Ultimately, however, the

application range of DRIVE and the measurement range depend on which system is to be attested.

Chapter 7

State of the Art and Related Work

System security technologies are diverse and over the past years many of these technologies have made important contributions to significantly increase system security. When considering the runtime of a system, the available technologies can be divided into two different groups. The first group includes technologies that are used by programs before or during the loading process. Within this group, a further distinction can be made between technologies based on digitally signed programs and technologies measuring and verifying the integrity of programs only. Typically, these technologies are carried out on the file level and do not take system memory content into account.

The second group of technologies addresses the actual runtime of programs. This means everything that affects the program's system memory parts after the program's loading process has finished. Within this second group, different approaches can be distinguished. Some of the technologies address different integrity properties of programs, factoring in certain characteristics of programs or the program behavior itself. Other technologies addresses the context in which the program is executed. This means that certain boundary conditions are defined which actively influence or control the program. For example, memory addresses can be randomized or certain access control restrictions can be set in memory.

In this thesis, the focus lies on the integrity properties of programs and the following section is about these integrity properties. Furthermore, related and relevant research is presented and discussed. In a first step, work related to the loading process will be discussed, which forms the basis for a secure operation of a system and partly utilizes building blocks, also used in this work. In a second step, technologies and research results that are part of the core area of this research will be presented and discussed. In particular, the measurement and verification of memory content based on integrity properties are laid out. In this context, it will be explained how existing technologies differ from the technology established in this work.

In the final step, further integrity-related technologies and research will be presented

that have has already been referred to and discussed in more detail throughout this work. These technologies make use of advanced integrity properties and therefore clearly distinguish themselves from the developed concepts and solution in this thesis. Above that, it will be explained if and to what extent existing technologies can be used with this thesis' protection technology.

7.1 File-based Integrity Protection during Load Time

File-based integrity protection technologies are typically applied before or during the actual load time of programs. The two main variants used are *Measured Boot* and *Secure Boot*. Both technologies measure the program by calculating a hash digest of the program's file, but differ in the subsequent operations conducted.

7.1.1 Measured Boot

Measured boot is a core concept of Trusted Computing technologies. It relies on a tamper-proof security module, typically a TPM, in order to anchor measured programs' integrity values securely for subsequent reporting and verification. The work from Sailer et al. [33] establishes the foundation of the enhanced measured boot concept in Linux-based systems today. Sailer's work is considered as the very first design and implementation of the Integrity Measurement Architecture (IMA) technology that extends the Trusted Computing attestation concepts all the way from BIOS to application level of a system. IMA is implemented as an LKM in the Linux OS kernel and measures program files, i.e. kernel, LKMs, executables, libraries and other security-related files, before they are loaded into the system memory. The implementation utilizes a TPM to anchor and store the measurements and makes use of certain TPM operations to generate a report. Additionally, it describes the basics of a remote attestation protocol used for exchanging reports with a remote attestation server. Furthermore, the work demonstrates the verification process that describes how the reported measurements can be verified on the basis of well-known reference values.

With regard to the DRIVE technology developed in this thesis, some characteristics of the work of Sailer and in particular IMA are recognizable. First, DRIVE uses a similar high level concept of measurement, reporting and verification, including the transmission protocol used. Due to the very general description of the original attestation concept, this high level concept seems to be applicable for all systems that want to use such a concept. Secondly, both technologies use security modules for the secure storage of the measured programs, especially TPMs. However, TPMs used by DRIVE are only one implementation option. IMA, on the other hand, was developed under the assumption of an existing TPM and therefore does not consider any alternative implementations.

However, the most important innovation this work contributes is that DRIVE measures, reports and verifies other system resources, i.e. memory content and metadata, and can be used multiple times. While Sailer's concept is limited to a one-time measurement of file-based resources at load time, DRIVE allows for a continuous and repeatable measurement of memory content during runtime. This also affects the reporting and verification of system states. The runtime integrity property is not provided by IMA because it does not consider threats or attacks after the actual loading process. As part of the security analysis, see 3, it has been demonstrated that runtime attacks are indeed a valid and common practice and therefore it is essential to continuously measure and attest system states throughout the entire runtime of a system. Nevertheless, DRIVE and IMA are two technologies that complement each other very well. In this case, IMA can detect whether malicious programs have been loaded and DRIVE detects whether previously benign programs have become malicious during runtime.

Other research is based on the basic concept of Sailer's work and on Trusted Computing technologies. The works of Jaeger [106] and Sadeghi [107] should be mentioned here in particular. Jaeger concludes that a mere attestation of programs is not sufficient and develops in his work a model that considers the data flows of programs. This allows a system to be divided into different domains and to define policies that allow, limit or prohibit specific communication between programs. Sadeghi's work is primarily concerned with the scalability of attestation. It is assumed that it would make more sense to attest only certain properties of a system instead of attesting programs themselves. Both their works have no intersections with the work presented in this thesis, as they assume a different view of the systems involved or program's behavior. However, follow-on research could investigate the extent to which DRIVE could contribute to the works of Jaeger and Sadeghi or incorporate ideas of their work.

7.1.2 Secure Boot

The second technology mentioned is Secure Boot, its core concept and architecture is based on the work of Arbaugh [19]. Arbaugh concludes that given a secure and trusted starting point, subsequent components and programs can be measured and integrity checked previously to their activation or loading. This leads to an implied chain of trust that intrinsically proves the trustworthiness of a system. On the technical side, this concept can be implemented with a so-called Root of Trust (RoT) and digital signatures. This RoT establishes the initial trust anchor that measures the subsequent component and uses a digital signature to guarantee the integrity of the measured component or program. Each such loaded component is thus implicitly trusted, which enables this now trusted component to measure its subsequent component. Microsoft® Windows implements secure boot at least until the so-called "Early Launch Anti-Malware System" takes over, see [108]. For Linux systems, IMA-appraisal [109] is available that limits the loading

mechanism to loading signed programs only. Secure boot is also available on mobile phones and is generally used by all major vendors, see [110–112].

Similar to the measured boot, secure boot offers its security features only in terms of load time integrity. From a conceptual point of view, however, secure boot and DRIVE have nothing in common. Secure boot is again file-based and is executed only once when loading components or programs. Both architectures are essentially different. Still, secure boot provides a very strong basis for a subsequent integrity verification at runtime, since it enforces a strict policy to ensure that no malicious components and programs have been loaded or executed that undermine integrity at loading time already. For this reason, secure boot is well suited as a complement to DRIVE.

7.2 Integrity Protection during Runtime of Programs

Research aimed at providing technologies to protect programs during runtime is the main focal point of this work. For this reason, especially technologies and research results that deal with the integrity measurement and verification of static program parts in the memory are discussed. Only literature on static program parts was taken into account since a verification of predictable dynamic memory content has not been considered by any of the related research. The transition between predictable and unpredictable memory content forms a natural boundary in this area. There are no known technologies or research results that consider or develop security technologies for both variants. Therefore, following the integrity verification of static memory, related research on protecting the integrity of these unpredictable memory areas will be briefly presented and discussed.

7.2.1 Integrity Protection of Static Memory Content

It is generally possible to differentiate between technologies and concepts that are implemented with or without hypervisors.

Non Hypervisor Based Integrity Protection

In this research area, researched technologies and solutions utilize different techniques to collect and verify state information on the targeted system parts. Specifically, Linux kernel rootkit detection was a particular target of research interest.

Copilot, as proposed by Petroni et al. in [89], aims at detecting persistent kernel rootkits on a host system. Copilot utilizes a PCI monitor card, in order to create snapshots of certain well-known critical kernel memory regions, which can be reported to a so called *admin station*. The verification of the measured snapshots is based on well-known hashes for collected content. Those well-known hashes are taken at some point in time, where the measured content are believed to be in a correct and benign state. The concept is similar

to user-mode detection programs, like, for instance KSTAT, chkrootkit, Rootkit Hunter, and Rkscan, which also calculate hash values from believed benign sources and compare measurements against those references. Copilot, specifically, focuses on the measurement and verification of the kernel's and LKM's `.text` segments and `jump tables`. Illicit modification can be detected, if well-known hashes are available.

The measurement and verification concept of Copilot is fundamentally different to DRIVE, when it comes to measurement acquisition. Since Copilot makes use of a PCI monitor card, it provides some arguably strong security guarantees. When using an external measuring instance, it is almost impossible for an attacker to manipulate the measuring process himself. One disadvantage that Petroni addresses is that the PCI card cannot ensure that it also measures the correct memory area. Since the PCI card is based on physical memory and works with fixed address ranges, an attacker can modify the page table of the kernel and reallocate functions or the entire kernel to another unmonitored area. Since Copilot has no knowledge about the virtual memory, it can be bypassed by changing the page table.

In principle, this negates the advantages of the external measurement instance for a powerful adversary. Therefore, Copilot does not provide any advantage regarding assured security properties in the context of the measurement process. For this reason, DRIVE and Copilot provide comparable security guarantees in this regard. The advantage of DRIVE over Copilot is that DRIVE is not limited to measuring the kernel. On top of that, Copilot does not offer the possibility to measure structures other than the kernel itself, since it has no semantic runtime information. But, this semantic runtime information is necessary to measure user space programs effectively and to verify them, see Section 4.3.2. In summary, this means that the Copilot does not offer any significant advantages over DRIVE. On the other hand, DRIVE offers a variety of additional measurement and verification options that cannot be covered by Copilot.

Another work, proposed by Rutkowska in [113], compares `.text` segments of memory content with their corresponding files counterparts. Rutkowska argues that it should be feasible to detect illicit modifications by employing this verification mechanism. The work focuses primarily on Windows-based systems and aims mainly kernel root kits as its primary target. According to Rutkowska, the research result is implemented as a PoC, the System Virginity Verifier, which implements the verification based on a comparison of in-memory `.text` segments against their file-based counterparts. This promising work was not continued after 2006 and neither the PoC nor further documentation is available.

The research results presented in this thesis confirm the assumption made by Rutkoskwa. Furthermore, the present work shows that not only static `.text` segments but also all predictable memory content and metadata can be attested in order to prove the system state to an external observer.

Finally, *LKIM* [114, 115] was proposed as another approach for Kernel Integrity Mon-

itoring. LKIM utilizes similar mechanisms to DRIVE during reference value generation, called base-lining, i.e. it also generates cryptographic hash values based on simulation of the loading process of the kernel and LKMs. Regarding dynamic LKM behavior, LKIM does not detail its base-lining and verification mechanisms. For this reason, a comparison with the research presented in this work is not possible.

Apart from the discussed research, different tools for memory forensics and extraction are available for different OSs. For instance, LIME [15] targets the extraction of memory images, whereas the *Volatility* Framework [16] and *FATKit* [14] additionally support further analysis of the extracted memory content. Memory extraction is usually implemented as an LKM using the internal kernel APIs and data-structures (c.f. 6.1.1). Volatility is considered the most recent and advanced tool and provides a huge amount of analysis plug-ins [12]. Typically, memory forensics is meant to analyze the behavior of infected systems. In other words, it is currently not used to detect or report malicious behavior.

Finally, it can be concluded that the presented related research focuses mainly on the integrity protection of kernels and their modules. None of the research discussed or known to the author addresses such a broad spectrum as the concepts and solutions presented in this thesis. The related architectures do not take into account the use of security modules or describe their application insufficiently. Apart from that, there has been no known research in this area. It is not known whether one of the solutions discussed is widely used. More specifically, no evidence has been found that any of the concepts presented have been used other than in academia. With regard to available memory extraction tools, Volatility seems to be used in the field of memory forensics though.

Hypervisor Based Integrity Protection

Hypervisor-based technologies have the great advantage of being able to monitor examined virtual machine states without restrictions. So it is common practice with these solutions to jump back into the hypervisor-based monitor for certain actions that are executed in the VM to perform attestation. In particular, the hypervisor has full control over the page tables of the virtual machine, enabling it to detect and and react to changes in access permission control structures in a targeted fashion. Litty proposes *Patagonix* [116], a technology that from its basic principle comes closest to the attestation mechanisms used by DRIVE. Patagonix uses a virtual IRQ-based trapping mechanism to jump into the hypervisor when code execution in a VM is detected. Subsequently, Patagonix executes measurement and verification mechanisms depending on the concrete type of executed code. PIC is represented and verified by hashes similar to DRIVE. However, RCC verification differs from DRIVE since it is implemented by applying the reverse function of the corresponding loader during measurement. This method was mentioned in Section 5.2.1 as an alternative, but not implemented or considered, because of information loss and complexity during during measurements. In addition, one major challenge by applying this

reverse method is to determine whether a transformation was valid or invalid. Patagonix does not describe how it comes to this decision. With regard to hypervisor-based technologies many other research in this area is available that is mostly based on similar methods, c.f. [18, 74–79, 117–119]. Most recently, measurement of a VM was also demonstrated by Chang in [120], utilizing Linux paging mechanisms and hypervisor introspection. Similar to Patagonix measurement and verification are performed on the hypervisor. However, in Chang´s work, the executed on-disk files are decomposed into individual pages and cryptographic hash values are calculated as reference values instead. Measurement acquisition is again similar to DRIVE; however, the proposed solution employs intrusive enforcement mechanisms relying on an anterior verification similar to Patagonix. That is, VM operations are only performed after successful verification in the hypervisor and are triggered whenever modifications in the page-table are detected. The solution utilizes a TPM for verification, yet, concrete verification mechanisms have not been published. Moreover, the role and exact tasks of the TPM are not explained.

The wide use of trapping mechanisms in the aforementioned research is not unexpected because the semantic gap, one of the fundamental problems in virtual machine introspection, c.f. [95], must be circumvented or resolved before a meaningful integrity- or any other state-based verification method can be used. The same problems encountered for introspection are also relevant for hardware-backed isolation technologies.

Similar to the presented work, DRIVE is also suitable for providing integrity protection in the area of hypervisor-based solutions. Since DRIVE is similar in terms of the functionality provided by Patagonix, it is perfectly possible to leverage a VM-based trapping mechanism for measurement and verification. The flexible architecture described in this thesis does support this on a conceptual level. Since in this case the hypervisor system is assumed to be implicitly trusted and the use of a security module would not be necessary. In addition, by using DRIVE, a more reliable verification would be available because DRIVE does not rely on the reverse function applied by the loader. The anterior verification mechanisms of Patagonix and Chang's work are currently not considered by DRIVE, but could be implemented retrospectively. In this respect, DRIVE's architecture has no limitation.

Other Solutions

Hardware-backed solutions have also been used to attest or enforce certain properties on systems comparable to hypervisor-based systems. Specifically, *SPROBES* [84] utilizes ARM TrustZones to enforce invariants detecting illicit modifications to Linux kernel code during runtime. More precisely, certain normal world instructions are rewritten to redirect the control flow to the secure world, where the invariants get evaluated. This technique is comparable to aforementioned trapping mechanisms. The invariants rely on enforced strong memory access permissions and evaluate different metadata to deduce whether

the system was maliciously altered or not. Consequently, the invariants and control flow transition instructions are chosen and designed so that unintended modification may not occur undetected.

SPROBES is a very promising concept that aligns well with DRIVE. In particular when DRIVE is implemented as part of the OS kernel or as an LKM, SPROBES is able to provide strong security guarantees that reduce the attack surface to disable or circumvent DRIVE's measurement process. The security guarantees are enforced on a higher security domain than the OS kernel itself.

Not closely related, but worth to be noted, is *SKEE* [88], a novel isolation approach that introduces a protected address space at the same privilege-level as the Linux OS kernel for the ARM platform. SKEE does not rely on hardware-based ARM TrustZone or other virtualization extensions, and thus is considered a lightweight alternative for a Trusted Execution Environment tailored explicitly for kernel-level security monitoring and protection. While concrete monitoring and protection schemes have not been described or published, SKEE seems to be a promising concept and suitable for a DRIVE MA implementation, similar to SPROBES.

7.3 Control Flow and Data Flow Integrity

To summarize the related work from the previous section, DRIVE is a flexible and practical orthogonal security technology that enhances the established state-of-the-art by providing system monitoring capabilities at runtime on a broad scale. In general, DRIVE supports hypervisor-based and non-hypervisor-based approaches, although the hypervisor-based approaches were not an explicit design focus of this thesis. DRIVE significantly increases overall system security and closes the gap to runtime integrity technologies, such as CFI and DFI that aim at providing security technologies to protect against other memory related attacks.

The limitations of DRIVE were previously discussed in Section 4.3.3. Furthermore, some related work is presented and discussed briefly in Sections 3.2.4, 3.3.5, and 4.3.3. Hereinafter, this section briefly revisits the research field and describe how the research established by the thesis relates to this area.

7.3.1 Control Flow Integrity

A lot of research has been done in the field of CFI over the last few years. One of the first publications in this field was presented by Abadi in [49, 121]. CFI serves as a security mechanism to prevent code reuse attacks and its basic idea is to recognize deviations from the original CFG in order to prevent an execution in case of a detected deviation.

Perfect CFI is to mitigate all code reuse attacks; but, due to its high overhead, perfect CFI is impractical. For this reason, coarse-grained CFI is often deployed, balancing be-

tween security and performance, c.f. [122, 123]. Coarse-grained CFI provides only partial protection and has different issues that have been discussed in [122–125]. In particular, Carlini argues in [39] that CFI without backward-edge protection is insecure and thus shadow stacks are essential for CFI to guarantee a secure solution.

There are countless possibilities for implementing fine- and coarse-grained CFI directives. In the first years, the concept of code tagging was mostly used as proposed by Abadi. This was followed by dozens of proposals for solutions which, however, can only be distinguished conceptually by variations in the actual recognition method. The implemented enforcement policy after detection is always the same, i.e. the execution is terminated immediately after a successful detection. A complete analysis of all techniques would go beyond the scope of this work. The interested reader will find a very detailed overview of available CFI solutions and implementations in [53] and especially for hardware-based approaches in [56].

In addition to research, CFI is adopted in some recent compiler versions, for instance for gcc and clang, c.f. [50, 51, 126], and Microsoft C/C++ StackGuard [65]. In addition to that, hardware based solutions are also adopted, e.g. shadow stacks [54] for Intel CPUs or, most recently, pointer authentication [127] for ARMv8.3 CPUs.

To summarize, CFI is a technology used to detect code reuse attacks only; this cannot be implemented by DRIVE. From a conceptual point of view, both technologies are completely different. CFI considers in particular changes in unpredictable highly dynamic memory areas. In this context, content-based methods for checking integrity are not useful and have never been considered in the literature to the best knowledge of the author. However, this limitation also applies to the fact that CFI is not applicable for the integrity check of other memory areas. Specifically, this means that CFI does not provide a solution to perform a necessary content-based verification. From this perspective it can be concluded that both technologies, DRIVE and CFI solutions complement each other very well and should therefore be used together.

7.3.2 Data Flow Integrity

Non-control data attacks are the next evolution of attack techniques, but were described in their basic form relatively early by Chen [128] in 2005. The basic form of non-control data attacks is limited to attacking specific data structures, where only data variables and data pointers are explicitly modified, but never code pointers. For this reason, these attacks are not recognizable by an analysis of the control flow and require other methods. Furthermore, relatively up-to-date, more complex attack methods have been developed which are no longer limited to the modification of individual data structures. These techniques were presented together with the proof of their Turing completeness by Hu [129] in 2016.

A first defense strategy against this type of attack was presented by Castro [130] in 2006. The term Data-flow Integrity was used here for the first time. The core idea is to extract

the data flow graph of a program and use it for verification. This means that if deviations from the regular data flow graph occur, an attack is recognized. Still, this method has a significant performance overhead of approximately 104% on average, c.f. [131]. A further countermeasure has also been proposed, known as Dynamic Taint Analysis. Here, a tainted program is analyzed during its runtime. For example, by analyzing valid user input. On the basis of the collected analysis results, it is possible to detect deviations with respect to the collected analysis data, c.f. [132] and [133]. There is currently a lot of research in this area, which means that further attacks and new defenses are to be expected. A deeper analysis of non-control data-based attacks and defenses would go beyond the scope of this work. A comprehensive overview is provided by [134] and [135]. In relation to DRIVE, non-control data-based attacks are generally not detectable. But there is one exception. If data has been manipulated in areas that are dynamic but predictable, it is possible to perform a content-based verification. This was briefly discussed in Section 5.4.3. However, such an approach should be further explored and is therefore left open for future research activities.

Chapter 8

Conclusion and Outlook

8.1 Conclusion

Many cyberattacks of the past years have impressively shown that every computer system can be compromised. Despite all the security mechanisms that are in place, it is almost impossible to completely rule out an attack on a computer system. This is based on the fact that today's systems run software that simply contains vulnerabilities. All efforts notwithstanding, this fact will not change in the near future.

In this thesis, the novel runtime protection technology DRIVE has been presented. DRIVE increases overall system security by bridging an existing gap between well-known and future security technologies. This bridge is the continuous reliability and trustworthiness assessment of runtime system states. This improvement has been achieved on the basis of the contributions made in this thesis that will be reviewed in the following section.

8.1.1 Contributions

(C1) Security analysis This thesis has made the contribution (C1) by providing a detailed security analysis that focused specifically on runtime attacks. The contribution is based on published results in [1].

First, today's malware has been briefly introduced and classified based on the key properties of stealth and persistence, which were found to be interdependent. Second, a detailed study of runtime attack techniques – the building blocks of malware – has been carried out, presented on the basis of control flow graphs and classified with regard to persistence and stealth. It has been confirmed that an intelligent combination of these attack techniques, which have been introduced as hybrid attacks, can lead to the avoidance of almost all countermeasures used today. Third, common countermeasures have been presented and briefly discussed. It has been concluded that all countermeasures increase

the complexity of attacks but are eventually avoidable by using hybrid attacks.

Finally, the long term-goals of adversaries have been described and attack methods have been developed that implement multi-step hybrid attacks to deactivate countermeasures for system takeover. For this purpose, a model has been defined that allows the modeling of multilevel hybrid attacks. On the basis of this model, concrete attack scenarios have been presented that were used for later evaluation of the concept and prototype.

(C2) Novel and holistic runtime protection technology The main contribution this thesis has made is the provisioning of DRIVE, a novel and holistic runtime protection technology that supports granular, reliable and continuous attestation of various memory data artifacts during software runtime. This contribution has been published in [1]. In this regard, DRIVE has filled the identified gap between load-time attestation and future CFI and DFI security technologies and thus contributes to enhancing overall system security considerably. This has been achieved in particular by providing novel attestation mechanisms that determine system runtime states based on static and predictable dynamic memory contents and metadata. A high-level attestation concept and flexible architecture have been defined and designed that support different instantiations. This flexibility increases the applicability of DRIVE for use cases that rely on different systems or that need to adopt constrained attestation concepts. Based on this architecture, a specific software architecture has been instantiated and corresponding building blocks have been defined that allow different implementations and enable use case-specific realizations.

Based on the provided use case and the conducted deployment analysis, one specific software architecture has been instantiated and described in detail. This has been done by a classification of memory artifacts and description of the necessary data structures, mechanisms and a data transmission protocol for the most basic static, predictable data case. This initial solution has established a secure measurement of memory content on a SuE, the creation of a security module anchored SSR, the secure transfer of the SSR to a VS based on an attestation protocol and the evidence proof verification of the collected measurement data. Subsequently, the initial solution has been successively refined to attest further memory artifacts and metadata. For this purpose, the data structures have been expanded and the necessary mechanisms have been described. The final solution supports attestation for predictable static and dynamic memory runtime data and metadata-based attestation for unpredictable runtime memory data.

In addition, a security analysis has been carried out that has evaluated the final solution's capabilities to detect attacks with regard to the derived attack scenarios. The result of this analysis was that all attack scenarios, except for one, are detectable by DRIVE. This result has been in line with the anticipated outcome.

(C3) implementation and evaluation The contribution (C3) has been made by implementing DRIVE as a PoC and describing and analyzing the PoC with regard to its software components, security capabilities and effects of the measurement component. The implementation was published in [2], the security analysis in [1] and the performance evaluation in [1, 3].

DRIVE's MA has been implemented as an LKM on SuE utilizing a TPM for secure measurement anchoring and report generation. The VA and RVG have been implemented as normal applications. The PoC is based on guidelines, an agreement that defines specific measurement and verification policies, and supports attestation of user space applications and libraries, LKMs and the kernel. The PoC is considered as a mature solution, has been successfully deployed on different architectures and different kernel versions, and has been used as a foundation for multiple different product implementations[37]. The security analysis has been carried out by simulating all defined attack scenarios and subsequent attestation after the attacks. As a consequence, it has been confirmed that the PoC is able to detect all anticipated attacks. An evaluation of MA on SuE has been carried out. The results of the evaluation demonstrate that the measurement process affects the SuE performance but stays within reasonable bounds. Furthermore, different optimization strategies have been evaluated, with the result that the optimizations reduced the effects to SuE significantly.

To summarize, the PoC has provided convincing arguments to demonstrate DRIVE's capabilities and applicability. For this reason, an adaptation by the industry has been achieved.

8.1.2 Research Questions and Objectives

This section will review the defined research questions (Q1)-(Q4) and set them in relation to the research objectives (RO1)-(RO4). This will be done by providing a detailed summary of the results in accordance to the research objectives. Consequently, the results established within the context of the research objectives represent the answers to the research questions raised.

Determination of Capabilities, Limitations and Characteristics of Runtime Attacks

The results were established in the security analysis in Chapter 3 and address the research question (Q1): ¨*What are the capabilities, limitations and characteristics of software runtime attacks?*¨ Accordingly, the goal of the security analysis was to identify and analyze threats and attacks at runtime with regard to the research objective (RO1).

[37] Parts of the DRIVE technology and architecture were implemented in mobile phones, base stations and routers. Details of specific products or manufacturers may not be included in this thesis for reasons of intellectual property protection.

To this end, malware was first examined for its key properties of persistence and stealth. The result of these studies is that both properties are related. Persistence is generally determined by which concrete system component was attacked and where exactly the malware was placed. Stealth is determined by the placement of the malicious code, but it must be determined whether the malicious code was placed in predictable or unpredictable data. Additionally, it has been shown that both properties, persistence and stealth, influence one another and therefore are mutually dependent. As a general rule, an increase in persistence reduces the secrecy, and an increase in stealth reduces persistence. As a result, it has also been concluded that shorter attacks are harder to detect than long-lasting attacks.

In the next step, attacking techniques were introduced and analyzed on the basis of the key characteristics established. To this end, runtime attack techniques were primarily investigated. It was found that only a change of the control flow graph allows the execution of the malicious code at runtime. Furthermore, it was shown that high persistence can only be achieved by also modifying persistent data. In contrast to this, a high degree of stealth is achieved by modifying short-term data, which results in the short-term execution of the malicious code. Since persistent data tends to be predictable and short-term data tends to be unpredictable, it is not possible to gain high stealth and high persistence at the same time.

It was discovered that the capabilities of the malicious code play only a secondary role. The capability of malicious code is limited solely by which concrete software component is being attacked. In the attacked component itself, however, malicious code can implement arbitrary malicious functionality. Moreover, countermeasures were presented and discussed that are readily available and activated on almost all modern computer systems. However, the assumption was that countermeasures are avoidable. To prove this point, hybrid attack techniques were introduced that allow a combination of various attack techniques. It was found that by skillful arrangement of different attack techniques, every countermeasure can be deactivated or circumvented. In addition, hybrid attacks can also reduce the complexity of attacks. For example, it is possible to deactivate only targeted countermeasures with very complex attacks and then use simpler attack techniques to place and activate malicious code in the system.

Based on these hybrid attacks, a simple model was developed that allows us to define and describe these hybrid attacks. To this end, the model describes specific phases and uses complex attack techniques to disable defensive measures and simple attack techniques to place and activate malicious code. On the basis of this model, various attack scenarios were then developed that represent real attacks on programs. Accordingly, multiple attack scenarios were defined and discussed. These attack scenarios are further used in the course of this thesis to verify their effectiveness on a conceptual and practical level.

The most important discovery made in this chapter was that all existing measures

can be circumvented by a clever combination of different attack techniques. Computer systems can therefore be attacked and completely compromised at runtime. For this reason, it was necessary to provide a mechanism that could detect whether a system has been compromised or is still in a sound state. Consequently, the runtime protection technology developed in this work adds an important contribution to the state-of-the-art in security. The assumption that it is not sufficient to verify or attest a software only once at the time of loading has been confirmed. Runtime attacks can occur at any time after loading, and thus a validation at loading time only reflects a small part of the current system's state.

Finally, it can be concluded that the security analysis has contributed very important insights and fully implements and confirms the research objective (RO1). As stated in the research goal, the results of this security analysis are very useful in the further course of the thesis and thus serve as a basis for the elaboration of the next research goal (RO2). The research question (Q1) is hence answered by the results provided in Chapter 3.

Provision of DRIVE's Implementable Architecture that Enables a Runtime Attestation Concept

Based on the results of the security analysis, the research question (Q2) was addressed in Chapter 4: *"What is necessary to establish a protection technology that implements continuous and reliable monitoring of the runtime state of systems?"* The question is in accordance with the results of the security analysis, which confirms that the current countermeasures are insufficient. Referring to the research question (Q2), the answer is provided in the results of Chapter 4.

In order to realize the research objective (RO2), a concept and an architecture was developed on the basis of known attestation technologies which allow the collection and evaluation of the runtime state of a system. This concept and the architecture were then progressively refined and analyzed in the course of the chapter. Two systems were considered in the concept: SuE, which carries out a measurement and reporting of the system state, and VS, which evaluates the system state on the basis of measured data received. This high-level concept is the foundation for the runtime protection technology DRIVE which was developed in the course of the work.

On the basis of the high-level concept, an architecture was designed that arranges the components of the concept into building blocks and describes them. For this purpose, a MA was defined as a central building block on SuE, which measures the system memory, anchors the measuring results in a security module and creates a measurement report linked to the anchored information. This measurement report is received on the verification side and is processed by the building block VA. The task of VA is to verify the measurement report by using reference values. This architecture described the building blocks and involved systems on an abstract level and was subsequently refined to a con-

crete software architecture to specify and describe the involved components, mechanisms and data structures more precisely.

Subsequently, a deployment analysis was carried out to analyze and discuss the different implementation variants of the components. In particular, the measurement and verification were considered and evaluated for their reliability with respect to isolation variants. The most important results of this analysis are that measurement and verification should ideally be realized on two different and physically isolated systems. This is the only way to achieve a high reliability of the attestation result. A measurement should ideally be realized in a higher protection domain than the measurement targets and should incorporate a security module. A measurement by physical separation is possible only to a limited extent and can only be implemented appropriately with the help of the operating system kernel. Accordingly, the operating system kernel is the best implementation option here.

The chapter concludes with an analysis of DRIVE's limitations. It was found that DRIVE is not able to detect code reuse or non-control data-based attacks due to its architecture. In the manner DRIVE realizes the attestation, it is not possible to make decisions based on the program flow. These can only be done within the program itself, just as CFI technologies are implemented. It is therefore recommended to use DRIVE and CFI together.

The most important result of the chapter is the provision of an implementable architecture that enables an attestation concept for the assessment of runtime system states. Important findings have been acquired from the analysis which will have an impact on the components and mechanisms to be incorporated and which must be taken into account when implementing DRIVE as a solution.

The research objective (RO2) has thus been achieved and serves as a foundation for the technical implementation of the runtime protection technology DRIVE, which is further addressed in research objective (RO3).

Provision of DRIVE's Capabilities, Procedures and Constraints

The goal of Chapter 5 was to develop and provide the technical solution of the runtime protection technology DRIVE. The technical solution is based on the software architecture developed and considers the analytical inputs from Chapter 4 accordingly. The research question (Q3) raised was: *"What are the capabilities, procedures and constraints of a continuous runtime protection technology?"*

In order to provide an answer to this question, Chapter 5 addressed each raised subtopic individually. At first the details of the necessary procedures and mechanisms were identified, defined and thoroughly described.

To this end, measurable and verifiable runtime memory artifacts were identified and analyzed. The result of this analysis was that a distinction between static, predictable dy-

namic and unpredictable dynamic data is most appropriate. This is because this division enabled the development of general data structures and procedures based on the easiest memory structures, i.e. the static memory artifacts. Subsequently, in the course of development, the individual data structures and attestation procedures were developed for each type, i.e. static, predictable and unpredictable. Consequently, the individual precise attestation mechanisms were described one after another. This was done by describing all the technical details coherently and by utilizing the same general data-structures for measurement collection, anchoring, reporting and verification.

In summary, the following data structures and procedures have been developed. The measurement concept was used to collect and provide the measurement data from the system memory of the SuE and was thus implemented in the MA. The measurement data were collected in an SSR and additionally anchored in a security module. This allowed an authenticity and integrity verification of the measurement data on the verification side. The SSR was transmitted to the VA on the VS using a customized remote attestation protocol. There, the transferred data was checked for authenticity and integrity. The actual verification of the measurement data was then carried out on the basis of reference values. To this end, the process of generating reference values was first described, which was specified in different variations depending on the measured data, i.e. statically, predictably dynamic or unpredictably dynamic. The verification of the measurement data was then carried out on the basis of calculated reference values or other guidelines. If all verification steps were successful, the runtime state of the system was classified as trustworthy.

The capabilities and constraints were derived from a security analysis of the developed procedures. First, the capabilities were analyzed on the basis of the defined attack scenarios taken from the security analysis.

The objective of this analysis was to determine whether the concepts developed were sufficient to detect defined attacks. The results of the security analyses were therefore a classification of the developed concepts with regard to the detection of the defined attacks. To summarize the results, all attacks that were supposed to be detected were covered by at least one of the concrete methods. However, the defined non-control data-based attacks were not detected by any method and access permissions are only detectable if they remain persistent.

The constraints were derived from an additional analysis that considers unpredictable dynamic data attestation. The result of this analysis is consistent with the findings from Chapter 4. DRIVE cannot detect code reuse and non-control data-based attacks in dynamic unpredictable memory areas. Certain effects of these attacks, which can be determined by metadata verification, are detectable as long as they are active.

All aforementioned results and findings provide the answers to the research question (Q3). DRIVE is a novel runtime protection technology able to detect many relevant attacks

that affect the runtime state. If modifications happen in predictable memory portions, which is often the case for persistent modifications, DRIVE is able to detect these attacks and thus can repeatedly and reliably assess whether a system has been attacked or is in a trustworthy runtime state. For this reason, the research objective (RO3) is regarded as accomplished.

Realization and Assessment of DRIVE

The goal of the final technical Chapter 6 was to demonstrate that the developed runtime protection technology DRIVE from Chapters 4 and 5 can be adopted and implemented as a concrete software implementation and deployed on a system to carry out a runtime system attestation. Moreover, based on the deployed solution, an evaluation of its promised security capabilities and effects on the SuE will be analyzed. The results provide the answers to the final research question (Q4): *"How can a runtime protection technology be realized and assessed on the basis of a designed runtime protection technology?"*

The results were presented on the basis of the description of a prototypical implementation of DRIVE, a subsequent security evaluation, and a performance and scalability evaluation of the PoC.

The implemented software architecture of the PoC was based on the concept and architecture of Chapter 4 and implemented mechanisms as described in Chapter 6. The PoC consisted of three components in total: DMC, RVG and DVC.

DMC realized the MA and incorporated the measurement and reporting mechanisms. It was implemented as an LKM and allowed the measurement and reporting of user space processes, LKMs and the kernel. A TPM 2.0 was used as a security module to securely anchor the collected measured values. The RVG component was implemented as a standard user space program and accumulated its reference value on the SuE. It collected the reference values on a freshly installed system and accumulated the reference values in a database that was later provided on the VS. DVC represented the VA and was also implemented as a standard user space program. The VA needed access to the provided reference value database and could then verify transmitted SSRs. At first the authenticity and integrity of the SSR was verified and afterward the DML was extracted. Depending on the type of data included in the extracted DML, different verification mechanisms were used.

The successful implementation of the PoC constitutes an important step in determining the feasibility of the work. It demonstrates that the developed architecture and mechanisms can be integrated and implemented.

The subsequent security evaluation simulated the attack scenarios defined in Chapter 3 on the SuE. After each attack, a measurement was performed and a generated SSR was verified. In the process, it should be determined whether the executed attacks were also detected by the PoC implementation. The results of the security evaluation confirmed that

all attacks that were expected to be detectable were detected in at least one verification step. This showed that the PoC implementation worked as planned and delivered all expected results. However, the PoC could not detect the non-control data attack A6, as expected.

In a final performance and scalability evaluation, the PoC was analyzed with regard to the performance effects on the systems involved. In particular, the DMC deployed on SuE was evaluated because in this case, the effects of DRIVE on this system were particularly significant. The measured effects were recognizable, but it was possible to carry out regular complete system attestations. Some optimizations were also tested to further reduce the effects on the SuE. The optimizations evaluated were very effective in this regard but had an impact on the security guarantees of DRIVE. Of particular interest was the discovery of the long-lasting TPM operations discovered during evaluation, which were totally unexpected. However, the reason for their long duration could not be determined.

The PoC demonstrated that DRIVE is well suited to be realized in a software implementation and not only in a conceptual or theoretical work. The security evaluation showed convincing results that confirmed successful detection of all anticipated attack scenarios. Effects on the SuE were noticeable, especially during complete system measurements. The effects could, however, be greatly reduced by incorporation of certain optimization strategies. Nevertheless, a more detailed analysis for real operation is recommended. The results of this chapter show that an implementation with subsequent evaluation is a very useful approach for validating and demonstrating designed concepts, architectures and mechanisms. The research question (Q4) can therefore be answered on the basis of the feasibility and applicability of the PoC and the results of the conducted evaluation studies.

As a consequence, the research objective (RO4) is considered to be fully achieved.

8.2 Outlook and Future Work

In the course of the work developed and presented in this thesis, some problems and challenges were discovered that are interesting for future work in the area of verification and evaluation of runtime memory. Some of these findings are now briefly presented and classified in the topic area.

Sophisticated Event-based Measurement Triggering The current process to trigger DRIVE measurements is either by explicit request, for instance after an attestation request was made, or based on a fixed timer. For the purpose of the PoC implementation, this is more than sufficient. For real operational deployments, however, it is expected that a more complex triggering mechanism for measurements is required, especially when a decision to continue an operation depends on a fast ad-hoc attestation. Providing event-based measurement triggering on the basis of monitoring execution of programs in the

OS scheduler or memory access permission operations were briefly mentioned as current problems. Yet, further approaches that address this problem exist and should be researched and studied more closely. This would enable DRIVE to be considered in more use-cases and increase its applicability.

Hardware Memory Bitflip Error Detection DRIVE offers the possibility of enabling the detection of unintended modification of predictable memory portions. These modifications are not necessarily carried out by explicit write operations; they may also occur on the basis of hardware-based memory attacks like Rowhammer [105] or even due to environmental effects, for instance, electrical or magnetic interference on the memory itself. In cases where error-correcting code memory is not used or too many bit-flips occurred to be handled properly, the software is in an undefined state. This can lead to deliberate or arbitrary errors during execution. DRIVE already detects these memory errors, but no studies have been conducted to investigate this behavior in more detail. Therefore, it is possible to carry out an evaluation for this particular error which analyzes this error more closely to offer strategies to remedy this problem. A concrete proposal for remediation is presented hereafter.

Self-healing Self-healing is a remediation strategy aimed at restoring a reliable state by recovering from modifications that have occurred. For the concept of self-healing, it is assumed that it is irrelevant whether an attack has occurred or whether environmental influences have caused a modification. The goal of self-healing is to repair the affected memory after an error has been recognized. The first ideas for implementing self-healing are, among others, restarting the affected program, using error correction codes or overwriting the memory on the basis of a trustworthy source. Although DRIVE is currently able to detect the error, it cannot determine the exact location in which the error occurred. This is a problem for the repair based on trusted sources because a hash digest-based measurement is too imprecise in this regard. For this reason, alternative algorithms should be evaluated that support a concrete localization of the error. In any case, research must consider the aforementioned and further possibilities in order to arrive at a sensible solution for self-healing.

Verification in Isolated Components In the course of this thesis, a remote attestation was described as the most secure variant of an attestation method since two physically isolated systems are involved. However, there are many applications where either no trustworthy third party exists or a remote attestation cannot be executed for other reasons. However, if there is another trusted component on the SuE, such as a TEE, a hypervisor or an enclave, verification can also be performed within this trusted component. Although the attestation procedures are similar, as described, these approaches should be studied in

order to determine the precise requirements, constraints and capabilities. One important question that must be solved in this regard is how to acquire and manage trusted reference data within the isolated component, especially considering that there might exist limitations with regard to available storage space or computational power.

Measurement Processes in Isolated Components Measuring processes that are to be carried out in isolated environments, i.e. in a component isolated from the operating system kernel, are a broad and very complex topic. Although related work methods have been presented to address this problem, they are far from a practical solution to the actual problem. Objectively speaking, the methods shown bypass the problem or accept the limitations associated with it. The core problem for virtualization solutions is also known as *semantic gap*, c.f. [95]. Similarly, this semantic gap exists whenever an external component does not have sufficient information. Thus, this problem is a general problem and can be mapped to enclaves or hardware-based components in the same way. For this reason, there is a need for research in this area to implement a measurement in trustworthy external components. This becomes necessary if the operating system kernel cannot be completely trusted. Particularly in cases in which there is no possibility of jumping back into a supervised hypervisor mode, a solution is considered to be very complex. Pfoh describes in [96] different methods to solve the problem for virtualized environments. Here it would be interesting to analyze to what extent these methods could also be applied to external components that do not offer hypervisor functionality.

Bibliography

[1] Andre Rein. "DRIVE: Dynamic Runtime Integrity Verification and Evaluation". In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security - ASIA CCS '17*. ACM Press, 2017, pp. 728–742. ISBN: 978-1-4503-4944-4. URL: http://dl.acm.org/citation.cfm?doid=3052973.3052975.

[2] Kai-Oliver Detken, Marcel Jahnke, Thomas Rix, and Andre Rein. "Software-design for internal security checks with dynamic Integrity Measurement (DIM)". In: *2017 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*. Vol. 1. IEEE, Sept. 2017, pp. 367–373. DOI: 10.1109/idaacs.2017.8095106.

[3] Andre Rein et al. "Trust Establishment in Cooperating Cyber-Physical Systems". In: *Cybersecurity of Industrial Control Systems, Security of Cyber Physical Systems*. Springer International Publishing, 2016, pp. 31–47. DOI: 10.1007/978-3-319-40385-4_3.

[4] Ralph Langner. "Stuxnet: Dissecting a cyberwarfare weapon". In: *IEEE Security & Privacy* 9.3 (2011), pp. 49–51. DOI: 10.1109/msp.2011.67.

[5] Ryan Gallagher. *OPERATION SOCIALIST The Inside Story of How British Spies Hacked Belgium's Largest Telco*. online. 2014. URL: https://theintercept.com/2014/12/13/belgacom-hack-gchq-inside-story/.

[6] Jessica Silver-Greenberg, Matthew Goldstein, and Nicole Perlroth. "JPMorgan Chase Hack Affects 76 Million Households". In: *New York Times* 2 (2014).

[7] netzpolitik.org. *Digital Attack on German Parliament: Investigative Report on the Hack of the Left Party Infrastructure in Bundestag*. online. 2015. URL: https://netzpolitik.org/2015/digital-attack-on-german-parliament-investigative-report-on-the-hack-of-the-left-party-infrastructure-in-bundestag/.

[8] GovCERT.ch. *Technical Report about the Espionage Case at RUAG*. Tech. rep. 2016. URL: https://www.melani.admin.ch/dam/melani/en/dokumente/2016/technical%20report%20ruag.pdf.download.pdf/Report_Ruag-Espionage-Case.pdf.

[9] Aleph One. "Smashing the Stack for Fun and Profit". In: *Phrack Volume 7, Issue 49* 7.49 (1996). URL: http://www.phrack.com/issues.html?issue=49&id=14.

[10] Kyung-Suk Lhee and Steve J Chapin. "Buffer overflow and format string overflow vulnerabilities". In: *Software: Practice and Experience* 33.5 (2003), pp. 423–460.

[11] Jesus Vigo. *Fileless malware: An undetectable threat*. online. June 2017. URL: https://www.techrepublic.com/article/fileless-malware-an-undetectable-threat/.

[12] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons, 2014. ISBN: 978-1-118-82509-9.

[13] Aravind Prakash, Eknath Venkataramani, Heng Yin, and Zhiqiang Lin. "On the Trustworthiness of Memory Analysis—An Empirical Study from the Perspective of Binary Execution". In: *Dependable and Secure Computing, IEEE Transactions on* 12.5 (2015), pp. 557–570.

[14] Nick L Petroni, Aaron Walters, Timothy Fraser, and William A Arbaugh. "FATKit: A framework for the extraction and analysis of digital forensic data from volatile system memory". In: *Digital Investigation* 3.4 (Dec. 2006), pp. 197–210. DOI: 10.1016/j.diin.2006.10.001.

[15] Joe Sylve. "Lime-linux memory extractor". In: *Proceedings of the 7th ShmooCon conference*. 2012.

[16] Volatility_Foundation. *Volatility Framework*. online. URL: http://www.volatilityfoundation.org.

[17] Chris Mitchell. *Trusted Computing (Professional Applications of Computing)*. IEE, 2005. ISBN: 978-0863415258.

[18] Tal Garfinkel et al. "Terra: A virtual machine-based platform for trusted computing". In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 193–206. DOI: 10.1145/1165389.945464.

[19] William A Arbaugh, David J Farber, and Jonathan M Smith. "A secure and reliable bootstrap architecture". In: *1997 IEEE Symposium on Security and Privacy (SP)*. IEEE. 1997, pp. 65–71.

[20] G. McGraw and G. Morrisett. "Attacking Malicious Code: A Report to the Infosec Research Council". In: *IEEE Software* 17.5 (Sept. 2000), pp. 33–41. ISSN: 0740-7459. DOI: 10.1109/52.877857.

[21] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014. ISBN: 978-1118804926.

[22] TIS Committee et al. "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2". In: *TIS Committee* (1995).

[23] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004. ISBN: 978-0131453487.

[24] Free Software Foundation. *ld(1) Linux User's Manual*. online. 2018. URL: http://man7.org/linux/man-pages/man1/ld.1.html.

[25] John R. Levine. *Linkers and Loaders*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999. ISBN: 1558604960.

[26] Ulrich Drepper. *How to write shared libraries*. online. Oct. 2011. URL: https://www.akkadia.org/drepper/dsohowto.pdf.

[27] Murugiah Souppaya and Karen Scarfone. "Guide to Malware Incident Prevention and Handling for Desktops and Laptops". In: *International Journal of Computer Research* 20.4 (2013), p. 417. DOI: 10.6028/nist.sp.800-83r1.

[28] Andrew S. Tanenbaum. *Modern Operating Systems*. 3rd. Upper Saddle River, NJ, USA: Prentice-Hall, 2007. ISBN: 0136006639.

[29] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. "A taxonomy of computer program security flaws". In: *ACM Computing Surveys (CSUR)* 26.3 (1994), pp. 211–254.

[30] Ken Dunham and Jim Melnick. *Malicious bots: an inside look into the cyber-criminal underground of the internet*. CrC Press, 2008. ISBN: 978-1420069037.

[31] Joanna Rutkowska. "Introducing stealth malware taxonomy". In: *COSEINC Advanced Malware Labs* (2006), pp. 1–9. URL: http://blog.invisiblethings.org/papers/2006/rutkowska_malware_taxonomy.pdf.

[32] Fred Cohen. "Computer viruses: theory and experiments". In: *Computers & security* 6.1 (1987), pp. 22–35.

[33] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. "Design and Implementation of a TCG-based Integrity Measurement Architecture." In: *USENIX Security Symposium*. Vol. 13. 2004, pp. 223–238. URL: https://www.usenix.org/legacy/event/sec04/tech/full_papers/sailer/sailer_html/.

[34] Peter Szor. *The art of computer virus research and defense*. Pearson Education, 2005. ISBN: 978-0321304544.

[35] Nick L. Petroni Jr and Michael Hicks. "Automated detection of persistent kernel control-flow attacks". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 103–115. URL: http://dl.acm.org/citation.cfm?id=1315260.

[36] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. "Countering persistent kernel rootkits through systematic hook discovery". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2008, pp. 21–38.

[37] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. "Sok: Eternal war in memory". In: *2013 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2013, pp. 48–62. URL: https://people.eecs.berkeley.edu/~dawnsong/papers/Oakland13-SoK-CR.pdf.

[38] Mathias Payer. *Software Security: Memory Safety*. online. Department of Computer Science Purdue University, 2017. URL: https://nebelwelt.net/teaching/17-527-SoftSec/slides/02-memory_safety.pdf.

[39] Nicholas Carlini et al. "Control-flow bending: On the effectiveness of control-flow integrity". In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 161–176. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini.

[40] Solar Designer. *Getting around non-executable stack (and fix)*. online. 1997. URL: http://www.securityfocus.com/archive/1/7480.

[41] Hovav Shacham. "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)". In: *Proceedings of the 14th ACM conference on Computer and communications security*. ACM. 2007, pp. 552–561. DOI: 10.1145/1315245.1315313.

[42] Hovav Shacham, E Buchanan, R Roemer, and S Savage. "Return-oriented programming: Exploits without code injection". In: *Black Hat USA Briefings (August 2008)* (2008).

[43] Stephen Checkoway et al. "Return-oriented programming without returns". In: *Proceedings of the 17th ACM conference on Computer and communications security*. ACM. 2010, pp. 559–572. DOI: 10.1145/1866307.1866370.

[44] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. "Jump-oriented Programming: A New Class of Code-reuse Attack". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. Hong Kong, China: ACM, 2011, pp. 30–40. ISBN: 978-1-4503-0564-8. URL: http://doi.acm.org/10.1145/1966913.1966919.

[45] Erik Bosman and Herbert Bos. "Framing Signals - A Return to Portable Shellcode". In: *2014 IEEE Symposium on Security and Privacy (SP)*. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 243–258. ISBN: 978-1-4799-4686-0. URL: http://dx.doi.org/10.1109/SP.2014.23.

[46] Felix Schuster et al. "Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications". In: *2015 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2015. DOI: 10.1109/SP.2015.51.

[47] Tim Rains, Matt Miller, and David Weston. "Exploitation Trends: From Potential Risk to Actual Risk". In: *RSA Conference*. 2015.

[48] Ben Gras et al. “ASLR on the Line: Practical Cache Attacks on the MMU”. In: *Proceeding of the Network and Distributed System Security Symposium (NDSS '17)*. 2017. URL: http://www.cs.vu.nl/~giuffrida/papers/anc-ndss-2017.pdf.

[49] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. “Control-flow integrity”. In: *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353. URL: http://dl.acm.org/citation.cfm?id=1102165.

[50] The Clang Team. *Clang 7 documentation: Control Flow Integrity*. online. 2018. URL: https://clang.llvm.org/docs/ControlFlowIntegrity.html.

[51] Caroline Tice et al. “Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM.” In: *USENIX Security Symposium*. 2014, pp. 941–955.

[52] Microsoft®. *Control Flow Guard*. online. 2016. URL: https://docs.microsoft.com/en-us/cpp/build/reference/guard-enable-control-flow-guard.

[53] Nathan Burow et al. “Control-Flow Integrity: Precision, Security, and Performance”. In: *ACM Computing Surveys* 50.1 (Apr. 2017), 16:1–16:33. ISSN: 0360-0300. URL: http://doi.acm.org/10.1145/3054924.

[54] Intel®. *Control-flow Enforcement Technology*. online. July 2017. URL: https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf.

[55] Mauro Conti et al. “Losing Control: On the Effectiveness of Control-Flow Integrity Under Stack Attacks”. In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 952–963. ISBN: 978-1-4503-3832-5. URL: http://doi.acm.org/10.1145/2810103.2813671.

[56] Ruan de Clercq and Ingrid Verbauwhede. “A survey of Hardware-based Control Flow Integrity (CFI)”. In: *arXiv preprint arXiv:1706.07257* (2017).

[57] Adam Shostack. *Threat Modeling: Designing for Security*. John Wiley & Sons, 2014. ISBN: 978-1118809990.

[58] Microsoft®. *The STRIDE Threat Model*. online. 2005. URL: https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx.

[59] Starr Andersen and Vincent Abella. *Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies*. 2004.

[60] V. Katoch. *Whitepaper on Bypassing ASLR/DEP*. online. URL: http://www.exploit-db.com/wp-content/themes/exploit/docs/17914.pdf.

[61] OWASP. *Definition of format string attacks*. https://www.owasp.org/index.php/Format_string_attack.

[62] Michael Howard and David LeBlanc. "Writing secure code". In: Pearson Education, 2003. Chap. 5, Array Indexing Errors, p. 144.

[63] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. "Return-oriented programming: Systems, languages, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 15.1 (2012), p. 2. DOI: 10.1145/2133375.2133377.

[64] Mathias Payer and Thomas R. Gross. "String Oriented Programming: When ASLR is Not Enough". In: *Proceedings of the 2Nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. PPREW '13. Rome, Italy: ACM, 2013, 2:1–2:9. ISBN: 978-1-4503-1857-0. URL: http://doi.acm.org/10.1145/2430553.2430555.

[65] Perry Wagle, Crispin Cowan, et al. "Stackguard: Simple stack smash protection for gcc". In: *Proceedings of the GCC Developers Summit*. Citeseer. 2003, pp. 243–255.

[66] Tobias Klein. *RELRO - A (not so well known) Memory Corruption Mitigation Technique*. online. URL: http://tk-blog.blogspot.de/2009/02/relro-not-so-well-known-memory.html.

[67] David Drysdale. *Anatomy of a system call, part 1 [LWN.net]*. online. 2014. URL: https://lwn.net/Articles/604287/.

[68] Tyler Nichols. *Hooking the Linux System Call Table*. online. 2015. URL: https://www.tnichols.org/2015/10/19/Hooking-the-Linux-System-Call-Table/index.html.

[69] G.F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi. "Surgically Returning to Randomized lib(c)". In: *Computer Security Applications Conference, 2009. ACSAC '09. Annual*. IEEE, Dec. 2009, pp. 60–69. DOI: 10.1109/ACSAC.2009.16.

[70] Amol Sarwate. *The GHOST Vulnerability*. online. 2015. URL: https://blog.qualys.com/laws-of-vulnerabilities/2015/01/27/the-ghost-vulnerability.

[71] Checkpoint SoftwareTechnologies Ltd. *Whitepaper on Quadrooter*. URL: https://www.checkpoint.com/resources/quadrooter-vulnerability-enterprise/.

[72] Wen Xu et al. "From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel". In: *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*. CCS '15. Denver, Colorado, USA: ACM, 2015, pp. 414–425. ISBN: 978-1-4503-3832-5. URL: http://doi.acm.org/10.1145/2810103.2813637.

[73] CVEdetails.com. *Linux Kernel Vulnerabilities*. online. URL: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33.

[74] Tal Garfinkel, Mendel Rosenblum, et al. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *NDSS*. Vol. 3. 2003. 2003, pp. 191–206.

[75] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. "Secure virtual architecture: A safe execution environment for commodity operating systems". In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 6. ACM. 2007, pp. 351–366. DOI: 10.1145/1294261.1294295.

[76] Ahmed M Azab, Peng Ning, Emre C Sezer, and Xiaolan Zhang. "HIMA: A hypervisor-based integrity measurement agent". In: *Computer Security Applications Conference, 2009. ACSAC'09. Annual*. IEEE. 2009, pp. 461–470. DOI: 10.1109/acsac.2009.50.

[77] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, pp. 335–350. ISBN: 978-1-59593-591-5. URL: http://doi.acm.org/10.1145/1294261.1294294.

[78] Udo Steinberg and Bernhard Kauer. "NOVA: a microhypervisor-based secure virtualization architecture". In: *Proceedings of the 5th European conference on Computer systems*. ACM. 2010, pp. 209–222. DOI: 10.1145/1755913.1755935.

[79] Chung Hwan Kim et al. "CAFE: A Virtualization-Based Approach to Protecting Sensitive Cloud Application Logic Confidentiality". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ASIA CCS '15. Singapore, Republic of Singapore: ACM, 2015, pp. 651–656. ISBN: 978-1-4503-3245-3. URL: http://doi.acm.org/10.1145/2714576.2714594.

[80] Owen S Hofmann et al. "Inktag: Secure applications on an untrusted operating system". In: *ACM SIGARCH Computer Architecture News*. Vol. 41. 1. ACM. 2013, pp. 265–278. DOI: 10.1145/2499368.2451146.

[81] Raoul Strackx and Frank Piessens. "Fides: Selectively hardening software application components against kernel-level or process-level malware". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 2–13. DOI: 10.1145/2382196.2382200.

[82] Yueqiang Cheng, Xuhua Ding, and Robert H Deng. "Efficient virtualization-based application protection against untrusted operating system". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM. 2015, pp. 345–356. DOI: 10.1145/2714576.2714618.

[83] Ahmed M Azab et al. "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 90–102. DOI: 10.1145/2660267.2660350.

[84] Xinyang Ge, Hayawardh Vijayakumar, and Trent Jaeger. "Sprobes: Enforcing kernel code integrity on the trustzone architecture". In: *arXiv preprint arXiv:1410.7747* (2014).

[85] Matthew Hoekstra et al. "Using innovative instructions to create trustworthy software solutions." In: *HASP@ ISCA*. 2013, p. 11. DOI: 10.1145/2487726.2488370.

[86] Frank McKeen et al. "Innovative instructions and software model for isolated execution." In: *HASP@ ISCA*. 2013, p. 10. DOI: 10.1145/2487726.2488368.

[87] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. "Innovative technology for CPU based attestation and sealing". In: *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*. Vol. 13. 2013.

[88] Ahmed M Azab et al. "SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM". In: (2016). DOI: 10.14722/ndss.2016.23009.

[89] Nick L Petroni Jr, Timothy Fraser, Jesus Molina, and William A Arbaugh. "Copilot-a Coprocessor-based Kernel Runtime Integrity Monitor." In: *USENIX Security Symposium*. San Diego, USA. 2004, pp. 179–194.

[90] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. "Detecting kernel-level rootkits using data structure invariants". In: *IEEE Transactions on Dependable and Secure Computing* 8.5 (2011), pp. 670–684. DOI: 10.1109/tdsc.2010.38.

[91] Jan-Erik Ekberg, Kari Kostiainen, and N. Asokan. "Trusted Execution Environments on Mobile Devices". In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. CCS '13. Berlin, Germany: ACM, 2013, pp. 1497–1498. ISBN: 978-1-4503-2477-9. URL: http://doi.acm.org/10.1145/2508859.2516758.

[92] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[93] Intel®. *Intel Software Guard Extensions*. online. 2017. URL: https://software.intel.com/en-us/sgx.

[94] David Kaplan, Jeremy Powell, and Tom Woller. *AMD memory encryption*. 2016. URL: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf.

[95] Peter M. Chen and Brian D. Noble. "When virtual is better than real operating system relocation to virtual machines". In: *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. IEEE, 2001, pp. 133–138. DOI: 10.1109/hotos.2001.990073.

[96] Jonas Pfoh, Christian Schneider, and Claudia Eckert. "A Formal Model for Virtual Machine Introspection". In: *Proceedings of the 1st ACM Workshop on Virtual Machine Security*. VMSec '09. Chicago, Illinois, USA: ACM, 2009, pp. 1–10. ISBN: 978-1-60558-780-6. URL: http://doi.acm.org/10.1145/1655148.1655150.

[97] M. Haardt and M. Coleman. *ptrace (2)*. online. 1999. URL: http://man7.org/linux/man-pages/man2/ptrace.2.html.

[98] Thomas Kittel et al. "Code validation for modern os kernels". In: *Workshop on Malware Memory Forensics (MMF)*. 2014.

[99] Peter Silberman and Richard Johnson. "A comparison of buffer overflow prevention implementations and weaknesses". In: *IDEFENSE, August* (2004).

[100] PaX Team. *PaX mmap() and mprotect() restrictions*. 2003. URL: https://pax.grsecurity.net/docs/mprotect.txt.

[101] Qualys. *The Stack Clash*. online. June 2017. URL: https://blog.qualys.com/securitylabs/2017/06/19/the-stack-clash.

[102] Paul Kocher et al. "Spectre Attacks: Exploiting Speculative Execution". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01203.

[103] Moritz Lipp et al. "Meltdown". In: *ArXiv e-prints* (Jan. 2018). arXiv: 1801.01207.

[104] Carsten Bormann and Paul Hoffman. *Concise Binary Object Representation (CBOR)*. online. 2013. URL: https://tools.ietf.org/html/rfc7049.

[105] Mark Seaborn and Thomas Dullien. "Exploiting the DRAM rowhammer bug to gain kernel privileges". In: *Black Hat*. 2015, pp. 7–9. URL: https://www.%20blackhat%20com/docs/us-15/materials/us-15-Seaborn-Exploiting-The-DRAM-Rowhammer-Bug-To-Gain-Kernel-Privileges.pdf.

[106] Trent Jaeger, Reiner Sailer, and Umesh Shankar. "PRIMA: Policy-reduced Integrity Measurement Architecture". In: SACMAT '06 (2006), pp. 19–28. DOI: 10.1145/1133058.1133063.

[107] Ahmad-Reza Sadeghi and Christian Stüble. "Property-based Attestation for Computing Platforms: Caring About Properties, Not Mechanisms". In: *Proceedings of the 2004 Workshop on New Security Paradigms*. NSPW '04. Nova Scotia, Canada: ACM, 2004, pp. 67–77. ISBN: 1-59593-076-0. URL: http://doi.acm.org/10.1145/1065907.1066038.

[108] Microsoft®. *Secure boot*. online. May 2017. URL: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot.

[109] *IMA. Integrity Measurement Architecture*. online. URL: http://sourceforge.net/p/linux-ima/wiki/Home.

[110] Nikolay Elenkov. *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press, 2014.

[111] Ryan P Nakamoto. *Secure Boot and Image Authentication*. online. URL: https://www.qualcomm.com/media/documents/files/secure-boot-and-image-authentication-technical-overview.pdf.

[112] Apple®. *iOS Security*. online. Jan. 2018. URL: https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[113] Joanna Rutkowska. *System Virginity Verifier*. online. 2005. URL: http://www.cs.dartmouth.edu/~sergey/cs258/rootkits/hitb05_virginity_verifier.ppt.

[114] Peter A. Loscocco, Perry W. Wilson, J. Aaron Pendergrass, and C. Durward McDonell. "Linux Kernel Integrity Measurement Using Contextual Inspection". In: *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*. STC '07. Alexandria, Virginia, USA: ACM, 2007, pp. 21–29. ISBN: 978-1-59593-888-6. DOI: 10.1145/1314354.1314362.

[115] J. Aaron Pendergrass and Kathleen N. McGill. "LKIM: The Linux Kernel Integrity Measurer". In: *Johns Hopkins APL technical digest* 32.2 (2013), p. 509.

[116] Lionel Litty, H Andrés Lagar-Cavilla, and David Lie. "Hypervisor Support for Identifying Covertly Executing Binaries." In: *USENIX Security Symposium*. 2008, pp. 243–258.

[117] Arvind Seshadri et al. "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems". In: *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*. Vol. 39. SOSP '05. Brighton, United Kingdom: ACM, 2005, pp. 1–16. ISBN: 1-59593-079-5. URL: http://doi.acm.org/10.1145/1095810.1095812.

[118] Zhi Wang and Xuxian Jiang. "Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity". In: *2010 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2010, pp. 380–395. DOI: 10.1109/sp.2010.30.

[119] Bryan D Payne, Martim Carbone, Monirul Sharif, and Wenke Lee. "Lares: An architecture for secure active monitoring using virtualization". In: *2008 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2008, pp. 233–247. DOI: 10.1109/sp.2008.24.

[120] Chaowen Chang, Xin Chen, Shuai Wang, and Qinghai Xiao. "Research on Dynamic Integrity Measurement Model Based on Memory Paging Mechanism". In: *Discrete Dynamics in Nature and Society* 2014 (2014). DOI: 10.1155/2014/478985.

[121] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), p. 4. URL: http://dl.acm.org/citation.cfm?id=1609960.

[122] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. "Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection". und. In: OCLC: 254320948. Berkeley, Calif: USENIX Association, 2014. ISBN: 978-1-931971-15-7.

[123] Patrick Wollgast et al. "Automated Multi-architectural Discovery of CFI-Resistant Code Gadgets". In: *European Symposium on Research in Computer Security*. Springer, 2016, pp. 602–620. URL: http://link.springer.com/chapter/10.1007/978-3-319-45744-4_30.

[124] Nicholas Carlini and David Wagner. "ROP is Still Dangerous: Breaking Modern Defenses". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 385–399. ISBN: 978-1-931971-15-7. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[125] Enes Göktaş et al. "Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard". und. In: OCLC: 254320948. Berkeley, Calif: USENIX Association, 2014. ISBN: 978-1-931971-15-7.

[126] PaX Team. *Rap: Rip rop*. online. 2015. URL: http://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf.

[127] Qualcomm. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. online. Jan. 2017. URL: https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf.

[128] Shuo Chen et al. "Non-Control-Data Attacks Are Realistic Threats." In: *Usenix Security*. Vol. 5. 2005. URL: http://static.usenix.org/legacy/events/sec05/tech/full_papers/chen/chen_html/.

[129] Hong Hu et al. "Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks". In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 969–986. DOI: 10.1109/sp.2016.62.

[130] Miguel Castro, Manuel Costa, and Tim Harris. "Securing software by enforcing data-flow integrity". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 147–160. URL: http://dl.acm.org/citation.cfm?id=1298470.

[131] Chengyu Song et al. "Enforcing Kernel Security Invariants with Data Flow Integrity". In: *Proceedings of the 23th Annual Network and Distributed System Security Symposium*. 2016. URL: http://www.cc.gatech.edu/grads/c/csong43/ndss16-kenali.pdf.

[132] Jingfei Kong, Cliff C. Zou, and Huiyang Zhou. "Improving software security via runtime instruction-level taint checking". In: *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006, pp. 18–24. URL: http://dl.acm.org/citation.cfm?id=1181313.

[133] Pankaj Kohli. "Coarse-grained Dynamic Taint Analysis for Defeating Control and Non-control Data Attacks". In: *arXiv preprint arXiv:0906.4481* (2009). URL: https://arxiv.org/abs/0906.4481.

[134] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)". In: *2010 IEEE Symposium on Security and privacy (SP)*. IEEE, 2010, pp. 317–331. URL: http://ieeexplore.ieee.org/abstract/document/5504796/.

[135] Toktam Ramezanifarkhani and Mohammadreza Razzazi. "Principles of Data Flow Integrity: Specification and Enforcement." In: *J. Inf. Sci. Eng.* 31.2 (2015), pp. 529–546. URL: http://www.iis.sinica.edu.tw/page/jise/2015/201503_10.pdf.

Acronyms

ABI	Application Binary Interface
API	Application Binary Interface
APT	Advanced Persistent Threat
ASLR	Address Space Layout Randomization
BIOS	Basic Input/Output System
CBOR	Concise Binary Object Representation
CFB	Control Flow Bending
CFG	Control Flow Graph
CFI	Control Flow Integrity
CFM	Control Flow Manipulation
CHF	Cryptographic Hash Functions
CIAP	Code Injection Attack Pattern
COOP	Counterfeit Object Oriented Programming
COW	Copy-On-Write
CPM	Code Pointer Manipulation
CPMAP	Code Pointer Manipulation Attack Pattern
CPU	Central Processing Unit
DCI	DRIVE Control Interface
DDoS	Distributed Denial of Service
DEP	Data Execution Prevention
DFI	Data Flow Integrity
DMAP	Data Manipulation Attack Pattern
DMC	DRIVE Measurement Component
DML	Dynamic Measurement List

DRIVE Dynamic Integrity Runtime Verification and Evaluation
DVC DRIVE Verification Component

ELF Executable and Linkable Format

GDB GNU Debugger
GOT Global Offset Table

IDAP Information Disclosure Attack Pattern
IMA Integrity Measurement Architecture
IP Instruction Pointer
ISA Instruction Set Architecture

JIT Just in Time
JOP Jump Oriented Programming
JSON JavaScript Object Notation
JVM Java Virtual Machine

KML Kernel Module Loader

LKM Loadable Kernel Module
LV Local Verification

MA Measurement Agent
MMU Memory Management Unit

OS Operating System

PCR Platform Configuration Register
PIC Position Independent Code
PIE Position Independent Executable
PLT Procedure Linkage Table
PoC Proof of Concept
PTE Page Table Entry

RCAP Code Replacement Attack Pattern
RCC Relocatable Code
ROP Return Oriented Programming

RoT	Root of Trust
RVD	Reference Value Data
RVG	Reference Value Generator
RVS	Reference Value Storage
SCADA	Supervisory Control and Data Acquisition
SPI	Serial Peripheral Interface
SROP	Sigretrun Oriented Programming
SSR	System State Report
SuE	System under Evaluation
TCG	Trusted Computing Group Module
TCP	Transmission Control Protocol
TEE	Trusted Execution Environment
TLS	Transport Layer Security
TPM	Trusted Platform Module
TTP	Trusted Third Party
VA	Verification Agent
VAS	Virtual Address Space
VE	Virtualization Environment
VM	Virtual Machine
VMA	Virtual Memory Address
VMM	Virtual Machine Manager
VS	Verification System

Notations

AK	private attestation key
ak	public attestation key
CFP	calculated fingerprint
FP	security module anchored fingerprint
SFP	signed fingerprint
$SNonce$	signed nonce
mhd	measured hash digest
ehd	expected hash digest
map	memory access permissions
mea	memory end address
mf	mapped filename
mi	measured information
msa	memory start address
ms	memory size
MS	measurement set
HMS	hashed measurement set
S	measurement information set
$tptf$	temporary program text file